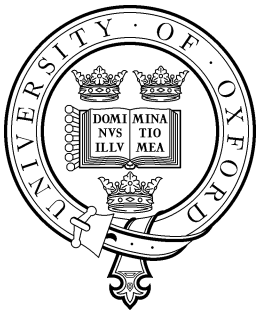
Verification of Asynchronous Concurrency and the Shaped Stack Constraint



Jonathan Kochems
Worcester College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

November 2014

To my late father, who sparked my interest in computing in the first place.

And, to my wonderful mother whose unwavering support enabled me to come this far.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor Luke Ong who has always had an open door for me. I have been very fortunate to receive Luke's mentorship and advice for many years both as an undergraduate and doctoral student.

My thanks to Matthew Hague, Stefan Kiefer, Subodh Sharma, Michael Tautschnig, and Emanuele D'Osualdo, who have helped me refine my ideas and arguments through many conversations and discussions, and to Sylvain Schmitz, Javier Esparza, Ranko Lazic, Jérôme Leroux, Pierre Ganty, Alain Finkel, and the anonymous reviewers of my articles, whose feedback considerably improved my work. I would also like to mention my examiners James Worrell and Rupak Majumdar and thank them for their insightful comments and inspiring questions.

I am very grateful to my funding body, the Engineering and Physical Sciences Research Council (EPSRC), and to the Department of Computer Science and the University of Oxford for giving me the opportunity to study at such a fantastic institution.

To my colleagues in office 347 — Robin Neatherway, Conrad Cotton-Barratt, Egor Ianovski, Martin Lester, Emanuele D'Osualdo, and Marcelo Sousa — I would like to say thanks for the supportive environment and for making the last few years a time that I will remember fondly.

Last but not least, I would like to thank my family and friends, and my beloved fiancée Celia for encouraging me and believing in me.

Abstract

In this dissertation, we study the verification of concurrent programs written in the programming language Erlang using infinite-state model-checking. Erlang is a widely used, higher order, dynamically typed, call-by-value functional language with algebraic data types and pattern-matching. It is further augmented with support for *actor concurrency*, i.e. asynchronous message passing and dynamic process creation.

With decidable model-checking in mind, we identify *actor communicating systems* (ACS) as a suitable target model for an abstract interpretation of Erlang. ACS model a dynamic network of finite-state processes that communicate over a fixed, finite number of unordered, unbounded channels. Thanks to being equivalent to Petri nets, ACS enjoy good algorithmic properties. We develop a verification procedure that extracts a sound abstract model, in the form of an ACS, from a given Erlang program; the resulting ACS simulates the operational semantics of the input. Using this abstract model, we can conservatively verify *coverability* properties of the input program, i.e. a weak form of safety properties, with a Petri net model-checker. We have implemented this procedure in our tool Soter, which is the first sound verification tool for Erlang programs using infinite-state model-checking. In our experiments, we find that Soter is accurate enough to verify a range of interesting and non-trivial benchmarks. Even though ACS coverability is EXPSPACE-complete, Soter's analysis of these verification problems is surprisingly quick.

In order to improve the precision of our verification procedure with respect to recursion, we investigate an extension of ACS that allows *pushdown* processes: *asynchronously communicating pushdown systems* (ACPS). ACPS that satisfy the empty-stack constraint (a pushdown process may receive only when its stack is empty) are a popular subclass of ACPS with good decision and complexity properties. In the context of Erlang, the empty stack constraint is unfortunately not realistic. We introduce a relaxation of the empty-stack constraint for ACPS called the *shaped stack* constraint. Stacks that fit the shape constraint may reach arbitrary heights. Further, a process may execute any communication action (be it process creation, message send or retrieval) whether or not its stack is empty. We prove that coverability for *shaped ACPS*, i.e. ACPS that satisfy the shaped constraint, reduces to the decidable coverability problem for *well-structured transition systems* (WSTS). Thus, shaped ACPS enable the modelling and verification of a larger class of message passing programs.

We establish a close connection between shaped ACPS and a novel extension of Petri nets: *nets with nested coloured tokens* (NNCT). Tokens in NNCT are of two types: *simple* and *complex*. Complex tokens carry an arbitrary number of *coloured* tokens. The rules of a NNCT can synchronise complex and simple tokens, inject coloured tokens into a complex token, and eject all tokens of a specified set of *active* colours to predefined places. We show that the coverability problem for NNCT is TOWER-complete, a new complexity class for non-elementary decision problems introduced by Schmitz. To prove TOWER-membership, we devise a geometrically inspired version of the *Rackoff technique*, and we obtain TOWER-hardness by adapting *Stockmeyer's ruler construction* to NNCT. To our knowledge, NNCT is the first extension of Petri nets (belonging to the class of nets with an infinite set of token *types*) that is proven to have primitive recursive coverability. This result implies TOWER-completeness of coverability for ACPS that satisfy the shaped stack constraint.

Contents

Chapter 1

Introduction

'Concurrent programs are hard to write and just as hard to verify.' This mantra is prevalent in the verification literature on concurrent systems and concisely states the challenge faced by the field. In this dissertation, we aim to present progress on the state of the art of a practical and theoretical nature. The verification of concurrent programs written in the programming language Erlang is the ultimate target of our work.

Erlang was originally designed to program fault-tolerant distributed systems at Ericsson in the late 1980s. It is now a widely used, open-source language with support for higher-order functions, concurrency, communication, distribution, on-the-fly code reloading, and multiple platforms [AVW93; CT09; Arm10]. Erlang's runtime system offers highly efficient process creation and message-passing communication. This makes Erlang a natural fit for programming multicore CPUs, networked servers, parallel databases, GUIs, and monitoring, control, and testing tools. Erlang is frequently described as the first *concurrency oriented* programming language [Lar09] and often called the gold standard for concurrent functional programming languages [Arm10]. Erlang's sequential fragment is a higher order, dynamically typed, call-by-value functional language with algebraic data types and pattern-matching. Following the *actor model* [Agh86], a concurrent Erlang computation consists of a dynamic network of processes that communicate by asynchronous message passing. Each process has a unique process identifier and is equipped with an unbounded mailbox. Messages are sent asynchronously. A process may attempt to retrieve a message that matches a specified pattern from its mailbox. This extracts the first matching message if one exists and blocks otherwise. Hence, the retrieval of messages from the mailbox is not FIFO but First-In-First-Firable-Out (FIFFO). For a highly readable introduction to Erlang, see Armstrong's *CACM* article [Arm10].

Even though Erlang is built from the ground up for concurrency, all classical faulty concurrent behaviours can and do appear in software written in Erlang. Races on shared resources, deadlock, livelock, unbounded systems, and traditional sequential errors are only an incomplete list of the software defects that arise in Erlang programs. The inherent difficulty in concurrent program design means that such software defects are readily introduced. Even extensive testing cannot detect all incorrect implementations. A class of software defects, called *heisenbugs*, require unlikely sequences of events to manifest themselves and are thus almost impossible to reproduce and localise through testing. Nevertheless such defects can have severe consequences.

There is thus a clear need for automated methods and tools that can either prove the absence of, or find, such software faults. Our work focusses on techniques that enable the verification of properties such as program point reachability, mutual exclusion, system boundedness, and absence of non-termination.

Challenges. The verification of sequential programs is already an undecidable problem. Nevertheless, in many instances it is possible to automatically uncover real software defects or algorithmically prove correctness. In the sequential case, this can often be achieved by sufficiently accurate *conservative* analyses, i.e. analyses that may erroneously classify a safe program as unsafe, yet may never identify an unsafe program as safe.

In the case of Erlang, the difficulties that arise in the verification of sequential programs are magnified. To algorithmically prove that an Erlang program satisfies a property, we must reason about the asynchronous communication of an unbounded set of messages across an unbounded set of Turing-powerful, higher-order processes. The inherent complexity of this task can be seen from several diverse sources of infinity in the state space [Huc99]:

(∞ 1) Function definitions are not necessarily tail-recursive. Thus, each concurrent process requires a local unbounded call-stack.

(∞ 2) Higher-order functions are first-class values. Arbitrary closures can be created, passed as arguments, and be returned by a function call.

(∞ 3) Data domains, and hence the message space, are unbounded: functions may return, and variables may be bound to, algebraic data structures of arbitrary size.

(∞ 4) Processes creation is dynamic. There is no *a priori* bound on the number of processes that may be spawned.

(∞ 5) Each process is equipped with a dedicated mailbox which has unbounded capacity.

These sources of infinity in the state space are a manifestation of Erlang's many diverse Turing-powerful fragments. Erlang offers a rich set of features such as higher-order functions, algebraic data, general recursion, FIFFO message passing, dynamic process and name creation, and name mobility. These features, in isolation or in combination, enable several distinct encodings of Turing machines. For example, Erlang's sequential fragment with higher-order functions and algebraic data [OR11], the concurrent first-order functional fragment with limited synchronisations [Ram00], and the concurrent, order-zero, finite-data, tail-recursive fragment [BZ83] are all Turing powerful. These undecidability results make the prospect of performing verification on Erlang look daunting, but not hopeless.

1.1 Overview

To reduce the complexity of the full Erlang programming language, we introduce λ_{Actor} as a formal object of study. λ_{Actor} is an idealised untyped functional language with actor concurrency. It is designed to capture the key features of single-node *Core Erlang* [Car01]—the official intermediate representation of Erlang code. In particular, λ_{Actor} subsumes Core Erlang's higher-order functional fragment with algebraic data and communication primitives. However, the fault-tolerance features and built-in functions of Core Erlang are not present in λ_{Actor}.

Infinite-State Verification. Developing verification methods for Erlang is difficult. Previous work has focussed on under-approximation [FS07; LS04] or semi-automatic techniques [Art+98; AEP04; Nys03] and restrictive fragments [Huc99; AD99; MW97]. It is our goal to improve on the state of the art by developing automatic verification procedures that make use of infinite-state model-checking. We approach the problem of verifying λ_{ACTOR} programs with the following strategy:

(i) Choose a computational model that is expressive enough to model key features of λ_{ACTOR} and has good algorithmic properties.

(ii) Generate a sound model from a λ_{ACTOR} program automatically using abstraction.

In Chapter 3, we introduce *actor communicating systems* (ACS) which model a dynamic network of finite-state processes that communicate over a finite number of unordered, unbounded channels. The capabilities of ACS make it a suitable candidate target model for the abstract interpretation of λ_{ACTOR} programs. In principle, it is possible to model dynamic process creation (∞ 4) and the unbounded capacity of mailboxes (∞ 5) in ACS. However, it is necessary to abstract general recursion (∞ 1), higher-order functions (∞ 2), and data (∞ 3) to obtain an ACS from a λ_{ACTOR} program. ACS give rise to well-known and well-studied infinite state systems: ACS are equivalent to Petri nets [Muk+98b] and thus enjoy good algorithmic properties. In order to obtain a procedure that abstracts (∞ 1), (∞ 2), and (∞ 3), we apply the principled methodology of *abstracting abstract machines* (AAM) [HM11; HM12] introduced by Horn and Might. The AAM methodology provides guidance in the design of a systematic abstraction interpretation [CC77] for higher-order functional languages. We apply the AAM methodology in the novel setting of actor-style concurrency to *automatically* extract an abstract model, in the form of an ACS, that simulates the operational semantics of an input λ_{ACTOR} program. Thus, we can use the generated ACS to conservatively answer *coverability* questions on the input program. Coverability is a reachability-like decision problem which asks given a start s_0 and a target configuration s_{cov} whether it is possible to reach a configuration s from s_0 such that s *covers* s_{cov} ($s \geq s_{\text{cov}}$) with respect to some order $\geq$ on configurations. Coverability is an important decision problem in the context of safety verification. We have implemented this procedure in our tool Soter which is the first sound verification tool for Erlang programs using infinite-state model-checking. Soter generates an ACS model from Erlang source code that can be annotated with assertions. The assertions and the abstract model are then compiled to a query for a Petri net coverability model-checker. Our experimental results for Soter are encouraging and indicate that infinite-state model-checking can be applied successfully to the verification of Erlang.

The Shaped Stack Constraint. Even though Soter is able to verify a range of non-trivial benchmarks, we observe that the control-flow abstractions that we have to employ impose a cost on precision. The fact that processes in ACS are finite-state forces us to significantly abstract the recursive (higher-order) processes arising in λ_{ACTOR}. The loss of precision is considerable. For example, consider a program $\mathcal{P}$ with procedure F and a finite-state program $\hat{\mathcal{P}}$ obtained by abstraction from $\mathcal{P}$. In the general case, an unbounded call-stack is required to remember the context of a call to F. After a call to F completes, control returns to a program point that is determined by this calling context. Unfortunately, we cannot use an unbounded call-stack in a finite-state program. In a finite-state program $\hat{\mathcal{P}}$ we can only remember a finite part of the con-

text in which F is called. When F returns in $\hat{\mathcal{P}}$ we must therefore introduce non-determinism to remain sound. This non-determinism leads to spurious returns, i.e. a call to F in context C may return to a call site in $\hat{\mathcal{P}}$ to which it is not possible to return to in $\mathcal{P}$ in the same context C. Thus, a finite-state program that is obtained by abstraction from a fully recursive program cannot, in general, faithfully model recursive procedure calls.

If we extend ACS by allowing processes to be recursive or *pushdown*, we obtain *asynchronously communicating pushdown systems* (ACPS). Similarly to ACS, ACPS model a dynamic network of *pushdown* processes that communicate over a fixed, finite number of unordered, unbounded channels. Unfortunately, ACPS are Turing powerful [Ram00]. In the general case, we cannot perform algorithmic verification on them.

Faced with this undecidability frontier, modelling the recursive definitions that arise in λ_{ACTOR} poses an interesting challenge: Can we find a subclass of ACPS that (i) has good algorithmic properties, (ii) extends the capabilities of ACS, and (iii) is expressive enough to model a wide range of recursive function definitions? An answer to this question is, of course, of much wider interest than just to improve the precision of our verification procedure for Erlang.

We dedicate Chapters 4 and 5 to an investigation into this question. Previous work established that ACPS that satisfy the *empty-stack constraint* [SV06] give rise to a subclass with good decision properties. The empty-stack constraint requires that a pushdown process may only receive a message when its stack is empty. Unfortunately, the empty-stack constraint is unrealistic in the context of Erlang. Erlang provides parametric implementations of common concurrent programming patterns through functional interfaces as *Erlang behaviours*. For example, a generic server template can be instantiated with arbitrary code that handles client requests and which may use communication and other Erlang behaviours within potentially multiple layers of programming abstractions. This promotes the construction of Erlang programs that do *not* satisfy the empty stack constraint.

In order to improve upon the empty-stack constraint, we introduce a novel class of ACPS— *shaped ACPS*— and study its decision and complexity properties for key verification problems. In the setting of unordered message passing, we may view *non-blocking* concurrency side-effects, such as message sends and spawns, as *commutative* because the order in which such concurrency actions are performed cannot be observed by other processes. This is in contrast to blocking concurrency actions such as message receive which we thus call *non-commutative*. If we view ACPS as concurrent recursive programs we informally say a procedure is *commutative* if its full execution may only give rise to commutative concurrency side-effects. Intuitively, a procedure is *non-commutative* just if a blocking operation, such as a receive, may be invoked when running it. Shaped ACPS satisfy a semantic constraint called the *shaped stack constraint*. The shaped stack constraint requires that at all times, no more than an *a priori* fixed number of non-commutative procedure calls may reside in the stack of any ACPS process. Note that the number of commutative procedure calls is *not* restricted and hence shaped constrained stacks can grow to arbitrary heights. Further, a process of a shaped ACPS may perform any communication action irrespective of whether its stack is empty or not. Shaped ACPS strictly subsume ACPS that satisfy the empty-stack constraint. We find that coverability for shaped ACPS reduces to the decidable coverability problem for *well-structured transition systems* (WSTS) (Chapter 4). WSTS [FS01] are a widely-applicable class of infinite-state systems that enjoy good model-checking properties. Unfortunately, it is difficult to obtain tight complexity results from WSTS algorithms. How-

ever, recent work [GMR09; GM12] has successfully established complexity results for ACPS that satisfy the empty-stack constraint by exhibiting a connection to Petri nets [Pet62].

Nets with Nested Coloured Tokens. Encouraged by this advance, we look for a similar connection for shaped ACPS. For this purpose, we introduce a non-trivial extension of Petri nets, *nets with nested coloured tokens* (NNCT), in Chapter 5. Besides ordinary Petri net or *simple* tokens, NNCT feature *complex* tokens. Complex tokens carry a multiset of *coloured* tokens which may be injected into or ejected from a complex token by transitions of the net. Thus, NNCT belong to the class of nets with an infinite set of token *types*. Intuitively, the complex tokens of an NNCT can be used to model ACPS processes while simple tokens can represent the contents of channels. We exploit these similarities to establish reductions between NNCT and shaped ACPS that allow us to transfer complexity results obtained on NNCT to shaped ACPS. Thus, we can focus on establishing complexity results in the simpler setting of NNCT. For this purpose, we turn to fundamental results in the Petri net literature: the *Rackoff technique* [Rac78] and the *Lipton construction* [Lip76]. The Rackoff method was originally used to establish an Expspace upper bound for Petri net coverability, while the Lipton construction provided an Expspace lower bound. We show that NNCT coverability is Tower-complete in the sense of Schmitz [Sch13]. Tower is a class of non-elementary decision problems that is closed under Elementary reductions. To obtain a Tower upper bound for NNCT coverability, we devise a geometrically inspired version of the Rackoff method. Further, we use a Stockmeyer *ruler construction* [Sto74], which we obtain by augmenting Lipton's construction, to establish Tower-hardness of NNCT coverability. NNCT are thus the first Petri net extension in the class of nets with an infinite set of token *types* that are proven to have a primitive recursive coverability problem. These results imply that e.g. coverability for shaped ACPS is Tower-complete.

Implications. Returning to our question of whether there exists an expressive decidable subclass of ACPS that extends ACS, we note that shaped ACPS do subsume ACS, are capable of modelling a wide range of recursive function definitions, and they support the algorithmic verification of safety properties via coverability. Even though the worst-case complexity of the latter is high, we do *not* believe that the algorithmic verification of shaped ACPS is doomed to failure. Recently there have been remarkable advances in the design of *practical* algorithms for problems of comparable or higher worst-case complexity such as *higher-order model-checking* [RNO14] (Tower-complete) and *safety verification of concurrent C programs with broadcast* [KKW12] (non-primitive recursive). The verification algorithms are practical in the sense that when run on realistic (not degenerate) input programs written by humans, they terminate in a few tens of seconds as opposed to months or years. These advances encourage us in the belief that it is possible to develop practical verification algorithms for shaped ACPS/NNCT despite their Tower-completeness. We further believe that such algorithms would provide the basis for a verification procedure for Erlang programs that can leverage the additional expressiveness with respect to recursion.

Before we survey our contributions in more detail in Section 1.3, let us give a high-level introduction to λ_{Actor} and explain its operational semantics on a simple example.

1.2 A Prototypical Fragment of Erlang: λActor

λActor is an idealised untyped functional language with actor concurrency. Its sequential fragment consists of Erlang's full higher-order functional features with algebraic data structures and pattern-matching. Functional reductions follow an eager evaluation strategy. It is augmented with concurrency primitives for dynamic process creation and asynchronous message-passing over FIFFO mailboxes. It thus captures the key features of single-node *Core Erlang* [Car01]. However, λActor does not contain Core Erlang's fault-tolerance features or built-in functions.

Syntax. The syntax of λActor is defined by the grammar below. We call a term that can be generated by this grammar a λActor expression.

$$
\begin{aligned}
e \in \mathit{Exp} ::= \;& x \mid \mathsf{c}(e_1, \ldots, e_n) \mid e_0(e_1, \ldots, e_n) \mid \mathit{fun} \\
\mid\; & \textbf{letrec}\ x_1\text{=}\mathit{fun}_1 . \cdots x_n\text{=}\mathit{fun}_n .\ \textbf{in}\ e \\
\mid\; & \textbf{case}\ e\ \textbf{of}\ \mathit{pat}_1 \to e_1 ; \ldots ; \mathit{pat}_n \to e_n\ \textbf{end} \\
\mid\; & \textbf{receive}\ \mathit{pat}_1 \to e_1 ; \ldots ; \mathit{pat}_n \to e_n\ \textbf{end} \\
\mid\; & \textbf{send}(e_1, e_2) \mid \textbf{spawn}(e) \mid \textbf{self}\ () \\
\mathit{fun} ::= \;& \textbf{fun}(x_1, \ldots, x_n) \to e \\
\mathit{pat} ::= \;& x \mid \mathsf{c}(\mathit{pat}_1, \ldots, \mathit{pat}_n)
\end{aligned}
$$

where we let the metavariable c range over a fixed, finite set of constructors. In definitions, formal statements, or proofs we let the metavariables x, y, z, and f range over variables. We adopt the usual definition of the set of free variables $\mathit{FV}(e)$ of an λActor expression e. Note that we omit parentheses for nullary constructors.

Remark. When presenting λActor code in concrete examples, we frequently use Erlang syntax which is similar and often more convenient. Let us briefly explain a few cosmetic differences. A fresh, unnamed variable for discarded values is denoted by '_', and variable names in Erlang start with an upper case letter, except in a **letrec**. The *let* construct $(\mathsf{X} = e_1,\ e_2)$ is a shorthand for $(\textbf{fun}(\mathsf{X}) \to e_2)(e_1)$ and the *sequencing* expression (e_1, e_2) is syntactic sugar for $(\_ = e_1,\ e_2)$. The *cons* operator for lists is denoted by [_|_] and the empty list is written as [], which can be translated to λActor as cons(_,_) and nil respectively. Further, a n-ary tuple is denoted by {e_1, ..., e_n}, which is interpreted as the λActor expression $\mathsf{tup}_n(e_1, \ldots, e_n)$, and the character '%' signifies the start of a comment in Erlang. Also note that functions are uncurried in both Erlang and λActor.

λActor Programs. We say a λActor expression $\mathcal{P}$ is a program if $\mathcal{P}$ is a closed λActor expression i.e. $\mathit{FV}(\mathcal{P}) = \emptyset$. Given a λActor program $\mathcal{P}$, we annotate each subexpression e of $\mathcal{P}$ with a unique label ℓ. These labels may be viewed as the *program locations* of $\mathcal{P}$. We write $\ell : e$ to mean e is the subexpression of $\mathcal{P}$ at program location/label ℓ. Labels are useful to distinguish subexpressions of $\mathcal{P}$ that may be identical as expressions, but appear at different program locations in $\mathcal{P}$. To unclutter notation, we often omit the label ℓ when there is no confusion. We further introduce two operations on labels: suppose $\ell\colon (\ell_0\colon e_0)(\ell_1\colon e_1, \ldots, \ell_n\colon e_n)$ is a subexpression of $\mathcal{P}$, then

we define $\ell.\text{arg}_i := \ell_i$ and $\text{arity}(\ell) := n$. Lastly, let us remark on a common setting in Erlang. If a program $\mathcal{P}$ consists only of a top level **letrec** with entry point main(), i.e.

$$\mathcal{P} = \textbf{letrec } f_1\text{=}\textbf{fun}(\boldsymbol{x}_1) \to e_1.\cdots f_n\text{=}\textbf{fun}(\boldsymbol{x}_n) \to e_n.\ \textbf{in } \mathsf{main}(),$$

then $\mathcal{P}$ is simply written as $f_1(\boldsymbol{x}_1) \to e_1.\cdots f_n(\boldsymbol{x}_n) \to e_n.$ in Erlang (see Example 1).

A Rewriting Semantics. In Chapter 3, we define an abstract-machine-based operational semantics for λACTOR which serves as its formal reference semantics. However, in order to illustrate λACTOR and its model of concurrency, we present a natural rewriting operational semantics here. A state of the computation of a λACTOR program is a set Π of processes $\pi_1, \ldots, \pi_n$ running in parallel which we denote by $\Pi = \pi_1 \parallel \cdots \parallel \pi_n$. In this rewriting operational semantics, we use the notation $\langle e \rangle^{\iota}_{\mathfrak{m}}$ to denote a process π that (i) is identified by a process identifier (pid) ι, (ii) evaluates an expression e that may contain process identifiers, and (iii) is equipped with a mailbox $\mathfrak{m}$ which is a sequence of messages (closed λACTOR expressions). The set of eager, left-to-right evaluation contexts is defined by the following grammar

$$\begin{aligned} E[-] ::= \quad & [-] \mid E[[-](e_1, \ldots, e_n)] \\ \mid \quad & E[v(v_1, \ldots, v_{i-1}, [-], e_{i+1}, \ldots, e_n)] \\ \mid \quad & \textbf{case } [-] \textbf{ of } \mathit{pat}_1 \to e_1; \ldots; \mathit{pat}_n \to e_n \textbf{ end} \end{aligned}$$

where we let v, v_1,... range over irreducible expressions such as function abstractions, constructors, pids, and un-applied primitives.

Purely Functional Reductions. We use the standard rewrite rules for purely functional reductions such as function application, pattern-matching, and **letrec**. Such functional reductions are freely interleaved with the reductions of other processes. The following rule evaluates a function application

$$\langle E[(\textbf{fun}(x_1, \ldots, x_n) \to e)(v_1, \ldots, v_n)] \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \to \langle E[\theta(e)] \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi$$

where the substitution θ maps each variable x_i to the value v_i. We write $\theta(e)$ for the expression obtained by substituting each variable x_i in e by the expression $\theta(x_i)$. A **case** construct pattern-matches a value v against patterns $p_1, \ldots, p_n$. If there is a match, i.e. $v = \theta(p_i)$, then control is given to e_i:

$$\langle E[\textbf{case } v \textbf{ of } p_1 \to e_1; \ldots; p_n \to e_n \textbf{ end}] \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \to \langle E[\theta(e_i)] \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi$$

If there are multiple matching patterns the textually first is chosen. A **letrec** binds a number of function abstractions to variables recursively:

$$\langle E[\textbf{letrec } x_1\text{=}f_1.\cdots x_n\text{=}f_n.\ \textbf{in } e] \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \to \langle E[\theta(e)] \rangle^{\iota}_{\mathfrak{m}} \parallel \Pi$$

where the fixpoint substitution θ maps each x_i to the expression **letrec** $x_1\text{=}f_1.\cdots x_n\text{=}f_n.$**in** f_i.

Reductions with Concurrency Side-Effects. Let us now present the rules for the primitives with side-effects. A **spawn** expression, **spawn**(**fun**()→e), creates a new process with a fresh process identifier ι' that evaluates the expression e and has an empty mailbox. The process identifier ι' is returned to the spawning process as the result of the evaluation:

$$\langle E[\textbf{spawn}(\textbf{fun}()\to e)]\rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \to \langle E[\iota']\rangle^{\iota}_{\mathfrak{m}} \parallel \langle e\rangle^{\iota'}_{\epsilon} \parallel \Pi$$

A **send** expression, **send**(ι, v), sends the message v to the mailbox of the process with pid ι. This is achieved by appending v to the end of ι's mailbox. Note that the evaluation of expression **send**(ι, v) is non-blocking and returns v:

$$\langle E[\textbf{send}(\iota, v)]\rangle^{\iota'}_{\mathfrak{m}'} \parallel \langle e\rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \to \langle E[v]\rangle^{\iota'}_{\mathfrak{m}'} \parallel \langle e\rangle^{\iota}_{\mathfrak{m}\cdot v} \parallel \Pi$$

In contrast to other concurrency primitives, a **receive** construct can cause the evaluation to block. If a process with mailbox $\mathfrak{m} = m_1 \cdots m_k$ evaluates the expression **receive** $p_1 \to e_1 \cdots p_n \to e_n$ **end**, then each message $m_1, \ldots, m_k$ is considered one-by-one. If m_i is the first message in $\mathfrak{m}$ to match one pattern, then it is removed from the mailbox and the textually first matching **receive** clause, e.g. $p_j \to e_j$, fires. This yields the mailbox $\mathfrak{m}' = m_1 \cdots m_{i-1} \cdot m_{i+1} \cdots m_k$. If there is no message in $\mathfrak{m}$ that matches a pattern then the evaluation blocks until a matching message arrives in the mailbox.

$$\langle E[\textbf{receive}\ p_1 \to e_1 \cdots p_n \to e_n\ \textbf{end}]\rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \to \langle E[\theta(e_j)]\rangle^{\iota'}_{\mathfrak{m}'} \parallel \Pi \text{ where } m = \theta(p_j)$$

This means that messages may be skipped if they do not match a pattern. Hence, message passing is *not* First-In-First-Out but rather First-In-First-Fireable-Out (FIFFO).

Lastly, the **self** primitive gives a process access to its own pid. If a process with pid ι evaluates the expression **self**() it obtains ι:

$$\langle E[\textbf{self}()]\rangle^{\iota}_{\mathfrak{m}} \parallel \Pi \to \langle E[\iota]\rangle^{\iota}_{\mathfrak{m}} \parallel \Pi$$

The **self** primitive allows a process to make decision based on its identity. Hence, two processes may act differently even if they evaluate the same λ_{Actor} expression e. Let us now explain the rewriting semantics on a simple example.

Example 1. The λ_{Actor} program we present below is inspired by an example by Fredlund and Svensson [FS07]. It runs three processes A, B, and C such that A sends two messages to C, one relayed through B and one directly. We modify the example slightly to illustrate FIFFO message passing and let A send a message to B which is *noise* and which B ignores. Consider the following three expressions:

$$\begin{aligned} e_A &= \textbf{fun}(\mathsf{X}, \mathsf{Y}) \to \textbf{send}(\mathsf{X}, \mathtt{c}), \textbf{send}(\mathsf{X}, \{\mathsf{Y}, \mathtt{b}\}), \textbf{send}(\mathsf{Y}, \mathtt{a}) \\ e_B &= \textbf{receive}\ \{\mathsf{P}, \mathsf{M}\} \to \textbf{send}(\mathsf{P}, \mathsf{M})\ \textbf{end} \\ e_C &= \textbf{receive}\ \mathtt{a} \to 0; \mathtt{b} \to 1\ \textbf{end}. \end{aligned}$$

We intend to let processes A, B, and C run expressions e_A, e_B, and e_C respectively. Note that e_A is a function: it takes two arguments ι_X and ι_Y, which are assumed to be the pids of B and

C. First, it sends a *noise* message c and a tuple, comprised of ι_Y and the message b, to the pid ι_X. Afterwards, it sends the message a to ι_Y. The expression e_B is a simple receive that waits for a message which is a pair comprised of a pid ι_P and a message m. On reception of such a message, it sends the message m to the prescribed recipient ι_P. Process C evaluates expression e_C and simply waits for either message a or b to arrive first. Let us now define λ_{ACTOR} program $\mathcal{P}$ that sets up our three processes A, B, and C:

$$\mathsf{main}() \to \mathsf{A} = e_A, \mathsf{B} = \textbf{spawn}(\textbf{fun}() \to e_B), \mathsf{C} = \textbf{spawn}(\textbf{fun}() \to e_C), A(B, C).$$

It first binds the function e_A to variable A, then spawns two processes that evaluate e_B and e_C, and binds its pids to the variables B and C. As a last step, A is invoked with B and C as arguments. Let us present a run induced by the rewriting semantics of λ_{ACTOR}:

$$\begin{aligned}
&\langle \textbf{letrec } \mathsf{main} = \textbf{fun}() \to \mathsf{A} = e_A, \mathsf{B} = \textbf{spawn}(\textbf{fun}() \to e_B) \dots \textbf{ in } \mathsf{main}() \rangle_{\epsilon}^{\iota_A} \\
&\quad \to \langle (\textbf{fun}() \to \mathsf{A} = e_A, \mathsf{B} = \textbf{spawn}(\textbf{fun}() \to e_B), \dots)() \rangle_{\epsilon}^{\iota_A} \\
&\quad \to \langle \mathsf{A} = e_A, \mathsf{B} = \textbf{spawn}(\textbf{fun}() \to e_B), \mathsf{C} = \textbf{spawn}(\textbf{fun}() \to e_C), A(B, C) \rangle_{\epsilon}^{\iota_A} \\
&\quad \to \langle \mathsf{B} = \textbf{spawn}(\textbf{fun}() \to e_B), \mathsf{C} = \textbf{spawn}(\textbf{fun}() \to e_C), e_A(B, C) \rangle_{\epsilon}^{\iota_A} \\
&\quad \to \langle \mathsf{B} = \iota_B, \mathsf{C} = \textbf{spawn}(\textbf{fun}() \to e_C), e_A(B, C) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\epsilon}^{\iota_B} \\
&\quad \to \langle \mathsf{C} = \textbf{spawn}(\textbf{fun}() \to e_C), e_A(\iota_B, C) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\epsilon}^{\iota_B} \\
&\quad \to^* \langle e_A(\iota_B, \iota_C) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\epsilon}^{\iota_B} \parallel \langle e_C \rangle_{\epsilon}^{\iota_C}
\end{aligned}$$

At this point, we have reached a configuration where we have set up the three processes with pids ι_A, ι_B, and ι_C which are executing e_A, e_B, and e_C as desired. We now continue with the evaluation of e_A:

$$\begin{aligned}
&\langle e_A(\iota_B, \iota_C) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\epsilon}^{\iota_B} \parallel \langle e_C \rangle_{\epsilon}^{\iota_C} \\
&\quad \to \langle \textbf{send}(\iota_B, \mathsf{c}), \textbf{send}(\iota_B, \{\iota_C, \mathsf{b}\}), \textbf{send}(\iota_C, \mathsf{a}) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\epsilon}^{\iota_B} \parallel \langle e_C \rangle_{\epsilon}^{\iota_C} \\
&\quad \to \langle \mathsf{c}, \textbf{send}(\iota_B, \{\iota_C, \mathsf{b}\}), \textbf{send}(\iota_C, \mathsf{a}) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\mathsf{c}}^{\iota_B} \parallel \langle e_C \rangle_{\epsilon}^{\iota_C} \\
&\quad \to \langle \textbf{send}(\iota_B, \{\iota_C, \mathsf{b}\}), \textbf{send}(\iota_C, \mathsf{a}) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\mathsf{c}}^{\iota_B} \parallel \langle e_C \rangle_{\epsilon}^{\iota_C} \\
&\quad \to^* \langle \textbf{send}(\iota_C, \mathsf{a}) \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\mathsf{c} \cdot \{\iota_C, \mathsf{b}\}}^{\iota_B} \parallel \langle e_C \rangle_{\epsilon}^{\iota_C} \\
&\quad \to \langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle e_B \rangle_{\mathsf{c} \cdot \{\iota_C, \mathsf{b}\}}^{\iota_B} \parallel \langle e_C \rangle_{\mathsf{a}}^{\iota_C}
\end{aligned}$$

In this configuration, the receive in expression e_B is enabled and we can extract message $\{\iota_C, \mathsf{b}\}$ from B's mailbox skipping message c. Thus, message b can be relayed to process C.

$$\begin{aligned}
&\langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \textbf{receive } \{\mathsf{P}, \mathsf{M}\} \to \textbf{send}(\mathsf{P}, \mathsf{M}) \textbf{ end} \rangle_{\mathsf{c} \cdot \{\iota_C, \mathsf{b}\}}^{\iota_B} \parallel \langle e_C \rangle_{\mathsf{a}}^{\iota_C} \\
&\quad \to \langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \textbf{send}(\iota_C, \mathsf{b}) \rangle_{\mathsf{c}}^{\iota_B} \parallel \langle e_C \rangle_{\mathsf{a}}^{\iota_C} \\
&\quad \to \langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \mathsf{b} \rangle_{\mathsf{c}}^{\iota_B} \parallel \langle \textbf{receive } \mathsf{a} \to 0; \mathsf{b} \to 1 \textbf{ end} \rangle_{\mathsf{a} \cdot \mathsf{b}}^{\iota_C} \\
&\quad \to \langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \mathsf{b} \rangle_{\mathsf{c}}^{\iota_B} \parallel \langle 0 \rangle_{\mathsf{b}}^{\iota_C}
\end{aligned}$$

Since message a appears before b in C's mailbox, the receive in e_C fires the first clause. Note that this receive could have fired as soon as a reached process C's mailbox. Further, a different

order in C's mailbox is possible, if we change the scheduling of processes. Suppose for example, that before process A sends a to C, process B is scheduled and sends message b to C first. This leads to the configuration $\langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \mathsf{b} \rangle_{\mathsf{c}}^{\iota_B} \parallel \langle e_C \rangle_{\mathsf{b}\cdot\mathsf{a}}^{\iota_C}$ which means that we ultimately reach $\langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \mathsf{b} \rangle_{\mathsf{c}}^{\iota_B} \parallel \langle 1 \rangle_{\mathsf{a}}^{\iota_C}$.

Even in this simple example, we use several features of Erlang/λ_{ACTOR} in a typical manner. We create a first-class function, use asynchronous message passing, dynamically create two processes, and pass their identities as first-class values both within one process and over messages. Further, process A sends an algebraic data structure, i.e. a tuple, as a message to process B which is received by pattern-matching.

Verification for λ_{ACTOR}. As we mentioned, message c is noise and should not reach process C's mailbox. Thus, we can pose the question whether it is possible to reach the configuration $\langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \mathsf{c} \rangle_{\{\iota_C, \mathsf{b}\}}^{\iota_B} \parallel \langle e_C \rangle_{\mathsf{a}\cdot\mathsf{c}}^{\iota_C}$.

Safety Verification. Such a question can be posed as a *reachability problem*: given an initial configuration, e.g. $\langle \mathcal{P} \rangle_{\epsilon}^{\iota_A}$, does there exist a run that *reaches* a given target configuration, e.g. $\langle \mathsf{a} \rangle_{\epsilon}^{\iota_A} \parallel \langle \mathsf{c} \rangle_{\{\iota_C, \mathsf{b}\}}^{\iota_B} \parallel \langle e_C \rangle_{\mathsf{a}\cdot\mathsf{c}}^{\iota_C}$. Reachability is the prototypical safety verification problem. A safety-property states that *something bad may never happen* [Lam77]. Thus, to verify whether a safety property is ever violated, we need to check whether a *bad state* is reachable. We observe that process B can under no circumstances extract message c from its mailbox since it does not match the pattern $\{\_, \_\}$. Hence, process B can never get into a state of the form $\langle \mathbf{send}(\iota_C, \mathsf{c}) \rangle_{\mathfrak{m}'}^{\iota_B}$, and therefore message c can never reach the mailbox of process C.

A weaker variant of reachability is *coverability* which requires a *covering preorder* $\leq$ on configurations. Given an initial configuration s_0 and a target configuration s_{cov}, a coverability problem asks whether it is possible to reach a configuration s from s_0 such that s *covers* s_{cov}, i.e. $s \geq s_{\text{cov}}$. For example, both the question whether process B can ever reach the state $\langle \mathbf{send}(\iota_C, \mathsf{c}) \rangle_{\mathfrak{m}'}^{\iota_B}$ and whether the message c can ever reach the mailbox of process C can be posed as a coverability problem for a suitable preorder. In general, safety verification problems such as program location reachability, mutual exclusion, and explicit bound-checks can be phrased as coverability problems.

Liveness Verification. Another interesting property of our example is that there is no infinite run, i.e. we eventually reach a final configuration which has no successor. Thus, $\mathcal{P}$ is *terminating*. Deciding whether there exists an infinite run from a given configuration is called the *termination problem*. Termination is an example of a liveness verification problem. A liveness property states that *eventually something good must happen* [Lam77]. It is known that verifying liveness properties reduces to checking for *fair termination* [Var91], i.e. is it the case that, provided non-deterministic choices are selected *fairly*, every run terminates.

Verifying Boundedness. In contrast to sequential programs, many concurrent systems, such as web servers or GUI applications, are non-terminating. It is often more interesting to ask whether a concurrent program will only require *bounded resources* (bounded number of processes, mailboxes, call-stacks, etc.). Such a question can be phrased as a boundedness problem. Given an initial configuration s_0, the *boundedness* problem asks whether the set of configurations reachable from s_0 is finite. Note that the λ_{ACTOR} program $\mathcal{P}$ gives rise to a bounded system. In the

context of dynamic process creation and asynchronous message passing, it is often interesting to specialise a boundedness question. For example, we may want to establish whether the capacity of a particular mailbox is bounded, or whether there exists a bound on the number of processes that are executing simultaneously. Such decision questions are sometimes referred to as *selective (un)boundedness* [Dem10; KM69].

In this dissertation, we focus on the easier but important problems: boundedness, termination, and in particular coverability. Unfortunately, λ_{Actor} is a Turing powerful language and thus even these easier problems are undecidable.

Turing Powerful Fragments of λ_{Actor}. Surprisingly λ_{Actor} subsumes several distinct Turing powerful formalisms: Already the sequential fragment of λ_{Actor} corresponds to higher-order functional programs with algebraic data structures and pattern-matching [OR11]. If we restrict ourselves to first-order λ_{Actor} programs, we can easily encode concurrent pushdown systems [Ram00]. Using tail-recursive first-order functions and dynamic process creation, it is also possible to encode the π-calculus [MPW92]. Further, FIFO-mailboxes allow us to simulate *communicating finite-state machines* [BZ83] and also *reactive FIFO systems* [Her+02] which are Turing powerful. A reactive FIFO system is essentially a finite-state machine with a single FIFO mailbox. The results of Herbreteau et al. [Her+02] imply that a single λ_{Actor} process running a finite-state program can simulate a two-counter Minsky-machine [Min61] with its unbounded FIFO mailbox. There are two further, known fragments of λ_{Actor} that are expressive enough to encode Minsky-machines [DKO11]. The first encoding is a (i) first-order λ_{Actor} program and requires (ii) a fixed, finite number of processes with (iii) mailboxes that may not contain more than one message, and uses only (iv) nullary constructors. Thus, the set of constructor terms and set of messages is finite. The second encoding of Minsky-machines into λ_{Actor} also requires (ii) and (iii), but uses (iii') constructors of arity at most two, and (iv') is a λ_{Actor} program of order-0, i.e functions do not take arguments. These undecidability results are overwhelming. They emphasise the fact that to perform verification on λ_{Actor}, we have to consider abstractions of both data and control-flow. In particular, identifying decidable computational models that can be the target of an abstract interpretation of λ_{Actor} is of interest to us.

Now that we have motivated λ_{Actor}, our interest in central verification problems, and explained their difficulty, let us give a detailed overview of our contributions together with an outline of the material that we present in this dissertation.

1.3 Contributions and Document Structure

In Chapter 2, we review essential background on transition systems and give a brief account of the complexity classes in the hierarchy of fast-growing complexity classes [Sch13]. Further, we review fundamental models for concurrency (Section 2.3), such as Petri nets [Pet62] and well-structured transition systems [Fin87], and we introduce pushdown systems and context-free grammars as models for recursive programs (Section 2.4).

We reserve Chapters 3, 4, and 5 for our contributions. In Chapter 3, we present a conservative verification procedure for λ_{Actor} and its implementation. We study asynchronously communicating pushdown systems (ACPS), the shaped stack constraint, and decision properties of shaped

ACPS in Chapter 4. In Chapter 5, we investigate the complexity of key verification problems for shaped ACPS via a novel extension of Petri nets: nested nets with coloured tokens (NNCT). We survey the contents of Chapters 3, 4, and 5 in more detail in the following paragraphs.

Lastly, we conclude with a summary of our contributions and perspectives on future work in Chapter 6.

Chapter 3. We identify *actor communicating systems* (ACS) as a computational model with good model-checking properties that is capable of modelling key features of λ_{ACTOR} (Section 3.1). Thus, we design an abstraction interpretation of λ_{ACTOR} using the methodology of abstracting abstract machines (AAM) [HM11; HM12], which we review in Section 3.2.

- Following the AAM methodology, we give a machine-based concrete operational semantics for λ_{ACTOR} that is designed to have a recursion-free state space which in turn facilitates a structural abstraction (Section 3.3). We then present a parametric abstract interpretation of this operational semantics and show how we can obtain a finite abstract transition system that simulates the concrete operational semantics (Section 3.4). This constitutes the first application of the AAM methodology in the context of message passing concurrency. Further, we show how we can generate an ACS while exploring the abstract state space. This ACS is guaranteed to simulate the concrete operational semantics of the input program (Section 3.5). We can use this abstract model to conservatively verify coverability properties of the input program.
- Thus, our conservative verification procedure is a two step process: (i) extract a sound ACS from the input program while exploring the state space of its abstract operational semantics, and (ii) verify coverability properties on the generated ACS using an existing cutting-edge Petri net model-checker. We have implemented this verification procedure in our tool Soter. We give an overview of the implementation in Section 3.6 and present a worklist algorithm for the exploration of the abstract state space. Soter is the first sound verification tool for Erlang programs that uses infinite-state model-checking. We illustrate the use of Soter on a worked example, which is a concurrent implementation of Eratosthenes sieve (Section 3.6), and evaluate Soter on a set of non-trivial benchmarks (Section 3.8). The results of our experiments are encouraging and indicate that infinite-state model-checking can be successfully applied to the verification of Erlang. We find that our worklist exploration algorithm yields a significant speed-up and that the abstractions employed by Soter are accurate enough to verify a range of interesting and non-trivial Erlang programs.

Chapter 4. In order to improve the expressivity of ACS with respect to recursion, we investigate an extension of ACS that allows processes to be *pushdown*. In the general case, such an extension yields Turing-powerful *asynchronously communicating pushdown systems* (ACPS). In the literature, ACPS that satisfy the empty-stack restriction [SV06] have been studied. In the context of Erlang, the empty stack constraint is unfortunately not realistic.

- We study ACPS in Section 4.1 and show that for the purposes of deciding coverability, boundedness, and termination we can consider ACPS in *normal form* (Proposition 1), which

is reminiscent of a context-free grammar (CFG) in *Chomsky normal form*. We further show that we can consider an equivalent, simplified version of coverability, *simple coverability*, which restricts the form of coverability queries (Lemma 1).

- In Section 4.2, we observe that in the setting of unordered message passing the relative order of some concurrency actions, such as message send or process creation, is immaterial. We thus partition concurrency actions into *commutative* and *non-commutative* ones. We formalise this sensitivity to order as an independence relation and lift the classification of non-/commutativity to stack characters. We build upon known results on how CFGs interact with an independence relation and introduce a hybrid model, *asynchronous partially commutative pushdown systems* (APCPS). Processes are modelled by a variant of context-free grammars which distinguish commutative and non-commutative concurrency actions. We then show that for ACPS in normal form simple coverability polynomial-time interreduces with APCPS (Proposition 2). Hence, we focus on developing technical results for APCPS.

- In Section 4.3, we define an alternative semantics for APCPS which exploits commutativity to simplify the standard semantics of APCPS. We show that for the purposes of deciding simple coverability the standard and alternative semantics are equivalent (Theorem 7).

- The alternative semantics of APCPS suggests a natural restriction on processes: the shaped stack constraint (Definition 21). The shaped stack constraint limits the number of non-commutative stack characters that may reside in the stack of any process. However, the number of commutative stack characters is unconstrained. The alternative semantics for APCPS that satisfy the shaped stack constraint gives rise to a strict, post-effective WSTS that is well-partially-ordered and hence coverability and boundedness become decidable (Theorem 8). In addition, we find an easy-to-check syntactic, sufficient condition for the shaped stack constraint (Proposition 4). We further show that the shaped stack constraint can be recovered on ACPS in normal form (Proposition 2). Hence, we can transfer decision and complexity results obtained on APCPS to ACPS. Note that shaped ACPS, i.e. ACPS that satisfy the shaped stack constraint, subsume ACPS that satisfy the empty stack constraint, thus advancing the state of the art.

Chapter 5. In Chapter 5 we focus on establishing complexity results for shaped ACPS. Encouraged by the connection between ACPS that satisfy the empty-stack constraint and Petri nets, we introduce a novel extension of Petri nets, *nets with nested coloured tokens* (NNCTs) (Section 5.1), that is expressive enough to model the alternative semantics of APCPS. Further, we identify an expressive subclass of NNCT: *total transfer* NNCT.

- We show that NNCT give rise to a strict WSTS and thus NNCT enjoy good model-checking properties. Coverability, boundedness, and termination are decidable (Theorem 9).

- Further, we show that simple coverability for shaped APCPS on the alternative semantics EXPTIME-reduces to coverability on NNCT (Theorem 10). Moreover, we give a proof that

simple coverability, boundedness, and termination for total transfer NNCT EXPTIME-reduces to their respective counterparts for shaped APCPS on the alternative semantics (Theorem 11).

- We then move on to establish TOWER membership of NNCT coverability (Theorem 12). Our proof uses a geometrically inspired extension of the Rackoff argument, which was originally used to establish EXPSPACE-membership of coverability for Petri nets [Rac78].

- We obtain a non-elementary lower-bound, i.e. TOWER-hardness, for simple coverability, boundedness, and termination by adapting Stockmeyer's ruler construction [Sto74] to NNCT (Theorem 13). We establish these reductions from the parametrised halting problem of $2 \uparrow\uparrow n$-bounded two counter machines, which is the canonical TOWER-hard decision problem. To this end, we exhibit a polynomial-time computable total-transfer NNCT that can simulate a given $2 \uparrow\uparrow n$-bounded two counter machine. We arrive at this total-transfer NNCT by augmenting Lipton's construction [Lip76], which established EXPSPACE lower bounds for coverability, boundedness, and termination for Petri nets.

- Hence, coverability for NNCT and shaped ACPS/APCPS is TOWER-complete (Theorems 14 and 15). To our knowledge, NNCT is the first extension of Petri nets (belonging to the class of nets with an infinite set of token *types*) that is proven to have primitive recursive coverability.

A Note to The Reader

In order to improve readability, we relegate detailed technical arguments and formal proofs to the appendix, while we give a high level explanation in the main text that we mark with *Proof Idea*. For the proofs of more intricate results such as Theorem 7, Theorem 10, or Theorem 12 we dedicate a section to an explanation of the proof. For some conjectures we provide a high level argument that we believe can be turned into a formal proof and mark it by *Proof Sketch*. Further, we would like to mention that we review related work at the end of each chapter instead of in a dedicated chapter. Lastly, in order to help the reader, we sometimes repeat definitions that have been introduced elsewhere in the text.

Notation.

Sets and Multisets. Let us write $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$ and $\langle n \rangle = \{1, \dots, n\}$. Let U be a set and $V \subseteq U$, we denote V's *set complement* by V^{c}, i.e. $V^{\mathsf{c}} = U \setminus V$. We write $\mathscr{P}[V]$ for the *powerset* of V, i.e. the set of all subsets of V. Further, we write $\mathbb{M}[U]$ for the set of multisets over the set U. We use $[\cdot]$ to denote multisets explicitly; e.g. we write $[u, u, v^2]$ to mean the multiset containing two occurrences each of u and v. Moreover, we write $\emptyset$ for both the empty set and the empty multiset. We say that u is an element of the multiset $M \in \mathbb{M}[U]$, written $u \in M$, if $M(u) \geq 1$. Given multisets M_1 and M_2, we write $M_1 \oplus M_2$ for the multiset union of M_1 and M_2. If $M = M_1 \oplus M_2$ then we define $M_1 = M \ominus M_2$. We order multisets (in $\mathbb{M}[U]$) standardly: $M_1 \leq_{\mathbb{M}[U]} M_2$ just if $M_1 = [u_1, \dots, u_n]$, $M_2 = [u'_1, \dots, u'_m]$ and there is an injective map $h : \langle n \rangle \to \langle m \rangle$ such that for all $1 \leq i \leq n$ we have $u_i \leq_U u'_{h(i)}$. Let $M \in \mathbb{M}[U]$ and $U_0 \subseteq U$.

We define $M \restriction U_0$ to be the multiset M restricted to U_0 i.e. $(M \restriction U_0) : u \mapsto M(u)$ if $u \in U_0$, and 0 otherwise.

Sequences. We write $U^{*\omega}$ for the set of (finite or infinite) sequences and U^* for the set of finite sequences over U. If $u, v, \ldots$ range of U, we write $\boldsymbol{u}$, $\boldsymbol{v}$ to range over U^*. We also let the metavariables $\beta, \gamma, \mu, \nu, \ldots$ range over U^* and we denote the *length* of sequence β by $|\beta|$. We denote the empty sequence by ϵ and we write '$\cdot$' for the concatenation operator. Sequences in U^* are maps from $\mathbb{N} \to U$. If $(U, \leq_U)$ is a preordered set then we extend $\leq_U$ to a preorder $\leq_{\text{Hig}}$ on U^* such that $\beta \leq_{\text{Hig}} \beta'$ if there is a strictly monotone function $h : \langle|\beta|\rangle \to \langle|\beta'|\rangle$ and for all $i \in \langle|\beta|\rangle$ we have $\beta(i) \leq_U \beta'(h(i))$.

Disjoint Unions. We write $U_1 + U_2$ for the disjoint union of sets U_1 and U_2. Suppose we have maps $f_1 : U_1 \to V_1$ and $f_2 : U_2 \to V_2$, we write $f_1 + f_2 : U_1 + U_2 \to V_1 + V_2$ where $+$ is the coproduct bifunctor, i.e., $(f_1 + f_2)(\mathit{in}_i(u)) := \mathit{in}_i(f_i(u))$, for $i \in \{1, 2\}$, where in_i is the canonical injection of U_i into $U_1 + U_2$. Henceforth, to save writing, we will elide in_i and write, for example, $(f_1 + f_2)(u) = f_i(u)$ if $u \in U_i$.

Functions. We write $f\,[u_1 \mapsto u_1', \ldots, u_n \mapsto u_n']$ for the function f' such that $f'(u) = f(u)$ if $u \neq u_i$ for all $i \in \langle n\rangle$, and $f'(u_i) = u_i'$. We extend the operators $\oplus$, $\ominus$ to functions, i.e., if $h_1, h_2 : V \to \mathbb{M}[U]$ then $(h_1 \oplus h_2)(v) := h_1(v) \oplus h_2(v)$ and $(h_1 \ominus h_2)(v) := h_1(v) \ominus h_2(v)$. Further, let us define for all sets U, V the map $\mathbf{0} : U \to \mathbb{M}[V]$ by $\mathbf{0}(u) = \emptyset$ for all $u \in U$; and abbreviate $\mathbf{0}\,[u_1 \mapsto u_1', \ldots, u_n \mapsto u_n']$ to $[u_1 \mapsto u_1', \ldots, u_n \mapsto u_n']$. Lastly, the set of finite partial functions from U to V is denoted by $U \rightharpoonup V$.

In the next chapter, we introduce a number of computational models and technical notions that we will assume as background for the remainder of this dissertation. Among these are a brief overview of transition systems (Section 2.1), the complexity classes of the fast growing-hierarchy (Section 2.2), models of concurrency (Section 2.3), and models for recursive programs (Section 2.4).

Chapter 2

Background

The material of this dissertation builds on several fundamental concepts to which we give a brief introduction in this chapter. First we present definitions of several classes of *transition systems* and introduce central decision problems in the verification literature that arise in their study (Section 2.1). A significant part of this dissertation is concerned with the analysis of decision problems for infinite state models; the complexity of the latter can often be accurately placed within the hierarchy of *fast growing complexity classes* which we will briefly present in Section 2.2 alongside the commonly used complexity classes. Further, we give a brief exposition of models of concurrency such as *Petri nets* and *well-structured transition systems* (Section 2.3), and present our model of choice for recursive programs, *pushdown systems* (Section 2.4).

2.1 Labelled Transition Systems

Transition systems are essentially directed graphs. In the verification literature they are commonly used to describe how a program, a concurrent system, or other computational model evolves. Transition systems are comprised of two parts: a *state space* which is often used to describe the configuration or state of the modelled system; and a *transition relation* that details how the modelled system changes state. The transition relation may be annotated or *labelled* to indicate, for example, the reason why a transition is possible, an action that is performed along a transition, or a property that holds at the successor state. A transition system with a labelled transition relation is referred to as a *labelled transition system*.

Definition 1. A *labelled transition system* (LTS) $\mathcal{S}$ is a triple $(S_{\mathcal{S}}, \to_{\mathcal{S}}, L_{\mathcal{S}})$ where $S_{\mathcal{S}}$ is a set of *states*, $L_{\mathcal{S}}$ is a set of *transition labels*, and $\to_{\mathcal{S}}$ is a *labelled transition relation* defined as a subset of $S_{\mathcal{S}} \times L_{\mathcal{S}} \times S_{\mathcal{S}}$. We use the notation $s \xrightarrow{l}_{\mathcal{S}} s'$ to denote a transition $(s, l, s') \in \to_{\mathcal{S}}$ and we drop subscripts whenever $\mathcal{S}$ is clear from the context. Suppose there exists a sequence $\boldsymbol{l} \in L_{\mathcal{S}}^*$ of labels such that there are transitions $s_0 \xrightarrow{\boldsymbol{l}(1)}_{\mathcal{S}} s_1 \xrightarrow{\boldsymbol{l}(2)}_{\mathcal{S}} \cdots \xrightarrow{\boldsymbol{l}(n)}_{\mathcal{S}} s_n$ then we write $s_0 \xrightarrow{\boldsymbol{l}}_{\mathcal{S}}^* s_n$. If $L_{\mathcal{S}}$ contains the empty string ϵ as a special label then we identify, as usual, two label sequences $\boldsymbol{l}, \boldsymbol{l}'$ if they are equal after repeatedly applying $\gamma \cdot \epsilon = \epsilon \cdot \gamma = \gamma$. Further, we omit the label from $\xrightarrow{l}$ and $\xrightarrow{l}^*$ to obtain the unlabelled one-step transition relation and its transitive closure.

An *(unlabelled) transition system* $\mathcal{S}$ is a LTS for which L is the singleton-set $\{\epsilon\}$. For brevity we use the unlabelled transition relation $s \to_{\mathcal{S}} s'$ if $s \xrightarrow{\epsilon}_{\mathcal{S}} s'$ and specify $\mathcal{S} = (S, \to_{\mathcal{S}})$ if it is understood that $\mathcal{S}$ is unlabelled.

Remark. As defined unlabelled transition systems constitute a special case of LTS. However, it is equally easy to obtain an unlabelled transition system from an LTS by omitting labels. Hence, it is common practice to apply any notion defined on one to the other, unless explicitly ruled out. We follow this practice and take the more convenient choice depending on the definition at hand.

Suppose that $\mathcal{S}$ is a transition system. We say $\boldsymbol{s} \in S^{*\omega}$ is a *path in* $\mathcal{S}$, just if for all $1 \leq i \leq |\boldsymbol{s}|$, $\boldsymbol{s}(i) \to_{\mathcal{S}} \boldsymbol{s}(i+1)$. If $|\boldsymbol{s}| < \omega$, i.e. $\boldsymbol{s}$ is a finite sequence, we say $\boldsymbol{s}$ is a *finite* path; otherwise $\boldsymbol{s}$ is said to be an *infinite* path. We define $\mathit{Reach}_{\mathcal{S}}(s_0)$, the set of *reachable configurations* from a start configuration s_0 in $\mathcal{S}$, as the set:

$$\mathit{Reach}_{\mathcal{S}}(s_0) := \{s \in S \mid s_0 \to^*_{\mathcal{S}} s\}.$$

In the context of verification, it of central interest to find answers to questions such as: Is it possible to reach an "error" state? Is a given program guaranteed to terminate? Does a software system require ever more resources in the course of its computation? Questions like the former can be phrased on a transition system as the decision problems: *reachability*, *boundedness*, and *termination*.

Decision Problems (Reachability, boundedness, and termination). Let $\mathcal{S} = (S, \to_{\mathcal{S}})$ be a transition system and $s_0 \in S$ be an initial configuration. We say the triple $(\mathcal{S}, s_0, s)$ is a *yes-instance* for *reachability* if s is a reachable configuration from s_0, i.e. $s_0 \to^*_{\mathcal{S}} s$. The *boundedness problem* is to decide whether the reachability set $\mathit{Reach}_{\mathcal{S}}(s_0)$ is finite and the *termination problem* is to decide whether there exists an infinite path from s_0 in $\mathcal{S}$.

Remark. It is customary to include an initial state in the definition of transition systems, sometimes also referred to as *pointed* transition systems, or other computational models. In this dissertation we opt to define all transition systems and computational models as *unpointed*, i.e. without an initial state. Instead we include an initial state as part of decision problems.

A transition system $(S, \to)$ is said to be *finite state* if S is a finite set. Finite state transition systems are an important class of transition systems and play an important rôle especially in the field of hardware and software model-checking, and finite-state verification. A transition system $(S, \to)$ is said to be *infinite state* if S is an infinite set. Thus, to decide the boundedness problem we have to answer the question of whether a general transition system is a *de facto* finite state transition system.

For some transition systems, especially in the context of concurrent systems, it makes sense to think of the state space as describing the *configuration* of many systems each with a *local state*. Such a view induces the notion of subsystem, i.e. a preorder on the state space. From a verification standpoint, it is of interest to consider the reachability of a "bad" subsystem as a decision problem in this setting. Such a structure can be made formal as a *pre-structured transition system*, also known as *preordered transition systems*.

Pre-Structured Transition Systems. Let us fix a few preliminary definitions. A *preorder* is a binary relation that is reflexive and transitive. A triple $\mathcal{S} = (S_{\mathcal{S}}, \to_{\mathcal{S}}, \leq_{\mathcal{S}})$ is said to be a *pre-structured transition system* (PSTS) if $\leq_{\mathcal{S}}$ is a preorder on $S_{\mathcal{S}}$ and $(S_{\mathcal{S}}, \to_{\mathcal{S}})$ is a transition system. For $s, s' \in S$ we say s' *covers* s just if $s \leq_{\mathcal{S}} s'$. A path $\boldsymbol{s}$ in $\mathcal{S}$ is said to be *covering* for s' from s if $\boldsymbol{s}(|\boldsymbol{s}|)$ covers s' and $\boldsymbol{s}(1) = s$.

Decision Problem (Coverability). Let $\mathcal{S} = (S, \to_{\mathcal{S}}, \leq_{\mathcal{S}})$ be a PSTS and $s_0 \in S$ be an initial configuration. We say $Q = (\mathcal{S}, s_0, s_{\text{cov}})$ is a *coverability query* where s_{cov} is said to be the target marking. The coverability query Q is said to be a *yes-instance* for *coverability* if there exists a reachable configuration $s' \in S$ that covers s_{cov}, i.e. $s_0 \to_{\mathcal{S}}^* s'$ and $s_{\text{cov}} \leq_{\mathcal{S}} s'$.

For a subset $U \subseteq S$ we can define U's *upward closure*: $\uparrow U := \{u' \mid u \leq u', u \in U\}$ and U's *downward closure*: $\downarrow U := \{u' \mid u' \leq u, u \in U\}$. Intuitively, the downward closure operator saturates a set with all its subsystems. Thus, we can relate the coverability and reachability problems and see that coverability formalises subsystem reachability: a reachability instance $(\mathcal{S}, s_0, s)$ asks whether $s \in \mathit{Reach}_{\mathcal{S}}(s_0)$; a coverability instance $(\mathcal{S}, s_0, s)$ asks whether $s \in \downarrow\mathit{Reach}_{\mathcal{S}}(s_0)$.

Simulation and Bisimulation. In the study of decision properties of transition systems, it is often fruitful to establish a connection between two LTS. Exhibiting a *simulation* or *bisimulation* is a technique that is commonly employed. Informally, a simulation between two LTS $\mathcal{S}$ and $\mathcal{S}'$ guarantees that any (labelled) path that is possible in $\mathcal{S}$ can be replicated in $\mathcal{S}'$; a bisimulation between $\mathcal{S}$ and $\mathcal{S}'$ ensures that this replication of behaviour is feasible in both directions. Let us give a formal definition of simulation, bisimulation, and their weaker variants.

Definition 2. Suppose $\mathcal{S} = (S, \to_{\mathcal{S}}, L)$ and $\mathcal{S}' = (S', \to_{\mathcal{S}'}, L')$ are labelled transition systems such that $L = L'$. We say a relation $\mathcal{R} \subseteq S \times S'$ is a *simulation* just if for all $(s, s') \in \mathcal{R}$ the existence of a transition $s \xrightarrow{l}_{\mathcal{S}} t$ for some $t \in S$ and $l \in L$ implies that there exists $t' \in S'$ such that $s' \xrightarrow{l}_{\mathcal{S}'} t'$ and $(t, t') \in \mathcal{R}$.

In order to define two weakened variants of simulation, we define the following two conditions for a choice of $s' \in S'$, $l \in L$, and $t \in S$:

(W) there exists $t' \in S'$ such that $s' \xrightarrow{l}_{\mathcal{S}'}^* t'$ and $(t, t') \in \mathcal{R}$;
(R) either $(t, s') \in \mathcal{R}$ and $l = \epsilon$, or there exists $t' \in S'$ such that $s' \xrightarrow{l}_{\mathcal{S}'} t'$ and $(t, t') \in \mathcal{R}$.

The relation $\mathcal{R}$ is said to be a *weak simulation (reflexive simulation)* if for all $(s, s') \in \mathcal{R}$ such that $s \xrightarrow{l}_{\mathcal{S}} t$ for some $t \in S$ and $l \in L$ condition (W) (respectively condition (R)) holds for s', l and t. We say $\mathcal{R}$ is a (weak/reflexive) *bisimulation* just if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are (weak/reflexive) simulations where we define the inverse R^{-1} of a relation R as $R^{-1} := \{(b, a) \mid (a, b) \in R\}$.

Simulations and bisimulations are useful to transfer decision properties from a transition system $\mathcal{S}$ to another $\mathcal{S}'$. For example, suppose there is a simulation $\mathcal{R}$ for $\mathcal{S}$ and $\mathcal{S}'$, $(s, s') \in \mathcal{R}$, and t is reachable from s in $\mathcal{S}$, then we know that some t' is reachable in $\mathcal{S}'$ such that $(t, t') \in \mathcal{R}$. Further, suppose $\mathcal{R}$ is a bisimulation, then $\mathit{Reach}_{\mathcal{S}}(s)$ is a finite set if, and only if, $\mathit{Reach}_{\mathcal{S}'}(s')$ is a finite set. In this dissertation we make frequent use of this strategy when establishing the interreducibility of two decision problems involving transition systems. Sometimes it is more natural to exhibit a

relation between a state of one transition system and a set of states of another. In this case it is often possible to apply the definitions of simulation and bisimulation to a *powerset lifting* of one of the transition systems.

Powerset Lifting. A powerset lifting of a transition system $\mathcal{S} = (S, \rightarrow_{\mathcal{S}})$ is a transition system with state space $\mathscr{P}[S]$. However, there are many choices to lift the transition relation to sets of states. We will present three possible powerset liftings of $\mathcal{S}$: the *existential*, the *universal*, and the *co-universal*.

Definition 3. Suppose $\mathcal{S} = (S, \rightarrow_{\mathcal{S}})$ is a transition system. We define $\mathscr{P}^{\exists}[\mathcal{S}]$ the *existential powerset lifting of* $\mathcal{S}$ by $\mathscr{P}^{\exists}[\mathcal{S}] = (\mathscr{P}[S], \rightarrow_{\mathscr{P}^{\exists}[\mathcal{S}]})$, $\mathscr{P}^{\forall}[\mathcal{S}]$ the *universal powerset lifting of* $\mathcal{S}$ by $\mathscr{P}^{\forall}[\mathcal{S}] = (\mathscr{P}[S], \rightarrow_{\mathscr{P}^{\forall}[\mathcal{S}]})$, and $\mathscr{P}^{\mathsf{A}}[\mathcal{S}]$ the *co-universal powerset lifting of* $\mathcal{S}$ by $\mathscr{P}^{\mathsf{A}}[\mathcal{S}] = (\mathscr{P}[S], \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{S}]})$, where we define the transition relations as follows:

- $S \rightarrow_{\mathscr{P}^{\exists}[\mathcal{S}]} S'$ if $\exists s \in S$, $\exists s' \in S'$ such that $s \rightarrow_{\mathcal{S}} s'$,
- $S \rightarrow_{\mathscr{P}^{\forall}[\mathcal{S}]} S'$ if $\forall s \in S$, $\exists s' \in S'$ such that $s \rightarrow_{\mathcal{S}} s'$, and
- $S \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{S}]} S'$ if $\forall s' \in S'$, $\exists s \in S$ such that $s \rightarrow_{\mathcal{S}} s'$.

A universal powerset lifting can be useful to show that properties have a *forward implication*: suppose, for example, we have sets U and U' that consist of all states that satisfy property P and P' respectively and there is a transition $U \rightarrow_{\mathscr{P}^{\forall}[\mathcal{S}]} U'$, then we know that for all states satisfying P any of its successors will satisfy P'. The co-universal powerset lifting allows us to reason backwards: a transition $U \rightarrow_{\mathscr{P}^{\forall}[\mathcal{S}]} U'$ allows us to deduce that every state satisfying P' has a predecessor that satisfies P.

The decision problems that we are concerned with in this dissertation involve finite and infinite state transition systems that are generated by computational models. In the following sections we will review commonly used complexity classes and two strands of computational models that are often associated with concurrent computations and recursive programs.

2.2 Classes of Computational Complexity

In this section we introduce very briefly the hierarchy of fast growing complexity classes alongside the standard collection of complexity classes. Let us begin with characterising the computational complexity of a decision problem by the *time* required by a Turing machine to provide a solution. Let n denote the size of the input and f be a function; we write

$$\textsc{Time}(f(n)), \qquad \textsc{NTime}(f(n)), \qquad \textsc{AltTime}(f(n))$$

for the sets of problems decidable in *time* $f(n)$ by a deterministic, non-deterministic, and alternating (respectively) Turing machine. Similarly, there is such a standard characterisation with respect to space requirements. We write

$$\textsc{Space}(f(n)), \qquad \textsc{NSpace}(f(n)), \qquad \textsc{AltSpace}(f(n))$$

to mean the sets of problems decidable in *space* $f(n)$ by a deterministic, non-deterministic, and alternating (respectively) Turing machine.

The standard collection of complexity classes can then be defined by constraining the choice of f: let p be a meta variable denoting a polynomial, we then define:

$$\text{P} = \bigcup_p \textsc{Time}(p(n)), \qquad \text{NP} = \bigcup_p \textsc{NTime}(p(n)),$$
$$\textsc{Exptime} = \bigcup_p \textsc{Time}\left(2^{p(n)}\right), \quad \textsc{NExptime} = \bigcup_p \textsc{NTime}\left(2^{p(n)}\right),$$
$$\textsc{Expspace} = \bigcup_p \textsc{Space}\left(2^{p(n)}\right), \qquad \textsc{NExpspace} = \bigcup_p \textsc{NSpace}\left(2^{p(n)}\right),$$
$$\textsc{AltExpspace} = \bigcup_p \textsc{AltSpace}\left(2^{p(n)}\right).$$

In order to define a more complex class, it is useful to introduce the operator that produces a tower of exponentials $2^{2^{2^{\cdots}}}$ of height n: the *tetration* operator $- \uparrow\uparrow -$. Tetration is defined inductively: $k \uparrow\uparrow 0 = 1$ and $k \uparrow\uparrow (n+1) = k^{k \uparrow\uparrow n}$; we write slog_k, *super logarithm*, for its inverse, i.e. $\text{slog}_k(k \uparrow\uparrow n) = n$ and for $k = 2$ we drop the subscript and write slog to mean slog_2. We can then define the complexity class Elementary which contains the problems that are decidable in time bounded by some tower of exponentials:

$$\textsc{Elementary} = \bigcup_{i \geq 0} \textsc{Time}(2 \uparrow\uparrow (i + \text{slog}(n))).$$

For a comprehensive treatment of complexity theory and the standard complexity classes, we refer the reader to Sipser's excellent book [Sip97] on the subject.

Fast-Growing Complexity Classes. Some decision problems involving infinite state computational models exhibit a computational complexity that lies beyond the class Elementary. In order to provide meaningful complexity classes for such decision problems, Schmitz recently introduced the hierarchy of *fast-growing complexity classes* [Sch13] based on the *extended Grzegorczyk hierarchy* of functions [LW70].

Ordinals, Cantor Normal Form, and Fundamental Sequences. Both the extended Grzegorczyk hierarchy and the hierarchy of fast-growing functions are indexed by *ordinals*. An ordinal α is said to be a *successor ordinal* if $\alpha = \alpha' + 1$ for an ordinal α'. Otherwise if $\alpha > 0$ we say α is a *limit ordinal*. We restrict ourselves in the following to ordinals below ϵ_0, i.e. the least ordinal α to satisfy $\alpha = \omega^\alpha$, which can be expressed in *Cantor Normal Form* (CNF):

$$\alpha = \omega^{\alpha_1} \cdot c_1 + \cdots \omega^{\alpha_n} \cdot c_n \text{ where } \alpha > \alpha_1 > \cdots > \alpha_n \text{ and } \omega > c_1, \ldots, c_n > 0.$$

A limit ordinal α can be written as $\alpha = \alpha_0 + \omega^{\alpha_1}$ such that $\alpha_1 > 0$ and either $\alpha_0 = 0$ or α_0 is a limit ordinal. A sequence of ordinals $\alpha_1, \ldots, \alpha_n, \ldots$ that converges to a limit ordinal α from below is said to be a fundamental sequence for α. We define the *(canonical) fundamental sequence* $\boldsymbol{\alpha}$ for the limit ordinal α by induction:

$$\boldsymbol{\alpha}(i) = \alpha_0 + \omega^{\alpha_1} \cdot (i+1) \qquad \text{if } \alpha = \alpha_0 + \omega^{\alpha_1 + 1}, \text{ and}$$
$$\boldsymbol{\alpha}(i) = \alpha_0 + \omega^{\boldsymbol{\alpha_1}(i)} \qquad \text{if } \alpha = \alpha_0 + \omega^{\alpha_1} \text{ and } \alpha_1 \text{ is a limit ordinal.}$$

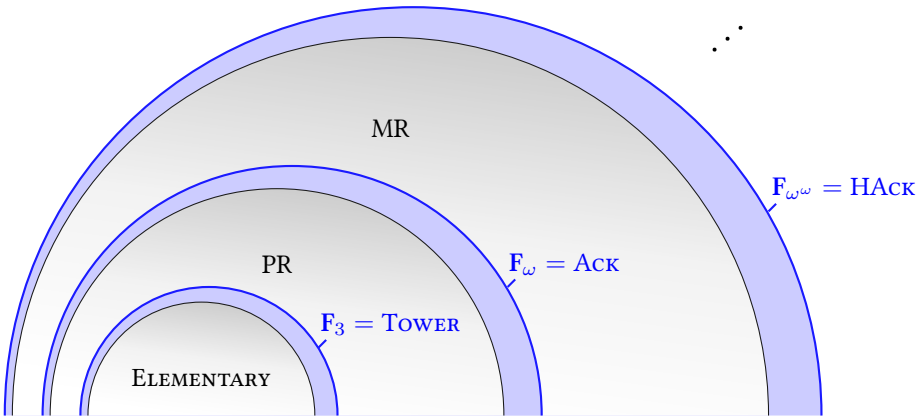


Figure 2.1: The complexity classes Elementary, Tower, primitive recursive (PR), Ack, multiply recursive (MR), and HAck. Reproduced with permission and with minor modifications from [Sch13].

Fast-Growing Functions. For each ordinal α the *fast-growing function* $F_\alpha : \mathbb{N} \to \mathbb{N}$ is defined inductively: $F_0(i) = i + 1$ as the base case,

$$\begin{aligned} F_{\alpha'+1}(i) &= \left(F_{\alpha'}\right)^{i+1}(i) && \text{if } \alpha = \alpha' + 1 \text{ is a successor ordinal, and} \\ F_\alpha(i) &= F_{\boldsymbol{\alpha}(i)}(i) && \text{if } \alpha \text{ is a limit ordinal.} \end{aligned}$$

The extended Grzegorczyk hierarchy of functions $(\mathscr{F}_\alpha)_\alpha$ is then defined using the fast-growing functions F_α: $\mathscr{F}_\alpha$ denotes the set of functions f such that there exists a constant $c < \omega$ so that $f(n)$ can be computed by a deterministic Turing machine in time $O\left((F_\alpha)^c(n)\right)$. Writing $\mathscr{F}_{<\alpha} = \bigcup_{\alpha'<\alpha} \mathscr{F}_{\alpha'}$ the *hierarchy of fast-growing complexity classes* is then defined by

$$\mathbf{F}_\alpha = \bigcup_{g \in \mathscr{F}_{<\alpha}} \textsc{Time}\left(F_\alpha(g(n))\right).$$

Using many-one $\mathscr{F}_{<\alpha}$-reductions gives rise to a notion of completeness for the complexity class $\mathbf{F}_\alpha$ that is familiar from the standard complexity classes. Schmitz defines three complexity classes: Tower $= \mathbf{F}_3$ the class of the "easiest" non-elementary decision problems closed under Elementary reductions; Ack $= \mathbf{F}_\omega$ the class of the least difficult non-primitive recursive problems closed under primitive recursive reductions; and HAck $= \mathbf{F}_{\omega^\omega}$ a class of hyper-Ackermannian decision problems closed under multiply-recursive reductions [Sch13].

We use the last part of this section to quote Savitch's Theorem relating NSpace and Space and to remind the reader that beyond Tower it is unnecessary to distinguish between time and space complexity:

Theorem (Savitch's Theorem [Sav70]). $\textsc{NSpace}(f(n)) \subseteq \textsc{Space}\big((f(n))^2\big)$.

Lemma. $\textsc{Space}\left(f(n)\right) \subseteq \textsc{Time}\left(2^{f(n)}\right)$.

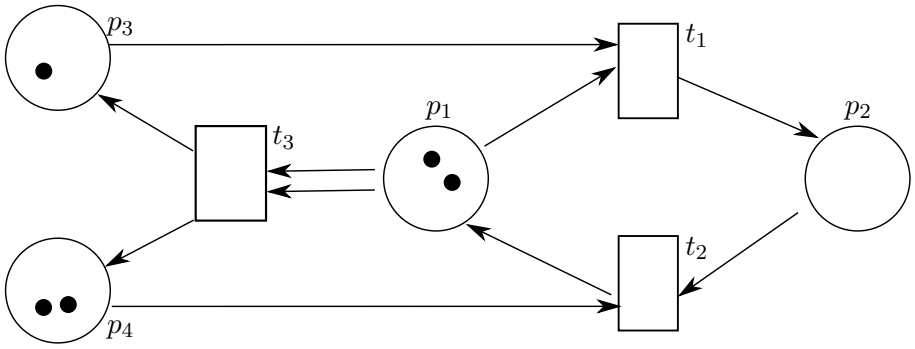


Figure 2.2: A graphical representation of the Petri net $\mathcal{V}$ in Example 2.

2.3 Computational Models for Concurrency

Concurrency is one of the central yet elusive concepts of computer science. It has been the subject of many decades of research and still today many central questions are unanswered. The study of *Petri nets* has shed much light on the nature of concurrency and to the present day they remain an important subject of study.

Petri Nets: Two Definitions and a Graphical Representation

Petri nets [Pet62] are widely recognised as one of the fundamental models for concurrency. Since their inception in the middle of the 20th century by Carl Adam Petri, they have stimulated a large body of research. Many decision results are known [Esp98], numerous extensions have been proposed [AK76; Cia94; FMP04; Rei08], and various equivalent definitions exist:

Definition 4. A *Petri net* (PN) is a pair $\mathcal{V} = (P, T)$ where P is a finite set of *places* and T is a finite set of *transitions*. A transition t is a pair $t = (I, O)$ where both I and O are mappings of type $P \to \mathbb{N}$. A configuration s is a distribution of *tokens* over places which is represented by a mapping $P \to \mathbb{N}$. We say a transition $t = (I, O)$ is enabled at a configuration s if $(s - I)(p) \geq 0$ for all places p. The Petri net $\mathcal{V}$ induces a transition system over configurations where the (T-labelled) transition relation $\to_{\mathcal{V}}$ is defined by $s \xrightarrow{t}_{\mathcal{V}} s'$ just if $t = (I, O) \in T$, $s' = s - I + O$, and t is enabled at s.

Example 2. A Petri net may be represented graphically. Let us consider the Petri net $\mathcal{V} = (P, T)$ where $P = \{p_1, p_2, p_3, p_4\}$, $T = \{t_1, t_2, t_3\}$, and the transitions $t_1, \ldots, t_4$ are as follows:

$$t_1 = ([p_1 \mapsto 1, p_3 \mapsto 1], [p_2 \mapsto 1]), \qquad t_2 = ([p_2 \mapsto 1, p_4 \mapsto 1], [p_1 \mapsto 1]), \text{ and}$$
$$t_3 = ([p_1 \mapsto 1], [p_3 \mapsto 1, p_4 \mapsto 1]).$$

In Figure 2.2 we display $\mathcal{V}$. Places are represented as circles and transitions as boxes. We use arrows to display the pair of mappings making up a transition. For a transition $t = (I, O)$ we draw $I(p)$ many incoming arrows, originating from p ending in t, and $O(p)$ arrows from t to p

for each place p. For example transition t_1 in $\mathcal{V}$: there is an arrow from p_1 to t_1, one from p_3 to t_1, and one from t_1 to p_2.

Using this representation it is easy to see that the transition relation describes the well-known *token game*. Transitions allow us to remove tokens from some places and add them to others. For example, in the configuration displayed in Figure 2.2 we can fire t_1 to move one token from p_1 to p_2 while removing a token from p_3. At this point we can fire t_2 to move the token in p_2 back to p_1 removing a token from p_4. The configuration we reach is the one displayed in Figure 2.2 except that p_3 is empty and p_4 contains only one token.

There is an equivalent definition for Petri nets in terms of the more general class of affine nets [FMP04].

Definition 5 (Petri net, affine net). An *affine net* (AN) is a pair $\mathcal{V} = (d, F)$ where $d \in \mathbb{N}^+$ is the *dimension* and $F \subseteq \mathbb{N}^{d\times d} \times \mathbb{Z}^d \times \mathbb{N}^d$ is a set of *transition* functions. The transition relation is defined for $s, s' \in \mathbb{N}^d$ and $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) \in F$ we have $s \xrightarrow{(\boldsymbol{A},\boldsymbol{B},\boldsymbol{C})}_{\mathcal{V}} s'$ just if $s \geq \boldsymbol{C}$, $s' = \boldsymbol{A}\, s + \boldsymbol{B}$ and $s' \in \mathbb{N}^d$. We say an AN $\mathcal{V} = (d, F)$ is a *Petri net* just if for all $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) \in F$, $\boldsymbol{A} = id$ the unique matrix such that $id \cdot s = s \cdot id = s$ in which case we treat F as $F \subseteq \mathbb{Z}^d \times \mathbb{N}^d$.

Defining Petri nets as a class of affine nets makes the well-known connection between and Petri nets and *vector addition systems* obvious.

Definition 6. A *vector addition system* (VAS) is an AN $\mathcal{V} = (d, F)$ such that all transitions $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) \in F$ satisfy $\boldsymbol{A} = id$ and $\boldsymbol{C} = 0$. The set of transitions is thus commonly treated as a subset of $\mathbb{Z}^d$.

It is well-known that VAS are equivalent to Petri nets. Petri nets subsume VAS and VAS can weakly bisimulate Petri nets.

It is not hard to see that the common extension of $\leq_{\mathbb{N}}$ to $\mathbb{N}^d$, i.e. $a \leq_{\mathbb{N}^d} b$ if $a(i) \leq_{\mathbb{N}} b(i)$ for all $i \in \langle d \rangle$, gives rise to a preorder on $\mathbb{N}^d$. We note that AN and thus Petri nets and VAS give rise to PSTS when equipped with $\leq_{\mathbb{N}^d}$.

Decision and Complexity Results for Petri Nets and Extensions

Many complexity and decision results are known for Petri nets: coverability and boundedness are decidable [KM69] and Expspace-complete [Rac78; Lip76]. Moreover, decidability and Expspace-completeness results for termination [KM69; Lip76; HRY91], place-boundedness [KM69; Lip76; Dem13], model-checking LTL [Esp94; Hab97], and model-checking fragments of Presburger CTL [BS11b] have been established. The reachability problem was proved to be decidable [ST77; May81; Kos82; Ler11a] more than 30 years ago yet its complexity is still open. An extensive survey on the decision and complexity properties of Petri nets by Esparza and Nielsen can be found in [EN94].

Numerous extensions to Petri nets or their VAS equivalent have been devised and studied. Petri nets with *reset arcs* [AK76] or *transfer arcs* [Cia94] can be seen as an instance of affine nets. A reset transition that resets place p is an affine transition $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ where $\boldsymbol{A} = id[(p, p) \mapsto 0]$. A transfer transition which transfers the contents from place p to p' is an affine transition $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ where $\boldsymbol{A} = id[(p, p) \mapsto 0, (p, p') \mapsto 1]$. Petri nets with reset arcs or transfer arcs

are then Petri nets that allow reset transitions or transfer transitions respectively. Reachability is known to be undecidable for Petri nets with reset or transfer arcs [AK76]. In the case of reset Petri nets this remains true for the boundedness problem which implies the undecidability of place-boundedness for transfer Petri nets [DFS98]. On the other hand, boundedness remains decidable for transfer Petri nets [DFS98]. Coverability and termination for Petri nets with reset or transfer arcs is known to be decidable [DFS98] and Ack-complete [Sch02; Sch10; Fig+11]. Of course, affine nets themselves are an extension of Petri nets. One subclass of AN called *strongly increasing affine nets* (SIAN) [BFP12] effectively prohibits reset or transfer like transitions. A strongly increasing affine transition $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ is required to satisfy $\boldsymbol{A} \geq id$. Coverability and boundedness are known to be Expspace-complete [BFP12] for SIAN, yet reachability becomes undecidable in the presence doubling arcs [DFS98]. The decision and complexity properties of general AN are inherited from reset Petri nets (reachability and boundedness: undecidable, coverability and termination: Ack-complete [Fig+11]).

VAS may be extended by adding a finite control-state component. The resulting model is called *VAS with states* (VASS) which is equivalent to VAS [HP79]. Petri nets may also be extended with *inhibitor arcs* or *zero tests*. A zero test transition is labelled by a place p and is only enabled at a configuration s if $s(p) = 0$. The ability to test two places for zero renders such an augmented Petri net Turing powerful [Min61]. Perhaps surprisingly, positive decision results exist if only one place may be tested for zero (VAS_0) [Rei08; Bon11b] and if tests are hierarchical [Rei08], i.e. if $p_1, \ldots, p_n$ may be tested for zero then a zero test transition may require all places $p_1, \ldots, p_i$ to be zero for some $i \in \langle n \rangle$. Positive decidability results have been established for both VAS_0 and VAS with hierarchical zero tests for reachability and coverability [Rei08; Bon11b; AM09], LTL model-checking [Bon11a], boundedness, and place-boundedness [Bon+10]. However, except for coverability which is known to be at least as hard as Petri net reachability [BFP12], no information is available on the complexity of the former decision problems.

There is a pattern that underlies decision results for a wide range of Petri net extensions and variants. This pattern has lead to the introduction of *well-structured transition systems* [Fin90] by Finkel.

Well-Structured Transition Systems

All the extensions of Petri nets we have discussed so far describe pre-structured transition systems with a preordered state space isomorphic to $(\mathbb{N}^d, \leq_{\mathbb{N}^d})$. Moreover, the preorder $\leq_{\mathbb{N}^d}$ is a *well-quasi-order*. Let $\leq_U$ be an preorder over a set U. We say $\leq_U$ is a *well-quasi-order* (WQO) just if for all infinite sequences $u_1, u_2, \ldots$ there exist $i < j$ such that $u_i \leq_U u_j$.

Karp and Miller exploited the WQO property of $\leq_{\mathbb{N}^d}$ to show that a VAS has a finite coverability tree, also known as the *Karp-Miller tree*, and thereby provided decision procedures for coverability, termination, boundedness, and place-boundedness. Many further decision results on Petri nets have been developed using the Karp-Miller tree [BS11b; VVN81; FS08]. VAS have a special property with respect to $\leq_{\mathbb{N}^d}$: the transition relation is *monotone* with respect to $\leq_{\mathbb{N}^d}$, i.e. we may think of $\leq_{\mathbb{N}^d}$ as a simulation relation. The combination of $\leq_{\mathbb{N}^d}$ being a WQO and the monotonicity of the transition relation has important consequences: suppose $\boldsymbol{s}$ is a potentially infinite path; since $\leq_{\mathbb{N}^d}$ is a WQO either $\boldsymbol{s}$ is finite or there exists i, j such that $\boldsymbol{s}(i) \leq_{\mathbb{N}^d} \boldsymbol{s}(j)$ and monotonicity implies that any transition that is possible from $\boldsymbol{s}(i)$ is also possible from $\boldsymbol{s}(j)$.

The transitions $\boldsymbol{s}(i), \boldsymbol{s}(i+1), \ldots, \boldsymbol{s}(j)$ can thus be seen as a (potentially increasing) loop. In the Karp-Miller tree construction the loop $\boldsymbol{s}(i), \boldsymbol{s}(i+1), \ldots, \boldsymbol{s}(j)$ is accelerated and $\boldsymbol{s}(j)$ is widened to its limit. The key properties that underlie this construction were distilled and generalised by Finkel and led to the definition of *well-structured transition systems* [Fin87]:

Definition 7. A *well-structured transition system* (WSTS) is a PSTS $\mathcal{S} = (S, \to_{\mathcal{S}}, \leq_{\mathcal{S}})$ such that $\leq_{\mathcal{S}}$ is a WQO over S and $\to_{\mathcal{S}} \subseteq S \times S$ is monotone with respect to $\leq_{\mathcal{S}}$, i.e. if $s \to_{\mathcal{S}} s'$ and $s \leq_S t$ then there exists t' such that $t \to_{\mathcal{S}} t'$ and $s' \leq_{\mathcal{S}} t'$.

We say $\mathcal{S}$ is a *strict WSTS* if $\to_{\mathcal{S}}$ satisfies *strict monotonicity*: if $s \to_{\mathcal{S}} s'$ and $s <_S t$ then there exists t' such that $t \to_{\mathcal{S}} t'$ and $s' <_{\mathcal{S}} t'$.

WSTS have emerged as a standard tool for decision results for infinite-state transition systems and many instances of WSTS are know to date [FS01]. Importantly, extensive research has shown that WSTS enjoy good model-checking properties [Fin87; Abd+96; FS98; Abd+00; FS01]. Let us define the set of one-step predecessors of a set U as $\mathit{Pred}_{\mathcal{S}}(U) := \{s \mid s \to_{\mathcal{S}} u, u \in U\}$, and we write $\mathit{Pred}^*(U)$ for the set of predecessors of a set U, i.e. $\mathit{Pred}^*(U) = \{s \mid s \to^*_{\mathcal{S}} u, u \in U\}$. The coverability, termination, and boundedness problems are decidable for a WSTS $\mathcal{S}$ provided that for any given $s \in S$ the set $\uparrow \mathit{Pred}(\uparrow\{s\})$ is effectively computable, the wqo $\leq_{\mathcal{S}}$ is decidable, and in the case of the boundedness problem $\mathcal{S}$ is a strict WSTS [FS01].

Monotonicity in WSTS allows us to relate coverability and reachability in a *backward* fashion. A reachability instance $(\mathcal{S}, s_0, s)$ asks whether $s_0 \in \mathit{Pred}^*_{\mathcal{S}}(s)$. In comparison, a coverability instance $(\mathcal{S}, s_0, s)$ asks whether $s_0 \in \uparrow \mathit{Pred}^*_{\mathcal{S}}(s)$. Note that monotonicity implies the equivalence: $s_0 \in \uparrow \mathit{Pred}^*_{\mathcal{S}}(s)$ if, and only if, $s \in \downarrow \mathit{Reach}_{\mathcal{S}}(s_0)$. For any subset $U \subseteq S$ the upward closure $\uparrow U$ can be represented by a finite set of minimal elements of U thanks to $\leq_{\mathcal{S}}$ being a WQO. These facts are exploited by the general WSTS backward fixpoint algorithm [Abd+96] for coverability:

$$\uparrow\{s\} \subseteq \uparrow \mathit{Pred}_{\mathcal{S}}(\uparrow\{s\}) \cup \uparrow\{s\} \subseteq \cdots \subseteq \bigcup_{i=0}^{k} \left(\uparrow \mathit{Pred}_{\mathcal{S}}\right)^i (\uparrow\{s\})$$

which is guaranteed to terminate in a finite number of steps [FS01].

Composing WQOs An important part of demonstrating that a PSTS $\mathcal{S}$ is a WSTS is to show that $\leq_{\mathcal{S}}$ is a WQO. Fortunately, the WSTS toolbox comes with a variety of results on constructing WQOs. WQOs can be composed in various ways which makes decision results for WSTS applicable to a wide variety of infinite-state models. In the following we recall a few results on the composition of WQO.

(WQO-a) If $(A_i, \leq_i)$ are WQO sets for $i = 1, ..., k$ then $(A_1 \times \cdots \times A_k, \leq_1 \times \cdots \times \leq_k)$ is a WQO set. (*Dickson's Lemma*)

(WQO-b) If A is a finite set then $(A, =)$ is a WQO set.

(WQO-c) If $(A, \leq_A)$ is a WQO then $(A^*, \leq_{A^*})$ is a WQO where we define $\boldsymbol{a} \leq_{A^*} \boldsymbol{a}'$ if there is a strictly monotone map $h : \langle|\boldsymbol{a}|\rangle \to \langle|\boldsymbol{a}'|\rangle$ such that $\boldsymbol{a}(i) \leq_A \boldsymbol{a}'(h(i))$ for all $i \in \langle|\boldsymbol{a}|\rangle$. (generalised *Higman's Lemma*)

(WQO-d) If $(A, \leq_A)$ is a WQO then $(\mathbb{M}[A], \leq_{\mathbb{M}[A]})$ is a WQO where we order $M \leq_{\mathbb{M}[A]} M'$ just if $M = [m_1, \ldots, m_n], M' = [m'_1, \ldots, m'_{n'}]$ and there is an injective function $h : \langle n\rangle \to \langle n'\rangle$ and $m_i \leq_A m'_{h(i)}$ for all $i \in \langle n\rangle$.

(WQO-e) If $(A, \leq_A)$ and $(B, \leq_B)$ are WQO sets, then $(A \cdot B, \leq_A \cdot \leq_B)$ is a WQO set, where we define $\gamma \cdot \gamma' \leq_A \cdot \leq_B \delta \cdot \delta$ just if $\gamma \leq_A \delta$ and $\gamma' \leq_B \delta'$.

(WQO-f) If $(A, \leq_A)$ and $(B, \leq_B)$ are WQO set, then $(A + B, \leq_A + \leq_B)$ is a wqo set, where $a \leq_A + \leq_B b$ just if $a, b \in A$ and $a \leq_A b$ or $a, b \in B$ and $a \leq_B b$.

Examples of WSTS. WSTS are ubiquitous. Petri nets, VAS, and affine nets are all instances of WSTS. For well chosen WQO other computational models are also WSTS: (i) *lossy counter machines* [BM99] which have unreliable counters that may non-deterministically decrement, (ii) *lossy channel systems* [AJ93] — communicating finite state machines with unreliable channels that may loose messages, (iii) the *depth-bounded π-calculus* [Mey08] — an expressive fragment of Milner et al.'s process calculus [MPW92], (iv) context-free grammars [FS01], and (v) *weak class memory automata* [CMO14] — a class of automata accepting languages over infinite alphabets.

Other Models of Concurrency. For completeness we would like to point the reader to other models of concurrency such as *communicating finite state machines* (CFSM) [BZ83], Milner's *calculus of communicating systems* (CCS) [Mil80], the *π-calculus* [MPW92], and Hoare's *communicating sequential processes* (CSP) [Hoa78]. Further, we would like to mention the process algebras of the hierarchy of *Process Rewrite Systems* (PRS) [May00]: (i) *basic parallel processes* (BPP) [CHM93] which are equivalent to communication-free Petri nets [Esp97], (ii) the *basic process algebra* (BPA) [BK84] describing sequential context-free processes, (iii) the *process algebra* (PA) which extends BPA with parallel composition but without synchronisation, and (iv) *PAD* (PA + pushdown processes) which subsumes PA and BPA [May00]. Lastly, *process algebra nets* (PAN) [May97], which are also part of the PRS hierarchy, are of interest since they subsume both PA and Petri nets. Thus, calls and returns may be modelled. However, synchronisations in PAN are restrictive. For a set of processes $\pi_1, \ldots, \pi_n$ to synchronise, they are restricted to be leafs of the same branch in the tree topology, and $\pi_1, \ldots, \pi_n$ are required to equal the premise of one of the finite number of rules form $\pi_1 \parallel \cdots \parallel \pi_n \to \pi$. Hence, it is impossible, in general, for a stack to be remembered along a synchronisation.

2.4 Computational Models for Recursive Programs

Pushdown systems are a commonly employed model for the analysis of sequential recursive programs. In this section we give a brief introduction to pushdown systems and illustrate the connection between recursive programs and pushdown systems by an example. Let us first recall their definition:

Definition 8 (Pushdown system). Let Σ be a stack alphabet. A *pushdown system* is a triple $\mathcal{P} = (\Sigma, \mathcal{Q}, \mathcal{T})$ where $\mathcal{Q}$ is a set of *control states* and $\mathcal{T} \subseteq \mathcal{Q} \times \Sigma^* \times \mathcal{Q} \times \Sigma^*$ is a finite set of transition rules. In the following we write $(q, \gamma) \to (q', \gamma')$ for a rule $(q, \gamma, q', \gamma') \in \mathcal{T}$.

A pushdown system $\mathcal{P}$ gives rise to an infinite-state transition system $(\mathcal{Q} \times \Sigma^*, \to_{\mathcal{P}})$ where the transition relation $\to_{\mathcal{P}}$ is defined by $(q, \gamma\delta) \to_{\mathcal{P}} (q', \gamma'\delta)$ for all $\delta \in \Sigma^*$ and $(q, \gamma) \to (q', \gamma') \in \mathcal{T}$.

```
1  readBoolean() →
2    X := false ,
3    X := read() ,
4    case X of
5      true  → readBoolean(), printTrue() ;
6      false → printFalse () ;
7    end.
8  printTrue() →
9    print "true".
10
11 printFalse () →
12   print "false".
13
14 main() → readBoolean().
```

Figure 2.3: A recursive program reading boolean values from input. As long as the input provided is true, the program continues to prompt for input; otherwise the programs prints out the list of inputs given in reverse order. The sequences of interaction always have the form $(\text{read})^n$ print "*false*" $(\text{print "}\textit{true}\text{"})^n$.

Example 3 (Recursive Programs and Pushdown Systems). Consider the program P in Figure 2.3. The procedure readBoolean() reads a boolean from input and calls itself recursively if the input was true; otherwise it prints all received inputs in reverse order.

We will present two different pushdown systems $\mathcal{P}_1$ (Figure 2.4) and $\mathcal{P}_2$ (Figure 2.5) that implement P. In Figure 2.4 we show a graphical representation of the rules $\mathcal{P}_1$. The control states of $\mathcal{P}_1$ are tuples of line numbers and valuations of the local variable X except for the entry point main. $\mathcal{P}_1$ follows P in the execution of line numbers and uses its stack to remember the return state conforming to the stack frame model. Nodes in the graph represent control states and edges represent rules. For example, the edge labelled with 'push $(\mathrm{L}_9, X = T)$' represents the rule $((\mathrm{L}_4, X = T), \epsilon) \to ((\mathrm{L}_2, X = T), (\mathrm{L}_9, X = T))$. To guide the intuition of the reader, we have labelled transitions with their input/output behaviour; the latter is not explicitly modelled by the pushdown system. We note that it is immediately visible that (i) $\mathcal{P}_1$ pushes the stack symbol $(\mathrm{L}_9, X = T)$ on its stack every time it receives the input '*true*' and (ii) every time $(\mathrm{L}_9, X = T)$ is popped off the stack $\mathcal{P}_1$ outputs '*true*'. We present the rules of $\mathcal{P}_2$ in Figure 2.5. In contrast to $\mathcal{P}_1$, the control states of $\mathcal{P}_2$ comprise the control states q_v, where v is a valuation of X, and the single control state q indicating *no variable in scope*. The pushdown system $\mathcal{P}_2$ uses the stack to remember which program points still need to be executed. The top of the stack signifies the current program point. A program point of P is either a line number (L_i), an end of line (EOL_i) or a procedure name. To help the reader we annotate $\mathcal{P}_2$'s rules similarly to the rules of $\mathcal{P}_1$ with reads and prints. We note that the rules of $\mathcal{P}_2$ are reminiscent of the code of P. In contrast to $\mathcal{P}_1$'s stack frame model implementation of P, the pushdown system $\mathcal{P}_2$ appears more like a rewrite system.

Complexity and Decision Results and Extensions. A well-known result due to Büchi is that the language of stack-contents of the reachable states of a pushdown-system forms a regular language [Bü64]. The regular automaton recognising this language can be computed in polynomial time [Cau88]. It follows that reachability, control-state reachability, and boundedness are decidable in polynomial time. Termination is also decidable [MS85], via decidability of MSO, and a polynomial time algorithm exists [BEM97]. Büchi's result has been exploited to decide complex properties, for example, to show that model-checking the alternation-free fragment of the μ-calculus is decidable and EXPTIME-complete [BEM97].

Pushdown systems have been extended to highly expressive computational models such as higher-order pushdown systems [Mas76], which feature nested stacks, and collapsible push-

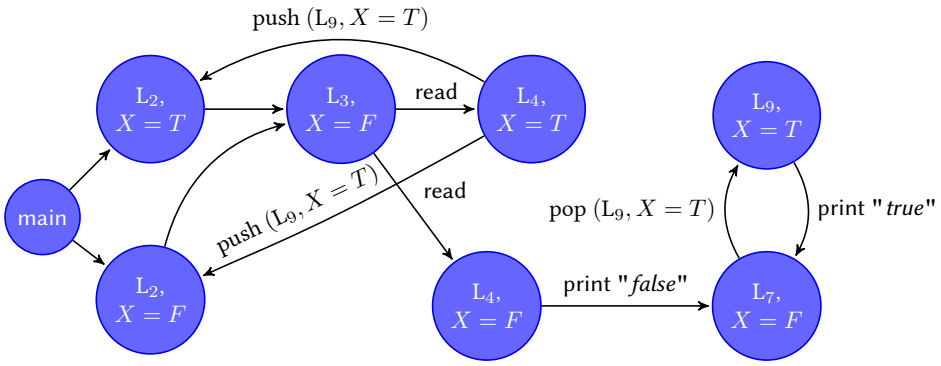


Figure 2.4: The pushdown system $\mathcal{P}_1$ implementing the program in Figure 2.3 using its stack to remember return-points.

down systems [Hag+08] that are an extension of higher-order pushdown systems corresponding to higher-order recursion schemes. Reachability and model-checking modal μ-calculus for both higher-order and collapsible pushdown systems is decidable and Tower-complete [Ong06; Ong13].

Pushdown Systems and Context-Free Grammars. We can simplify reasoning about pushdown systems by imposing a *normal form*. We will discuss two syntactic restrictions on pushdown systems that do not lose expressivity. The first, which we will refer to as *push/pop normal form*, limits stack pushes and pops to one symbol at a time. Hence all rules are of the form:

$$(q, A) \to (q', \epsilon), \qquad (q, \epsilon) \to (q', \epsilon), \text{ or} \qquad (q, \epsilon) \to (q', A).$$

A general pushdown system can be transformed to fit push/pop normal form by introducing control states and performing pushes and pops one-by-one. The second normal form, called *Chomsky normal form*, limits a pushdown system to use only a single control state q_0 and rules are required to be of the form:

$$(q_0, A) \to (q_0, \epsilon), \qquad (q_0, A) \to (q_0, B), \text{ or} \qquad (q_0, A) \to (q_0, B\,C).$$

Given a pushdown system $\mathcal{P}$ fitting the push/pop normal form, it is possible to encode $\mathcal{P}$'s control states into the stack alphabet. Given a stack character A the transformation produces $A^{q,q'}$ for all control states q, q' of $\mathcal{P}$. To obtain the rules of the pushdown system in Chomsky normal form, the following mapping is applied to $\mathcal{P}$'s rules:

$$\begin{aligned}
(q, A) \to (q', \epsilon) \quad &\mapsto \quad (q_0, A^{q,q'}) \to (q_0, \epsilon) \\
(q, \epsilon) \to (q', \epsilon) \quad &\mapsto \quad (q_0, B^{q,q''}) \to (q_0, B^{q',q''}) \\
(q, \epsilon) \to (q', A) \quad &\mapsto \quad (q_0, B^{q,q''}) \to (q_0, A^{q',q'''}\, B^{q''',q''})
\end{aligned}$$

where B ranges over stack characters and a fresh stack character Θ, and the meta variables q'', q''' range over control states. Thus, each rule of $\mathcal{P}$ yields a set of rules. A state $(q, A_1 \cdots A_n)$ of

$$
\begin{aligned}
(q, \mathsf{readBoolean}) &\to (q_{X=b}, \mathrm{L}_2\,\mathrm{L}_3\,\mathrm{L}_4, \mathrm{L}_7),\\
(q_{X=b}, \mathrm{L}_2) &\to (q_{X=F}, \epsilon),\\
(q_{X=b}, \mathrm{L}_3) &\xrightarrow{\mathsf{read}} (q_{X=T}, \epsilon), \qquad (q_{X=b}, \mathrm{L}_3) \xrightarrow{\mathsf{read}} (q_{X=F}, \epsilon),\\
(q_{X=T}, \mathrm{L}_4) &\to (q, \mathsf{readBoolean\ printTrue}\,\mathrm{EOL}_5), \qquad (q, \mathrm{EOL}_5) \to (q_{X=T}, \epsilon),\\
(q_{X=F}, \mathrm{L}_4) &\to (q, \mathsf{printFalse}\,\mathrm{EOL}_6), \qquad (q, \mathrm{EOL}_6) \to (q_{X=F}, \epsilon),\\
(q_{X=b}, \mathrm{L}_7) &\to (q, \epsilon),\\
(q, \mathsf{printTrue}) &\xrightarrow{\mathsf{print\ "}\mathit{true}\mathsf{"}} (q, \epsilon),\\
(q, \mathsf{printFalse}) &\xrightarrow{\mathsf{print\ "}\mathit{false}\mathsf{"}} (q, \epsilon),\\
(q, \mathsf{main}) &\to (q, \mathsf{readBoolean})
\end{aligned}
$$

Figure 2.5: The rules of $\mathcal{P}_2$ implementing the program in Figure 2.3 where b, b' range over $\{T, F\}$. The stack of $\mathcal{P}_2$ represents the lines of code to be executed.

the pushdown system $\mathcal{P}$ is represented by $(q_0, A_1^{q,q_1} A_2^{q_1,q_2} \cdots A_n^{q_{n-1},q_n} \Theta^{q_n,q_{n+1}})$ for some control states $q_1, \ldots, q_{n+1}$; Θ is intended to signify the end of the stack. The control state q_i in this representation is a guess whether a rule $(q^\dagger, C) \to (q_i, \epsilon)$ will be used to pop the i-th stack character C. This is enabled by a rule $(q_0, C^{q^\dagger,q_i}) \to (q_0, \epsilon)$ in the transformed pushdown system. The transformation we have outlined above is, of course, a description of the well-known connection between pushdown systems and *context-free grammars*.

Definition 9. Let Σ be an alphabet of terminal symbols. A *context-free grammar* (CFG) in *Chomsky normal form* is a triple $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R})$ where $\mathcal{R}$ is a set of rewrite rules of the following types: (i) $X \to Y\ Z$ and (ii) $X \to a$ where $X, Y, Z \in \mathcal{N}$ and $a \in \Sigma \cup \{\epsilon\}$.

Traditionally a CFG $\mathcal{G}$ is a computational device that generates words, i.e. sequences, in Σ^*. Words are generated or *derived* from $\mathcal{G}$ using the derivation relation $\to_\mathcal{G} \subseteq (\Sigma \cup \mathcal{N})^* \times (\Sigma \cup \mathcal{N})^*$ which is defined as $\beta\, A\, \gamma \to_\mathcal{G} \beta\, \delta\, \gamma$ for all words $\beta, \gamma \in (\Sigma \cup \mathcal{N})^*$ and for all rules $A \to \delta$. There are two sub-relations of $\to_\mathcal{G}$ the *leftmost* derivation relation $\to_{\mathcal{G};L}$ and the *rightmost* derivation relation $\to_{\mathcal{G};R}$ which are defined by $A\,\gamma \to_{\mathcal{G};L} \delta\,\gamma$ and $\gamma\, A \to_{\mathcal{G};R} \gamma\,\delta$ for all words $\gamma \in (\Sigma \cup \mathcal{N})^*$ and rules $A \to \delta$. The *language* of a non-terminal N induced by $\mathcal{G}$, denoted by $\mathcal{L}_\mathcal{G}(N)$, is defined as the words over Σ derivable from N: $\mathcal{L}_\mathcal{G}(N) = \{w \in \Sigma : N \to_\mathcal{G}^* w\}$. It is well known that any word in $\mathcal{L}_\mathcal{G}(N)$ may also be derived by a leftmost or rightmost derivation. The choice of derivation relation is thus immaterial when words in $\mathcal{L}_\mathcal{G}(N)$ are considered.

Let $k \in \mathbb{N}$. A more restrictive subrelation of $\to_\mathcal{G}$ is the *k-index derivation relation* $\to_{\mathcal{G};k}$ which restricts the number of non-terminals appearing in a derivation step to be at most k, i.e. $\gamma \to_{\mathcal{G};k} \gamma'$ just if $\gamma \to_\mathcal{G} \gamma'$ and in both γ and γ' there are at most k occurrences of non-terminals. Surprisingly, there exists a k such that any word w in $\mathcal{L}_\mathcal{G}(N)$ may be derived using a k-index derivation provided we regard two words as equivalent if they can be obtained from each other by a permutation of letters [Esp+11]. We elaborate on this result in the next section.

Commutativity and Parikh's Theorem

One way to disregard the order of a sequence $\boldsymbol{u} \in U^*$ is to consider $\boldsymbol{u}$'s *Parikh image*. The *Parikh image* of $\boldsymbol{u}$ to is defined to be the multiset

$$\mathbb{M}_U(\boldsymbol{u}) := [u \mapsto |\{i \mid \boldsymbol{u}(i) = u\}|]\,;$$

we drop the subscript and write $\mathbb{M}(\boldsymbol{u})$ whenever it is clear from the context. Further, we extend $\mathbb{M}(-)$ to subsets $V \subseteq U$ in the obvious way: $\mathbb{M}(V) := \{\mathbb{M}(\boldsymbol{v}) \mid \boldsymbol{v} \in V\}$.

Parikh's Theorem relates the Parikh images of the language generated by a CFG to the language generated by a subclass of context-free grammars, *regular grammars*:

Definition 10. A *regular grammar* is a CFG $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{R})$ such that all rules in $\mathcal{R}$ are of the form: (i) $A \to a$, (ii) $A \to a\,B$ where $A, B \in \mathcal{N}$ and $a \in \Sigma \cup \{\epsilon\}$.

Regular grammars are equivalent in expressivity to finite automata, and finite-state labelled transitions systems, and are powerful enough to generate context-free languages up to equivalence of Parikh images.

Theorem 1 (Parikh's Theorem [Par66]). *For all context-free grammars $\mathcal{G}$ and non-terminals N there exists a regular grammar $\mathcal{G}_r$ and non-terminal N_r such that the* Parikh image *of the languages they generate coincide, i.e.*

$$\mathbb{M}(\mathcal{L}_{\mathcal{G}}(N)) = \mathbb{M}(\mathcal{L}_{\mathcal{G}_r}(N_r)).$$

Independence Relation and Commutativity. Another common technique to formalise a sensitivity to order is the use of an *independence relation*. An *independence relation* I over a set U is a symmetric irreflexive relation over U. It induces a congruence relation $\simeq_I$ on U^* defined as the least equivalence relation R containing I and satisfying: $(\mu, \mu') \in R \implies \forall \nu_0, \nu_1 \in U^* : (\nu_0\,\mu\,\nu_1, \nu_0\,\mu'\,\nu_1) \in R$. We obtain the *full commutativity* congruence relation $\simeq_C$ by defining the independence relation C to be $C = U \times U \setminus \{(u, u) : U\}$.

We may thus investigate the words derivable by a CFG $\mathcal{G}$ modulo $\simeq_C$, i.e. we identify words that are equivalent under $\simeq_C$ which are thus elements of the quotient ${}^{\Sigma^*}/_{\simeq_C}$. A context-free (regular) grammar endowed with $\simeq_C$ is referred to as a *commutative context-free grammar* (CCFG) (commutative regular grammar respectively). In this context it is possible to reformulate Parikh's Theorem:

Theorem 2. *For all commutative context-free grammars $\mathcal{G}$ and non-terminals N there exists a commutative regular grammar $\mathcal{G}_r$ and non-terminal N_r such that*

$$\mathcal{L}_{\mathcal{G}}(N) = \mathcal{L}_{\mathcal{G}_r}(N_r).$$

Recent work by Esparza et al. has shown that for every commutative context-free grammar $\mathcal{G}$ and non-terminal N there exists a $k \geq 1$, polynomial in the size of $\mathcal{G}$, such that the language $\mathcal{L}_{\mathcal{G}}(N)$ can be generated by derivations of index k [Esp+11]. This result has recently been employed [GM12] to show that a connection between CCFG and Petri nets is tighter than it was previously known [Esp95].

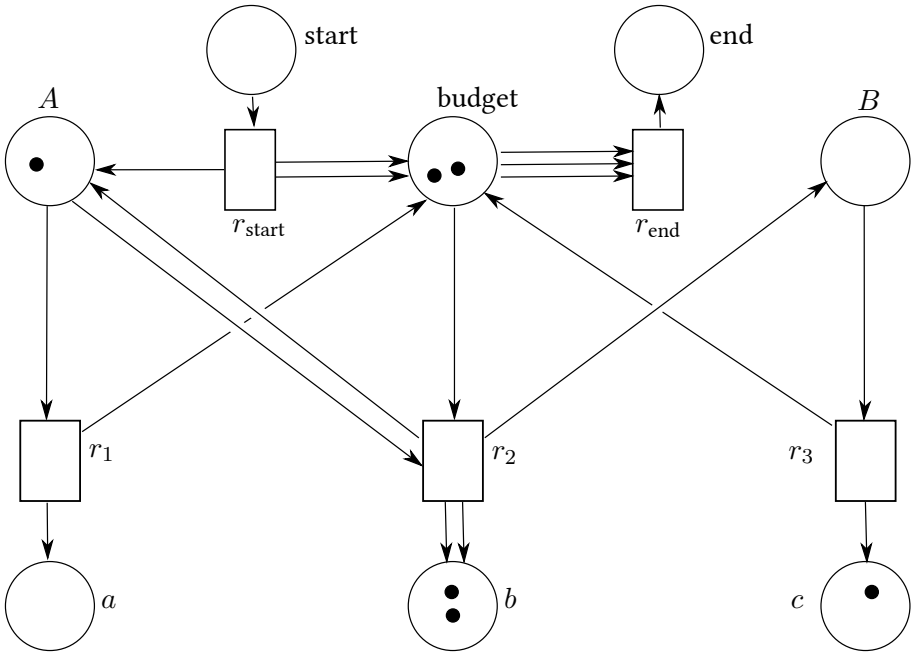


Figure 2.6: The CCFG widget *à la* Ganty and Majumdar of the CCFG $\mathcal{G}$. This example is taken from [Esp95] and modified to present Ganty and Majumdar's encoding.

A Connection between Petri Nets and Commutative Context-Free Grammars.

In [Esp95] Esparza showed that for every commutative context-free grammar $\mathcal{G}$ there exists a *communication-free* Petri net $\mathcal{V}$ such that a membership query on $w \in \mathcal{L}_{\mathcal{G}}(N)$ may be answered by a reachability query on $\mathcal{V}$. Esparza's Petri net encoding has recently been augmented by Ganty and Majumdar [GM12] using the fact that any word generated by $\mathcal{G}$ can be generated by a k-index derivation [Esp+11]. Ganty and Majumdar added a budget place that makes sure that only k-derivations are simulated. Their encoding allows a membership query $w \in \mathcal{L}_{\mathcal{G}}(N)$ to be tested by a Petri net transition. A consequence of this is that a CCFG derivation can be performed by a Petri net *widget*, i.e. a sub Petri net, which can interact with the rest of the net. We will present Ganty and Majumdar's CCFG in an example:

Example 4. In Figure 2.6 we present the Petri net encoding of the CCFG $\mathcal{G}$:

$$r_1 : A \to a, \quad r_2 : A \to b\,A\,b\,B, \quad r_3 : B \to c$$

In Esparza's encoding of a CCFG into a Petri net there is a place for every non-terminal N and for every terminal a. In the case of $\mathcal{G}$, we have non-terminals A and B and terminals a, b, and c, and there is a place for each of them. Configurations of the Petri net maintain the *partial word invariant*: a configuration represents the Parikh image of a partial word along a derivation of $\mathcal{G}$, i.e. the number of tokens in (non-) terminal places correspond to the number of occurrences of (non-)terminals in the intermediate word. In Esparza's encoding, rule r_2 leads to the Petri net transition in Figure 2.6 labelled with r_2 provided we remove the arcs from and to the budget

place, i.e the transition removes one token from place A and adds a token to place A and B each, and two tokens to place b. This maintains the partial word invariant. Ganty and Majumdar augment Esparza's encoding by adding a budget place to enforce k-index derivations. For $\mathcal{G}$, $k = 3$ is enough to derive all words in $\mathcal{L}_{\mathcal{G}}(A)$. In general, we can take $k = m|\mathcal{N}| + 1$ where $|\mathcal{N}|$ is the number of non-terminals and $m + 1$ is the maximal number of non-terminals on the RHS of a CCFG rule [Esp+11]. In Ganty and Majumdar's encoding the following additional invariant is maintained: the sum of tokens in non-terminal places and the budget place is always k, the *index invariant*. We note that in the Petri net in Figure 2.6, the arcs from and to the budget place for rules r_1, r_2, and r_3 ensure the index invariant is maintained. For example, rule r_2 rewrites one non-terminal to two non-terminals. Hence, rule r_2 consumes one token from the budget. This ensures that the budget is at least 1 and the index of the current partial word is at most $k-1$. The index invariant allows us to use this Petri net encoding as a widget: adding two places *start* and *end*, and two rules r_{start} and r_{end}. A CCFG computation can be started by placing a token into the place *start*. The rule r_{start} then places a token into place A and $k - 1$ tokens into the budget place. This initialises both invariants. The rule r_{end} requires that there are k tokens in the budget place. Hence the index invariant ensures that if r_{end} is enabled then all non-terminal places are empty and thus the Parikh image of a generated word has been added to the terminal places. Firing rule r_{end} then places a token into the place *end* and removes all tokens from the budget and non-terminal places. This signals to the rest of the net that the CCFG computation has terminated and that the widget may be used again by placing a token into *start*. We note that the terminal places do not necessarily have to be empty before the CCFG widget is started. However, after firing r_{end} it is guaranteed that the Parikh image has been added to the terminal places in addition to their previous contents.

We will now briefly formalise Ganty and Majumdar's CCFG widget. Suppose we have a CCFG $\mathcal{G}$ in Chomsky normal form, Ganty and Majumdar's CCFG widget $\mathcal{W}$ can then be obtained by applying the following mapping on rules:

$$\begin{aligned}
A \to B\,C \quad &\mapsto \quad ([A \mapsto 1, \text{budget} \mapsto 1],[B \mapsto 1, C \mapsto 1]) \\
A \to a \quad &\mapsto \quad ([A \mapsto 1,[a \mapsto 1, \text{budget} \mapsto 1]]) \\
A \to \epsilon \quad &\mapsto \quad ([A \mapsto 1,[\text{budget} \mapsto 1]])
\end{aligned}$$

together with the two rules $r_{\text{start}} = ([\text{start} \mapsto 1], [A_0 \mapsto 1, \text{budget} \mapsto |\mathcal{N}|])$ and $r_{\text{end}} = ([\text{budget} \mapsto |\mathcal{N}| + 1], [\text{end} \mapsto |\mathcal{N}|])$ where $|\mathcal{N}|$ is the number of $\mathcal{G}$'s non-terminals. Note that this sets the choice $k = |\mathcal{N}| + 1$.

Ganty and Majumdar's CCFG widget then guarantees the following:

Theorem 3 ([GM12]). *Let $\mathcal{G}$ be a CFG, A_0 be a non-terminal of $\mathcal{G}$, and $\mathcal{W}$ be Ganty and Majumdar's widget derived from $\mathcal{G}$. If s is a configuration of a Petri net containing $\mathcal{W}$, then*

$$s \oplus [\textit{start} \mapsto 1] \to_{\mathcal{W}}^{*} s \oplus [\textit{end} \mapsto 1] \oplus s'$$

if, and only if, $s' = \mathbb{M}(w)$ for some $w \in \mathcal{L}_{\mathcal{G}}(A_0)$.

This result has been exploited to show that various verification problems of *asynchronous programs with atomic procedure calls* [SV06] are interreducible with decision problems on Petri nets

and therefore e.g. safety verification is EXPSPACE-complete [GM12]. Asynchronous programs can be seen as a family of CFGs $(\mathcal{G}_p)_p$ where each CFG $\mathcal{G}_p$ implements an atomic procedure p. The alphabet of the CFGs $\mathcal{G}_p$ are interpreted as actions that post new asynchronous calls into unordered buffers [GM12]. It is thus not difficult to believe that a posting of a procedure call can be seen as commutative. Hence, we can view $(\mathcal{G}_p)_p$ as a family of CCFGs which implement an asynchronous program. Such commutativity assumptions are clearly not always valid and we may want to consider CFGs augmented with a *partially commutative* independence relation.

Partially Commutative Context Free Grammars

A commutative context-free grammar is a special case of endowing a CFG with an arbitrary independence relation. In the general case this yields a *partially commutative context-free grammar*.

Definition 11. Let Σ be an alphabet of terminal symbols and $I \subseteq \Sigma \times \Sigma$ an independence relation over Σ. A *partially commutative context-free grammar* (PCCFG) is a quadruple $\mathcal{G} = (\Sigma, I, \mathcal{N}, \mathcal{R})$ such that $(\Sigma, \mathcal{N}, \mathcal{R})$ is a CFG.

The derivation relation $\to$ is defined over the quotient by $\simeq_I$, so the words generated are congruence classes induced by $\simeq_I$, i.e. the derivation relation $\to$ is defined as a binary relation over $(\Sigma \cup \mathcal{N})^*/_{\simeq_I}$ by as $\gamma\, X\, \gamma' \to \gamma'\, \delta\, \gamma'$ if $X \to \delta$ is a rule of $\mathcal{G}$.

We will use PCCFG in Chapter 4 as a formalism to describe a class of asynchronous pushdown systems. Our work leverages the connection between CCFG and Petri nets to exhibit a model of asynchronous pushdown systems that subsumes asynchronous programs. PCCFG was originally introduced by Czerwinski et al. as a study in process algebra. They investigated the decidability of bisimulation for BPC [CFL09], a class of processes described by PCCFG where the commutativity of sequential composition is constrained by an independence relation on nonterminals. BPC subsumes both BPA and BPP. Similarly to PA and BPP, processes are not allowed to synchronise.

Summary

In this chapter we have given a brief exposition of labelled, unlabelled, and pre-structured transition systems. We have also defined the decision problems reachability, boundedness, termination, and coverability (for PSTS) which are of central interest in verification. Further, we have introduced the hierarchy of fast-growing complexity classes $\mathbf{F}_\alpha$ alongside the standard complexity classes such as P, EXPTIME, EXPSPACE, and ELEMENTARY. We have given a short overview of models of computation for concurrency, in particular Petri nets and WSTS, and of recursive programs, i.e. pushdown systems and context-free grammars. Lastly, we have illustrated the connection between CCFG and Petri nets, and given the definition of partially commutative context-free grammars — a generalisation of CCFG.

Chapter 3

Actor Communicating Systems and an Abstract Interpretation of λACTOR

This chapter describes joint work with Emanuele D'Osualdo and Luke Ong, and has been published in [DKO12; DKO13]. The definition of the concrete and abstract operational semantics (Figures 3.4, 3.5, 3.6, and 3.7), and Figure 3.8 are due to Emanuele D'Osualdo as well as the implementation of the ACS simplification procedure, Soter's UI, and the interface to BFC. The contributions of the author to this work are the exploration algorithm of the abstract state space, ***WORKLISTEXPLORE****, and its implementation (Section 3.6) in addition to the proofs of soundness (Theorem 4 and 6).*

In this chapter we describe an approach to verify λACTOR and Erlang programs. Given a λACTOR program $\mathcal{P}$, our goal is to provide a verification procedure that analyses, for example, a coverability query on the transition system generated by $\mathcal{P}$. Since λACTOR is a Turing powerful formalism, we know that this is impossible in general. Thus, we opt for a conservative approach, i.e. we develop a verification procedure that is sound but may result in an inconclusive analysis. Given a λACTOR program $\mathcal{P}$, we compute an abstract model $\mathcal{A}_\mathcal{P}$ that simulates the transition system generated by $\mathcal{P}$ and that supports algorithmic verification. We introduce *actor communicating systems* (ACS) as a suitable abstract computational model (Section 3.1). ACS can model a dynamic network of finite state processes that communicate asynchronously over unbounded, unordered channels. Thus, ACS are sufficiently expressive to model the key features of λACTOR and Erlang programs. Further, ACS are easily seen to be equivalent to Petri nets [Muk+98b]. Therefore, the decision results and tool support for Petri nets can be exploited in an implementation. To conservatively approximate the behaviour of a λACTOR program, we turn to the framework of *abstract interpretation* [CC77]. Obtaining an abstract interpretation of λACTOR is by no means trivial. For this purpose, we turn to a systematic methodology that derives a sound abstract interpretation: *abstracting abstract machines* (AAM) [Mig10; HM11; HM12; Joh+13] (Section 3.2). We apply the AAM methodology in the novel setting of message passing concurrency and give λACTOR a concrete operational semantics in terms of a *control*, *environment*, *store*, and *(store allocated) continuation* (CESK*) machine that is adapted to fit λACTOR's natural semantics. Conforming to the AAM framework, the machine's state space is designed to break the cyclical dependency of domains. The concrete operational semantics generates a transition system $\mathcal{P} = (State, \rightarrow_\mathcal{P})$ that serves as the formal semantics for λACTOR (Section 3.3). We then present an abstract interpret-

ation of this machine semantics that is parametric in the choice of several basic domains. This abstract interpretation yields a finite state machine that may be used to compute a range of static properties. The finite state machine generates an abstract transition system $\hat{\mathcal{P}} = (\widehat{State}, \rightarrow_{\hat{\mathcal{P}}})$ which simulates $\mathcal{P}$. The accuracy of this simulation is dependent on the choice the basic domains (Section 3.4). We present a common instantiation of the basic domains based on *contours* [Shi91]. Our verification procedure explores the abstract state space of $\hat{\mathcal{P}}$ induced by this choice to generate an ACS $\mathcal{A}_{\mathcal{P}}$ that simulates the transition system $\mathcal{P}$ (Section 3.5). The ACS $\mathcal{A}_{\mathcal{P}}$ can then be used to conservatively answer verification questions on the input program $\mathcal{P}$.

With Emanuele D'Osualdo, we have implemented this verification procedure as the verification tool Soter. Soter is the first verification tool for Erlang that employs infinite-state verification techniques (see Section 3.7 for related work). We give an overview of Soter, present its worklist exploration algorithm to explore the abstract state space (Section 3.6), and evaluate it on a set of benchmarks (Section 3.8). Our experiments show that infinite-state verification techniques can be successfully applied to Erlang programs and that the worklist algorithm provides significant speedups over a baseline. We present the use of Soter on a worked example and explain the information that Soter computes (Section 3.6). Further, we discuss the theoretical limitations of ACS and the verification procedure (Section 3.8). In the next section we introduce ACS, clarify their relationship with Petri nets, and present an example.

3.1 Actor Communicating Systems

Even though λActor is an idealised fragment of Erlang, the computational features that it provides are very rich. Thus, finding an abstract model that captures λActor's most important features and supports algorithmic verification is not an easy task. Moreover, the abstract model is a pivotal choice in our approach to verify λActor and Erlang programs. As we have noted in Section 1.2, several fragments of λActor are already Turing powerful. The functional fragment with pattern-matching [OR11], the concurrent pushdown fragment with simple synchronisations [Ram00], and even systems with one finite-state process equipped with a FIFFO mailbox [Her+02], are already too expressive to allow algorithmic verification. Constrained by these decidability frontiers, we choose a model of concurrent computation that captures the interaction of an unbounded set of finite-state processes which communicate asynchronously over unbounded and *unordered* channels.

Definition 12. An *actor communicating system* (ACS) $\mathcal{A}$ is a quadruple $(\mathcal{Q}, \mathit{Chan}, \mathit{Msg}, \mathcal{R})$ composed of the finite sets of *control states* $\mathcal{Q}$, *channel names* Chan, *messages* Msg, and *rules* $\mathcal{R}$. The set of rules $\mathcal{R}$ is a finite subset of $\mathcal{Q} \times \Lambda \times \mathcal{Q}$ where

$$\Lambda := \{c\,?\,m, c\,!\,m, \nu q : c \in \mathit{Chan}, m \in \mathit{Msg}, q \in \mathcal{Q}\} \cup \{\epsilon\}$$

is the set of *communication side-effects*. We denote a rule $(q, \lambda, q') \in \mathcal{R}$ by $q \xrightarrow{\lambda} q'$.

An action λ of the form $c\,!\,m$ denotes the sending of the message m to channel c, $c\,?\,m$ denotes the retrieval of message m from channel c, and νq denotes the spawning of a new process that begins execution from q.

An ACS $\mathcal{A}$ gives rise to a transition system with state space $\mathbb{M}[\mathcal{Q}] \times (\mathit{Chan} \to \mathbb{M}[\mathit{Msg}])$. We write $q \parallel q' \lhd [m_a, m_b, m_b]^{c_1} \oplus [m_c]^{c_2}$ for a configuration $([q, q'], [c_1 \mapsto [m_a, m_b, m_b], c_2 \mapsto [m_c]])$ to simplify notation. We abbreviate a set of processes running in parallel as Π and a set of channels by Γ with names in *Chan*. The transition relation $\to_{\mathcal{A}}$ generated by the ACS $\mathcal{A}$ is defined as follows: suppose $q \xrightarrow{\lambda} q' \in \mathcal{R}$ then

$$q \parallel \Pi \lhd \Gamma \to_{\mathcal{A}} q' \parallel \Pi \lhd \Gamma \qquad \text{if } \lambda = \epsilon \tag{3.1}$$

$$q \parallel \Pi \lhd \Gamma \to_{\mathcal{A}} q' \parallel q_0 \parallel \Pi \lhd \Gamma \qquad \text{if } \lambda = \nu q_0. \tag{3.2}$$

$$q \parallel \Pi \lhd \Gamma \to_{\mathcal{A}} q' \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]] \qquad \text{if } \lambda = c\,!\,m \tag{3.3}$$

$$q \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]] \to_{\mathcal{A}} q' \parallel \Pi \lhd \Gamma \qquad \text{if } \lambda = c\,?\,m \tag{3.4}$$

In the following we illustrate the operational semantics of ACS using an example. We also take the opportunity to discuss the difference between unordered message passing and FIFFO.

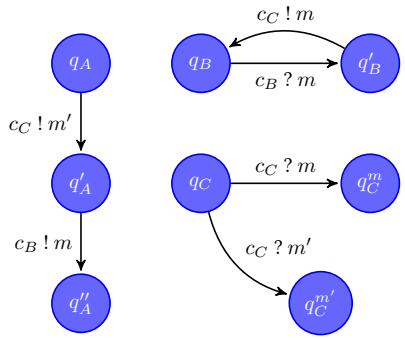
Example 5. Consider the ACS $\mathcal{A}$ displayed in Figure 3.1 which is inspired by Fredlund and Svensson's example [FS07] that has been the basis of Example 1. The rules of $\mathcal{A}$ are displayed as a graph, nodes are control states, directed edges represent rules. For example, the edge from q_B to q'_B labelled with $c_B\,?\,m$ represents rule $q_B \xrightarrow{c_B?m} q'_B$. The ACS $\mathcal{A}$ models three processes A, B, and C with dedicated channels c_B and c_C for B and C respectively. Control states with subscript X are intended to belong to process X where X ranges over A, B, and C. If we fix $q_A \parallel q_B \parallel q_C \lhd \emptyset$ as a starting state, we can see that A sends two messages m and m' which are intended for C. The message m' is sent directly to C, the message m is relayed to C via B. In contrast to Example 1, we stay faithful to Fredlund and Svensson's example here and send m' to C before m is sent to B. The right side of Figure 3.1 displays this informal description of $\mathcal{A}$ in a simple graph. Since channels are unordered and processes can proceed at will, there is a path in $\mathcal{A}$ where C receives message m from channel c_C instead of m', which is first sent to C:

$$\begin{aligned} q_A \parallel q_B \parallel q_C \lhd \emptyset &\to_{\mathcal{A}} q'_A \parallel q_B \parallel q_C \lhd \left[m'\right]^{c_C} \to_{\mathcal{A}} q''_A \parallel q_B \parallel q_C \lhd \left[m'\right]^{c_C} \oplus [m]^{c_B} \\ &\to_{\mathcal{A}} q''_A \parallel q'_B \parallel q_C \lhd \left[m'\right]^{c_C} \to_{\mathcal{A}} q''_A \parallel q_B \parallel q_C \lhd \left[m', m\right]^{c_C} \\ &\to_{\mathcal{A}} q''_A \parallel q_B \parallel q^m_C \lhd \left[m'\right]^{c_C} \end{aligned}$$

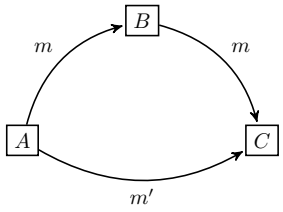
We would like to point out that this illustrates a major difference between ACS and λ_{Actor}. We can easily implement the processes A, B, and C as a λ_{Actor} program by, for example, slightly modifying the program in Example 1. Mailboxes in λ_{Actor} extract messages using a FIFFO discipline. Since m' is sent first to C the operational semantics of λ_{Actor} guarantees that C can only remove m' from its mailbox. Such a path can also be found for $\mathcal{A}$. Using the path above up to the penultimate configuration we obtain:

$$q_A \parallel q_B \parallel q_C \lhd \emptyset \to^*_{\mathcal{A}} q''_A \parallel q_B \parallel q_C \lhd \left[m', m\right]^{c_C} \to_{\mathcal{A}} q''_A \parallel q_B \parallel q^{m'}_C \lhd [m]^{c_C}.$$

Remark. It would appear as if the unordered nature of channels is a great loss of precision. Let us take the opportunity to make the case *for* using unordered message passing in the abstract model. λ_{Actor} is motivated by single-node Core Erlang [Fre01], i.e. Erlang code running on a single machine. The operational semantics of single-node Erlang gives the programmer very



(a) The rules of the ACS $\mathcal{A}$.



(b) A simplified graph of communication in $\mathcal{A}$.

Figure 3.1: From the configuration $q_A \parallel q_B \parallel q_C \lhd \emptyset$ the ACS $\mathcal{A}$ models the evolution of three processes: A, B, and C starting in q_A, q_B, and q_C respectively. A sends two messages m and m' to C where m is relayed through B. The ACS $\mathcal{A}$ is inspired by an example from [FS07].

strong guarantees on the arrival order of messages in the mailbox. However, these guarantees are greatly weakened when distributed Erlang is considered [CS05; SF07]. Consider again the simple graph description of $\mathcal{A}$ in Figure 3.1b. Suppose the processes A, B, and C are deployed on different physical machines and the physical channels along which the messages m and m' are sent have different speeds or delays. In these circumstances, it is conceivable that m travels faster from A to B and from B to C than it is possible for m' to arrive at C. The distributed Erlang semantics thus allows reorderings of processes' mailboxes and only gives the guarantee that *'message passing between a pair of processes is assumed to be ordered'* [Arm03]. Let us consider a process π executing the λACTOR expression $e = \textbf{receive}\ \mathrm{p}_1 \to \ldots;\ \ldots\ \mathrm{p}_n \to \ldots;\ \textbf{end}$. Provided that (i) the patterns p_1, ..., p_n have no free variables, (ii) π is receiving messages from processes $\pi_1, \ldots, \pi_k$, and (iii) the messages sent by each process π_i are guaranteed to match at most one pattern p_j, π's mailbox may be accurately modelled by an unordered mailbox. Thus, ordering guarantees in the distributed setting are weak. Further, Erlang advocates a *develop-locally-distribute-seamlessly* philosophy: *'programs can be developed on a single node and deployed on a cluster without major changes'* [Arm10]. Hence, if our verification procedure reports a false negative that is due only to the use of unordered channels, it points to a possible over-reliance on ordering guarantees that may hinder seamless distribution.

ACS and VAS. It turns out that ACS and vector addition systems (VAS) are equivalent. Petri net models for finite state machines that communicate asynchronously via unordered channels were first investigated by Mukund et al. [Muk+98b; Muk+98a]. The translation from an ACS $\mathcal{A}$ to a VAS $\mathcal{V}$ is mainly straightforward: for every control-state q and every pair $(c, m) \in \mathit{Chan} \times \mathit{Msg}$ there is a place in $\mathcal{V}$. The rules of $\mathcal{V}$ are then obtained by the following mapping:

$$q \xrightarrow{\epsilon} q' \quad \mapsto [q \mapsto -1, q' \mapsto 1] \qquad q \xrightarrow{c?m} q' \ \mapsto [q \mapsto -1, q' \mapsto 1, (c,m) \mapsto -1]$$

$$q \xrightarrow{\nu q''} q' \ \mapsto [q \mapsto -1, q' \mapsto 1, q'' \mapsto 1] \qquad q \xrightarrow{c!m} q' \ \mapsto [q \mapsto -1, q' \mapsto 1, (c,m) \mapsto 1]$$

The VAS $\mathcal{V}$ thus uses a place q to count the number of processes in control-state q while the places (c, m) represent the number of occurrences of the m in channel c. This relation between configurations of $\mathcal{A}$ and $\mathcal{V}$ is easily seen to be a bisimulation.

Decision Problems for ACS. We can turn ACS into PSTS by adding an order on configurations: $\Pi \lhd \Gamma \leq_{ACS} \Pi' \lhd \Gamma'$ if $\Pi \leq_{\mathbb{M}[\mathcal{Q}]} \Pi'$ and $\Gamma(c) \leq_{\mathbb{M}[Msg]} \Gamma(c')$ for all $c \in Chan$. Hence reachability, coverability, boundedness, and termination are well-defined, decidable, and for the latter three, EXPSPACE-complete by translation to VAS. It is fortunate that for Petri nets and VAS there are a range of mature coverability tools available, such as BFC [KKW12], PETRUCHIO [MS10], and MIST [Gan+07].

In the next section, we review the methodology of *abstracting abstract machines* (AAM) which we use to obtain a sound approximation of a λ_{ACTOR} program in the form of an ACS.

3.2 Methodology: Abstracting Abstract Machines

In this section, we illustrate the design choices advocated by the AAM methodology [Mig10; HM11; HM12; Joh+13] by applying it to a sequential fragment of λ_{ACTOR}. An important choice in designing an abstract interpretation is the concrete semantics of a program. The AAM methodology advocates the use of a machine based, concrete operational semantics which is designed to facilitate a structural abstraction.

The natural rewriting semantics of λ_{ACTOR} is not suitable. It conflates the construction of algebraic data structures, evaluation contexts, and environments. Thus, structural abstractions are difficult to apply in a systematic manner. To make a first step, we present an operational semantics that makes environments and evaluation contexts explicit. Suppose we have a λ_{ACTOR} program $\mathcal{P}$, i.e. $\mathcal{P}$ is a closed expression, from the fragment of λ_{ACTOR} that corresponds to the λ-calculus. Hence, $\mathcal{P}$ is taken from λ_{ACTOR}'s sequential fragment and contains only unary function abstractions and applications. The operational semantics generates a transition system $(\mathit{State}, \rightarrow)$. The state space *State* is a set of triples of *program locations* of $\mathcal{P}$, *environments*, and *continuations* i.e. $\mathit{State} = \mathit{ProgLoc} \times \mathit{Env} \times \mathit{Kont}$. Each subexpression e of $\mathcal{P}$ is a program location and is labelled by a unique label ℓ. We denote this by $\ell : e$ and define the set of program locations as $\mathit{ProgLoc} = \{\ell : e \mid e \text{ subexpression of } \mathcal{P}\}$; we drop ℓ if no confusion arises. Environments are partial mappings from variables to *closures*, i.e. $\mathit{Env} = \mathit{Var} \rightharpoonup \mathit{Closure}$, where a *closure* is a pair comprising a program location and an environment, i.e. $\mathit{Closure} = \mathit{ProgLoc} \times \mathit{Env}$. The set of *continuations Kont* is defined as the smallest set satisfying the following set equality:

$$\mathit{Kont} = \{\bullet\} + \mathit{ProgLoc} \times \mathit{Env} \times \mathit{Kont} + \mathit{ProgLoc} \times \mathit{Closure} \times \mathit{Env} \times \mathit{Kont}$$

We define the natural embeddings $\text{Stop}(-) : \{\bullet\} \rightarrow \mathit{Kont}$, $\text{Arg}_0\langle - \rangle : \mathit{ProgLoc} \times \mathit{Env} \times \mathit{Kont} \rightarrow \mathit{Kont}$, and $\text{Arg}_1\langle - \rangle : \mathit{ProgLoc} \times \mathit{Closure} \times \mathit{Env} \times \mathit{Kont} \rightarrow \mathit{Kont}$. We usually drop the argument of Stop for brevity. Continuations represent evaluation contexts. Using the three embeddings we can illustrate the connection:

- Stop represents the empty context $[-]$.
- $\text{Arg}_0\langle \ell, \rho, \kappa \rangle$ represents an evaluation context $E[[-](e_1')]$ which is related to ℓ, ρ, and κ as follows. Suppose $e_0(e_1)$ is the subexpression of $\mathcal{P}$ labelled ℓ, i.e. $\ell : e_0(e_1)$, then ρ specifies

the environment that closes the expression e_1 to e_1', i.e. $\rho(e_1) = e_1'$, and κ is the continuation representing the enclosing evaluation context E.
• $\mathsf{Arg}_1\langle \ell, (v_0, \rho_0), \rho, \kappa\rangle$ represents the context $E[v_0'([-])]$ where $\ell : e_0(e_1)$, ρ is an environment that originally closed the term $e_0(e_1)$, ρ_0 closes v_0 to the term v_0', and κ is the continuation representing the enclosing evaluation context E.

Note that there is an equivalent inductive definition of *Kont* generated by the above three constructs. The small step operational semantics breaks down functional reductions into four rules:

$$\begin{aligned}
(\ell\colon e_0(e_1), \rho, \kappa) &\to (e_0, \rho, \mathsf{Arg}_0\langle \ell, \rho, \kappa\rangle),\\
(v, \rho, \mathsf{Arg}_0\langle \ell, \rho', \kappa\rangle) &\to (e_1, \rho', \mathsf{Arg}_1\langle \ell, (v, \rho), \rho', \kappa\rangle) \text{ if } \ell\colon e_0(e_1),\\
(v, \rho, \mathsf{Arg}_1\langle \ell, (\mathbf{fun}(x) \to e, \rho_0), \rho', \kappa\rangle) &\to (e, \rho'[x \mapsto (v, \rho)], \kappa),\\
(x, \rho, \kappa) &\to (v, \rho', \kappa) \text{ where } \rho(x) = (v, \rho').
\end{aligned}$$

The first rule starts the evaluation of an application according to a leftmost and strict evaluation strategy. First e_0 is evaluated while the continuation $\mathsf{Arg}_0\langle \ell, \rho, \kappa\rangle$ stores the new evaluation context. The second rule fires when the functional part e_0 of an application $e_0(e_1)$ has been evaluated to v. The rule initiates the evaluation of e_1 and stores the appropriate continuation. Note that we use the meta variable v for an irreducible expression such as a constructor expression or functional abstraction in *head-normal-form*. The third rule is enabled when an application $e_0(e_1)$ has been evaluated to $(\mathbf{fun}(x) \to e)(v)$. After performing the substitution, which is realised by updating the original environment ρ' to $\rho'[x \mapsto (v, \rho)]$, the evaluation progresses to the function body e. The last rule specifies that if we meet a variable during the evaluation we look up its binding in the environment.

Example 6. Let us illustrate the small step operational semantics on the expression e where $e = \ell_0 : (\mathbf{fun}(x) \to x)\, y$ with the environment $\rho = [y \mapsto (v, \emptyset), y' \mapsto (v', \emptyset)]$. The evaluation produces the following run:

$$\begin{aligned}
(e, \rho, \mathsf{Stop}) &\to (\mathbf{fun}(x) \to x, \rho, \mathsf{Arg}_0\langle \ell_0, \rho, \mathsf{Stop}\rangle)\\
&\to (y, \rho, \mathsf{Arg}_1\langle \ell_0, (\mathbf{fun}(x) \to x, \rho), \rho, \mathsf{Stop}\rangle) && \rho(y) = (v, \emptyset)\\
&\to (v, \emptyset, \mathsf{Arg}_1\langle \ell_0, (\mathbf{fun}(x) \to x, \rho), \rho, \mathsf{Stop}\rangle)\\
&\to (x, \rho[x \mapsto (v, \emptyset)], \mathsf{Stop}) && \rho[x \mapsto (v, \emptyset)](x) = (v, \emptyset)\\
&\to (v, \emptyset, \mathsf{Stop}).
\end{aligned}$$

The operational semantics now separates the construction of algebraic data and evaluation contexts from the program. Any expression found in any part of the state space is guaranteed to be a subexpression of $\mathcal{P}$. However, arbitrary chains of continuations and nested closures can be constructed.

Recursive State Space. An interesting view is offered by Might [Mig10]: there is recursion in the definition of *State*. This recursion manifests itself in cycles in the dependency graph of the state space. The dependency graph for the sets involved in the definition of *State* is displayed in Figure 3.2. There are two cycles: a self-cycle at *Kont* and the two node cycle along *Env* and

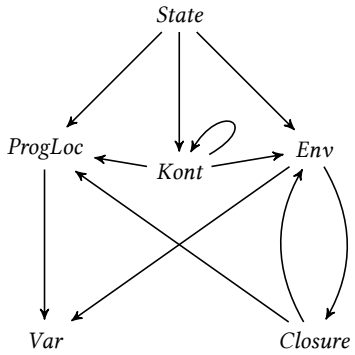


Figure 3.2: The dependency graph of domains.

Closure. The ultimate goal is to design an *abstract interpretation* that is both *sound* and *finite*. A cycle in the dependency graph complicates a structural abstraction. To find a finite set to represent abstract continuations, we need a finite set to represent abstract continuations! The AAM methodology offers a solution for this dilemma. Cycles can be broken by introducing an indirection, i.e. recursive structures can be represented with the help of *pointers*. For example, instead of representing environments as partial mappings from variables to closures, we can define an environment to map a variable to an *address*. Such a value address may then be looked up in a *store* which is a mapping from addresses to closures. Using such indirections, the dependency graph turns into a *directed acyclic graph*. This enables a structural abstraction. We only need to abstract the leaves of the dependency graph and lift the abstraction along the dependency tree. In order to restructure the state space, we have to revisit the definition of *State*, introduce the appropriate redirections, and amend the operational semantics.

Restructuring the Operational Semantics. We need to break two cycles in the dependency graph: one at *Kont* and the other at *Env* and *Closure*. We break both cycles by introducing *value pointers* and *continuation pointers*. We augment the definition of *State* with two additional sets for *value addresses VAddr* and *continuation addresses KAddr*. *Stores* will be used to resolve both value and continuation pointers:

$$Store = \{\sigma_V + \sigma_K \mid \sigma_V : VAddr \rightharpoonup Closure, \sigma_K : KAddr \rightharpoonup Kont\}.$$

Environments are now mappings from variables to value addresses i.e. $Env = Var \rightharpoonup VAddr$, and the set of continuations can now be defined without recursion:

$$Kont = \{\bullet\} + ProgLoc \times Env \times KAddr + ProgLoc \times Closure \times Env \times KAddr.$$

We continue to use the natural embeddings Stop, $\mathrm{Arg}_0\langle - \rangle$, and $\mathrm{Arg}_1\langle - \rangle$ except that continuation addresses replace actual continuations. From the λ_{ACTOR} program $\mathcal{P}$, the restructured operational semantics generates a transition system $(State, \rightarrow_{\mathcal{P}})$ to which we will also refer to as $\mathcal{P}$. The state space is $State = ProgLoc \times Env \times Store \times KAddr$. The rules of the restructured operations semantics are similar to the original version except that it incorporates the indirection through the store.

The following rules define a small step $\varsigma \to_{\mathcal{P}} \varsigma'$ where we will use ς and ς' to denote the pre and post states:

$$
\begin{array}{rl}
(\ell : e_0(e_1), \rho, \sigma, a) \to_{\mathcal{P}} (e_0, \rho, \sigma[a' \mapsto \kappa], a') & \text{where } \kappa = \mathsf{Arg}_0\langle \ell, \rho, a\rangle, a' = \mathrm{new}_{ka}(\kappa, \varsigma), \\
(v, \rho, \sigma, a) \to_{\mathcal{P}} (e_1, \rho', \sigma[a' \mapsto \kappa], a') & \text{if } \ell\colon e_0(e_1), \sigma(a) = \mathsf{Arg}_0\langle \ell, \rho', c\rangle, \text{ and} \\
 & \kappa = \mathsf{Arg}_1\langle \ell, (v, \rho), \rho', c\rangle, a' = \mathrm{new}_{ka}(\kappa, \varsigma) \\
(v, \rho, \sigma, a) \to_{\mathcal{P}} (e, \rho'[x \mapsto b], \sigma[b \mapsto (v, \rho)], a') & \text{if } \sigma(a) = \mathsf{Arg}_1\langle \ell, (\textbf{fun}(x) \to e, \rho_0), \rho', a'\rangle, \\
 & \text{and } b = \mathrm{new}_{\mathrm{va}}(x, \varsigma), \\
(x, \rho, \sigma, a) \to_{\mathcal{P}} (v, \rho', \sigma, a) & \text{if } \sigma(\rho(x)) = (v, \rho').
\end{array}
$$

In the restructured operational semantics, the first rule has been amended to create a new address a' for the new continuation $\kappa = \mathsf{Arg}_0\langle \ell, \rho, a\rangle$. The address a' is created using the helper function $\mathrm{new}_{ka} : \mathit{Kont} \times \mathit{State} \to \mathit{KAddr}$ which may take into account κ and the entire pre state ς to generate a new continuation address. The store σ is then updated to $\sigma[a' \mapsto \kappa]$ which maps a' to κ. The second rule is similarly amended to manage the creation of a new continuation. The third rule is analogously changed to use variable addresses for creating the bindings in the environment. Here the helper function $\mathrm{new}_{\mathrm{va}} : \mathit{Var} \times \mathit{State} \to \mathit{VAddr}$ may use the variable and the pre state ς to generate a new variable address b. The environment ρ' is then updated to bind x to b and the store σ is modified to map b to (v, ρ). The last rule remains essentially the same except that the variable lookup of x now goes through the store with $\sigma(\rho(x))$.

Example 7. We demonstrate the restructured operational semantics on the same expression e of Example 6. However, the environment now maps to addresses, i.e $\rho = [y \mapsto a_1, y' \mapsto a_2]$, which are resolved by a store $\sigma = [a_1 \mapsto (v, \emptyset), a_2 \mapsto (v', \emptyset), * \mapsto \mathsf{Stop}]$.

$$
\begin{array}{rll}
(e, \rho, \sigma, *) & \to_e (\textbf{fun}(x) \to x, \rho, \sigma[b_1 \mapsto \mathsf{Arg}_0\langle \ell_0, \rho, *\rangle], b_1) & \\
 & \to_e (y, \rho, \sigma_1[b_2 \mapsto \mathsf{Arg}_1\langle \ell_0, (\textbf{fun}(x) \to x, \rho), \rho, *\rangle, b_2]) & \sigma_1(\rho(y)) = (v, \emptyset) \\
 & \to_e (v, \emptyset, \sigma_2, b_2) \to_e (x, \rho[x \mapsto a_3], \sigma_2[a_3 \mapsto (v, \emptyset)], *) & \sigma_3(a_3) = (v, \emptyset) \\
 & \to_e (v, \emptyset, \sigma_3, *), &
\end{array}
$$

where the intermediate stores σ_1, σ_2, and σ_3 are $\sigma_1 = \sigma[b_1 \mapsto \mathsf{Arg}_0\langle \ell_0, \rho, *\rangle]$,

$$
\sigma_2 = \sigma_1[b_2 \mapsto \mathsf{Arg}_1\langle \ell_0, (\textbf{fun}(x) \to x, \rho), \rho, *\rangle], \qquad \sigma_3 = \sigma_2[a_3 \mapsto (v, \emptyset)].
$$

This abstract machine semantics we have just described is called a *CESK* machine*, a variant of Felleisen and Friedman's CESK machine with store allocated continuations [HM12]. The CESK machine is an abstract machine that executes the λ-calculus with assignments using an eager evaluation strategy [FF87]. Fortunately, the (restructured) CESK* operational semantics is equivalent to the operational semantics we presented earlier [HM12].

As a result of restructuring the operational semantics, the dependency graph (see Figure 3.3) is cycle-free. We have obtained a directed acyclic graph. The leafs of the dependency graphs are *VAddr*, *KAddr*, and *Var* (and its superset *ProgLoc*). Hence, we have been successful in pushing the sources of an infinite state space into the leaves. If we can ensure that *KAddr* and *VAddr* are finite sets, then the finiteness of *State* is ensured. We want to achieve a sound over-approximation of the operational semantics. A common framework to obtain such a finite over-approximation is *abstract interpretation* [CC77].

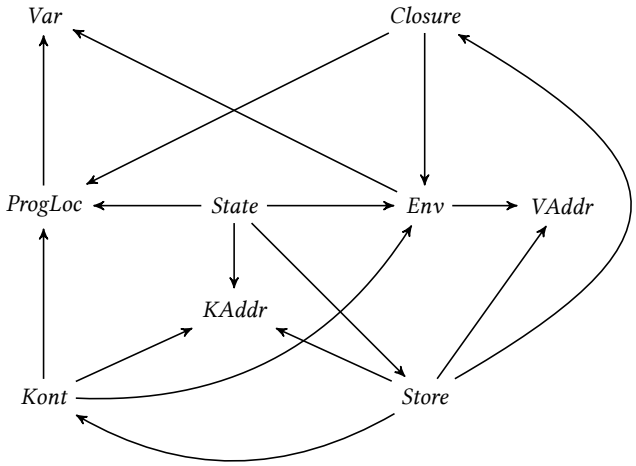


Figure 3.3: The dependency graph of the restructured state space.

Abstract Interpretation. An *analysis* on a transition system $\mathcal{S} = (S, \rightarrow_{\mathcal{S}})$ can be thought of as a pair consisting of a transition system $\hat{\mathcal{S}} = (\hat{S}, \rightarrow_{\hat{\mathcal{S}}})$ and a simulation relation $\mathcal{R}$ between $\mathcal{S}$ and $\hat{\mathcal{S}}$. Typically $\hat{\mathcal{S}}$ is a finite transition system or has good decision properties. The simulation relation $\mathcal{R}$ then relates states of $\mathcal{S}$ to states of $\hat{\mathcal{S}}$. The simulation relation gives us an *abstract* view on the *concrete* states of $\mathcal{S}$ which is maintained by the transition relation. In the AAM methodology, the existence of such a simulation relation is referred to as *soundness* of $\hat{\mathcal{S}}$ with respect to $\mathcal{S}$ and allows conservative, over-approximative deductions on $\mathcal{S}$ by inspecting $\hat{\mathcal{S}}$. The framework of *abstract interpretation* [CC77] provides an extensive study of analyses—on transition systems, denotational semantics, or other programming semantics—, the relationship between concrete and abstract objects, soundness, and methods for computing an abstraction. We focus our presentation of abstract interpretation on the setting of transition systems. Typically the transition systems $\mathcal{S}$ and $\hat{\mathcal{S}}$ are endowed with a complete lattice structure $(S, \leq_{\mathcal{S}}, \sqcup_{\mathcal{S}}, \sqcap_{\mathcal{S}}, \bot_{\mathcal{S}}, \top_{\mathcal{S}})$ and $(\hat{S}, \leq_{\hat{\mathcal{S}}}, \sqcup_{\hat{\mathcal{S}}}, \sqcap_{\hat{\mathcal{S}}}, \bot_{\hat{\mathcal{S}}}, \top_{\hat{\mathcal{S}}})$ to which we also refer to as $\mathcal{S}$ and $\hat{\mathcal{S}}$. Elements of $\mathcal{S}$ and $\hat{\mathcal{S}}$ are related using an *abstraction function* $\alpha : S \to \hat{S}$ and a *concretisation function* $\gamma : \hat{S} \to S$. A quadruple $(\mathcal{S}, \alpha, \gamma, \hat{\mathcal{S}})$ is said to be a *Galois connection* if (i) α and γ are monotone functions, (ii) $(\gamma \circ \alpha)(s) \geq_{\mathcal{S}} s$ for all $s \in S$, and (iii) $(\alpha \circ \gamma)(\hat{s}) \leq_{\hat{\mathcal{S}}} \hat{s}$ for all $\hat{s} \in \hat{S}$.

Representation and Extraction Functions. A common way to construct a Galois connection in the AAM methodology uses *representation* and *extraction* functions. Suppose V is a set and L is a complete lattice then a function $\beta : V \to L$ is said to be a representation function. A representation function β gives rise to a Galois connection $(\mathscr{P}[V], \alpha_\beta, \gamma_\beta, L)$ where $\alpha_\beta(U) = \bigsqcup\{\beta(v) \mid v \in U\}$ and $\gamma_\beta(l) = \{v \in V \mid \beta(v) \leq_L l\}$ [NNH99]. A special case arises when L is the powerset lattice over some set W, i.e. $L = (\mathscr{P}[W], \subseteq, \bigcup, \bigcap, \emptyset, W)$, and β is generated by a function $\eta : V \to W$ i.e. $\beta(v) = \{\eta(v)\}$. In this case, η is said to be an *extraction function* and we write $\beta = \beta_\eta$ to emphasise that β is induced by η. The Galois connection induced by β_η can then be rewritten as $(\mathscr{P}[V], \alpha_\eta, \gamma_\eta, \mathscr{P}[W])$ where

$$\alpha_\eta(U) = \{\eta(v) \mid v \in U\} \text{ and } \gamma_\eta(U') = \left\{w \mid \eta(w) \in U'\right\} \text{ [NNH99].}$$

Extraction functions are easily composed and thus are useful in the structural lifting of an abstraction. A common set of structural liftings proposed by the AAM methodology are the following:

- If $\eta : A \to B$ is an extraction function and $\eta' : A' \to B'$ is an extraction function then both $\eta \times \eta' : A \times A' \to B \times B'$ and $\eta + \eta' : A + A' \to B + B'$ are extraction functions.
- If $\eta : A \to B$ is an extraction function then $\eta' : (C \to A) \to (C \to B)$ defined by $\eta'(f) = \eta \circ f$ is an extraction function.
- If $\eta_0 : A \to B$ and $\eta_1 : C \to D$ are extraction functions then $\eta : (C \to A) \to (D \to \mathscr{P}[B])$ defined by $\eta(f) = \lambda \hat{x}. \bigcup \{\eta_0(f(x)) \mid \eta_1(x) = \hat{x}\}$ is an extraction function.

These properties of extraction functions are exploited in the AAM methodology to lift an extraction function for a leaf of the dependency graph to the entire state space.

Deriving an Analysis. The framework of abstract interpretation provides guidance in the design of an analysis. Given a concrete semantics and a Galois connection, the *optimal abstract semantics* can be derived. In our setting of transition systems, we can make this more concrete. Let $\mathcal{S} = (S, \to_{\mathcal{S}})$ be a transition system, such that $\mathcal{S}$ forms a complete lattice and $\to_{\mathcal{S}}$ is monotone wrt to $\leq_{\mathcal{S}}$, and let $(\mathcal{S}, \alpha, \gamma, \hat{\mathcal{S}})$ be a Galois connection. We can derive the optimal abstract transition system $\hat{\mathcal{S}} = (\hat{S}, \to_{\hat{\mathcal{S}}})$ where the transition relation $\to_{\hat{\mathcal{S}}}$ is defined through its relational inverse: $(\to_{\hat{\mathcal{S}}})^{-1} = \alpha \circ (\to_{\mathcal{S}})^{-1} \circ \gamma$ where α, γ are considered as relations. Equivalently:

$$\hat{s} \to_{\hat{\mathcal{S}}} \hat{s}' \iff \exists s, s'. s \to_{\mathcal{S}} s', \gamma(\hat{s}) = s, \alpha(s') = \hat{s}';$$

$\hat{\mathcal{S}}$ is guaranteed to be sound with respect to $\mathcal{S}$ since $\{(s, \hat{s}) \mid \alpha(s) \leq_{\hat{\mathcal{S}}} \hat{s}\}$ is a simulation for $\mathcal{S}$ and $\hat{\mathcal{S}}$ [CC79].

Calculating the Abstract Semantics. Equipped with the definition of extraction functions and the basic notions of abstract interpretation, we can now present how to calculate the abstract operational semantics from the restructured semantics. Let us first fix two finite sets $\widehat{\mathit{VAddr}}$, for abstract variable addresses, and $\widehat{\mathit{KAddr}}$, for abstract continuation addresses, and two extraction functions $\eta_{\mathit{VAddr}} : \mathit{VAddr} \to \widehat{\mathit{VAddr}}$ and $\eta_{\mathit{KAddr}} : \mathit{KAddr} \to \widehat{\mathit{KAddr}}$. In the following, we omit the subscripts for extraction functions and write simply η if no confusion arises. Following the dependency graph, we obtain extraction functions for abstract environments $\widehat{\mathit{Env}} = \mathit{Var} \rightharpoonup \widehat{\mathit{VAddr}}$, with $\eta(\rho) = \eta \circ \rho$, abstract closures $\widehat{\mathit{Closure}} = \mathit{Exp} \times \widehat{\mathit{Env}}$, with $\eta(t, \rho) = (t, \eta(\rho))$, and abstract continuations

$$\widehat{\mathit{Kont}} = \{\mathsf{Stop}\} + \mathit{ProgLoc} \times \widehat{\mathit{Env}} \times \widehat{\mathit{KAddr}} + \mathit{ProgLoc} \times \widehat{\mathit{Closure}} \times \widehat{\mathit{Env}} \times \widehat{\mathit{KAddr}}$$

where the extraction function η can be written using pattern-matching on the embeddings $\eta(\mathsf{Arg}_0\langle \ell, \rho, a\rangle) = \mathsf{Arg}_0\langle \ell, \eta(\rho), \eta(a)\rangle$, $\eta(\mathsf{Arg}_1\langle \ell, d, \rho, a\rangle) = \mathsf{Arg}_1\langle \ell, \eta(d), \eta(\rho), \eta(a)\rangle$, and $\eta(\mathsf{Stop}) = \mathsf{Stop}$. Note that we also write Stop, $\mathsf{Arg}_0\langle -\rangle$, and $\mathsf{Arg}_1\langle -\rangle$ for the embeddings for $\widehat{\mathit{Kont}}$. The abstract store $\widehat{\mathit{Store}}$ is obtained by

$$\widehat{\mathit{Store}} = \{\hat{\sigma}_V + \hat{\sigma}_K \mid \hat{\sigma}_V : \widehat{\mathit{VAddr}} \to \mathscr{P}[\widehat{\mathit{Closure}}], \hat{\sigma}_K : \widehat{\mathit{KAddr}} \to \mathscr{P}[\widehat{\mathit{Kont}}]\}$$

and the extraction function η_{Store} is derived to be the disjoint union of the two mappings:

$$\{\sigma_V \mapsto \lambda\hat{v}.\bigsqcup\{\eta_{Closure}(\sigma_V(v)) \mid \eta_{VAddr}(v) = \hat{v}\} \mid \sigma_V : VAddr \rightharpoonup Closure\},$$
$$\{\sigma_K \mapsto \lambda\hat{a}.\bigsqcup\{\eta_{Kont}(\sigma_K(a)) \mid \eta_{KAddr}(a) = \hat{a}\} \mid \sigma_K : KAddr \rightharpoonup Kont\}.$$

Lifting the derived sets to the abstract state space $\widehat{State}$ then yields $\widehat{State} = Exp \times \widehat{Env} \times \widehat{Store} \times \widehat{Kont}$ where lifting the extraction function from *State* to $\widehat{State}$ is achieved naturally by the definition $\eta(e, \rho, \sigma, \kappa) = (e, \eta(\rho), \eta(\sigma), \eta(\kappa))$. At this point we note that (i) η is an extraction function from *State* to $\widehat{State}$, and hence there is a Galois connection $(\mathscr{P}[State], \alpha_\eta, \gamma_\eta, \mathscr{P}[\widehat{State}])$, and (ii) $\widehat{Store}$ is a complete lattice. It is common in the AAM methodology to lift the complete lattice of $\widehat{Store}$ to the abstract state space $\widehat{State}$ by turning the sets *Exp*, $\widehat{Env}$, and $\widehat{Kont}$ into *flat lattices*. The complete flat lattice $U^\flat$ derived from the set U is $(U \cup \{\top, \bot\}, \leq_{U^\flat}, \sqcap_{U^\flat}, \sqcup_{U^\flat}, \bot, \top)$ where the order $\leq_{U^\flat}$ is essentially equality: $\bot \leq_{U^\flat} u$, $\top \geq_{U^\flat} u$, and $u \leq_{U^\flat} u'$ if $u = u'$; $u \sqcap_{U^\flat} u' = u$ if $u = u'$ and $\bot$ otherwise; and $u \sqcup_{U^\flat} u' = u$ if $u = u'$ and $\top$ otherwise. In this setting, η turns out to be a representation function from *State* to the complete lattice $\widehat{State}$. Hence, we obtain the Galois connection $(\mathscr{P}[State], \alpha, \gamma, \widehat{State})$ where α and γ are induced by the representation function η, i.e. $\alpha(S) = \bigsqcup\{\eta(s) \mid s \in S\}$ and $\gamma(\hat{s}) = \{s \mid \eta(s) \leq_{\widehat{State}} \hat{s}\}$.

Using the abstract interpretation framework, we can derive the optimal sound abstract operational semantics with state space $\widehat{State}$ from a transition system $\mathcal{S}'$ with state space $\mathscr{P}[State]$ and transition relation $\rightarrow_{\mathcal{S}'}$ that is monotone with respect to $\subseteq$. However, we started out with the transition system $\mathcal{P} = (State, \rightarrow_{\mathcal{P}})$. If we apply a powerset lifting to $\mathcal{P}$, we may still derive an optimal abstract interpretation.

An Optimal Abstract Interpretation via Powerset Lifting. First we need to choose a powerset lifting of $\mathcal{P}$ that ensures the lifted transition relation is monotone with respect to $\subseteq$. A natural choice is the co-universal powerset lifting $\mathscr{P}^{\mathsf{A}}[\mathcal{P}] = (\mathscr{P}[State], \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]})$ (Section 2.1). We note that if $S_0 \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]} S'$ and $S_0 \subseteq S_1$, then $S_1 \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]} S'$ and hence $\rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}$ is monotone with respect to $\subseteq$. Hence, we know that the optimal abstract transition system is $\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]} = (\widehat{State}, \rightarrow_{\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}})$ where $\rightarrow_{\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}}$ is defined by its relational inverse

$$\left(\rightarrow_{\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}}\right)^{-1} = \alpha \circ \left(\rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}\right)^{-1} \circ \gamma.$$

Further, we know that $\{(S, \hat{s}) \mid \alpha(S) \leq_{\widehat{State}} \hat{s}\}$ is a simulation relation between $\mathscr{P}^{\mathsf{A}}[\mathcal{P}]$ and $\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}$. Notice that this and $\alpha(\{s\}) = \eta_{State}(s)$ implies that $\{(s, \hat{s}) \mid \eta_{State}(s) \leq_{\widehat{State}} \hat{s}\}$ is a simulation relation between $\mathcal{P}$ and $\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}$. Thus, it is customary to write α_{State} for η_{State} in the AAM methodology while remembering that α_{State} is a representation function; we will follow this practice. To prove soundness for a particular choice of abstract semantics $\hat{\mathcal{P}}$, one may either exhibit a simulation between $\mathcal{P}$ and $\hat{\mathcal{P}}$ or $\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}$ and $\hat{\mathcal{P}}$. The former is often simpler while inspecting $\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}$ provides a valuable guide in defining $\hat{\mathcal{P}}$.

Deriving the Abstract Semantics. Especially for practical considerations, it is often inconvenient to implement $\rightarrow_{\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}}$. Instead, it is common to directly calculate the rules of the abstract

transition relation. The restructured operational semantics is defined by rules that are readily adapted using the representation function $\alpha_{\widehat{State}}$. This yields the abstract transition system $\hat{\mathcal{P}} = (\widehat{State}, \rightarrow_{\hat{\mathcal{P}}})$. The rules below define a small step $\hat{\varsigma} \rightarrow_{\hat{\mathcal{P}}} \hat{\varsigma}'$ where we use $\hat{\varsigma}$ and $\hat{\varsigma}'$ to denote the pre and post states. We note that store lookups of variable and continuation addresses are now non-deterministic and hence hard updates turn into joins.

$$
\begin{array}{rl}
(\ell : e_0(e_1), \hat{\rho}, \hat{\sigma}, \hat{a}) \rightarrow_{\hat{\mathcal{P}}} (e_0, \hat{\rho}, \hat{\sigma} \sqcup [\hat{a}' \mapsto \{\hat{\kappa}\}], \hat{a}') & \text{where } \hat{\kappa} = \mathsf{Arg}_0\langle \ell, \hat{\rho}, \hat{a}\rangle, \hat{a}' = \widehat{\mathsf{new}_{ka}}(\hat{\kappa}, \hat{\varsigma}), \\
(v, \hat{\rho}, \hat{\sigma}, \hat{a}) \rightarrow_{\hat{\mathcal{P}}} (e_1, \hat{\rho}', \hat{\sigma} \sqcup [\hat{a}' \mapsto \{\hat{\kappa}\}], \hat{a}') & \text{if } \ell : e_0(e_1), \mathsf{Arg}_0\langle \ell, \hat{\rho}', \hat{c}\rangle \in \hat{\sigma}(\hat{a}), \text{ and} \\
& \hat{\kappa} = \mathsf{Arg}_1\langle \ell, (v, \hat{\rho}), \hat{\rho}', \hat{c}\rangle, \hat{a}' = \widehat{\mathsf{new}_{ka}}(\hat{\kappa}, \hat{\varsigma}) \\
(v, \hat{\rho}, \hat{\sigma}, \hat{a}) \rightarrow_{\hat{\mathcal{P}}} \left(e, \hat{\rho}'\left[x \mapsto \hat{b}\right], \hat{\sigma} \sqcup \left[\hat{b} \mapsto \{(v, \hat{\rho})\}\right], \hat{a}'\right) & \text{if } \hat{b} = \widehat{\mathsf{new}_{\mathrm{va}}}(x, \hat{\varsigma}), \text{ and} \\
& \mathsf{Arg}_1\langle \ell, (\textbf{fun}(x) \rightarrow e, \hat{\rho}_0), \hat{\rho}', \hat{a}'\rangle \in \hat{\sigma}(\hat{a}), \\
(x, \hat{\rho}, \hat{\sigma}, \hat{a}) \rightarrow_{\hat{\mathcal{P}}} (v, \hat{\rho}', \hat{\sigma}, \hat{a}) & \text{if } (v, \hat{\rho}') \in \hat{\sigma}(\hat{\rho}(x))
\end{array}
$$

The first rule is essentially the same as in the concrete operational semantics. It operates on abstract elements indicated by $\hat{-}$. An abstract helper function $\widehat{\mathsf{new}_{ka}}$ is used to generate a new abstract continuation address $\hat{a}'$ for the new continuation $\hat{\kappa}$. The hard update of the store, $\sigma[a' \mapsto \kappa]$ in the concrete, is replaced by a join $\hat{\sigma} \sqcup [\hat{a}' \mapsto \hat{\kappa}]$. In the second rule, the binding of the abstract variable address $\hat{b}$, which is generated by the abstract helper function $\widehat{\mathsf{new}_{\mathrm{va}}}$, is also realised by a join $\hat{\sigma} \sqcup [\hat{b} \mapsto (v, \hat{\rho})]$. The lookup of the continuation in the second rule, $\sigma(a) = \mathsf{Arg}_0\langle \ell, \rho', c\rangle$ in the concrete, becomes non-deterministic: $\mathsf{Arg}_0\langle \ell, \hat{\rho}', \hat{c}\rangle \in \hat{\sigma}(\hat{a})$. This is also the case for the continuation lookup in the third rule and the variable lookup in the fourth: $(v, \hat{\rho}') \in \hat{\sigma}(\hat{\rho}(x))$.

Example 8. We continue with our running example and present a run of the abstract semantics. To illustrate approximations, we use only a single abstract value address $\hat{a}$. The abstract environment thus becomes $\hat{\rho} = [y \mapsto \hat{a}, y' \mapsto \hat{a}]$, which is resolved by the abstract store $\sigma = [\hat{a} \mapsto \{(v, \emptyset), (v', \emptyset)\}, * \mapsto \{\mathsf{Stop}\}]$. This gives rise to the run:

$$
\begin{array}{rll}
(e, \rho, \hat{\sigma}, *) \rightarrow_{\hat{e}} (\textbf{fun}(x) \rightarrow x, \rho, \hat{\sigma}_1, b_1) \rightarrow_{\hat{e}} (y, \rho, \hat{\sigma}_2, b_2) & & \sigma_1(\rho(y)) \ni (v', \emptyset) \\
\quad \rightarrow_{\hat{e}} (v', \emptyset, \hat{\sigma}_2, b_2) \rightarrow_{\hat{e}} (x, \rho[x \mapsto \hat{a}], \hat{\sigma}_2, *) & & \sigma_2(\rho[x \mapsto \hat{a}](x)) \ni (v', \emptyset) \\
\quad \rightarrow_{\hat{e}} (v', \emptyset, \sigma_2, *), & &
\end{array}
$$

where the intermediate stores $\hat{\sigma}_1$ and $\hat{\sigma}_2$ are

$$
\hat{\sigma}_1 = \hat{\sigma}[b_1 \mapsto \{\mathsf{Arg}_0\langle \ell_0, \rho, \mathsf{Stop}\rangle\}], \quad \hat{\sigma}_2 = \hat{\sigma}_1[b_2 \mapsto \{\mathsf{Arg}_1\langle \ell_0, (\textbf{fun}(x) \rightarrow x, \rho), \rho, *\rangle\}].
$$

We note that the lookup of $(v', \emptyset) \in \sigma_1(\rho(y))$ is spurious and that the store $\hat{\sigma}$ is already saturated at the address $\hat{a}$ from the start. This is a result of the address abstraction.

Defining the abstract operational semantics in this fashion requires an *a posteriori* soundness proof. In the AAM methodology, it is common to establish soundness by showing that the relation $\{(s, \hat{s}) \mid \alpha_{\widehat{State}}(s) \leq_{\widehat{State}} \hat{s}\}$ is a simulation between the concrete and abstract operational semantics. This proof commonly assumes that appropriate relationships between $\mathsf{new}_{\mathrm{va}}$, new_{ka} and $\widehat{\mathsf{new}_{\mathrm{va}}}$, $\widehat{\mathsf{new}_{ka}}$ respectively hold. The abstract operational semantics above then coincides with the optimal abstract interpretation $\widehat{\mathscr{P}^{\mathsf{A}}[\mathcal{P}]}$ provided appropriate choices for $\widehat{\mathsf{new}_{\mathrm{va}}}$, $\widehat{\mathsf{new}_{ka}}$ are made. Note that in general this may not be the case. It is possible to derive abstract counterparts of the auxiliary functions from their soundness constraints [MJ08]. They are thus correct by construction.

Choosing Concrete and Abstract Addresses. An important part of the AAM methodology is that initially the nature of the sets *KAddr*, *VAddr* and $\widehat{KAddr}$, $\widehat{VAddr}$, and the extraction functions η_{KAddr} and η_{VAddr} are left unspecified. They may thus be instantiated in different ways with mild assumptions on the relationships between new_{va}, new_{ka} and $\widehat{\text{new}_{\text{va}}}$, $\widehat{\text{new}_{ka}}$. If both $\widehat{KAddr}$ and $\widehat{VAddr}$ are chosen to be finite sets, the cycle-free dependency graph guarantees that the abstract operational semantics only requires a finite state space. It is customary in applications of AAM to instrument the concrete semantics further by, for example, a *time* component. This further instrumentation can then be used in the generation of variable addresses to allow more precise abstractions in the abstract operational semantics.

From Higher-Order Control-Flow Analysis to AAM. The methodology of AAM arises from the literature of *higher-order control-flow analysis* (CFA). In the presence of higher-order functions, traditional data flow analyses cannot be applied since no static control-flow graph is available. The purpose of a CFA is thus to statically compute an over-approximation of the control flow graph for which the data-flow to high-order variables needs to be determined. Shivers points out the inherent difficulty in this task: *'In order to do* [*data*] *flow analysis, we need a control flow graph. In order to determine control flow graphs, we need to do* [*data*] *flow analysis'* [Shi88]. Since the seminal works of Jones [Jon81] and Shivers [Shi88], CFA has generated a large body of research [Shi91; HJ94; JW95; MS06; MJ08; Mig10; MH11; VS10]. For an overview, we refer the reader to the excellent survey by Midtgaard [Mid12]. A family of analyses named k-CFA [Shi91] has been developed based on *contours*: a contour is a finite string of program locations representing the dynamic calling context of a variable. Phrasing k-CFA in the AAM framework requires a component *Time* that is instantiated to contours. Value addresses *VAddr* and continuation addresses *KAddr* are then set to be $\mathit{Var} \times \mathit{Time}$ and $\mathit{ProgLoc} \times \mathit{Time}$. The k-CFA abstraction then limits contours to be of length less than k by disregarding any length-exceeding suffix. This forms the most common instantiation in the AAM literature. The abstract machine representation of k-CFA has lead to non-trivial extensions increasing the precision or scope of analyses: Γ-CFA [MS06; MS08] introduces garbage collection into the machine semantics to improve precision of abstract stores, k-CFA for shared-state concurrency [MH11], and CFA2 [VS10] changes the abstraction to obtain a pushdown system.

In the following sections we expand on the abstract interpretation for the fragment of λ_{ACTOR} that we have developed. We show how to use a 0-CFA analysis to generate a sound abstract ACS for a given λ_{ACTOR} program $\mathcal{P}$ and discuss our implementation.

3.3 An Operational Semantics for λ_{ACTOR}

In this section, we extend the concrete operational semantics for the λ-calculus fragment of λ_{ACTOR} to the general case. The full concrete operational semantics associates with each concurrent λ_{ACTOR}-process a *process identifier* or *pid* $\iota \in \mathit{Pid}$ which is also used to reference ι's mailbox. The state space has one global store to realise the indirection for environments, closures, and continuations. Let us fix a closed λ_{ACTOR} program $\mathcal{P}$ for the rest of this chapter.

A Concrete Semantics. Without loss of generality, we assume that in $\mathcal{P}$ all variables are distinct, and constructors and cases are only applied to variables. The concrete operational semantics generates a transition system $(\mathit{State}, \rightarrow_{\mathcal{P}})$ where we define the set *State* as follows:

$$\begin{aligned} s \in \mathit{State} &:= \mathit{Procs} \times \mathit{Mailboxes} \times \mathit{Store} \\ \pi \in \mathit{Procs} &:= \mathit{Pid} \rightharpoonup \mathit{ProcState} \\ \mu \in \mathit{Mailboxes} &:= \mathit{Pid} \rightharpoonup \mathit{Mailbox} \end{aligned}$$

An element of *Procs* π associates a process with pid ι with its *local process state* $\pi(\iota)$. An element of *Mailboxes* μ associates a pid ι with its mailbox $\mu(\iota)$. The *local state* of a process:

$$q \in \mathit{ProcState} := (\mathit{ProgLoc} + \mathit{Pid}) \times \mathit{Env} \times \mathit{KAddr} \times \mathit{Time}$$

is similar to what we saw in the sequential semantics in Section 3.2. It is a quadruple $\langle e, \rho, a, t \rangle$ consisting of e which is a pid or a program location of $\mathcal{P}$, an environment ρ, a continuation address a, and a new component: a time-stamp t.

The *Store* remains virtually the same. However, in the full fragment of λ_{ACTOR} variables may be bound to a pid. Thus, we redefine stores to map value addresses to hold *values*:

$$\sigma \in \mathit{Store} := \{\sigma_V + \sigma_K \mid \sigma_V : \mathit{VAddr} \rightharpoonup \mathit{Value}, \sigma_K : \mathit{KAddr} \rightharpoonup \mathit{Kont}\}$$

where values are either closures or pids, i.e. $d \in \mathit{Value} := \mathit{Closure} + \mathit{Pid} \times \mathit{Env}$. The definition of the sets *Closure* and *Env* remain the same: $\mathit{Env} := \mathit{Var} \rightharpoonup \mathit{VAddr}$ and $\mathit{Closure} := \mathit{ProgLoc} \times \mathit{Env}$. We amend the definition of the set of continuations to allow arbitrary arity of function abstractions and applications in $\mathcal{P}$:

$$\mathit{Kont} = \{\bullet\} + \sum_{i=0}^{ar} \mathit{ProgLoc} \times \mathit{Value}^i \times \mathit{Env} \times \mathit{KAddr}.$$

where we write *ar* for the maximal arity of any function abstraction or application in $\mathcal{P}$. We continue to use the natural embeddings Stop and $\mathsf{Arg}_0\langle - \rangle, \ldots, \mathsf{Arg}_{ar}\langle - \rangle$ as before.

Mailboxes. A *mailbox* is a finite sequence of values, i.e $\mathfrak{m} \in \mathit{Mailbox} := \mathit{Value}^*$. It is supported by two operations:

$$\begin{aligned} \mathrm{mmatch}\colon\ & \mathit{pat}^* \times \mathit{Mailbox} \times \mathit{Env} \times \mathit{Store} \to (\mathbb{N} \times (\mathit{Var} \rightharpoonup \mathit{Value}) \times \mathit{Mailbox})_\perp, \\ \mathrm{enq}\colon\ & \mathit{Value} \times \mathit{Mailbox} \to \mathit{Mailbox}. \end{aligned}$$

The function mmatch takes a list of patterns, a mailbox, the current environment, and a store as arguments. The store is used to resolve pointers in the values stored in the mailbox. If matching is successful, mmatch returns the index of the matching pattern, a substitution witnessing the match, and the mailbox resulting from the extraction of the matched message. Otherwise mmatch returns $\perp$. To model FIFFO mailboxes we set $\mathrm{enq}(d, \mathfrak{m}) := \mathfrak{m} \cdot d$ and define:

$$\mathrm{mmatch}(p_1 \ldots p_n, \mathfrak{m}, \rho, \sigma) := (i, \theta, \mathfrak{m}_1 \cdot \mathfrak{m}_2)$$

such that

$$\mathfrak{m} = \mathfrak{m}_1 \cdot d \cdot \mathfrak{m}_2 \qquad \forall d' \in \mathfrak{m}_1 \,.\, \forall j \,.\, \text{match}_{\rho,\sigma}(p_j, d') = \bot$$
$$\theta = \text{match}_{\rho,\sigma}(p_i, d) \qquad \forall j < i \,.\, \text{match}_{\rho,\sigma}(p_j, d) = \bot$$

where $\text{match}_{\rho,\sigma}(p, d)$ is the concrete matching function. The function $\text{match}_{\rho,\sigma}(p, d)$ seeks to match the value d against the pattern p, following the pointers in environment ρ through the store σ, if necessary. If pattern matching is successful, the result is a substitution θ, otherwise $\text{match}_{\rho,\sigma}(p, d)$ yields $\bot$.

Addresses, Pids, and Time-Stamps. In principle, the concrete operational semantics supports arbitrary concrete representations of time-stamps, addresses, and pids. Here, we opt for an instantiation based on *contours* [Shi91]. A way to represent a *dynamic* occurrence of a symbol is the history of the computation at the point of its creation. This history can be formalised as *contours* which are sequences of program locations, i.e. $t \in \mathit{Time} := \mathit{ProgLoc}^*$. Thus, the contour that we use to start the execution of $\mathcal{P}$ is the empty sequence $t_0 := \epsilon$. At a function call we update contours using the function tick: $\mathit{ProgLoc} \times \mathit{Time} \to \mathit{Time}$ which is defined as $\text{tick}(\ell, t) := \ell \cdot t$.

A value addresses $b \in \mathit{VAddr}$ is represented by a tuple (ι, x, δ, t) comprised of the pid ι, variable x, and the time stamp t at the point of binding, in addition to the bound data value δ. An address $a, c \in \mathit{KAddr}$ for a continuations κ is represented by a tuple (ι, ℓ, ρ, t) comprising the pid ι, program location ℓ, environment ρ, and contour t at the point of creation of κ. The distinguished address $*$ is reserved for the continuation Stop, i.e.

$$\mathit{VAddr} := \mathit{Pid} \times \mathit{Var} \times \mathit{Data} \times \mathit{Time}, \qquad \mathit{KAddr} := (\mathit{Pid} \times \mathit{ProgLoc} \times \mathit{Env} \times \mathit{Time}) + \{*\}.$$

Including *Pid* into the address space ensures a separation in the global store along pids. The set *Data* is the set of closed λ_{ACTOR} expressions. The function $\text{res} : \mathit{Store} \times \mathit{Value} \to \mathit{Data}$ resolves all the pointers of a value through the store σ, returning the corresponding closed expression:

$$\text{res}(\sigma, (\iota, \rho)) := \iota, \qquad \text{res}(\sigma, (e, \rho)) := e[x \mapsto \text{res}(\sigma, \sigma(\rho(x))) \mid x \in \mathit{FV}(e)].$$

To enable data abstraction in our framework, the address of a value d contains the data δ which is the result of applying res. The function new_{va} creates such value addresses for us:

$$\text{new}_{\text{va}} : \mathit{Pid} \times \mathit{Var} \times \mathit{Data} \times \mathit{ProcState} \to \mathit{VAddr}$$
$$\text{new}_{\text{va}}(\iota, x, \delta, \langle \_, \_, \_, t \rangle) := (\iota, x, \delta, t)$$

With this instrumentation of value addresses, we can use the embedded information on data in the abstract semantics to fine-tune the data-sensitivity of our analysis.

In order to simplify the presentation of the rules of the concrete semantics, we use two specialised helper functions $\text{new}_{\text{kpush}}$ and new_{kpop} to generate new continuation addresses:

$$\text{new}_{\text{kpush}} : \mathit{Pid} \times \mathit{ProcState} \to \mathit{KAddr}$$
$$\text{new}_{\text{kpush}}(\iota, \langle \ell, \rho, \_, t \rangle) := (\iota, \ell.\text{arg}_0, \rho, t)$$

$$\text{new}_{\text{kpop}} : \mathit{Pid} \times \mathit{Kont} \times \mathit{ProcState} \to \mathit{KAddr}$$
$$\text{new}_{\text{kpop}}(\iota, \kappa, \langle \_, \_, \_, t \rangle) := (\iota, \ell.\text{arg}_{i+1}, \rho, t) \quad \text{where } \kappa = \text{Arg}_i \langle \ell, \ldots, \rho, \_ \rangle$$

Functional reductions

FunEval

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \ell : e_0(e_1, \ldots, e_n), \rho, a, t\rangle \\ \quad b := \text{new}_{\text{kpush}}(\iota, \pi(\iota)) \\ \text{then } \pi' = \pi[\iota \mapsto \langle e_0, \rho, b, t\rangle] \\ \quad \sigma' = \sigma[b \mapsto \text{Arg}_0\langle \ell, \epsilon, \rho, a\rangle]\end{array}$$

Apply

$$\begin{array}{l}\text{if } \pi(\iota) = \langle v, \rho, a, t\rangle,\ \text{arity}(\ell) = n \\ \quad \sigma(a) = \text{Arg}_n\langle \ell, d_0 \ldots d_{n-1}, \rho', c\rangle \\ \quad d_0 = (\textbf{fun}(x_1 \ldots x_n) \to e, \rho_0) \\ \quad d_n := (v, \rho) \\ \quad b_i := \text{new}_{\text{va}}(\iota, x_i, \text{res}(\sigma, d_i), \pi(\iota)) \\ \quad t' := \text{tick}(\ell, \pi(\iota)) \\ \quad \rho'' := \rho'[x_i \mapsto b_i \mid i \in \langle n\rangle] \\ \text{then } \pi' = \pi[\iota \mapsto \langle e, \rho'', c, t'\rangle] \\ \quad \sigma' = \sigma[b_1 \mapsto d_1 \ldots b_n \mapsto d_n]\end{array}$$

Vars

$$\begin{array}{l}\text{if } \pi(\iota) = \langle x, \rho, a, t\rangle \\ \quad \sigma(\rho(x)) = (v, \rho') \\ \text{then } \pi' = \pi[\iota \mapsto \langle v, \rho', a, t\rangle]\end{array}$$

ArgEval

$$\begin{array}{l}\text{if } \pi(\iota) = \langle v, \rho, a, t\rangle, \ell : e_0(e_1, \ldots, e_n), \\ \quad \sigma(a) = \kappa = \text{Arg}_i\langle \ell, d_0 \ldots d_{i-1}, \rho', c\rangle \\ \quad d_i := (v, \rho) \\ \quad b := \text{new}_{\text{kpop}}(\iota, \kappa, \pi(\iota)) \\ \text{then } \pi' = \pi[\iota \mapsto \langle e_{i+1}, \rho', b, t\rangle] \\ \quad \sigma' = \sigma[b \mapsto \text{Arg}_{i+1}\langle \ell, d_0 \ldots d_i, \rho', c\rangle]\end{array}$$

LetRec

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \textbf{letrec } x_1 = f_1 \cdots x_n = f_n \textbf{ in } e, \rho, a, t\rangle \\ \quad \rho' := \rho[x_i \mapsto b_i \mid i \in \langle n\rangle], \\ \quad b_i := \text{new}_{\text{va}}(\iota, x_i, \text{res}(\sigma', d_i), \pi(\iota)) \text{ and} \\ \quad d_i := (f_i, \rho') \text{ for all } i \in \langle n\rangle \\ \text{then } \pi' = \pi[\iota \mapsto \langle e, \rho', a, t\rangle] \\ \quad \sigma' = \sigma[b_i \mapsto d_i \mid i \in \langle n\rangle]\end{array}$$

Case

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \textbf{case } x \textbf{ of } p_1 \to e_1 \cdots p_n \to e_n \textbf{ end}, \rho, a, t\rangle \\ \quad \theta = \text{match}_{\rho,\sigma}(p_i, (x, \rho)) \text{ and} \\ \qquad \text{for all } j < i \ \text{match}_{\rho,\sigma}(p_j, (x, \rho)) = \bot, \\ \quad \theta = [x_1 \mapsto d_1 \ldots x_k \mapsto d_k], \\ \quad b_m := \text{new}_{\text{va}}(\iota, x_m, \text{res}(\sigma, d_m), \pi(\iota)), \\ \quad \rho' := \rho[x_i \mapsto b_i \mid i \in \langle k\rangle], \\ \text{then } \pi' = \pi[\iota \mapsto \langle e_i, \rho', a, t\rangle] \\ \quad \sigma' = \sigma[b_i \mapsto d_i \mid i \in \langle k\rangle]\end{array}$$

Figure 3.4: The rules of the concrete operational semantics for functional reductions. The table defines the transition relation $s = \langle \pi, \mu, \sigma, \vartheta\rangle \to_{\mathcal{P}} \langle \pi', \mu', \sigma', \vartheta'\rangle = s'$ by cases. The primed components of the state are identical to the non-primed components, unless indicated otherwise. The meta-variable v stands for irreducible expressions such as λ-abstractions, constructor applications, and un-applied primitives.

Further, a pid $\iota \in \mathit{Pid}$ can be identified with the contour of the **spawn** that generated it. Thus, we define $\mathit{Pid} := \mathit{ProgLoc} \times \mathit{Time}$. We formalise the generation of new pids using the auxiliary function $\text{new}_{\text{pid}} : \mathit{Pid} \times \mathit{ProgLoc} \times \mathit{Time} \to \mathit{Pid}$ defined by $\text{new}_{\text{pid}}((\ell', t'), \ell, t) := (\ell, \text{tick}^*(t, \text{tick}(\ell', t')))$. The function tick^* is just the extension of tick that concatenates two contours to one another. When we start the execution of $\mathcal{P}$, the pid of the initial process is $\iota_0 := (\ell_0, \epsilon)$, where ℓ_0 is just the label at the root of $\mathcal{P}$.

Remark 1. We note that the dependency tree has now three leaves that are potentially infinite sets: *Time*, *Mailbox*, and *Data*. If we ensure that these three sets are finite, we can conclude that the state space is finite. Naturally, these three sets are the target for our abstractions.
The concrete operational semantics can be presented in a more general fashion. Following our demonstration in Section 3.2 we may choose not to determine the sets *KAddr*, *VAddr*, and *Pid*. This would yield more leaves in the dependency tree which offers more flexibility in choosing abstractions.

Concurrency

Receive

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \textbf{receive } p_1 \to e_1 \ldots p_n \to e_n \textbf{ end}, \rho, a, t\rangle \\ \quad \text{mmatch}(p_1 \ldots p_n, \mu(\iota), \rho, \sigma) = (i, \theta, \mathfrak{m}) \\ \quad \theta = [x_1 \mapsto d_1 \ldots x_k \mapsto d_k] \\ \quad b_j := \text{new}_{\text{va}}(\iota, x_j, \text{res}(\sigma, d_j), \pi(\iota)) \\ \quad \rho' := \rho[x_1 \mapsto b_1 \ldots x_k \mapsto b_k] \\ \text{then } \pi' = \pi[\iota \mapsto \langle e_i, \rho', a, t\rangle] \\ \qquad \mu' = \mu[\iota \mapsto \mathfrak{m}] \\ \qquad \sigma' = \sigma[b_1 \mapsto d_1 \ldots b_k \mapsto d_k]\end{array}$$

Send

$$\begin{array}{l}\text{if } \pi(\iota) = \langle v, \rho, a, t\rangle \\ \quad \sigma(a) = \kappa = \text{Arg}_2\langle \ell, d, (\iota', \_), \_, c\rangle,\ d = (\textbf{send}, \_) \\ \text{then } \pi' = \pi[\iota \mapsto \langle v, \rho, c, t\rangle] \\ \qquad \mu' = \mu[\iota' \mapsto \text{enq}((v, \rho), \mu(\iota'))]\end{array}$$

Spawn

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \textbf{fun}() \to e, \rho, a, t\rangle \\ \quad \sigma(a) = \text{Arg}_1\langle \ell, d, \rho', c\rangle \\ \qquad d = (\textbf{spawn}, \_) \\ \qquad \iota' := \text{new}_{\text{pid}}(\iota, \ell, \vartheta) \\ \text{then} \\ \qquad \pi' = \pi \begin{bmatrix} \iota \mapsto \langle \iota', \rho', c, t\rangle, \\ \iota' \mapsto \langle e, \rho, *, t_0\rangle \end{bmatrix} \\ \quad \mu' = \mu[\iota' \mapsto \epsilon]\end{array}$$

Self

$$\begin{array}{l}\text{if } \pi(\iota) = \langle \textbf{self}\,() , \rho, a, t\rangle \\ \text{then } \pi' = \pi[\iota \mapsto \langle \iota, \rho, a, t\rangle]\end{array}$$

Figure 3.5: The rules of the concrete operational semantics for dealing with concurrency.

Definition 13 (Concrete Semantics). The rules of the concrete semantics defines a transition system $\mathcal{P} = (\mathit{State}, \to_{\mathcal{P}})$. In Figure 3.4, we present the rules for application, letrecs, and pattern-matching. The rules for message passing and process creation are displayed in Figure 3.5. We denote the initial state that starts the execution of the λACTOR program $\mathcal{P}$ by $s_0 := \langle \pi_0, \mu_0, \sigma_0\rangle$ where $\pi_0 = [\iota_0 \mapsto \langle \mathcal{P}, \emptyset, *, t_0\rangle]$, $\mu_0 = [\iota_0 \mapsto \epsilon\,]$, and $\sigma_0 = [* \mapsto \text{Stop}]$.

The rules for the purely functional reductions are a simple lifting of the corresponding rules for the sequential CESK* machine that we have seen in Section 3.2. The rules **Apply**, **FunEval**, and **ArgEval** are essentially the same as before. We have amended them to allow arbitrary arity in applications. The rule **LetRec** constructs a fixpoint binding of the function abstractions $f_1, \ldots, f_n$ to the variables $x_1, \ldots, x_n$. This is achieved by ensuring the environment that resolves the variables $x_1, \ldots, x_n$ is used in the store binding. Pattern-matching is realised by the **Case** rule: match is used to find the first matching pattern p_i yielding a substitution θ. Control is then given to e_i after the pattern-matching variables $x_1, \ldots, x_k$ have been bound to their substitutions. The Rule **Receive** can only fire if mmatch returns a valid match from the mailbox of the process. In case there is a match, control is passed to the expression in the matching clause. The substitution θ witnessing the match is used to generate the bindings for the variables of the pattern. When applying the **Send** rule, the recipient's pid is first extracted from the continuation, before enq is called to dispatch the evaluated message to the designated mailbox. When performing a spawn (Rule **Spawn**), the argument must be an evaluated nullary function. A new process with a fresh pid is then created that executes the body of the function.

3.4 Parametric Abstract Interpretation of λACTOR

Inspecting the definitions of the concrete state space, we identify that its dependency tree has three leaves that are infinite sets: *Time*, *Mailbox*, and *Data*. In order to obtain a finite abstract

semantics, we need (i) abstract sets $\widehat{Time}$, $\widehat{Mailbox}$, and $\widehat{Data}$, together with (ii) three extraction functions $\eta_{Time} : Time \to \widehat{Time}$, $\eta_{Mailbox} : Mailbox \to \widehat{Mailbox}$, and $\eta_{Data} : Data \to \widehat{Data}$, and (iii) an initial contour $\hat{t}_0$. Equipped with such a choice, we can begin to define the abstract counterparts of the abstract state space by lifting and obtain the respective extraction functions as we do in Section 3.2. We start with abstract pids $\widehat{Pid}$ and abstract addresses $\widehat{VAddr}$ and $\widehat{KAddr}$:

$$\widehat{Pid} := (ProgLoc \times \widehat{Time}) + \{\hat{\iota}_0 := (\ell_0, \hat{t}_0)\}, \qquad \widehat{VAddr} := \widehat{Pid} \times Var \times \widehat{Data} \times \widehat{Time},$$
$$\eta(\ell, t) = (\ell, \eta(t)), \quad \eta(\iota_0) = \hat{\iota}_0; \qquad \eta(\iota, x, \delta, t) = (\eta(\iota), x, \eta(\delta), \eta(t));$$
$$\widehat{KAddr} := (\widehat{Pid} \times ProgLoc \times \widehat{Env} \times \widehat{Time}) + \{\hat{*}\}$$
$$\eta(\iota, \ell, \rho, t) = (\eta(\iota), \ell, \eta(\rho), \eta(t)), \quad \eta(*) = (\hat{*}).$$

It is then easy to lift the abstraction to abstract environments $\widehat{Env} := Var \rightharpoonup \widehat{VAddr}$ using the extraction function $\eta(\rho) = \lambda\hat{x}.\eta_{VAddr}(\rho(x))$, and to abstract closures, values, and continuations:

$$\widehat{Closure} := ProgLoc \times \widehat{Env}, \qquad \widehat{Value} := \widehat{Closure} + \widehat{Pid} \times \widehat{Env},$$
$$\eta(e, \rho) = (e, \eta(\rho)); \qquad \eta_{Value} = \eta_{Closure} + \lambda(\iota, \rho).(\eta(\iota), \eta(\rho));$$
$$\widehat{Kont} := \{\bullet\} + \sum_{i=0}^{ar} ProgLoc \times \widehat{Value}^{\,i} \times \widehat{Env} \times \widehat{KAddr},$$
$$\eta(\mathsf{Arg}_i\langle \ell, d_1 \dots d_i, \rho, a\rangle) = \mathsf{Arg}_i\langle \ell, \eta(d_1) \dots \eta(d_i), \eta(\rho), \eta(a)\rangle, \quad \eta(\mathsf{Stop}) = \mathsf{Stop}.$$

As before, we write ar as the maximal arity of any function abstraction or application in $\mathcal{P}$ and we write $\mathsf{Stop}, \mathsf{Arg}_0\langle - \rangle, \dots, \mathsf{Arg}_{ar}\langle - \rangle$ for the natural embeddings into $\widehat{Kont}$. Moving up a further level in the dependency tree, we can lift the abstraction to $\widehat{ProcState}$, $\widehat{Procs}$, and $\widehat{Store}$:

$$\widehat{ProcState} := (ProgLoc + \widehat{Pid}) \times \widehat{Env} \times \widehat{KAddr} \times \widehat{Time},$$
$$\eta(e, \rho, a, t) = (e, \eta(\rho), \eta(a), \eta(t)), \quad \eta(\iota, \rho, a, t) = (\eta(\iota), \eta(\rho), \eta(a), \eta(t));$$
$$\widehat{Procs} := \widehat{Pid} \to \mathscr{P}(\widehat{ProcState}),$$
$$\eta(\pi) = \lambda\hat{\iota}. \bigcup \{\eta_{ProcState}(\pi(\iota)) \mid \eta_{Pid}(\iota) = \hat{\iota}\};$$
$$\widehat{Store} := \{\sigma_V + \sigma_K \mid \sigma_V : \widehat{VAddr} \to \mathscr{P}(\widehat{Value}), \sigma_K : \widehat{KAddr} \to \mathscr{P}(\widehat{Kont})\},$$
$$\eta_V(\sigma_V) = \lambda\hat{b}. \bigcup \{\eta_{Value}(\sigma_V(b)) \mid \eta_{VAddr}(b) = \hat{b}\},$$
$$\eta_K(\sigma_K) = \lambda\hat{a}. \bigcup \{\eta_{Kont}(\sigma_K(a)) \mid \eta_{KAddr}(a) = \hat{a}\}, \text{ and } \eta_{Store} = \eta_V + \eta_K.$$

We note that both *Procs* and *Store* have become non-deterministic, e.g. a $\hat{\pi} \in \widehat{Procs}$ maps an abstract pid $\hat{\iota}$ to a set of abstract process states. In order to give us more choice in how to represent abstract mailboxes, we make the additional assumption that $\widehat{Mailbox}$ is a complete lattice. Therefore, $\eta_{Mailbox}$ is a representation function. We can then lift $\widehat{Mailbox}$ and complete the definition of the abstract state space:

$$\widehat{Mailboxes} := \widehat{Pid} \to \widehat{Mailbox}, \qquad \widehat{State} := \widehat{Procs} \times \widehat{Mailboxes} \times \widehat{Store},$$
$$\eta(\mu) = \lambda\hat{\iota}. \bigsqcup \{\eta_{Mailbox}(\mu(\iota)) \mid \eta_{Pid}(\iota) = \hat{\iota}\}; \qquad \alpha_{State}(\pi, \mu, \sigma) = (\eta(\pi), \eta(\mu), \eta(\sigma)).$$

We endow $\widehat{Procs}$, $\widehat{Mailboxes}$, and $\widehat{Store}$ with the partial orders below which turns all of the former into complete lattices:

$$\begin{aligned}\hat{\pi} \leq_{\widehat{Procs}} \hat{\pi}' &\iff \forall \hat{\imath} \in \widehat{Pid}.\hat{\pi}(\hat{\imath}) \subseteq \hat{\pi}'(\hat{\imath}),\\ \hat{\mu} \leq_{\widehat{Mailboxes}} \hat{\mu}' &\iff \forall \hat{\imath} \in \widehat{Pid}.\hat{\mu}(\hat{\imath}) \leq_{\widehat{Mailbox}} \hat{\mu}'(\hat{\imath}),\\ \hat{\sigma} \leq_{\widehat{Store}} \hat{\sigma}' &\iff \forall \hat{b} \in \widehat{VAddr}.\hat{\sigma}(\hat{b}) \subseteq \hat{\sigma}'(\hat{b}) \wedge \forall \hat{a} \in \widehat{KAddr}.\hat{\sigma}(\hat{a}) \subseteq \hat{\sigma}'(\hat{a}).\end{aligned}$$

The state space $\widehat{State}$ is thus turned into a complete lattice with the order $(\hat{\pi},\hat{\mu},\hat{\sigma}) \leq_{\widehat{State}} (\hat{\pi}',\hat{\mu}',\hat{\sigma}')$ if, and only if, $\hat{\pi} \leq_{Procs} \hat{\pi}'$, $\hat{\mu} \leq_{Mailboxes} \hat{\mu}'$, and $\hat{\sigma} \leq_{Store} \hat{\sigma}'$. Hence we know that α_{State} is a representation function and we obtain a Galois connection

$$(\mathscr{P}[State], \alpha, \gamma, \widehat{State}), \text{ where } \alpha(S) = \bigsqcup\{\alpha_{State}(s) \mid s \in S\}, \gamma(\hat{s}) = \{s \mid \alpha_{State}(s) \leq_{\widehat{State}} \hat{s}\}.$$

Before we can define the abstract operational semantics, we need to fix the abstract helper functions for generating abstract addresses and pids. In the concrete, these helper functions where defined in terms of operations on *Data*, *Time*, and *Mailboxes*. In the abstract, we assume that a choice on $\widehat{Data}$, $\widehat{Time}$, and $\widehat{Mailboxes}$ is endowed with abstract operations. Let us summarise our assumptions on a *basic domains abstraction* below.

Definition 14. (i) A *data abstraction* is a triple $\mathcal{D} = \langle \widehat{Data}, \eta_{Data}, \widehat{\text{res}} \rangle$ where $\widehat{Data}$ is a set of abstract data values, $\eta_{Data} : Data \to \widehat{Data}$ is an extraction function, and $\widehat{\text{res}} : \widehat{Store} \times \widehat{Value} \to \mathscr{P}[\widehat{Data}]$ is an abstract counterpart of the function res.
(ii) A *time abstraction* is a quadruple $\mathcal{T} = \langle \widehat{Time}, \eta_{Time}, \widehat{\text{tick}}, \hat{t}_0 \rangle$ where $\widehat{Time}$ is a set of abstract contours, $\eta_{Time} : Time \to \widehat{Time}$ is an extraction functions, $\hat{t}_0 \in \widehat{Time}$ is the initial abstract contour, and $\widehat{\text{tick}} : ProgLoc \times \widehat{Time} \to \widehat{Time}$ is an abstract counterpart of tick.
(iii) A *mailbox abstraction* is a quintuple $\mathcal{M} = \langle \widehat{Mailbox}, \eta_{Mailbox}, \widehat{\text{enq}}, \hat{\epsilon}, \widehat{\text{mmatch}} \rangle$ where $\widehat{Mailbox}$ is a complete lattice with least element $\hat{\epsilon} \in \widehat{Mailbox}$, $\eta_{Mailbox} : Mailbox \to \widehat{Mailbox}$ is a representation function, and the functions $\widehat{\text{enq}} : \widehat{Value} \times \widehat{Mailbox} \to \widehat{Mailbox}$ and

$$\widehat{\text{mmatch}} : pat^* \times \widehat{Mailbox} \times \widehat{Env} \times \widehat{Store} \to \mathscr{P}\left[\mathbb{N} \times (Var \rightharpoonup \widehat{Value}) \times \widehat{Mailbox}\right]$$

are monotone in mailboxes.
(iv) A *basic domains abstraction* is a triple $\mathcal{I} = \langle \mathcal{D}, \mathcal{T}, \mathcal{M} \rangle$ consisting of a data, a time, and a mailbox abstraction.

The abstract functions $\widehat{\text{new}_{\text{kpush}}}$, $\widehat{\text{new}_{\text{kpop}}}$, $\widehat{\text{new}_{\text{va}}}$, and $\widehat{\text{new}_{\text{pid}}}$ may be defined exactly as their concrete versions, but on the abstract domains:

$$\begin{aligned}\widehat{\text{new}_{\text{kpush}}}(\hat{\imath}, \langle \ell, \hat{\rho}, \_, \hat{t} \rangle) &:= (\hat{\imath}, \ell.\text{arg}_0, \hat{\rho}, \hat{t})\\ \widehat{\text{new}_{\text{kpop}}}(\hat{\imath}, \hat{\kappa}, \langle \_, \_, \_, \hat{t} \rangle) &:= (\hat{\imath}, \ell.\text{arg}_{i+1}, \hat{\rho}, \hat{t}) \text{ where } \hat{\kappa} = \text{Arg}_i\langle \ell, \ldots, \hat{\rho}, \_ \rangle\\ \widehat{\text{new}_{\text{va}}}(\hat{\imath}, x, \hat{\delta}, \langle \_, \_, \_, \hat{t} \rangle) &:= (\hat{\imath}, x, \hat{\delta}, \hat{t})\\ \widehat{\text{new}_{\text{pid}}}((\ell', \hat{t}'), \ell, \hat{t}) &:= (\ell, \widehat{\text{tick}}^*(\hat{t}, \widehat{\text{tick}}(\ell', \hat{t}'))\end{aligned}$$

The abstract counterpart of the matching function $\widehat{\text{match}}_{\hat{\rho},\hat{\sigma}}(p, \hat{d})$ uses non-deterministic lookups in the store to find a match of $\hat{d}$ against the pattern p. Lookups are only performed when necessary and since they are non-deterministic $\widehat{\text{match}}_{\hat{\rho},\hat{\sigma}}$ yields a set of substitutions. Note that an empty set is returned if matching is unsuccessful. The minimal lookup strategy ensures that the substitutions returned by $\widehat{\text{match}}_{\hat{\rho},\hat{\sigma}}$ bind variables to values that are either abstract pids or closures $(e, \hat{\rho})$ such that e is a subexpression of $\mathcal{P}$ [JA07]. We can then derive the rules of the abstract operational semantics:

Definition 15 (Abstract Semantics). The rules of the abstract operational semantics define a transition system $\hat{\mathcal{P}} = (\widehat{State}, \rightarrow_{\hat{\mathcal{P}}})$. In Figure 3.6, we present the abstract counterparts of the functional reduction rules for the operational semantics, and in Figure 3.7 we display the abstract rules handling concurrency.

Soundness. We now go on to show that the abstract semantics induces a sound analysis given a basic domain abstraction that satisfies a few mild constraints. We follow the AAM methodology in exhibiting a simulation relation. Let us first give a set of conditions on the basic domain abstraction to ensure that we obtain a sound analysis. The conditions below require that the abstract auxiliary functions behave as intended with respect to the abstraction functions and represent a sound abstraction of their concrete counterparts.

Definition 16. We say a basic domains abstraction $\mathcal{I}$ is *sound*, if (i) the abstract function $\widehat{\text{tick}}$ is determined by the time abstraction and the equation:

$$\eta_{Time}(\text{tick}(\ell, t)) = \widehat{\text{tick}}(\ell, \eta_{Time}(t)); \tag{3.5}$$

(ii) the abstract resolve function $\widehat{\text{res}}$ is monotone with respect to stores and is consistent with the concrete resolve function res:

$$\hat{\sigma} \leq \hat{\sigma}' \implies \widehat{\text{res}}(\hat{\sigma}, \hat{d}) \leq \widehat{\text{res}}(\hat{\sigma}', \hat{d}) \tag{3.6}$$

$$\forall \hat{\sigma} \geq \eta_{Store}(\sigma).\ \eta_{Data}(\text{res}(\sigma, d)) \in \widehat{\text{res}}(\hat{\sigma}, \eta_{Data}(d)); \tag{3.7}$$

(iii) the abstract enqueue helper function $\widehat{\text{enq}}$ is a consistent abstraction of its concrete counterpart, the empty mailbox is abstracted to $\hat{\epsilon}$, i.e.

$$\eta_{Mailbox}(\text{enq}(d, \mathfrak{m})) \leq \widehat{\text{enq}}(\eta_{Value}(d), \eta_{Mailbox}(\mathfrak{m})), \quad \eta_{Mailbox}(\epsilon) = \hat{\epsilon}, \tag{3.8}$$

and the abstract auxiliary function $\widehat{\text{mmatch}}$ is a sound approximation of its concrete counterpart, i.e. if $\text{mmatch}(\boldsymbol{p}, \mathfrak{m}, \rho, \sigma) = (i, \theta, \mathfrak{m}')$ then $\forall \hat{\mathfrak{m}} \geq \eta(\mathfrak{m}), \forall \hat{\sigma} \geq \eta(\sigma), \exists \hat{\mathfrak{m}}' \geq \eta(\mathfrak{m}')$ such that:

$$(i, \eta(\theta), \hat{\mathfrak{m}}') \in \widehat{\text{mmatch}}(\boldsymbol{p}, \hat{\mathfrak{m}}, \eta(\rho), \hat{\sigma}). \tag{3.9}$$

Equipped with these assumptions we can prove that any conforming instantiation of the basic domain abstraction gives rise to a sound analysis:

Functional abstract reductions

AbsFunEval

if $\hat{q} \in \hat{\pi}(\hat{\iota})$
$\hat{q} = \langle \ell : e_0(e_1, \ldots, e_n), \hat{\rho}, \hat{a}, \hat{t} \rangle$
$\hat{b} := \widehat{\text{new}_{\text{kpush}}}(\hat{\iota}, \hat{q})$
then $\hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle e_0, \hat{\rho}, \hat{b}, \hat{t} \rangle\}]$
$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b} \mapsto \{\text{Arg}_0\langle \ell, \epsilon, \hat{\rho}, \hat{a}\rangle\}]$

AbsArgEval

if $\hat{\pi}(\hat{\iota}) \ni \langle v, \hat{\rho}, \hat{a}, \hat{t} \rangle, \ell : e_0(e_1, \ldots, e_n)$,
$\hat{\sigma}(\hat{a}) \ni \hat{\kappa} = \text{Arg}_i\langle \ell, \hat{d}_0 \ldots \hat{d}_{i-1}, \hat{\rho}', \hat{c}\rangle$
$\hat{d}_i := (v, \hat{\rho})$
$\hat{b} := \widehat{\text{new}_{\text{kpop}}}(\hat{\iota}, \hat{\kappa}, \hat{q})$
then $\hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle e_{i+1}, \hat{\rho}', \hat{b}, \hat{t} \rangle\}]$
$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b} \mapsto \{\text{Arg}_{i+1}\langle \ell, \hat{d}_0 \ldots \hat{d}_i, \hat{\rho}', \hat{c}\rangle\}]$

AbsApply

if $\hat{\pi}(\hat{\iota}) \ni \hat{q} = \langle v, \hat{\rho}, \hat{a}, \hat{t} \rangle$,
$n = \text{arity}(\ell)$
$\hat{\sigma}(\hat{a}) \ni \text{Arg}_n\langle \ell, \hat{d}_0 \ldots \hat{d}_{n-1}, \hat{\rho}', \hat{c}\rangle$
$\hat{d}_0 = (\textbf{fun}(x_1 \ldots x_n) \to e, \hat{\rho}_0)$
$\hat{d}_n := (v, \hat{\rho})$
$\hat{\delta}_i \in \widehat{\text{res}}(\hat{\sigma}, \hat{d}_i,)$
$\hat{b}_i := \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_i, \hat{\delta}_i, \hat{q})$
$\hat{\rho}'' := \hat{\rho}'[x_1 \mapsto \hat{b}_1 \ldots x_n \mapsto \hat{b}_n]$
$\hat{t}' = \widehat{\text{tick}}(\ell, \hat{t})$
then $\hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle e, \hat{\rho}'', \hat{c}, \hat{t}' \rangle\}]$
$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b}_i \mapsto \{\hat{d}_i\} \mid i \in \langle n \rangle]$

AbsLetRec

if $\hat{\pi}(\hat{\iota}) = \langle \textbf{letrec } x_1 = f_1 \cdots x_n = f_n \textbf{ in } e, \hat{\rho}, \hat{a}, \hat{t} \rangle$
$\hat{\rho}' := \hat{\rho}[x_i \mapsto \hat{b}_i \mid i \in \langle n \rangle]$,
$\hat{b}_i := \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_i, \hat{\delta}_i, \hat{\pi}(\hat{\iota}))$ and
$\hat{\delta}_i \in \widehat{\text{res}}(\hat{\sigma}, \hat{d}_i)$
$\hat{d}_i := (f_i, \hat{\rho}')$ for all $i \in \langle n \rangle$
then $\hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle e, \hat{\rho}', \hat{a}, \hat{t} \rangle\}]$
$\sigma' = \hat{\sigma} \sqcup [\hat{b}_i \mapsto \{\hat{d}_i\} \mid i \in \langle n \rangle]$

AbsVars

if $\hat{\pi}(\hat{\iota}) \ni \langle x, \hat{\rho}, \hat{a}, \hat{t} \rangle$
$\hat{\sigma}(\hat{\rho}(x)) \ni (v, \hat{\rho}')$
then $\hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle v, \hat{\rho}', \hat{a}, \hat{t} \rangle\}]$

AbsCase

if $\hat{\pi}(\hat{\iota}) = \langle \textbf{case } x \textbf{ of } p_1 \to e_1 \cdots p_n \to e_n \textbf{ end}, \hat{\rho}, \hat{a}, \hat{t} \rangle$
$\theta \in \widehat{\text{match}}_{\hat{\rho},\hat{\sigma}}(p_i, (x, \hat{\rho}))$ and
$\theta = [x_1 \mapsto \hat{d}_1 \ldots x_k \mapsto \hat{d}_k]$,
$\hat{b}_j := \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_j, \hat{\delta}_j, \hat{\pi}(\hat{\iota}))$,
$\hat{\delta}_j \in \widehat{\text{res}}(\hat{\sigma}, \hat{d}_j)$
$\hat{\rho}' := \hat{\rho}[x_j \mapsto \hat{b}_j \mid j \in \langle k \rangle]$,
then $\hat{\pi}' = \hat{\pi} \sqcup [\iota \mapsto \{\langle e_i, \hat{\rho}', \hat{a}, \hat{t} \rangle\}]$
$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b}_j \mapsto \{\hat{d}_j\} \mid j \in \langle k \rangle]$

Figure 3.6: The rules of the abstract operational semantics for functional reductions. The tables define the transition relation $\hat{s} = \langle \hat{\pi}, \hat{\mu}, \hat{\sigma} \rangle \to_{\hat{\mathcal{P}}} \langle \hat{\pi}', \hat{\mu}', \hat{\sigma}' \rangle = \hat{s}'$ by cases. The primed components of the state are identical to the non-primed components, unless indicated otherwise. We write $\sqcup$ for the join operation of the appropriate domain.

Theorem 4 (Soundness of Analysis). *Given a sound abstraction of the basic domains, the relation* $\mathcal{R} = \{(s, \hat{s}) \mid \alpha_{State}(s) \leq_{\widehat{State}} \hat{s}\}$ *is a simulation relation between the concrete and the abstract operational semantics.*

Proof Idea. Every transition in the concrete system justified by rule **R** can be replicated in the abstract transition system using its abstract version **AbsR**. See Appendix A.1 for a detailed proof. □

Given appropriate auxiliary helper functions on data, time, and mailboxes the abstract operational semantics gives rise to the optimal abstract interpretation along the Galois connection $(\mathscr{P}[State], \alpha, \gamma, \widehat{State})$. However, Theorem 4's proof yields soundness also when the basic domains abstraction is not necessarily optimal.

Abstract Concurrency

AbsReceive

$$
\begin{array}{l}
\text{if } \hat{\pi}(\hat{\iota}) \ni \hat{q} = \langle e, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\
\quad e = \textbf{receive}\ p_1 \to e_1 \ldots p_n \to e_n\ \textbf{end} \\
\quad (i, \hat{\theta}, \hat{\mathfrak{m}}) \in \widehat{\text{mmatch}}(p_1 \ldots p_n, \hat{\mu}(\hat{\iota}), \hat{\rho}, \hat{\sigma}) \\
\quad \hat{\theta} = [x_1 \mapsto \hat{d}_1 \ldots x_k \mapsto \hat{d}_k] \\
\quad \hat{\delta}_j \in \widehat{\text{res}}(\hat{\sigma}, \hat{d}_j) \\
\quad \hat{b}_j := \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_j, \hat{\delta}_j, \hat{q}\,) \\
\quad \hat{\rho}' := \hat{\rho}[x_1 \mapsto \hat{b}_1 \ldots x_k \mapsto \hat{b}_k] \\
\text{then } \hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle e_i, \hat{\rho}', \hat{a}, \hat{t}\,\rangle\}] \\
\qquad \hat{\mu}' = \hat{\mu} \sqcup [\hat{\iota} \mapsto \hat{\mathfrak{m}}] \\
\qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{b}_1 \mapsto \{\hat{d}_1\} \ldots \hat{b}_k \mapsto \{\hat{d}_k\}]
\end{array}
$$

AbsSend

$$
\begin{array}{l}
\text{if } \hat{\pi}(\hat{\iota}) \ni \langle v, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\
\quad \hat{\sigma}(\hat{a}) \ni \text{Arg}_2\langle \ell, \hat{d}, \hat{\iota}', \_, \hat{c}\rangle, \quad \hat{d} = (\textbf{send}, \_) \\
\text{then } \hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle v, \hat{\rho}, \hat{c}, \hat{t}\,\rangle\}] \\
\qquad \hat{\mu}' = \hat{\mu}[\hat{\iota}' \mapsto \widehat{\text{enq}}((v, \hat{\rho}), \hat{\mu}(\hat{\iota}'))]
\end{array}
$$

AbsSpawn

$$
\begin{array}{l}
\text{if } \hat{\pi}(\hat{\iota}) \ni \langle \textbf{fun}() \to e, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\
\quad \hat{\sigma}(\hat{a}) \ni \text{Arg}_1\langle \ell, \hat{d}, \hat{\rho}', \hat{c}\rangle \\
\qquad \hat{d} = (\textbf{spawn}, \_) \\
\quad \hat{\iota}' := \widehat{\text{new}_{\text{pid}}}(\hat{\iota}, \ell, \hat{t}\,) \\
\text{then} \\
\hat{\pi}' = \hat{\pi} \sqcup \left[\begin{array}{l} \hat{\iota} \mapsto \{\langle \hat{\iota}', \hat{\rho}', \hat{c}, \hat{t}\,\rangle\}, \\ \hat{\iota}' \mapsto \{\langle e, \hat{\rho}, *, \hat{t}_0\rangle\} \end{array}\right] \\
\hat{\mu}' = \hat{\mu} \sqcup [\hat{\iota}' \mapsto \hat{\epsilon}\,]
\end{array}
$$

AbsSelf

$$
\begin{array}{l}
\text{if } \hat{\pi}(\hat{\iota}) \ni \langle \textbf{self}\,(), \hat{\rho}, \hat{a}, \hat{t}\,\rangle \\
\text{then } \hat{\pi}' = \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\langle \hat{\iota}, \hat{\rho}, \hat{a}, \hat{t}\,\rangle\}]
\end{array}
$$

Figure 3.7: The rules of the abstract operational semantics that deal with concurrency.

Termination. Since we have eliminated any recursion from the state space, we are ensured that for any given λACTOR program $\mathcal{P}$ we obtain a terminating analysis.

Theorem 5 (Decidability of Analysis). *Given a finite abstraction of the basic domains, the derived abstract transition relation $\to_{\hat{\mathcal{P}}}$ gives rise to a transition system with a finite reachability set $Reach_{\hat{\mathcal{P}}}(\alpha_{State}(s_0))$. It is also decidable if the associated auxiliary operations (in Definition 16) are computable.*

Proof. The proof is by a simple inspection of the rules: all the individual rules are decidable and the state space is finite. □

An Instantiation of the Abstract Interpretation

In order to generate a simulating ACS for $\mathcal{P}$, we consider a specific instantiation of the abstract semantics. The basic domain abstraction we consider abstracts data to expressions of *a priori* fixed depth D, contours to their prefixes of length at most k — yielding a k-CFA style abstraction, and mailboxes to sets of messages which disregards both the sequence and number of occurrences in mailboxes.

The Set Mailbox Abstraction. We may assume that by abstracting data and time the set of abstract values $\widehat{Value}$ will be finite. Thus, taking abstract mailboxes to be sets of abstract values, i.e. $\mathscr{P}[\widehat{Value}]$, yields a finite choice. Further, $\mathscr{P}[\widehat{Value}]$ forms the well-known complete powerset lattice: $(\mathscr{P}[\widehat{Value}], \subseteq, \cap, \cup, \emptyset, \widehat{Value})$. We can then define the representation function η_{set} from

concrete to abstract mailboxes by $\eta_{\text{set}}(\mathfrak{m}) := \{\eta_{\mathit{Value}}(d) \mid \exists i.\, \mathfrak{m}(i) = d\}$. It thus remains to define the abstract auxiliary functions $\widehat{\text{enq}}_{\text{set}}$ and $\widehat{\text{mmatch}}_{\text{set}}$. We may define $\widehat{\text{enq}}_{\text{set}}$ as set insertion and we let $\widehat{\text{mmatch}}_{\text{set}}$ apply the abstract match function non-deterministically to obtain a set of matches, i.e. $\widehat{\text{enq}}_{\text{set}}(\hat{d}, \hat{\mathfrak{m}}) := \{\hat{d}\} \cup \hat{\mathfrak{m}}$ and, writing $\boldsymbol{p} = p_1 \cdots p_n$,

$$\widehat{\text{mmatch}}_{\text{set}}(\boldsymbol{p}, \hat{\mathfrak{m}}, \hat{\rho}, \hat{\sigma}) := \{(i, \hat{\theta}, \hat{\mathfrak{m}}) \mid \exists i.\hat{d} \in \hat{\mathfrak{m}}, \hat{\theta} \in \widehat{\text{match}}_{\hat{\rho},\hat{\sigma}}(p_i, \hat{d})\}.$$

We can easily verify that choosing $\mathcal{M}_{\text{set}} := \langle \mathscr{P}[\widehat{\mathit{Value}}], \eta_{\text{set}}, \widehat{\text{enq}}_{\text{set}}, \emptyset, \widehat{\text{mmatch}}_{\text{set}} \rangle$ as a mailbox abstraction is sound.

Abstracting Data. In order to allow an abstraction take data into account, we augmented the definition of variable addresses *VAddr* with a data component. This allows the abstract semantics to distinguish states that differ on the abstract bindings in environments. Since *Data* is a leaf of the dependency graph, any finite data abstraction yields a finite abstract semantics.

A commonly used data abstraction for languages with algebraic data-types truncates subexpressions at a chosen depth D. Given a closure $(e, \hat{\rho})$ the set of terms $\lfloor (e, \hat{\rho}) \rfloor_{\hat{\sigma},D}$ is the result of this truncation where variables are looked up using the environment $\hat{\rho}$ and the store $\hat{\sigma}$:

$$\lfloor (e, \hat{\rho}) \rfloor_{\hat{\sigma},0} := \{\_\} \quad \lfloor (\textbf{fun}\ldots, \hat{\rho}) \rfloor_{\hat{\sigma},D+1} := \{\_\}$$
$$\lfloor (\mathrm{c}(x_1 \ldots x_n), \hat{\rho}) \rfloor_{\hat{\sigma},D+1} := \{\mathrm{c}(\hat{\delta}_1 \ldots \hat{\delta}_n) \mid \hat{d}_i \in \hat{\sigma}(\hat{\rho}(x_i)), \hat{\delta}_i \in \lfloor \hat{d}_i \rfloor_{\hat{\sigma},D}\}$$

where _ is a distinguished constructor representing discarded subexpressions. A D-deep abstraction $\lfloor - \rfloor_{\_}$ can be similarly defined for concrete values and concrete data. We write $\lfloor (e, \rho) \rfloor_{\sigma,D}$ for the former and $\lfloor e \rfloor_D$ for the latter. Hence, we can define the *depth-D* data abstraction to be $\mathcal{D}_D = \langle \mathit{Data}_D, \alpha_{\text{D}}, \widehat{\text{res}}_D \rangle$ where Data_D is defined inductively:

$$\mathit{Data}_{D+1} := \{\_\} \cup \{\mathrm{c}(\hat{\delta}_1 \ldots \hat{\delta}_n) \mid \hat{\delta}_i \in \mathit{Data}_D\}, \qquad \mathit{Data}_0 := \{\_\}$$

and we define the extraction function η_D and the abstract resolve function $\widehat{\text{res}}_D$ as follows:

$$\eta_D(\delta) := \lfloor \delta \rfloor_D, \qquad \widehat{\text{res}}_D(\hat{\sigma}, \hat{d}) := \lfloor \hat{d} \rfloor_{\hat{\sigma},D}.$$

The abstract resolve function $\widehat{\text{res}}_D$ is easily seen to to satisfy $\eta(\text{res}(\sigma, d)) \in \widehat{\text{res}}(\hat{\sigma}, \eta(d))$ for all $\hat{\sigma} \geq_{\widehat{\mathit{Store}}} \eta(\sigma)$. Further, $\widehat{\text{res}}_D$ is clearly monotone and hence Data_D is a sound data abstraction.

Abstracting Time. k-CFA specifies a family of contour abstractions. The parameter k determines the context-sensitivity that can be captured by abstract contours. For a fixed k the set of abstract contours becomes $\widehat{\mathit{Time}}_k := \bigcup_{0 \leq i \leq k} \mathit{ProgLoc}^i$. The extraction function η_k from *Time* to $\widehat{\mathit{Time}}_k$ is defined by:

$$\eta_k(\ell_1 \cdots \ell_j) := \ell_1 \ldots \ell_i, \quad \text{where } i = \min(j, k)$$

Thus, k-CFA merges two contours whose dynamic history have the same k-long prefix, which represents the k most recent call-contexts, in the abstract. The soundness requirement on the function $\widehat{\text{tick}}$ determines the definition: $\widehat{\text{tick}}(\ell, \hat{t}) = \eta_k(\ell \cdot \hat{t})$. Hence the k-CFA time abstraction $\mathcal{T}_k = \langle \widehat{\mathit{Time}}_k, \eta_k, \widehat{\text{tick}}, \epsilon \rangle$ is sound by construction.

In the next section we show how we generate an ACS while exploring the state space of the abstract operational semantics generated by $\mathcal{P}$.

3.5 Generating the Actor Communicating System

The abstract interpretation we describe in the previous section gives us a finite state transition system $\hat{\mathcal{P}}$ that simulates the concrete semantics $\mathcal{P}$. As an abstract description of the concrete transition system, $\hat{\mathcal{P}}$ is quite coarse. $\hat{\mathcal{P}}$ cannot accurately track the number of processes in each local process state. Further, $\hat{\mathcal{P}}$ cannot account for the number of messages buffered in mailboxes. However, $\hat{\mathcal{P}}$ gives us a sound finite description of the local process states, pids, and messages. We exploit this information to generate an ACS $\mathcal{A}_{\mathcal{P}}$ that simulates the concrete semantics $\mathcal{P}$. The ACS $\mathcal{A}_{\mathcal{P}}$ is able to keep track of the number of processes in a local process state, and the number of messages in mailboxes. The control states of $\mathcal{A}_{\mathcal{P}}$ are pairs $(\hat{\iota}, \hat{q})$ comprised of an abstract pid $\hat{\iota} \in \widehat{Pid}$ and an abstract local process state $\hat{q} \in \widehat{ProcState}$. A process $\pi = (\hat{\iota}, \hat{q})$ of $\mathcal{A}_{\mathcal{P}}$ represents a process in state $\hat{q}$ with pid $\hat{\iota}$. However, there is a discrepancy between ACS and λACTOR. λACTOR attaches a private mailbox to each running process. This is a feature which we cannot model in ACS since an ACS only supports a finite number of channels. As a solution, we break the link between the representation of a process and its mailbox. Instead, there is a channel for each abstract pid in $\mathcal{A}_{\mathcal{P}}$. Thus, the mailboxes of all concrete processes whose concrete pid ι are mapped to the abstract pid $\hat{\iota}$ are merged together in the channel $\hat{\iota}$. It remains to choose the set of messages for $\mathcal{A}_{\mathcal{P}}$. In the abstract semantics, messages are values that are resolved through the store and which may represent a regular tree language of λACTOR expressions. In order to obtain a finite set of messages, we require an additional data abstraction geared for messages. For the purposes of the ACS, such a message abstraction needs to resolve an abstract message just enough to uniquely identify a matching receive pattern.

We tie the generation of the ACS $\mathcal{A}_{\mathcal{P}}$ to the exploration of the abstract semantics in our presentation. This illustrates that the ACS $\mathcal{A}_{\mathcal{P}}$ can effectively be generated on-the-fly during the exploration of the abstract semantics.

Terminology. We identify a common pattern in the rules in Figures 3.6 and 3.7. In each rule **R**, the premise distinguishes an abstract pid $\hat{\iota}$ and an abstract process state $\hat{q} = \langle e, \hat{\rho}, \hat{a}, \hat{t} \rangle$ associated with $\hat{\iota}$, i.e. $\hat{q} \in \hat{\pi}(\hat{\iota})$, and the conclusion of the rule associates a new abstract process state $\hat{q}'$ with $\hat{\iota}$, i.e. $\hat{q}' \in \hat{\pi}'(\hat{\iota})$. Henceforth, we shall refer to $(\hat{\iota}, \hat{q}, \hat{q}')$ as the *active components* of the rule **R**.

Definition 17 (Generated ACS). Given a λACTOR program $\mathcal{P}$, a sound basic domains abstraction $\mathcal{I} = \langle \mathcal{T}, \mathcal{M}, \mathcal{D} \rangle$, and a sound data abstraction for messages $\mathcal{D}_{\text{msg}} = \langle \widehat{Msg}, \alpha_{\text{msg}}, \widehat{\text{res}}_{\text{msg}} \rangle$ the ACS $\mathcal{A}_{\mathcal{P}}$ generated by $\mathcal{P}$, $\mathcal{I}$, and $\mathcal{D}_{\text{msg}}$ is defined by $\mathcal{A}_{\mathcal{P}} := (\widehat{Pid} \times \widehat{ProcState}, \widehat{Pid}, \widehat{Msg}, R)$ where the set R is defined by induction over the following rules:

(i) If $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'$ is proved by rule **AbsFunEval**, **AbsArgEval**, **AbsLetRec**, **AbsCase**, or **AbsApply** with active components $(\hat{\iota}, \hat{q}, \hat{q}')$, then $(\hat{\iota}, \hat{q}) \xrightarrow{\epsilon} (\hat{\iota}, \hat{q}') \in R$.
(ii) If $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'$ is proved by **AbsReceive** with active components $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{d}$ is the message matched by $\widehat{\text{mmatch}}$ and $\hat{m} \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d})$, then $(\hat{\iota}, \hat{q}) \xrightarrow{\hat{\iota}?\hat{m}} (\hat{\iota}, \hat{q}') \in R$.
(iii) If $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'$ is proved by **AbsSend** with active components $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{d}$ is the abstract value that is sent and $\hat{m} \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d})$, then $(\hat{\iota}, \hat{q}) \xrightarrow{\hat{\iota}'!\hat{m}} (\hat{\iota}, \hat{q}') \in R$.
(iv) If $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'$ is proved by **AbsSpawn** with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota}_0$ is the new abstract pid that is generated in the premise of the rule, which gets associated with the process state $\hat{q}_0$ then $(\hat{\iota}, \hat{q}) \xrightarrow{\nu(\hat{\iota}_0, \hat{q}_0)} (\hat{\iota}, \hat{q}') \in R$.

While we explore the state space of the abstract semantics $\hat{\mathcal{P}}$ from the initial state $\alpha_{\mathit{State}(s_0)}$, we collect the rules of $\mathcal{A}_{\mathcal{P}}$. Whenever we fire an abstract rule $\boldsymbol{R}$, we identify an active component $(\hat{\iota}, \hat{q}, \hat{q}')$ and add a rule $(\hat{\iota}, \hat{q}) \xrightarrow{\lambda} (\hat{\iota}, \hat{q}')$ to $\mathcal{A}_{\mathcal{P}}$ where the side-effect λ depends on $\boldsymbol{R}$ and the abstract transition. Once we have exhaustively explored $\mathit{Reach}_{\hat{\mathcal{P}}}(\alpha_{\mathit{State}(s_0)})$, we have computed all rules of $\mathcal{A}_{\mathcal{P}}$.

Soundness of $\mathcal{A}_{\mathcal{P}}$. Let us now argue that the ACS $\mathcal{A}_{\mathcal{P}}$ is a sound abstraction of the concrete semantics. For this purpose, we define an abstraction function α_{acs} mapping a concrete state to a configuration of $\mathcal{A}_{\mathcal{P}}$, i.e. the abstraction function has type

$$\alpha_{\text{acs}} : \mathit{State} \to \left(\mathbb{M}\left[\widehat{\mathit{Pid}} \times \widehat{\mathit{ProcState}}\right] \times \left(\widehat{\mathit{Pid}} \to \mathbb{M}\left[\widehat{\mathit{Msg}}\right]\right)\right).$$

The mapping α_{acs} is reminiscent of a counter abstraction: α_{acs} maps every concrete state $s = \langle \pi, \mu, \sigma \rangle$ to a parallel composition of ACS processes $\Pi(\pi)$—dependent only on π—together with channel contents $\Gamma(\mu, \sigma)$—dependent only on μ and σ, i.e. $\alpha_{\text{acs}}(s) := \Pi(\pi) \lhd \Gamma(\mu, \sigma)$. The parallel composition $\Pi(\pi)$ runs for every concrete process with pid ι in control state q an ACS process $(\hat{\iota}, \hat{q})$. In $\Gamma(\mu, \sigma)$ every channel $\hat{\iota}$ is a multiset that contains the abstraction of every message in any mailbox $\mu(\iota)$ where ι's abstraction is $\hat{\iota}$. Formally, we define

$$\begin{aligned}\Pi(\pi) &= \left[(\hat{\iota}, \hat{q}) \mapsto \left|\{\iota \mid \eta(\iota) = \hat{\iota}, \eta(\pi(\iota)) = \hat{q}\}\right|\right], \\ \Gamma(\mu, \sigma)(\hat{\iota}) &= \left[\hat{m} \mapsto \left|\{(\iota, i) \mid \eta(\iota) = \hat{\iota}, \alpha_{\text{msg}}(\text{res}(\sigma, \mu(\iota)(i))) = \hat{m}\}\right|\right] \text{ for all } \hat{\iota}.\end{aligned}$$

Equipped with a definition of α_{acs} we may now exhibit a simulation between $\mathcal{P}$ and $\mathcal{A}_{\mathcal{P}}$.

Theorem 6 (Soundness of the generated ACS). *For all choices of $\mathcal{I}$ and $\mathcal{D}_{msg}$, the relation $\mathcal{R}$ defined by $\mathcal{R} = \{(s, \Pi \lhd \Gamma) \mid s \in \mathit{State}, \alpha_{acs}(s) \leq \Pi \lhd \Gamma\}$ is a simulation for $\mathcal{P}$ and $\mathcal{A}_{\mathcal{P}}$.*

Proof Idea. We appeal to Theorem 4 that for any concrete transition $s \to_{\mathcal{P}} s'$ using rule **R** of the concrete operational semantics with active component (ι, q, q') and there is a step of the abstract operational semantics $\hat{s} \to_{\mathcal{P}} \hat{s}'$ with active component $(\hat{\iota}, \hat{q}, \hat{q}')$. This gives us a rule $(\hat{\iota}, \hat{q}) \xrightarrow{\lambda} (\hat{\iota}, \hat{q}')$ in $\mathcal{A}_{\mathcal{P}}$ that we can use to simulate $s \to_{\mathcal{P}} s'$ in $\mathcal{A}_{\mathcal{P}}$.
See Appendix A.2 for a full proof. □

Theorem 6 allows us to relate paths of the transition system $\mathcal{P}$ and the ACS $\mathcal{A}_{\mathcal{P}}$.

Corollary 1. Let $\mathcal{A}_{\mathcal{P}}$ be the ACS derived from $\mathcal{P}$. Then, for each path $s \to_{\mathcal{P}} s_1 \to_{\mathcal{P}} s_2 \to_{\mathcal{P}} \ldots$ in the transition system $\mathcal{P}$, there exists a path $\Pi \lhd \Gamma \to_{\mathcal{A}_{\mathcal{P}}} \Pi_1 \lhd \Gamma_1 \to_{\mathcal{A}_{\mathcal{P}}} \Pi_2 \lhd \Gamma_2 \to_{\mathcal{A}_{\mathcal{P}}} \ldots$ in $\mathcal{A}_{\mathcal{P}}$ such that $\alpha_{\text{acs}}(s) = \Pi \lhd \Gamma$ and for all i, $\alpha_{\text{acs}}(s_i) \leq \Pi_i \lhd \Gamma_i$.

Hence, using $\mathcal{A}_{\mathcal{P}}$ we can conservatively answer reachability, coverability, and termination queries on the concrete operational semantics.

Corollary 2. (i) If there is no $\Pi \lhd \Gamma \geq \alpha_{\text{acs}}(s')$ such that $\alpha_{\text{acs}}(s) \to^*_{\mathcal{A}_{\mathcal{P}}} \Pi \lhd \Gamma$ then $s \not\to^*_{\mathcal{P}} s'$.
(ii) If there is no infinite path from $\alpha_{\text{acs}}(s)$ in $\mathcal{A}_{\mathcal{P}}$, then all paths from s are finite in $\mathcal{P}$.

The extraction function $\eta_{\mathit{ProcState}}$ may map an unbounded number of concrete process states into a single abstract process state. Similarly, α_{msg} may abstract an infinite number of concrete messages to the same abstract message. Thus, boundedness is not preserved by $\mathcal{A}_{\mathcal{P}}$, i.e if $\mathit{Reach}_{\mathcal{A}_{\mathcal{P}}}(\alpha_{\text{acs}}(s))$ is a finite set, we cannot conclude that $\mathit{Reach}_{\mathcal{P}}(s)$ is finite. However, if $\mathcal{A}_{\mathcal{P}}$ has a finite reachability set from $\alpha_{\text{acs}}(s)$, then there exists a bound such that the number of processes that can be created and the capacity of mailboxes do not exceed this bound.

Corollary 3. If $\mathit{Reach}_{\mathcal{A}_{\mathcal{P}}}(\alpha_{\text{acs}}(s))$ is a finite set, then there exists a bound B such that for all reachable configurations $\langle \pi, \mu, \sigma \rangle \in \mathit{Reach}_{\mathcal{P}}(s)$ the number of processes is bounded by B, i.e. $|\{\iota \mid \pi(\iota) = q \in \mathit{ProcState}\}| \leq B$, and for each pid ι the capacity of its mailbox is bounded by B, i.e. $|\mu(\iota)| \leq B$.

The Verification Procedure. In summary, our verification procedure for λACTOR programs is divided into two steps. Given a λACTOR program $\mathcal{P}$, we generate $\mathcal{A}_{\mathcal{P}}$ while exploring the abstract state space $\mathit{Reach}_{\hat{\mathcal{P}}}(\alpha_{\mathit{State}}(s_0))$. We then use $\mathcal{A}_{\mathcal{P}}$ to give conservative answers for verification questions posed on $\mathcal{P}$. Note that in our implementation we focus on coverability.

In the next section, we give an overview of the implementation of our verification procedure in the tool Soter. We explain our exploration algorithm and apply Soter on an example.

3.6 Implementation and a Worked Example

In this section, we present the implementation of our verification procedure as the tool Soter. Soter is written in Haskell and implements an exploration of the abstract space state and the generation of $\mathcal{A}_{\mathcal{P}}$.

Soter accepts a single Erlang module, annotated with *non-coverability assertions*, as input. In general, non-coverability assertions are specified in two parts. The user may label program points and mailboxes, and express *non-coverability constraints* on them. An non-coverability constraint specifies a *bad state*, e.g. an assertion may stipulate that a configuration in which two (or more) processes execute at label l is a *bad state* and hence should not be coverable. Thus, it is possible to assert safety properties such as program point reachability, mutual exclusion, and bound-checks on mailboxes. We have restricted Soter to only handle the subset of Erlang that is readily expressible in λACTOR. This includes the full higher-order fragment of Erlang together with its communication primitives.

We present the workflow of Soter in Figure 3.8. In phase 1, Soter uses the standard Erlang compiler `erlc` to compile the input Erlang module to *Core Erlang*. *Core Erlang* is the official intermediate representation of Erlang and close to λACTOR. The compilation to *Core Erlang* resolves syntactic sugar and normalises the code. Then, Soter parses the *Core Erlang* output — containing user's non-coverability assertions — rejecting any programs that make use of unsupported Erlang primitives. Another normalisation pass is then performed by Soter to ensure that the assumptions we make on λACTOR programs in Section 3.3 are satisfied, i.e. we use α-renaming to ensure variable names are distinct and add additional let-bindings to guarantee that constructors and cases are only applied to variables. The result of phase 1 is a λACTOR program $\mathcal{P}$ that is equivalent to the input Erlang program.

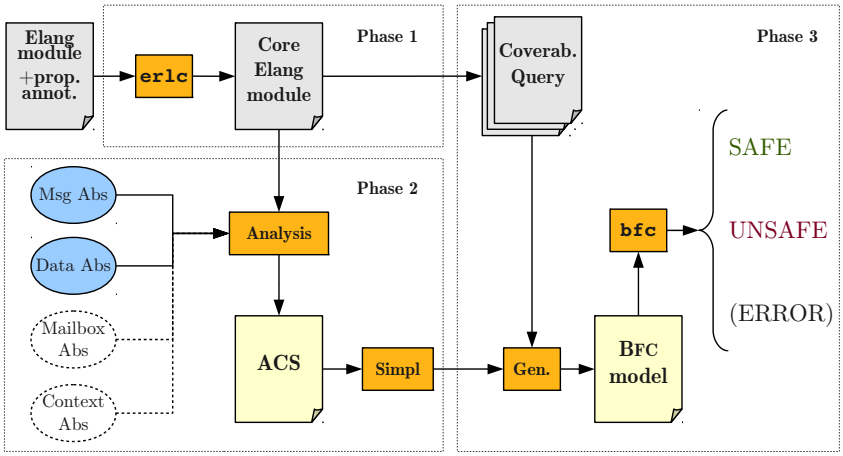


Figure 3.8: The workflow of Soter is divided into three phases. The first translates an arbitrary Erlang module to λ_{ACTOR}. Phase 2 explores the abstract operational semantics, generates an ACS, and simplifies the latter. The third phase invokes BFC to perform coverability checks.

Phase 2 then generates the ACS $\mathcal{A}_{\mathcal{P}}$ on-the-fly while exploring the abstract operational semantics $\mathit{Reach}_{\hat{\mathcal{P}}}(\alpha_{\mathit{State}}(s_0))$. The basic domain abstraction that we have implemented in Soter is $\langle \widehat{\mathit{Data}}_D, \widehat{\mathit{Time}}_0, \widehat{\mathit{Mailbox}}_{\text{set}} \rangle$ together with a data message abstraction $\widehat{\mathit{Data}}_M$. Hence, the analysis is parametric in D — the depth of the data abstraction, which is set to 0 by default — and M which is the depth of the message abstraction. By default, M is set to the maximal depth of receive guards plus D. The abstract operational semantics is implemented in a modular fashion, and we have made provisions to allow the implementation of other time, mailbox, or data abstractions. Before Soter enters phase 3, the generated ACS $\mathcal{A}_{\mathcal{P}}$ is simplified. This yields another ACS $\mathcal{A}'_{\mathcal{P}}$. In the simplification step, we collapse rules that simulate sequences of functional reductions into a single step. Thus, we can significantly reduce the size of $\mathcal{A}_{\mathcal{P}}$.

In phase 3, Soter combines the ACS $\mathcal{A}'_{\mathcal{P}}$ with the non-coverability assertions from the input Erlang module to produce a VAS and a coverability query for each non-coverability constraint. All coverability queries are generated for the tool BFC [KKW12] which Soter uses as a backend coverability engine. BFC implements a highly competitive algorithm for coverability which supports VAS and VAS with transfer arcs. BFC's algorithm tries to find small proofs of non-coverability of target states while running a Karp-Miller-tree forward-search in parallel [KKW12]. For each non-coverability assertion, Soter invokes BFC with the corresponding coverability instance. Finally, if BFC reports that a coverability instance is a no-instance, then we can conclude using Theorem 6 that the corresponding non-coverability constraint is never violated by the input program. Consequently, Soter reports that the non-coverability assertion is safe in this case. Otherwise, the analysis is inconclusive.

Now that we have given an overview of the workflow of Soter, we focus on how Soter explores the abstract state space and generates the ACS $\mathcal{A}_{\mathcal{P}}$ on-the-fly.

Exploring the Abstract State Space

The exploration of the abstract state space $\mathit{Reach}_{\hat{\mathcal{P}}}(\alpha_{\mathit{State}}(s_0))$ drives the generation of the ACS $\mathcal{A}_{\mathcal{P}}$. Every time a new active component $(\hat{\iota}, \hat{q}, \hat{q}')$ of a rule $\boldsymbol{R}$ is encountered in this exploration, an ACS rule $(\hat{\iota}, \hat{q}) \xrightarrow{\lambda} (\hat{\iota}, \hat{q}')$ is generated, where the label λ depends on the rule $\boldsymbol{R}$.

A Widening. We use the basic domains abstraction $\langle \widehat{\mathit{Data}}_0, \widehat{\mathit{Time}}_0, \widehat{\mathit{Mailbox}}_{\text{set}} \rangle$ in our implementation by default. With this choice many of the abstract domains collapse into singletons. Even in this case, the size of the abstract state space $\mathit{Reach}_{\hat{\mathcal{P}}}(\alpha_{\mathit{State}}(s_0))$ is exponential in the size of the input program $\mathcal{P}$. The reason for this is that each abstract state contains its own separate store, mailboxes, and mapping of pids to process states. Exploring $\mathit{Reach}_{\hat{\mathcal{P}}}(\alpha_{\mathit{State}}(s_0))$ exhaustively would thus lead to an exponential algorithm. Horn and Might advocate a widening to obtain an exploration algorithm that terminates in a polynomial number of steps [HM10, Section 7]. Instead of keeping separate stores, mailboxes, etc., we exploit the complete lattice structure of $\widehat{\mathit{State}}$ and apply a join. Thus, we keep just one global copy of each. The widening yields a new transition system $\sqcup\hat{\mathcal{P}} = (\widehat{\mathit{State}}, \rightarrow_{\sqcup\hat{\mathcal{P}}})$ where the transition relation is defined as $\hat{s} \rightarrow_{\sqcup\hat{\mathcal{P}}} \bigsqcup\{\hat{s}' \mid \hat{s} \rightarrow_{\hat{\mathcal{P}}} s'\}$. Note that already the transition relation $\rightarrow_{\hat{\mathcal{P}}}$ guarantees that any successor $\hat{s}'$ of an abstract states $\hat{s}$ (i.e. $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'$) satisfies $\hat{s} \leq_{\widehat{\mathit{State}}} \hat{s}'$. Hence, we can verify by inspecting the rules of the abstract operational semantics that the transition relation $\rightarrow_{\hat{\mathcal{P}}}$ satisfies the following diamond property: if $s \rightarrow_{\hat{\mathcal{P}}} s'$ and $s \rightarrow_{\hat{\mathcal{P}}} s''$, then $s' \rightarrow_{\hat{\mathcal{P}}} s' \sqcup s''$ and $s'' \rightarrow_{\hat{\mathcal{P}}} s' \sqcup s''$. The widening thus acts like a partial order reduction in our setting. Therefore, we can conclude that (i) $\sqcup\hat{\mathcal{P}}$ clearly simulates $\hat{\mathcal{P}}$, and (ii) exploring $\mathit{Reach}_{\sqcup\hat{\mathcal{P}}}(\alpha_{\mathit{State}}(s_0))$ instead of $\mathit{Reach}_{\hat{\mathcal{P}}}(\alpha_{\mathit{State}}(s_0))$ does not discover spurious active components $(\hat{\iota}, \hat{q}, \hat{q}')$. Further, the path in the abstract state space generated by $\rightarrow_{\sqcup\hat{\mathcal{P}}}$ forms an increasing chain in $\widehat{\mathit{State}}$. Note that the length of increasing chains in $\widehat{\mathit{State}}$ are polynomially bounded by the sizes of $\widehat{\mathit{Pid}}$, $\widehat{\mathit{ProcState}}$, and $\widehat{\mathit{Kont}}$. If we use a $\mathcal{D}_0$ data abstraction and a $\mathcal{T}_0$ time abstraction, the size of $\widehat{\mathit{ProcState}}$ and $\widehat{\mathit{Kont}}$ becomes polynomial and $\widehat{\mathit{Pid}}$ linear in the size of the input program[1].
Using the widening, we obtain an exploration algorithm SIMPLEEXPLORE that terminates in a number of steps that is polynomial in the size of the input program. We present SIMPLEEXPLORE in Figure 3.9. Given $\mathcal{P}$ we start the exploration from the state $\hat{s}_0 = \alpha_{\mathit{State}}(s_0)$. The ACS $\mathcal{A}_{\mathcal{P}}$ is generated on-the-fly. For every active component $(\hat{\iota}, \hat{q}, \hat{q}')$ that we discover, we add a number of rules to $\mathcal{A}_{\mathcal{P}}$. This number is bounded by $|\widehat{\mathit{Msg}}| + 1$. The runtime overhead of the ACS generation can thus be controlled by the choice of message abstraction. This is the motivation for letting the message abstraction be an independent choice. We can thus increase precision with respect to types of messages, which is computationally cheap, and keep the expensive precision on data as low as possible.

Considering a different choice than $\langle \widehat{\mathit{Data}}_0, \widehat{\mathit{Time}}_0, \widehat{\mathit{Mailbox}}_{\text{set}} \rangle$ as the basic domains abstraction can quickly lead to the need to explore an abstract state space that grows exponentially with respect to the size of the input. In particular, the size of $\widehat{\mathit{VAddr}}$ and thus $\widehat{\mathit{ProcState}}$ grows exponentially in the parameter D of the data abstraction $\widehat{\mathit{Data}}_D$.

[1] We assume that the maximal arity ar is a constant. Note that a currying transformation can be applied onto the input program to force $ar = 1$ which leads to a polynomial increase in the size of the program.

SimpleExplore$(\langle \hat{\pi}_0, \hat{\mu}_0, \hat{\sigma}_0 \rangle)$:
  $\hat{s} := \bot$;
  $\hat{s}' := \langle \hat{\pi}_0, \hat{\mu}_0, \hat{\sigma}_0 \rangle$;
  $R := \emptyset$;
  while $\hat{s}' \not\sqsubseteq_{\widehat{State}} \hat{s}$ **do**
    $\hat{s} := \hat{s}'$;
    for each $\hat{s}'_0$ such that $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'_0$ with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ and rule $\boldsymbol{R}$ **do**
      $\hat{s}' := \hat{s}' \sqcup \hat{s}'_0$;
      Determine λ *from* $\boldsymbol{R}$ *according to Definition 17;*
      $R := R \cup \left\{ (\hat{\iota}, \hat{q}) \xrightarrow{\lambda} (\hat{\iota}, \hat{q}'); \right\}$
  return $(\hat{s}, R)$;

Figure 3.9: The exploration algorithm **SimpleExplore**. Given a λ_{Actor} expression $\mathcal{P}$ we start the exploration from the state $\hat{s}_0 = \langle \hat{\pi}_0, \hat{\mu}_0, \hat{\sigma}_0 \rangle = \alpha_{State}(s_0)$.

A Worklist Algorithm. The algorithm **SimpleExplore** is unnecessarily inefficient. The fixpoint check $\hat{s}' \not\sqsubseteq_{\widehat{State}} \hat{s}$ is expensive to perform at every iteration, and all potential active components are re-examined at each iteration. The design of the abstract interpretation originates from the literature on higher-order control-flow analysis. The exploration of the abstract semantics is reminiscent of finding a solution to a data flow constraint system. In the context of flow analysis, a worklist algorithm called the *round robin algorithm* has been devised. The round robin algorithm takes the dependencies of the value- and control-flow of the program into account to speed up the computation of the fixpoint [NNH99]. Unfortunately, the round robin algorithm is based on the structure of the constraint system and cannot be directly applied in our setting [Joh+13]. However, we can seek inspiration from its design. In our setting, we can expose dependencies between parts of the abstract state. If a change is made e.g. in the store at an address $\hat{a}$, this has ramifications for all points at which $\hat{a}$ is involved at a lookup. We may thus maintain a frontier of the exploration as a worklist and repopulate it with a state whenever a dependency is triggered by an update. We note that in the abstract operational semantics (Figure 3.6 and Figure 3.7) there are easily visible dependencies:

(i) An abstract control state $\hat{q} = \langle v, \hat{\rho}, \hat{a}, \hat{t} \rangle$ that has arrived at an irreducible value v will perform a continuation lookup at $\hat{a}$. Hence, there is a dependency of $\hat{q}$ on $\hat{\sigma}(\hat{a})$. If the content of the latter changes we need to revisit $\hat{q}$.

(ii) A control state $\hat{q} = \langle x, \hat{\rho}, \hat{a}, \hat{t} \rangle$ has a dependency on $\hat{\sigma}(\hat{\rho}(x))$, i.e. the value part of the store. Control states $\hat{q}$ to which the rules **AbsCase**, and **AbsReceive** apply have similar dependencies on the store.

(iii) Further, a control state $\hat{q}$ at active pid $\hat{\iota}$ to which **AbsReceive** applies, depends on the contents of $\hat{\iota}$'s mailbox $\hat{\mu}(\hat{\iota})$.

Whenever our worklist algorithm algorithm explores a new active control state and discovers a dependency to a store address or a mailbox, it records this dependency in a lookup table *dependencies* of type $(\widehat{KAddr} + \widehat{VAddr} + \widehat{Pid}) \rightarrow \mathscr{P}[\widehat{Pid} \times \widehat{ProcState}]$. If the content at a store address or in a mailbox is updated, the dependent control states are added to the worklist for

WORKLISTEXPLORE$(\langle \hat{\pi}_0, \hat{\mu}_0, \hat{\sigma}_0 \rangle)$:
  worklist $:= [(\hat{\iota}_0, \hat{\pi}_0(\hat{\iota}_0))]$;
  $\hat{s} := \langle \hat{\pi}_0, \hat{\mu}_0, \hat{\sigma}_0 \rangle$;
  $R := \emptyset$;
  dependencies $:= \emptyset$;
  while *worklist* $\neq \epsilon$ **do**
    $(\hat{\iota}, \hat{q}) :=$ *dequeuemax*(*worklist*);
    for each $\hat{s}'_0$ *such that* $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'_0$ *with active component* $(\hat{\iota}, \hat{q}, \hat{q}')$ *and rule* $\boldsymbol{R}$ **do**
      $\hat{s} := \hat{s} \sqcup \hat{s}'_0$;
      Determine λ *from* $\boldsymbol{R}$ *according to Definition 17;*
      $R := R \cup \left\{ (\hat{\iota}, \hat{q}) \xrightarrow{\lambda} (\hat{\iota}, \hat{q}'); \right\}$
      if *the transition* $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'_0$ *dependends on store addresses* A **then**
        dependencies $:=$ *dependencies* $\sqcup [\hat{a} \mapsto \{(\hat{\iota}, \hat{q})\} \mid \hat{a} \in A]$;
      if *the transition* $\hat{s} \rightarrow_{\hat{\mathcal{P}}} \hat{s}'_0$ *is a receive transition* **then**
        dependencies $:=$ *dependencies* $\sqcup [\hat{\iota} \mapsto \{(\hat{\iota}, \hat{q})\}]$;
      newqs $:= \{\}$;
      if $\hat{q}' \notin \hat{\pi}(\hat{\iota})$ **then** *newqs* $:=$ *newqs* $\cup \{(\hat{\iota}, \hat{q}')\}$
      if $\lambda = \nu(\hat{\iota}'', \hat{q}''), \hat{q}'' \notin \hat{\pi}(\hat{\iota}'')$ **then** *newqs* $:=$ *newqs* $\cup \{(\hat{\iota}'', \hat{q}'')\}$
      if *address* $\hat{a}$ *is updated* **then** *newqs* $:=$ *newqs* $\cup$ *dependencies*$(\hat{a})$
      if *the mailbox of a* $\hat{\iota}''$ *is updated* **then** *newqs* $:=$ *newqs* $\cup$ *dependencies*$(\hat{\iota}'')$
      for each $(\hat{\iota}_1, \hat{q}_1) \in$ *newqs s.t.* $(\hat{\iota}_1, \hat{q}_1) \notin$ *worklist* **do**
        enqueue$((\hat{\iota}_1, \hat{q}_1),$ *priority*$(\hat{q}_1),$ *worklist*$)$;
  return $(\hat{s}, R)$;

Figure 3.10: The exploration algorithm **WORKLISTEXPLORE** which is used in Soter. Given a λACTOR expression $\mathcal{P}$ we start the exploration from the state $\hat{s}_0 = \langle \hat{\pi}_0, \hat{\mu}_0, \hat{\sigma}_0 \rangle = \alpha_{State}(s_0)$.

renewed exploration.

Revisiting a control state is necessary to discover new active components that are enabled by a change in a dependent store location or mailbox. In order to minimise the re-examination of a control state, it is profitable to delay a re-examination until all potential dependency updates have been performed [NNH99]. For this purpose, we employ a priority queue as a worklist. We note that control states that fire rules **AbsFunEval**, **AbsLetRec**, and **AbsSelf** have no need to be revisited, but add possibly new bindings in the store. Hence, we give such control states the highest priority. On the other side of the spectrum, we have control states $\hat{q}$ that fire the rules **AbsReceive** and **AbsCase** that may have deep, complicated dependencies in the store. We delay revisiting such control states for as long as possible. Of the remaining cases we favour control states that add bindings to the store or add messages to mailboxes. This prioritises control states of the form $\hat{q} = \langle v, \hat{\rho}, \hat{a}, \hat{t} \rangle$ over variable lookup control states $\hat{q} = \langle x, \hat{\rho}, \hat{a}, \hat{t} \rangle$. We implement this priority assignment in a simple function *priority*. We present our exploration algorithm **WORKLISTEXPLORE** in Figure 3.10. Note that in the worst case the input program can force **WORKLISTEXPLORE** to simulate **SIMPLEEXPLORE**. Thus, we cannot improve on the worst case complexity over **SIMPLEEXPLORE**. However, in practise we see significant improvements by

employing **WorklistExplore**.

Recent work published after the completion of Soter's implementation has conducted a systematic analysis of optimisations within the AAM framework in a sequential setting [Joh+13]. A frontier based semantics is proposed by Johnson et al. which gives rise to a similar exploration to ours. However, Johnson et al.'s approach does not make use of dependencies as we do. This may be due to the sequential setting. Revisiting control states is not necessary in Johnson et al.'s framework. In the concurrent case, a mailbox update may unlock a new receive branch which needs to be explored. The analysis is conservative and has to assume that at every program point one process may be waiting to execute. Thus, using dependencies to trigger such explorations is of importance to us. Our experimental evaluation (Section 3.8) shows that the dependency based worklist algorithm improves the runtime of the analysis significantly. Johnson et al. also advocate other optimisations such as the global store widening which we also adopt, imperative data structures, and abstract compilation. Abstract compilation reduces the administrative states introduced by evaluating the arguments of an application and can thus speed up the exploration. In the simplification step, we remove such administrative transitions from $\mathcal{A}_{\mathcal{P}}$. It may be fruitful to implement abstract compilation into our exploration algorithm as it could speed up the analysis and reduce the work of the simplification step.

The Size of $\mathcal{A}_{\mathcal{P}}$. The size of $\mathcal{A}_{\mathcal{P}}$ depends on the basic domain abstraction. The ACS $\mathcal{A}_{\mathcal{P}}$ has at most $|\widehat{ProcState}| \times |\widehat{Pid}|$ control states, $|\widehat{Msg}|$ messages and $|\widehat{Pid}|$ channels. Choosing the basic domain abstraction $\langle \widehat{Data}_0, \widehat{Time}_0, \widehat{Mailbox}_{\text{set}} \rangle$ ensures that the size of $\mathcal{A}_{\mathcal{P}}$ is polynomial in the size of the input program and the choice of $\widehat{Msg}$ (which is usually small). In Soter, coverability checks on $\mathcal{A}_{\mathcal{P}}$ are performed by translation to a Petri net $\mathcal{V}_{\mathcal{P}}$ with $(|\widehat{ProcState}| + |\widehat{Msg}|) \times |\widehat{Pid}|$ places. The simplification step on the ACS $\mathcal{A}_{\mathcal{P}}$ significantly reduces its size. We notice that many control states in the abstract operational semantics serve as intermediate states in functional reductions which may safely be collapsed. We conjecture that, after the reduction, the cardinality of $\mathcal{A}_{\mathcal{P}}$'s set of control states is quadratic only in the number of **send**, **spawn**, and **receive** of $\mathcal{P}$.

A Worked Example: A Concurrent Eratosthene's Sieve

We demonstrate the use of Soter on an example by Pike [Pik07]. The example is a concurrent implementation of Eratosthene's Sieve. Pike presents this program in a *Google Tech Talk* in the series *Advanced Topics in Programming Languages* to illustrate the concurrency model of the programming language *newsqueak*. Even though *newsqueak* features synchronous message passing, Pike's example is readily translated to Erlang.

We display such a translation in Figure 3.11 and Figure 3.12. In this example, we say a process implements a stream, if (i) it accepts request-messages of the form $\{\mathsf{poke}, \iota\}$, and (ii) it provides the next element d of the stream by sending a message $\{\mathsf{ans}, d\}$ back to the requestor ι.

The program consists of the functions `counter`, `sieve`, `filter`, `dump`, and `main`. At the start, `main` spawns two processes running `sieve` and `counter`. Both processes implement streams. The `counter` process implements the stream of integers starting from 2. The `sieve` process is connected to an incoming stream, initially the `counter` process. The `sieve` process relays any request-messages to this incoming stream and waits for an answer yielding a number p. Initially, the request

```
1  % Process: Pid(0), mailbox: {{prime, Z}}
2  main() ->
3      Me = self(),                              % Me = Pid(0)
4      Gen = spawn(fun()->counter(2)end),        % Gen = Pid97
5      spawn(fun()->sieve(Gen,Me)end),           % creates Pid37
6      dump().
7
8  dump() ->
9      receive
10       {prime,X} -> io:fwrite("~w~n", [X]),   % X = 2, N+1
11                    dump()
12     end.
13
14 % Process: Pid97, mailbox: {{poke, Pid37}, {poke, Pid56}}
15 counter(N) ->                                 % N = 2, N+1
16     ?label_mail("counter_mail"),
17     receive
18         {poke, From} ->                       % From = Pid37, Pid56
19             From ! {ans, N},
20             counter(N+1)
21     end.
```

Figure 3.11: Pike's concurrent implementation of Eratosthenes' Sieve translated to Erlang. We have annotated the code of functions main, dump, and counter with flow information that is computed by Soter.

yields the number 2 from the counter process. Before relaying the number p, the sieve process creates a new process running filter, equips it with a test that checks divisibility by p, and passes it its input stream. The sieve process then recurses and replaces its input stream by listening to the new filter process. As we would expect, the filter process relays request-messages to its input stream. However, it only relays an answer if it passes its Test. Otherwise, it continues to request messages from its input stream until a message can be relayed. Hence, whenever the sieve process receives a prime number from its input stream, it creates a filter process that is placed in between itself and the counter process. This process filters all multiples of the just received prime number. Since the first number generated by the counter process is 2, we obtain the familiar sieve of Eratosthenes algorithm. We note that we may view the system as a chain: on the one end the counter process, on the other the sieve process, and in between filter processes — one for each generated prime number.

We note that Soter provides only very limited support for arithmetic operations. We provide an encoding of natural numbers using Peano numbers. Modelling division is thus cumbersome. To overcome this, we provide the definition of the function divisible_by as follows:

```
51 -ifdef(SOTER).
52     divisible_by(X) -> fun(Y) -> ?any_bool() end.
53 -else.
54     divisible_by(X) -> fun(Y) -> case (Y rem X) of 0 -> true; _ -> false end end.
55 -endif.
```

This defines divisible_by as a function that returns true or false non-deterministically when run through Soter. Otherwise, its natural definition is supplied. The macro ?any_bool is actually implemented by a primitive **choice** that we introduce to encode a binary non-deterministic choice. We supply an implementation of **choice** as a random choice to the Erlang runtime. However, Soter interprets a call to **choice** as an actual non-deterministic choice.

```
22  % Process: Pid37, mailbox: {{ ans, Y}}
23  sieve (In, Out) →                                   % In  = Pid56, Pid97
24      ?label_mail ("sieve_mail"),                     % Out = Pid(0)
25      In ! {poke, self()},
26      receive
27          {ans, Z} →                                  % Z = 2, N+1
28              Out ! {prime, Z},
29              F = spawn(fun()→filter (divisible_by (Z), In) end),  % F = Pid56
30              sieve (F, Out)
31      end.
32
33  % Process: Pid56, mailbox: {{ ans, Y}, {ans, N}, {poke,Pid37}, {poke,Pid56}}
34  filter (Test, In) →                                 % Test = divisible_by (Z)
35      ?label_mail ("filter_mail "),                   % In   = Pid56, Pid97
36      receive
37          {poke, From} →                              % From = Pid37, Pid56
38              filter (Test, In, From)
39      end.
40
41  filter (Test, In, Out) →                            % Test = divisible_by (Z)
42      In ! {poke, self()},                            % In   = Pid56, Pid97
43      receive                                         % Out  = Pid37, Pid56
44          {ans, Y} →
45              case Test(Y) of                         % Y    = 2, N+1
46                  false → Out ! {ans, Y}, filter (Test, In);
47                  true  → filter (Test, In, Out)
48              end
49      end.
```

Figure 3.12: The code of functions sieve and filter of Pike's concurrent implementation of Eratosthenes' Sieve.

This implementation of Eratosthenes' Sieve was originally written for *newsqueak*'s synchronous message passing concurrency model. Hence, we would like to check that the translation to Erlang is faithful to this synchronous communication. One property we expect to hold is that the size of each mailbox is bounded by 1. We can write this as an non-coverability assertion for Soter. First, we need to label mailboxes, which can be done using the ?label_mail macro. We have placed mailbox labels at lines 16, 24, and 35. These macros mark the mailbox of any process running the functions counter, filter, and sieve. As a second step, we need to specify a *bad state* with non-coverability constraints:

```
-uncoverable("counter_mail > 1").
-uncoverable("filter_mail  > 1").
-uncoverable("sieve_mail > 1").
```

These directives define three non-coverability assertions. Each of them specifies a *bad state* as a configuration in which there is strictly more than one messages in a mailbox labelled by either counter_mail, filter_mail, or sieve_mail. In a similar fashion, we can label program points with the ?label macro. Non-coverability constraints can then express how many processes need to execute the labelled program locations to violate the assertion.

Soter produces various types of information when run on this example. It explores the abstract state spaces and generates an ACS that we display in Figure 3.11 and Figure 3.12. Inspecting the outcome of the abstract state space exploration, we obtain a store $\hat{\sigma}$, mailboxes $\hat{\mu}$, and processes $\hat{\pi}$. To illustrate the information we obtain, we have annotated the code with comments.

In lines 3, 4, 5, and 29 new processes are created. The abstract pids for the created processes are the program points of the creating spawn. To enhance readability, Soter displays pids by

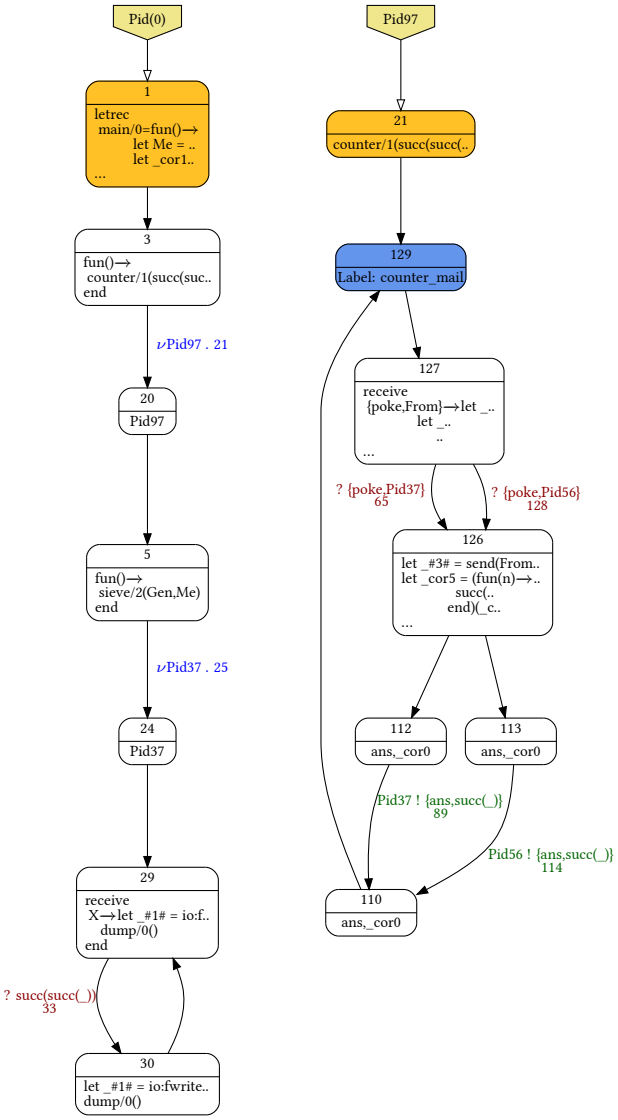


Figure 3.13: The rules corresponding to the abstract pids Pid (0) and Pid97 of the ACS that Soter generates for the sieve of Eratosthenes example.

computing a hash value. For example, the abstract pid running the sieve function can be identified with line 5. However, Soter displays it as Pid37 for simplicity. Since in this example it is easy to group the code which is executed by each abstract pid $\hat{\iota}$, we have placed a comment line just before the functions main and dump, counter, sieve, and filter noting the corresponding $\hat{\iota}$. Further, the comment contains the abstract mailbox $\hat{\mu}(\hat{\iota})$ of the respective abstract pid $\hat{\iota}$. For every variable x, we have included the possible abstract bindings $\hat{\sigma}(\hat{\rho}(x))$ as a comment at the end of the line. Since our basic domain abstraction uses a $\widehat{Time}_0$ abstraction, we note that environments are essentially identity mappings and we thus omit them.
The ACS presented in Figure 3.13 and Figure 3.14, which are actual outputs produced by Soter,

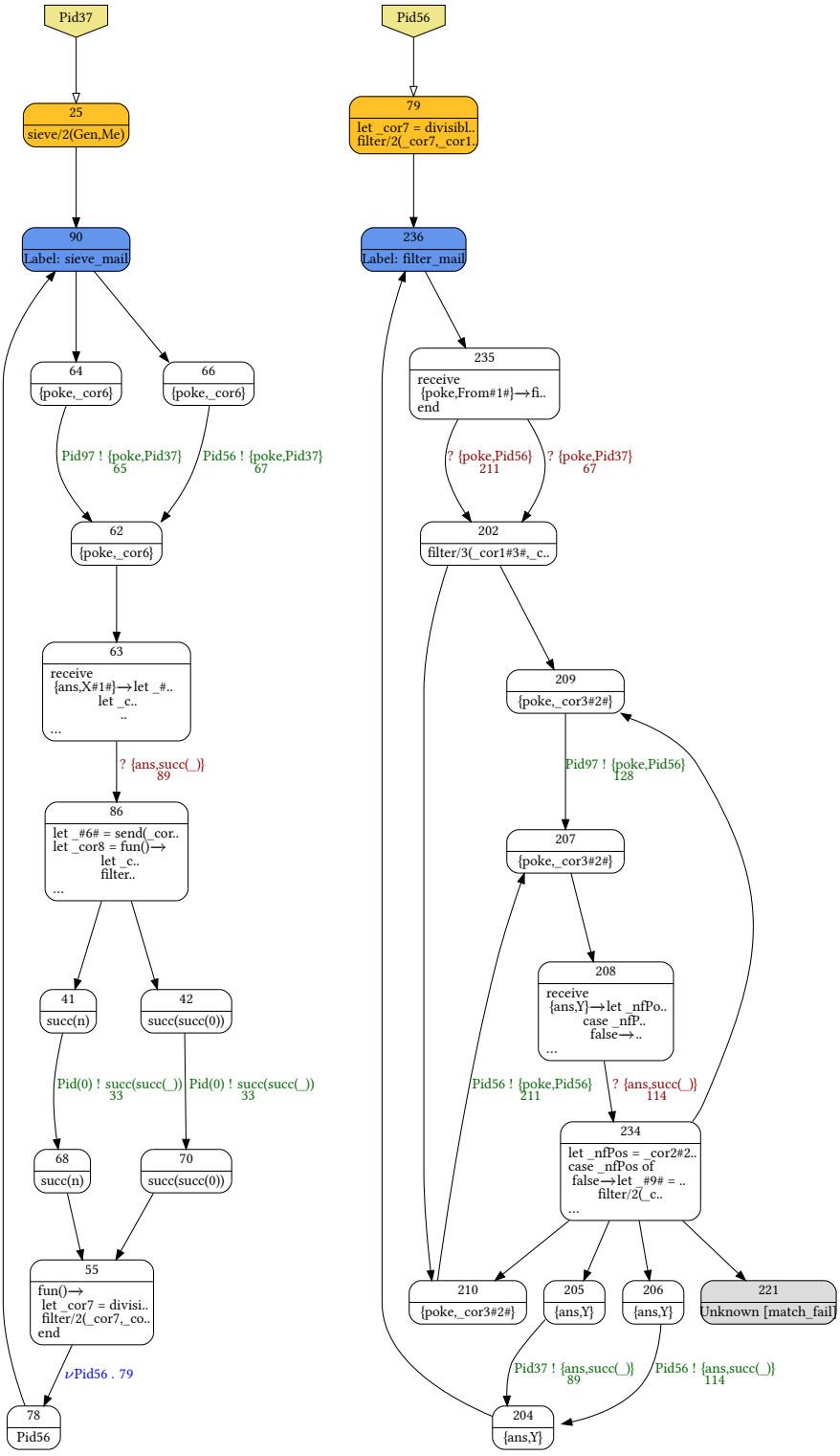


Figure 3.14: The rules corresponding to the abstract pids Pid37 and Pid56 executing sieve and filter respectively.

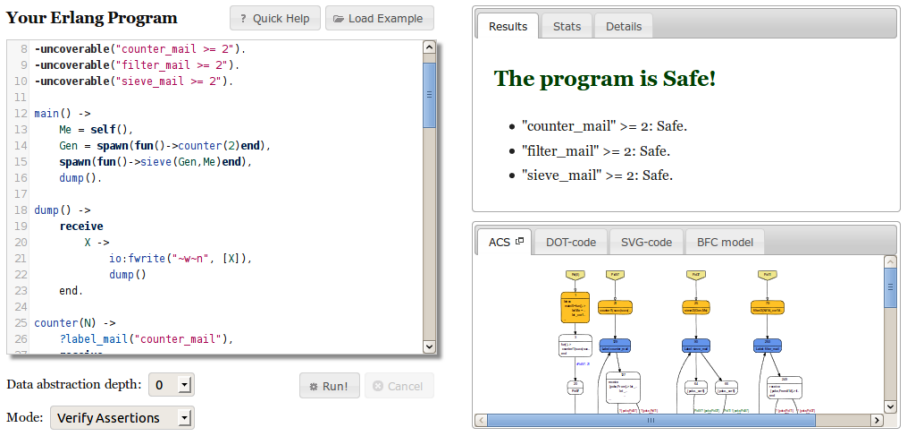


Figure 3.15: A screenshot of Soter's web interface reporting the verification outcome of the Eratosthenes' sieve example. Soter's web interface is available at http://mjolnir.cs.ox.ac.uk/soter/.

use a simplified display convention. The abstract pid associated with a connected component is indicated by an additional node above it rather than keeping an explicit tuple for every state. Further, in a receive label $\hat{\imath}\ ?\ \hat{m}$ we omit the abstract pid $\hat{\imath}$ which is understood to be the abstract pid of the connected component. Also, note that control states and messages are annotated with a unique number that corresponds to a place in the Petri net translation.

After the exploration of the abstract state space is complete, the ACS in Figure 3.13 and Figure 3.14 has been generated in its entirety. Together with the non-coverability constraints the ACS is translated to a Petri net coverability query. Soter invokes BFC to perform the model-checking and in the case of this example the results are no-instances. Hence, all non-coverability assertions are safe. A screenshot of Soter's web interface being run on the example presented in this section is displayed in Figure 3.15.

In the next section, we review related work and compare our verification procedure to other approaches from the literature.

3.7 Related Work

We start our survey of related work with the verification literature on Erlang. Many approaches have been investigated such as theorem proving, term rewriting techniques, abstract interpretation, model-checking, type systems, and static analysis. We present each of them in turn.

Verification of Erlang. The first contribution to Erlang's verification literature is a proof assistant due to Arts et al. The proof assistant allows the user to prove that an Erlang program satisfies a μ-calculus specification [Art+98]. Later work focussed on the verification of fixed properties for specific industrial Erlang systems. Arts et al. apply techniques for well-studied termination problems from the term-rewriting literature to verify a responsiveness property of

Erlang's Mnesia database [AD99] and the correct implementation of a telecommunication protocol [AG99; GA01].

Abstract Interpretation. Huch was the first to study an abstract interpretation of the operational semantics of Erlang [Huc99; Huc01; Huc02]. He develops an abstract interpretation framework for the purposes of model-checking LTL on the operational semantics generated by an Erlang program. His framework lifts a data abstraction to the entire state-space. However, other parts of the state-space remain untouched. Thus, Huch considers a restricted fragment of Erlang programs: (i) order-one, (ii) tail-recursive, (iii) mailboxes are bounded, and (iv) programs spawn an *a priori* bounded number of processes [Huc99]. These assumptions guarantee that the abstract state space is finite. Since algebraic data structures in Erlang may grow even when primitive data domains are finite, Huch proposes two data abstractions. The first is essentially the $\widehat{Data}_D$ abstraction we present in Section 3.4. The $\widehat{Data}_D$ abstraction disregards nested structures below depth D. The second data abstraction takes into account the patterns found in case and receive statements of the program. Terms are represented by the patterns they fit. However, this abstraction also imposes a restriction on the depth of nesting to arrive at a finite data abstraction. In subsequent work, Huch relaxes the tail-recursive requirement on functions to full recursion [Huc01; Huc02] by means of an abstraction that is reminiscent of a $\widehat{Time}_0$ abstraction. Huch's abstractions are designed to preserve path properties. Hence, if LTL model-checking the finite abstract model yields a safe-instance result, then the property holds for the input Erlang program.

A drawback of Huch's framework is the restriction to Erlang programs that allow only an *a priori* bounded number of processes with mailboxes of bounded size. Since our approach makes use of decidable infinite-state models such as ACS, we can overcome this restriction. Further, our approach naturally handles higher-order function definitions.

Model-Checking. *EtomCRL2* [AEP04; GD07; GDH08] provides a tool set that translates an Erlang program to the process algebra mCRL2. The process algebra μCRL, which supports algebraic data types, pattern-matching, first-order functions, and synchronisation of concurrent processes, is a precursor for mCRL2. The process algebra mCRL2 extends μCRL with higher-order functions among other things. Thus, both μCRL and mCRL2 are Turing powerful formalisms. EtomCRL2 is capable of automatically translating arbitrary Erlang code to mCRL2, but also provides specialised translation of commonly used implementation patterns, i.e. Erlang behaviours. Further, EtomCRL2 includes a discrete clock into the state space and can thus model time-outs [GD07]. The standard finite-state model-checker CADP [Cad] can then be used to verify an extensive set of properties such as LTL and the μ-calculus [Guo+10]. However, the EtomCRL2 verification process enumerates the entire state space before performing the verification process. Thus, it is unsuitable for the verification of infinite-state systems [Guo+10]. In order to guarantee termination in general, the input Erlang program needs to fall into a decidable fragment of mCRL2 which is supported by CADP.

McErlang is an explicit-state model-checker for Erlang programs developed by Fredlund and Svensson [FS07; EF12]. McErlang offers a high degree of flexibility. It replaces Erlang's runtime system with its own based on a formal operational semantics of distributed Erlang [Fre01; CS05]. The input program is instrumented to use McErlang's runtime system and then executed. The runtime system then ensures an on-the-fly depth-first model-checking traversal. Both properties

and abstractions may be provided to the runtime system as Erlang modules, however, McErlang's toolset also includes support for specifying properties such as Büchi automata or LTL formulæ. Recently, McErlang's runtime has been extended to handle a timed operational semantics which enables modelling of receive timeouts [EF12]. McErlang benefits from the fast Erlang runtime for functional reductions while retaining control over concurrent executions. Hence, it provides support for a substantial fragment of Erlang. However, since the exploration strategy is based on execution, McErlang's exploration may not terminate when the abstracted model is infinite-state. This poses a particular challenge, for example, when the number of processes of the modelled system cannot easily be bounded.

Type Systems. Marlow and Wadler were the first to propose a type system for a restricted functional fragment of Erlang [MW97]. Marlow and Wadler's type system provides decidable type inference and they suggest an extension of their type system to support concurrency features based on type and effect systems [TJ94]. In [Nys03] Nyström proposes a different, more expressive type-system at the cost of loosing type inference. Nyström's type-system uses a 0-CFA control-flow analysis for the purposes of type-checking. Concurrency features are type-checked against user annotations, i.e. the user needs to specify which messages a sub-program may send and receive. The type-system then verifies that the input program adheres to these specifications.

Sound for Error Detection Static Analyses. A different approach to static analysis of Erlang was initiated by Lindahl and Sagonas [LS04]. Lindahl and Sagonas developed a control-flow analysis based on set-based analysis [HJ94] for Erlang programs. The analysis is implemented in the tool Dialyzer which follows the design principle *sound for error-detection* [CS10], i.e. an analysis may only report a bug, if the bug is guaranteed to occur at runtime. While Dialyzer is successful in finding real bugs, its analyses cannot be used to establish correctness. Lindahl and Sagonas's analysis design philosophy is formalised in the notion of *success types* [LS05; LS06; SST13]. In contrast to traditional types, success types *'never disallows the use of a function that will not result in a type clash during runtime'* [LS06]. Success types provide a useful type notion for dynamically type languages. They are flexible enough to accept valid input programs while ruling out a large fraction of incorrect ones. Further, Lindahl and Sagonas' inference algorithm for success types is compositional and scales well.

Dialyzer has subsequently been extended by further targeted analyses. Even though Erlang is a purely functional language, the Mnesia database and some parts of the Erlang runtime may be accessed in an impure fashion. The race analysis of Christakis and Sagonas tries to statically detect such races [CS10]. Recently, Christakis and Sagonas designed an analysis for Dialyzer that detects common message passing errors. It is designed to find (i) receive statements that will never fire, and (ii) message sends that are never received by the recipient process.

Our analysis generates an ACS that simulates the operational semantics of the input program. Thus, we may say that our analysis extracts an abstract description of the *behaviour* of our input program. In the literature on type systems, *behavioural types* have been studied as appropriate types for processes.

Behavioural Types. In general, behavioural type systems are an instance of type an effect systems [NNH99]. However, an effect ϕ describes how a program $\mathcal{P}$ behaves, i.e. ϕ can be seen as a

computational model that simulates $\mathcal{P}$. Behavioural types were first introduced by Nielson and Nielson in a type system for *concurrent ML* (CML) [NN93]. The behaviours used in Nielson and Nielson's type system are terms of a process algebra that supports synchronous communication on channels and dynamic process creation. The idea of behavioural types has also been applied to the π-calculus [Yos96] to infer a simpler abstract process from a π-calculus term. Later type systems were designed to guarantee properties such as deadlock freedom [Kob97] and livelock-freedom [Kob00] for the typeable π-calculus terms. An expressive behavioural type system for the π-calculus that guarantees deadlock freedom and supports type inference is due to Kobayashi [Kob06]. The inference produces constraints on how channels can be used which are then checked for consistency. This check reduces to Petri net reachability [Kob06].

In contrast to the type system approach, our method takes two steps. First, we produce an abstract behavioural description, in the form of an ACS, for an arbitrary λ_{ACTOR} term. Then, we use the ACS to verify coverability properties. Since Erlang is a dynamically typed language, using an untyped static analysis seems to be the more natural approach. A type-sensitive control-flow analysis has been used by Reppy and Xiao to generate an extended control-flow graph for CML programs [RX07]. The generated extended control-flow graph is reminiscent of an ACS. Reppy and Xiao use this extended control-flow graph to detect communication patterns that can be specialised as part of a range of optimisations at compile time.

Other abstract interpretation frameworks have been proposed for concurrent functional languages and more abstract process calculi.

Abstract Interpretation. In the literature of AAM [MS06; MS08; HM11; MH11; VS10; EMH10; Ear+12] Might and Horn have extended the AAM methodology to concurrent Scheme [MH11]. The concurrency model is thread and shared-state based and allows synchronisations by a compare-and-swap primitive. In comparison, our abstract interpretation applies the AAM methodology to a message passing concurrency model. In particular, our concrete operational semantics is modelled after Fredlund's formalisation of single-node Core Erlang's operational semantic [Fre01].

Concurrent ML. In an abstract interpretation of CML, Colby proposes to represent process and channel identities by the a concurrent analogue of Shivers's contours, i.e. a process or channel is identified by the contour that creates it [Col95]. The analysis is *non-uniform*, i.e. it can distinguish between iterations in an infinite recursive communication pattern. This is made possible by abstracting the relationship of process contours, taking into account their common ancestor contour. However, Colby's analysis is specialised to detect which receives may match which message sends.

Process Calculi. Venet proposes an abstract interpretation framework for the π-calculus [Ven96]. His framework is specialised to allow abstractions of identities and name mobility. Further, Venet proposes rich abstractions that can be used for non-uniform analyses. Venet's framework was later generalised and extended by Feret to open π-calculus terms and other process algebras [Fer05]. Subsequently, Venet and Feret's abstract interpretation frameworks have been applied the the process calculus CAP by Garoche, Pantel, and Thirioux [GPT06]. CAP is a process calculus based on the actor model and features asynchronous message passing and actors as first class values. The abstractions of identifiers employed in this line of the literature are very

expressive and may provide a good addition to our work in order to improve the precision of abstract pids when required. Further, Feret's non-standard semantics can be seen as an alternative base framework to the AAM methodology, but tailored for process calculi.

In the next section, we present an empirical evaluation of Soter, before we discuss our contributions and the limitations of our verification procedure.

3.8 Discussion

In this section, we discuss our approach to verifying Erlang programs. We first present an empirical evaluation of Soter against a set of benchmarks. We then discuss the limitations of our method and offer perspectives on how they can be addressed. Lastly, we conclude this chapter with a summary of our contributions.

Empirical Evaluation of Soter

We evaluate Soter on a set of benchmark Erlang programs. The benchmarks consist of Erlang programs that are higher-order, use dynamic (and unbounded) process creation, and non-trivial synchronisation. The benchmark programs contain non-coverability assertions that require the verification of properties such as: (i) mutual exclusion of a critical section, (ii) non-coverability of a bad state, and (iii) checking explicit bounds on mailboxes.[2] As a whole, the set of benchmarks and the non-coverability properties that we verify lie outside of the scope of any comparable tool. The set of benchmarks feature non-trivial concurrent systems such as a firewall, a leader election protocol with a finite number of processes (finite_leader), and a generic resource server (reslock). We also include examples derived or extended from the literature: Pike's concurrent implementation of Erasthostenes' sieve [Pik07], a concurrent lookup table due to Huch (concdb) [Huc99], two examples setting up a pipe and ring topology (pipe and ring) inspired by Kobayashi et al. [KNY95], and a factory for stateful functions from Rosetta Code (state_factory). The annotated example programs in Table 3.1 can all be viewed and verified using Soter at the web interface `http://mjolnir.cs.ox.ac.uk/soter/`.

We evaluate the performance of Soter's worklist algorithm **WorklistExplore** against **SimpleExplore** as a baseline. In this choice we follow Johnson et al. who use the sequential analogue of **SimpleExplore** to evaluate the performance of the optimisations they propose [Joh+13]. We present our experimental results in Table 3.1. There are three observations we make from our experiments: (i) our abstractions are sufficiently precise to prove safety for a variety of non-trivial benchmarks, (ii) the simplification procedure is effective in reducing the size of the generated ACS, (iii) the worklist algorithm **WorklistExplore** provides a significant speedup over **SimpleExplore**, and (iv) despite the Expspace-completeness of the Petri net coverability problem the verification of our benchmarks is completed in seconds. A closer look at our experimental result yields that the simplification procedure yields an ACS whose size is

[2] Our benchmark suite has recently been included to evaluate the Petri net coverability checker Petrinizer [Esp+14]. Esparza et al. evaluated Petrinizer on coverability problems generated by Soter from our Erlang programs with various settings.

Example	LOC	P	R	ABS		ACS		TIME					
				D	M	Pl	Rat	WEA	SEA	SP	Smp	Bfc	Total
reslock	356	1	S	0	2	40	10%	0.59	7.23	12X	0.08	0.83	1.50
sieve	230	3	S	0	2	47	19%	0.26	2.35	9X	0.03	1.17	1.46
concdb	321	1	S	0	2	67	12%	0.99	8.57	9X	0.16	4.62	5.77
state_factory	295	2	S	0	1	22	4%	0.58	8.45	15X	0.13	0.48	1.19
pipe	173	1	S	0	0	18	8%	0.17	1.42	8X	0.03	0.05	0.25
ring	211	1	S	0	2	36	9%	0.53	3.29	6X	0.07	0.37	0.97
parikh	101	1	S	0	2	42	41%	0.05	0.23	5X	0.01	0.07	0.13
unsafe_send	49	1	U	0	1	10	38%	0.01	0.02	2X	0.00	0.00	0.01
safe_send	82	1	U*	0	1	33	36%	0.05	0.17	3X	0.01	0.00	0.06
safe_send	82	4	S	1	2	82	34%	0.21	0.65	3X	0.03	0.12	0.36
firewall	236	1	U*	0	2	35	10%	0.33	4.30	13X	0.05	0.16	0.54
firewall	236	1	S	1	3	74	10%	1.82	19.32	11X	0.30	0.06	2.18
finite_leader	555	1	U*	0	2	56	20%	0.40	1.90	5X	0.03	0.09	0.52
finite_leader	555	1	S	1	3	97	23%	0.70	3.95	6X	0.07	0.87	1.64
stutter	115	1	U*	0	0	15	19%	0.04	0.16	4X	0.00	0.00	0.05
howait	187	1	U*	0	2	29	14%	0.18	1.74	10X	0.02	0.05	0.25

Table 3.1: Results of the empirical evaluation of Soter. Column LOC reports the number of lines of Core Erlang code for each benchmark. Column P states the number of non-coverability assertions. In the R column, S means Soter can prove all non-coverability assertions safe, "U*" means the benchmark does not violate the assertions but verification was inconclusive; "U" Soter reports a genuine violation of an assertion. Columns D and M report the depths of the data and message abstractions. The column Pl states the number of places that are required to translate the generated ACS to a Petri net after the simplification. Column Rat reports on reduction on the ACS size achieved by the simplification as a ratio. Columns WEA and SEA report the time taken by **WorklistExplore** and **SimpleExplore** respectively. Column SP indicates the speedup gained by using **WorklistExplore**. Column Smp states the time taken to apply the simplification procedure. All times are in seconds.

on average[3] a sixth (16%) of the unsimplified one. The worklist algorithm provides an average speed up of a factor of 6.5X across our benchmarks. We would expect both of our optimisations to yield increasing benefits as the size of the input increases. If we exclude small input programs (LOC < 150) we observe that the simplification produces reductions of a factor of 9 (11%) and the speed up of the worklist algorithm increases to (almost) one order of magnitude (9X). Further, Bfc implements an algorithm that is Expspace-hard in the ACS size. Our experiments show there are other factors such as transition structure that strongly influence the runtime complexity of Bfc. Compare, for example, the time Bfc takes to perform the coverability checks on the concdb and finite_leader examples. Thus, the experimental outcomes encourages us to believe that infinite-state verification techniques can be applied to the verification of Erlang. However, in order to draw stronger conclusions, an evaluation on a set of more realistic benchmarks of larger size would be required. This requires more work to widen Soter's scope of applicability. There are obstacles of theoretical and practical nature that limit the scope of Soter. We elaborate on the limitations of our approach in the next section.

[3]All averages for ratios (i.e. speed ups and simplification-reduction ratio) are computed using the geometric mean.

Limitations

Practical Limitations. In order to make Soter applicable to larger real-word code bases, it is necessary to improve the support for more of Erlang's language features. λActor does not capture the module system, exception handling, arithmetic primitives, built-in data types, and I/O. The abstract interpretation framework could be extended without much difficulty and thus these features mainly represent an implementation challenge. However, providing this support would allow Soter to be applied to real-world Erlang code more readily. Modelling fault-tolerance features such as the monitor/link primitives require changes to the design of the concrete and abstract operational semantics, but could, in principle, be modelled by an ACS. Similarly, process registers, type guards, and timeouts can be integrated into the analysis.

Theoretical Limitations. The theoretical limitations concern the limits of expressivity of ACS and force us to choose abstractions that lose significant precision. From our perspective the choice of abstractions should be motivated by overcoming limitations of the abstract model. ACS allow us to model an unbounded number of finite-state processes communicating over a finite number of unbounded channels. Thus, our abstractions have to approximate three important capabilites: (C_1) the link between a process and its mailbox, (C_2) non tail-recursive function definitions, and (C_3) higher-order control-flow. We abstract the mailboxes of all processes with the same abstract pid into one unbounded, unordered channel. The difficulty lies in the dynamic creation of names that are first-class values and that can be used for communication. In order to model the link between a process and its mailbox, it seems the capability to dynamically create private names is necessary. However, this capability already allows us to encode the π-calculus and makes a potential abstract model Turing-powerful. Recently, progress has been made in isolating a rich fragment of the π-calculus called *depth-bounded* [Mey08]. The depth-bounded fragment allows the dynamic creation of private first-class names in a restricted fashion while retaining the decidability of coverability. In fact, a term of the depth-bounded π-calculus gives rise to a WSTS [Mey08] and could form the basis of more expressive abstract model than ACS that can lessen the need to approximate capability C_1.

In order to model non tail-recursive function definitions (C_2), we need to extend ACS to allow pushdown processes. If this extension is unconstrained, we would obtain a Turing-powerful model [Ram00]. However, there has recently been progress in the literature of concurrent pushdown systems. A decidable class of concurrent pushdown systems that can model an unbounded number of pushdown processes that may communicate over a finite number of unbounded, unordered channels has been identified. Processes are restricted to only receive messages when their stack is empty. This restriction is often referred to as the *empty-stack restriction* [SV06]. Coverability, among other decision problems, is decidable for concurrent pushdown systems that satisfy the empty-stack restriction [SV06] and a tight connection to Petri nets has been established [GM12] which implies many verification problems are in fact Expspace-complete. Further, the empty-stack restriction has been extended to other computational models such as concurrent higher-order pushdown systems [CV07]. The latter may form a basis for an extension of ACS that is capable of modelling restricted higher-order control flow (C_3). However, the empty-stack restriction is not entirely realistic for Erlang programs.

In the remainder of this dissertation, we present our work on extending ACS to allow pushdown processes with a restriction that goes beyond the empty-stack restriction. Before we em-

bark on this, we summarise the contributions we have presented in this chapter.

Summary In this chapter, we have presented a parametric abstract interpretation for λ_{ACTOR}, an idealised fragment of Core Erlang. We arrive at the abstract interpretation by applying the AAM methodology in the novel setting of message passing concurrency. We have shown how to use the abstract semantics to generate an ACS that simulates the operational semantics of the input program. ACS is an infinite-state model that can model the dynamic creation of finite state processes that communicate over a finite number of unbounded, unordered channels. Further, ACS are equivalent to Petri nets and thus algorithms for verification problems such as coverability exist. We propose to perform verification on Erlang programs via ACS and have implemented this verification approach in our tool Soter. Soter is the first verification tool for Erlang that makes use of infinite-state verification techniques. We have presented Soter's worklist algorithm to explore the abstract operational semantics and evaluated Soter, the worklist algorithm, and the ACS simplification procedure on a set of non-trivial benchmarks. The results of our empirical evaluation are encouraging. They suggest both the worklist algorithm and the ACS simplification are effective. Further, they demonstrate that infinite-state verification can be successfully applied to the verification of Erlang.

Chapter 4

Recursive Asynchronous Concurrency and the Shaped Stack Constraint

The work presented in this chapter is joint work with Luke Ong and has been published in [KO13].

In this chapter we investigate an extension of ACS that allows the modelling of restricted pushdown processes. In the verification method we present in Chapter 3, we need to abstract non tail-recursive function calls. This is a considerable source of imprecision. Following the AAM methodology, Vardoulakis and Shivers have shown that, in the sequential setting, abstract interpretations can be designed that lead to a pushdown system [VS10]. Such an analysis has the benefit that function calls and returns can be accurately matched in the abstract semantics. This leads to a substantially more precise abstract model of the input program, since values can be accurately returned to the correct call site [VS10]. Vardoulakis and Shivers's analysis could be extended to the concurrent case using suitable abstractions.

Hence, we are interested in *asynchronously communicating pushdown systems* (ACPS). ACPS are a model for dynamic networks of concurrent pushdown systems that communicate via a fixed, finite set of unbounded and unordered channels. ACPS are an extension of ACS in the sense that each process may be a pushdown process instead of a finite state process. Unfortunately, ACPS are Turing powerful [Ram00]. Thus, ACPS are in general unsuitable for algorithmic verification. However, we may look for a subclass of ACPS that retains good decision properties. Once such a suitable class has been identified, we could adapt our abstractions to approximate pathological non tail-recursive function definitions, i.e. function definitions that cannot be modelled. The abstract model could then be employed to model the benign recursive function definitions.

Of course, a subclass of ACPS that supports algorithmic verification is of much wider, independent interest. ACPS are a natural model for recursive programs that employ asynchronous communication. Hence, besides a wide range of concurrent and distributed systems, such as client-server environments and peer-to-peer networks, ACPS are a good model for programs written in event-driven programming style. Event-driven programming forms the basis of the implementation of many web-based applications, embedded systems, and sensor networks. Another popular programming paradigm that falls into the scope of ACPS is (task-based) asynchronous programming. Asynchronous calls, which return immediately, are often used to run

```
1  %%% Server
2  server () →
3    init_despatcher () , do_server() , post_task() ,
4    case (*) of
5      true → server () ;
6      false → system ? stop
7    end,
8    task_bag ! stop.
9
10 post_task() → task_bag ! task, task_bag ? ok.
11
12 init_despatcher () → task_bag ! init , task_bag ? ready.
13
14 despatcher() →
15   task_bag ? init , task_bag ! ready,
16   task_bag ? task, task_bag ! ok,
17   do_task() ,
18   case (*) of
19     true → despatcher() ;
20     false → task_bag ? stop, system ! despatcher_done
21   end.
22
23 main() → spawn(server), spawn(despatcher), system ! stop.
```

Figure 4.1: Server in asynchronous programming style. The procedures do_server and do_task may be arbitrary terminating recursive procedures that are allowed to send messages and spawn new processes.

work-intensive procedures concurrently. In recent years substantial advances have been made in the algorithmic verification of *asynchronous programs*, i.e. recursive programs with asynchronous atomic procedure calls [SV06; JM07; GMR09; GM12]. Asynchronous programs can be modelled naturally by ACPS subject to the *empty-stack restriction* [SV06]. The empty-stack restriction constrains a pushdown process to only receive messages when its stack is empty. ACPS satisfying the empty-stack restriction are closely related to Petri nets. For example, their respective coverability problems are polynomial-time interreducible [GM12]. Thus, many verification problems become Expspace-complete. Unfortunately, the empty-stack restriction is unrealistic for the purpose of verifying Erlang programs. One of our major contributions, that we present in this chapter, is the development of a more liberal restriction that permits concurrency and communication actions at any stack height, called the *shaped stack* constraint. An ACPS that satisfies the empty-stack constraint is guaranteed to satisfy the shaped stack constraint. Thus, the shaped stack constraint may be viewed as a relaxation of the empty-stack constraint and enables the safety verification of a large class of message passing programs.

Before we begin with the formal developments, let us give an intuitive explanation of ACPS and the empty-stack constraint, and compare it with the shaped stack restriction.

ACPS and the Empty Stack Constraint

ACPS are a general model for recursive programs with asynchronous concurrency and are an extension of ACS that allows each process to use a stack. In contrast to ACS, rules of an ACPS are of the form $(q, \gamma) \xrightarrow{\lambda} (q', \gamma')$ where γ and γ' are sequences of stack characters. Intuitively, the rule allows a process that is in control state q and can pop γ off its stack to make a transition to control state q', push γ', and perform the concurrency side-effect λ. Thus, a configuration $(q, \gamma \cdot \gamma'') \parallel \Pi \lhd \Gamma$ can transition to $(q', \gamma' \cdot \gamma'') \parallel \Pi' \lhd \Gamma'$ provided that Γ, Π', and Γ' are

$$
\begin{aligned}
&\text{server} \xrightarrow{\epsilon} \text{init\_despatcher do\_server post\_task } \mathrm{L}_4\ \mathrm{L}_8, \\
&\mathrm{L}_4 \xrightarrow{\epsilon} \text{server}, \qquad \mathrm{L}_4 \xrightarrow{\text{system?stop}} \epsilon, \\
&\mathrm{L}_8 \xrightarrow{\text{task\_bag!stop}} \epsilon, \\
&\text{post\_task} \xrightarrow{\epsilon} \mathrm{L}_{10:1}\ \mathrm{L}_{10:2}, \qquad \mathrm{L}_{10:1} \xrightarrow{\text{task\_bag!task}} \epsilon, \qquad \mathrm{L}_{10:2} \xrightarrow{\text{task\_bag?ok}} \epsilon, \\
&\text{do\_server} \xrightarrow{\epsilon} \ldots, \\
&\text{init\_despatcher} \xrightarrow{\epsilon} \mathrm{L}_{12:1}\ \mathrm{L}_{12:2}, \qquad \mathrm{L}_{12:1} \xrightarrow{\text{task\_bag!init}} \epsilon, \qquad \mathrm{L}_{12:2} \xrightarrow{\text{task\_bag?ready}} \epsilon, \\
&\text{despatcher} \xrightarrow{\epsilon} \mathrm{L}_{15:1}\ \mathrm{L}_{15:2}\ \mathrm{L}_{16:1}\ \mathrm{L}_{16:2}\ \text{do\_task}\ \mathrm{L}_{18}, \\
&\mathrm{L}_{15:1} \xrightarrow{\text{task\_bag?init}} \epsilon, \qquad \mathrm{L}_{15:2} \xrightarrow{\text{task\_bag!ready}} \epsilon, \\
&\mathrm{L}_{16:1} \xrightarrow{\text{task\_bag?task}} \epsilon, \qquad \mathrm{L}_{16:2} \xrightarrow{\text{task\_bag!ok}} \epsilon, \\
&\mathrm{L}_{18} \xrightarrow{\epsilon} \text{despatcher}, \qquad \mathrm{L}_{18} \xrightarrow{\epsilon} \mathrm{L}_{20:1}\ \mathrm{L}_{20:2}, \\
&\mathrm{L}_{20:1} \xrightarrow{\text{task\_bag?stop}} \epsilon, \qquad \mathrm{L}_{20:2} \xrightarrow{\text{system!despatcher\_done}} \epsilon, \\
&\text{do\_task} \xrightarrow{\epsilon} \ldots, \\
&\text{main} \xrightarrow{\epsilon} \mathrm{L}_{22:1}\ \mathrm{L}_{22:2}\ \mathrm{L}_{22:3}, \\
&\mathrm{L}_{22:1} \xrightarrow{\nu\text{server}} \epsilon, \qquad \mathrm{L}_{22:2} \xrightarrow{\nu\text{despatcher}} \epsilon, \qquad \mathrm{L}_{22:3} \xrightarrow{\text{system!stop}} \epsilon,
\end{aligned}
$$

Figure 4.2: Rules of the ACPS $\mathcal{P}$ implementing the task server program in Figure 4.1. Note the ACPS has a single control state q_0 that we omit to unclutter notation, i.e. instead of $(q_0, \gamma) \xrightarrow{\lambda} (q_0, \gamma')$ we write $\gamma \xrightarrow{\lambda} \gamma'$.

consistent with the side-effect λ. Since the capability to synchronise two stacks is enough to simulate a Turing machine, there is little hope to develop an algorithmic verification procedure for ACPS. In fact, in the general case any *'context-sensitive synchronization-sensitive analysis is undecidable'* [Ram00] for concurrent pushdown systems. It is surprising and fortunate that a decidable and meaningful class of ACPS can be captured by the empty-stack restriction. The empty-stack restriction says that a rule $r = (q, \epsilon) \xrightarrow{c?m} (q', \gamma')$ may only apply if the process' stack is empty. Hence, to satisfy the empty-stack restriction, a transition using the rule r may only be enabled in a configuration of the form $(q, \epsilon) \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]]$. There is a tight connection between ACPS that satisfy the empty-stack constraint and Petri nets [GM12]. Hence, a variety of verification problems are polynomial-time interreducible to decision problems on Petri nets. Coverability, for example, is thus Expspace-complete for ACPS satisfying the empty-stack restriction. However, the empty-stack constraint prohibits arbitrary synchronisations between processes, thus ruling out classes of interesting programs for analysis via ACPS.

Consider, for example, the program in Figure 4.1 which is presented in pseudo-code that

is reminiscent of Erlang. However, communication is over named, unordered, and unbounded channels. Thus, we write sends of a message *msg* to a channel *ch* as $ch\,!\,msg$, and receives of a message *msg* on a channel *ch* are denoted by $ch\,?\,msg$. The program spawns two processes, one running server, the other despatcher. The server process posts tasks to the channel task_bag. Tasks in this channel are continually removed by the despatcher process which then executes the selected tasks (possibly posting further tasks to the task_bag or spawning new processes), and then non-deterministically chooses to wait for the server process to send the stop message, or call itself recursively. If we inspect the synchronisations of the server and despatcher processes carefully, we observe that the procedures do_server and do_task cannot execute concurrently. We can thus consider a configuration in which they do as erroneous and ask the question whether such a configuration can be ever be reached. Another interesting question for this program is whether there is a moment during the execution when the messages ready and despatcher_done respectively are in the channels task_bag and system, which would be an error. Such a questions may be formulated as coverability problems. In Figure 4.2 we present ACPS $\mathcal{P}$ which is a model for the task server program and for which we can pose such a coverability problem. Note that we have chosen a translation that is reminiscent of a rewrite system. The ACPS has a single control state q_0 which we omit to unclutter notation. Each process uses its stack to remember which program points still need to be executed. The example run of $\mathcal{P}$ below shows how the main process spawns the server and despatcher process and progresses the server process until it blocks while waiting for the ready message in the task_bag channel:

$$\begin{aligned}
\mathsf{main} \lhd \emptyset &\to_{\mathcal{P}} \mathrm{L}_{22:1}\,\mathrm{L}_{22:2}\,\mathrm{L}_{22:3} \lhd \emptyset \\
&\to^*_{\mathcal{P}} \mathrm{L}_{22:3} \parallel \mathsf{server} \parallel \mathsf{despatcher} \lhd \emptyset \\
&\to_{\mathcal{P}} \mathrm{L}_{22:3} \parallel \mathsf{init_despatcher}\,\mathsf{do_server}\,\mathsf{post_task}\,\mathrm{L}_4\,\mathrm{L}_8 \parallel \mathsf{despatcher} \lhd \emptyset \\
&\to^*_{\mathcal{P}} \mathrm{L}_{22:3} \parallel \mathrm{L}_{12:2}\,\mathsf{do_server}\,\mathsf{post_task}\,\mathrm{L}_4\,\mathrm{L}_8 \parallel \mathsf{despatcher} \lhd [\mathsf{task_bag} \mapsto [\mathsf{init}]]
\end{aligned}$$

If we progress the despatcher process until it replies with the ready message, the server process can make a receive transition while it has a non-emtpy stack. Worse, each time the server process recurses a L_8 is added to its stack. Thus, receive transitions of the ready message by the server process occur with stacks of arbitrary height. Clearly, the task server program does not satisfy the empty-stack constraint. To prove algorithmically that the two undesired configurations

$$\emptyset \lhd [\mathsf{task\_bag} \mapsto [\mathsf{ready}]\,, \mathsf{system} \mapsto [\mathsf{despatcher\_done}]], \qquad \text{and} \qquad \mathsf{do\_server}\,\gamma \parallel \mathsf{do\_task}\,\gamma' \lhd \emptyset$$

are not coverable in $\mathcal{P}$ for some γ, γ', we need to consider a more expressive class of ACPS.

The empty-stack constraint appears restrictive when we consider Erlang programs. For example, standard programming patterns — *Erlang behaviours* —such as *finite-state machines*, *generic servers*, and *event-handlers* are provided to the programmer through a functional interface. The implementations of these patterns, which make extensive use of communication, are often parametrised by higher-order functions that can be arbitrarily instantiated by the programmer. For example, the generic server template needs to be supplied with code that handles client requests. This code may use further communication and may be mixed with other Erlang behaviours. These programming abstractions encourage programmers to use behaviours in a functional way and thus promote programs that do not satisfy the empty-stack constraint. Thus,

there is a need for a more liberal restriction than the empty-stack restriction such as the *shaped stack constraint*.

The Shaped Stack Constraint

Is it possible to relax the empty-stack restriction on ACPS while preserving the decidability of key verification problems? Fortunately, we can provide a positive answer to this question. The shaped stack constraint exploits the observation that processes have limited sensitivity to the order of concurrency events.

An Observation. Since channels are unordered, processes cannot observe the precise sequencing of concurrency actions such as message send and process creation. However, the sequencing of other actions, notably blocking actions such as message retrieval which requires synchronisation, is observable. To illustrate this, let us consider a sequence of concurrency actions performed by the server process:

$$(\mathsf{task\_bag}!\mathsf{init}\ \ \mathsf{task\_bag}?\mathsf{ready}\ \ \boldsymbol{\lambda}_{\mathsf{do\_server}}\ \ \mathsf{task\_bag}!\mathsf{task}\ \ \mathsf{task\_bag}?\mathsf{ok})^n\ (\mathsf{task\_bag}!\mathsf{stop})^n$$

where we assume $\boldsymbol{\lambda}_{\mathsf{do\_server}}$ is a sequence of sends and spawns produced by do_server. For illustration, suppose the function do_server sends log messages of its start and finishing onto the channel system, i.e.

$$\mathsf{do\_server}() \rightarrow \mathsf{system}\ !\ \mathsf{begin}, \mathsf{system}\ !\ \mathsf{end}.$$

From the perspective of any process receiving from channel system, the order in which the messages begin and end are sent is impossible to observe. Since the channel system is unordered, they may have been sent in any order. In particular, no process could distinguish the former definition of the function do_server from the following one

$$\mathsf{do\_server}() \rightarrow \mathsf{system}\ !\ \mathsf{end}, \mathsf{system}\ !\ \mathsf{begin}.$$

The situation is slightly different for the remainder of the messages sent by the server process. The despatcher process cannot see the order in which the messages init, task, and stop are sent to the task_bag channel. However, we know that the server process can only perform the send task_bag ! task after it has received the message ready. Thus, the relative order of a message send and a receive may be observed by another process.

Sensitivity to Order. How can we capture this sensitivity to order? Considering the sequence of concurrency actions, we may postulate that certain actions *commute* with each other (over sequential composition) while others do not. If we classify the concurrency actions message send and spawn as commutative and message receives as *non-commutative*, we obtain an equivalence relation on sequences of concurrency actions. Two sequences are equivalent if we can permute adjacent commutative concurrency actions to obtain one from the other. This equivalence relation captures our intuition that in a configuration we may substitute a process state (q, γ) by another (q', γ') that performs equivalent sequences of concurrency actions.

commutative ■	do_server, L_8, $L_{10:1}$, $L_{12:1}$, $L_{15:2}$, $L_{16:2}$, $L_{20:2}$, do_task, $L_{22:1}$, $L_{22:2}$, $L_{22:3}$, main
non-commutative ■	server, init_despatcher, post_task, L_4, $L_{10:2}$, $L_{12:2}$, despatcher, $L_{15:1}$, $L_{16:1}$, L_{18}, $L_{20:1}$

Figure 4.3: A parition of $\mathcal{P}$'s stack characters into commutative and non-commutative ones.

Exploiting Commutativity. In the case of the task server example we are fortunate. We can exploit the partition of concurrency actions into commutative and non-commutative to define a more liberal restriction than the empty-stack restriction. The ACPS $\mathcal{P}$ is in a *normal form* that guarantees that we can associate a set of sequences of concurrency actions to each stack symbol. When we rewrite a stack character to completion, one of these sequences of concurrency actions are performed. Let us say that a stack symbol is *commutative* if it produces only commutative concurrency actions and *non-commutative* otherwise. Intuitively, a stack character is *non-commutative* just if a blocking operation, such as receive, may be invoked when running it. In Figure 4.3 we present the partition of $\mathcal{P}$'s stack characters that arises from this classification. For example, the stack symbol L_8 just produces a message send. Hence, it is classified as commutative.

The Shaped Stack Constraint. For emphasis, let us mark a stack character in red if it is non-commutative and green otherwise. Colouring the last configuration of the example run in the previous section then yields:

$$L_{22:3} \parallel L_{12:2}\ \text{do\_server}\ \text{post\_task}\ L_4\ L_8 \parallel \text{despatcher} \lhd [\text{task\_bag} \mapsto \text{init}]$$

Notice that the colour pattern of each process fits a suffix of the shape ■■*■■*■■*. In fact, this configuration is representative. It is the case that all reachable processes of the task server example fit this shape. The *shaped stack constraint* formalises this pattern. For ACPS in normal form, the *K-shaped stack constraint*, where $K \geq 0$, limits the number of non-commutative stack symbols that may occur in any reachable process to no more than K. Thus, all reachable processes fit a suffix of the shape ■* $(■■^*)^K$. We can see that the task server example satisfies the 3-shaped stack constraint. It turns out that coverability is decidable for ACPS that satisfy the shaped stack constraint, i.e. they satisfy the K-shaped stack constraint for some K.

A Comparison. Stacks that fit the shape constraint may reach arbitrary heights while processes may perform concurrent actions (message-send, receive, or spawns) at any time, provided all reachable processes fit the K-shape constraint. To illustrate the expressivity we gain by moving from the empty-stack constraint to the shaped constraint, consider a receive transition of the server process:

$$L_{10:2}\ \text{do\_server}\ \text{post\_task}\ L_4\ L_8 \cdots L_8 \parallel L_{22:3} \parallel L_{16:1}\ L_{16:2}\ \text{do\_task}\ L_{18} \lhd [\text{task\_bag} \mapsto [\text{ready}]]$$
$$\rightarrow_{\mathcal{P}} \text{do\_server}\ \text{post\_task}\ L_4\ L_8 \cdots L_8 \parallel L_{22:3} \parallel L_{16:1}\ L_{16:2}\ \text{do\_task}\ L_{18} \lhd \emptyset$$

This transition would be impossible under the empty-stack constraint. To satisfy the latter the server process is forced to empty its stack before it can make a receive transition. Remembering

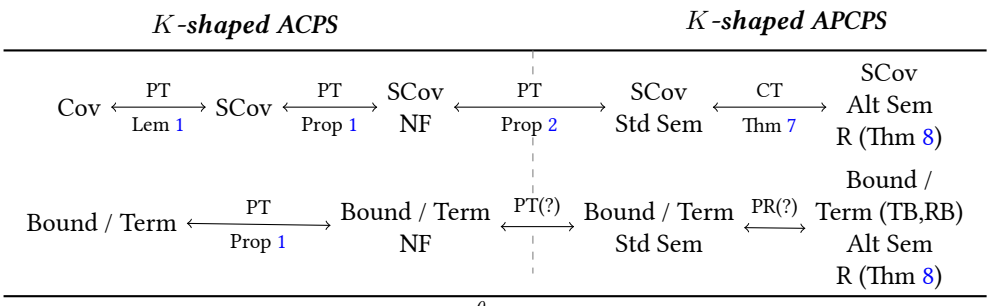


Figure 4.4: A map of this chapter's results. "$P_1 \xrightarrow{\theta} P_2$" means that (decision problem) P_1 θ-reduces to P_2. If the K-shaped restriction is dropped, the part of the graph excluding the rightmost column (Theorem 8) remains valid. *Legend: PT= polynomial time, CT= constant time, PR= primitive recursive time, R= decidable (recursive), NF= normal form, TB= termination bounded, RB= rewrite bounded, Std Sem= standard semantics, Alt Sem= alternative semantics, (?)= conjecture, Cov= coverability, SCov= simple coverability, Bound= boundedness, and Term= termination.*

to execute do_server () ,post_task () , to return to line 4 (L_4), and eventually to perform all recursive executions of line 8 (L_8) is not possible. In general, the K-shaped stack constraint allows a pushdown process to remember *infinite state* information along a receive transition, whereas the empty-stack constraint limits it to be *finite state*. Notice that the K-shaped ACPS model strictly extends the ACPS model with empty-stack restriction. In fact, the latter is guaranteed to satisfy the 1-shape constraint. For an ACPS that satisfies the empty-stack constraint all reachable processes fit either the shape 🟩*🟥 or 🟩*.

A Look Ahead. With the help of Figure 4.4, let us give a brief overview of the technical results presented in this chapter. As a first step, we formally introduce ACPS (Section 4.1) and show that it is possible to transform an arbitrary ACPS into *normal form* (Proposition 1). In normal form, it is evident that we may think of a pushdown process as a context-free grammar. Further, we formalise the notion of commutativity by means of an *independence relation*. Context-free grammars endowed with an independence relation are called *partially-commutative context-free grammars* (PCCFG) and form a good model for describing ACPS processes. In Section 4.2, we demonstrate how such context-free processes equipped with an *independence relation* give rise to a concurrent system which we call *asynchronous partially commutative pushdown systems* (APCPS). Further, we show how an ACPS gives rise to an APCPS and that coverability polynomial-time interreduces (Proposition 2). We define the shaped stack constraint (Definition 21) and present a sufficient easy-to-check syntactic condition (Proposition 4) which seems natural and readily satisfied by recursive programs humans write (Section 4.5). The independence relation of an APCPS enables us to simplify the operational semantics of APCPS. This leads to the definition of an *alternative semantics* for APCPS (Section 4.3). The standard semantics of an APCPS weakly bisimulates the co-universal powerset lifting of the alternative semantics (Proposition 3). Thus, we can use the alternative semantics to decide coverability (Theorem 7). We conjecture that we can answer termination and boundedness questions for APCPS on the alternative semantics (Conjectures 1 and 2). In Section 4.4, we present a major contribution of ours: the alternative semantics of a shaped APCPS gives rise to a strict WSTS. Hence, coverability and boundedness

are decidable, while termination is decidable for two subclasses of shaped APCPS (Theorem 8).

4.1 Recursive Asynchronous Concurrency

In this section, we formally introduce ACPS as a model of recursive asynchronous concurrency. Concurrent pushdown systems as the prototypical model for recursive concurrent programs have been extensively studied within the setting of different models of concurrency. For example, concurrent pushdown systems communicating over a shared state with locks [Kah09a], with hierarchical communication [BMOT05], synchronous message passing [EG11], and asynchronous message passing over FIFO channels [Heu+10] have been considered in the literature.

As in the case of ACS, we opt for a model of concurrency with asynchronous message passing over a fixed finite number of unbounded, unordered channels. Unordered message passing is a more natural fit for the potentially distributed systems that provide weak ordering guarantees, which arise in the verification of Erlang. Such concurrent pushdown systems with asynchronous concurrency have previously been studied [SV06; JM07; GMR09; GM12] which led to the introduction of the empty-stack constraint. Let us now introduce the formal definition of asynchronously communicating pushdown systems.

Definition 18. An *asynchronously communicating pushdown system* (ACPS) $\mathcal{P}$ is a quintuple $\mathcal{P} = (\mathcal{Q}, \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R})$ composed of a set of *control states* $\mathcal{Q}$, a *stack alphabet* $\mathcal{A}$, a set of *channel* names *Chan*, a set of *messages Msg*, and a set of *rules* $\mathcal{R}$. The set of rules $\mathcal{R}$ is a finite subset of $\mathcal{Q} \times \mathcal{A}^* \times \Lambda \times \mathcal{Q} \times \mathcal{A}^*$ where

$$\Lambda := \{c \,?\, m, c \,!\, m, \nu(q, \beta) : c \in \mathit{Chan}, m \in \mathit{Msg}, q \in \mathcal{Q}, \beta \in \mathcal{A}^*\} \cup \{\epsilon\}$$

is the set of *communication side-effects*. We denote a rule $(q, \beta, \lambda, q', \beta') \in \mathcal{R}$ by $(q, \beta) \xrightarrow{\lambda} (q', \beta')$.

The communication side-effects remain virtually unchanged from the ACS setting. An action λ of the form $c \,!\, m$ denotes a message send of m to channel c, $c \,?\, m$ denotes a receive of m from channel c, and $\nu(q, \beta)$ denotes a spawn of a new process that begins execution from (q, β). An ACPS $\mathcal{P}$ gives rise to a transition system $\mathcal{P} = (\mathit{State}, \rightarrow_{\mathcal{P}})$ with the potentially infinite state space

$$\mathit{State} = \mathbb{M}[\mathcal{Q} \times \mathcal{A}^*] \times (\mathit{Chan} \to \mathbb{M}[\mathit{Msg}]).$$

We use a similar notation for configurations as we did for ACS configurations. Hence, we write $(q, \beta) \parallel (q', \beta') \lhd [m_a, m_b, m_b]^{c_1} \oplus []^{c_2}$ for $([(q, \beta), (q', \beta')], \{c_1 \mapsto [m_a, m_b, m_b], c_2 \mapsto []\})$. As before, we abbreviate a set of processes running in parallel as Π and a set of channels by Γ with names in *Chan*. The transition relation $\rightarrow_{\mathcal{P}}$ is defined as follows: suppose $(q, \beta) \xrightarrow{\lambda} (q', \beta') \in \mathcal{R}$ and $\beta_0 \in \mathcal{A}^*$ then

$$\begin{aligned}
&(q, \beta\beta_0) \parallel \Pi \lhd \Gamma \rightarrow_{\mathcal{P}} (q', \beta'\beta_0) \parallel \Pi \lhd \Gamma && \text{if } \lambda = \epsilon,\\
&(q, \beta\beta_0) \parallel \Pi \lhd \Gamma \rightarrow_{\mathcal{P}} (q', \beta'\beta_0) \parallel (q_1, \beta_1) \parallel \Pi \lhd \Gamma && \text{if } \lambda = \nu(q_1, \beta_1),\\
&(q, \beta\beta_0) \parallel \Pi \lhd \Gamma \rightarrow_{\mathcal{P}} (q', \beta'\beta_0) \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]] && \text{if } \lambda = c \,!\, m,\\
&(q, \beta\beta_0) \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]] \rightarrow_{\mathcal{P}} (q', \beta'\beta_0) \parallel \Pi \lhd \Gamma && \text{if } \lambda = c \,?\, m.
\end{aligned}$$

Inspecting the operational semantics, it is immediate that ACS are subsumed by ACPS. However, ACPS are strictly more expressive. Every process has its own local stack in addition to its control state. The effects of communication actions along a transition, however, remain the same.

We augment ACPS with an order to yield a PSTS as follows. We define an order on processes by $(q, \beta) \leq_{\mathcal{Q}\times\mathcal{A}^*} (q', \beta')$ just if $q = q'$ and either $\beta = \epsilon$ or $\beta = A\cdot\beta_0, \beta' = A\cdot\beta'_0$ and $\beta_0 \leq_{\text{Hig}} \beta'_0$, i.e. we can obtain β_0 from β'_0 by erasing stack characters. Configurations are then ordered using the usual multiset and function extension: $\Pi \lhd \Gamma \leq_{ACPS} \Pi \lhd \Gamma$ just if $\Pi \leq_{\mathbb{M}[\mathcal{Q}\times\mathcal{A}^*]} \Pi'$ and if $\Gamma(c) \leq_{\mathbb{M}[Msg]} \Gamma'(c)$ for all $c \in \mathit{Chan}$.

Restrictions to Overcome Undecidability. With this choice of order, ACPS are PSTS and hence the decision problems coverability, reachability, termination, and boundedness are well-defined. However, we know that unconstrained ACPS are Turing powerful. Ramalingam studied static analyses on concurrent recursive programs that may perform very simple rendezvous synchronisations [Ram00]. In his setting, processes may have many possible execution paths that are constrained by synchronisation. An analysis that only considers execution paths that are valid with respect to synchronisations is called *synchronisation-sensitive*. Similarly, the call-stack or evaluation context of a process influences the possible execution paths. The call-stack determines the return points after a procedure call completes. An analysis that is accurate with respect to the constraints imposed by evaluation context or call-stack is referred to as *context-sensitive*. Ramalingam's result show that there is no computable analysis that is both context-sensitive and synchronisation-sensitive [Ram00]. These undecidability results carry over to ACPS since rendezvous synchronisations are easily encoded. Thus, the expressivity gained by combining the stack and synchronisation has to be tamed. The literature on concurrent pushdown systems may thus be placed onto a spectrum. On the one side of the spectrum constraints on synchronisations are imposed and on the other restrictions on the expressivity of the stack are applied. On the synchronisation-sensitive extreme we may place ACS, on the context-sensitive one we have non-interactive pushdown systems. On the context-sensitive side various under-approximation approaches have been successfully developed such as *context-bounded* [QR05], *phase-bounded* [BE12b], and *scope-bounded* model-checking [TN11]. Under-approximation methods are often formulated as relativised, bounded decision problems that apply to all ACPS. Context-bounded model-checking only considers execution paths in which an *a priori* fixed number of context-switches occur. The stack of each process is thus unconstrained, but the limit on context-switches enforces a limit on how many synchronisations can be performed. Phase-bounded model-checking considers message passing concurrency and categorises each process as either a *sender* or *receiver*. The phase bound limits the number of times processes may switch between sending and receiving mode along any considered execution path. Scope-bounded model-checking only considers execution paths along which the pushes and pops of stack characters satisfy the following property: a stack character that is pushed onto a process' stack is either popped within a bounded number of context switches or is never popped.

By contrast, the several restrictions that apply to the stack are more commonly formalised as subclasses of ACPS. An ACPS $\mathcal{P}$ satisfies the *empty-stack restriction* from an initial configuration $\Pi_0 \lhd \Gamma_0$, if for all paths $\boldsymbol{s}_0$ originating from $\Pi_0 \lhd \Gamma_0$ the following property holds: If for some i the transition $\boldsymbol{s}(i) = (q, \beta) \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} (q', \beta') \parallel \Pi \lhd \Gamma' = \boldsymbol{s(i+1)}$ uses a receive rule

where the process (q, β) is active, then $\beta = \epsilon$. This subclass was initially proposed by Sen and Viswanathan to model programs with *asynchronous atomic procedure calls* [SV06]. In contrast to the constraints imposed by context-bounded, phase-bounded, etc., the empty-stack restriction imposes less of a restriction onto synchronisations and more onto the stack. A restriction dual to the empty-stack restriction has been proposed by Heußner et al. for ACPS communicating over FIFO channels [Heu+10]. A process may only *send* a message when its stack is empty, but may receive at any time. Another restriction has been proposed by Babic and Rakamaric in the context of FIFO channels. They consider processes that are visibly pushdown [BR13]. In general, the visibly pushdown restriction is defined relative to an input alphabet which is partitioned into push, pop, and τ terminals. A visibly pushdown automata may (i) perform a push of a stack character while consuming a push input terminal, (ii) pop a stack symbol while consuming a pop terminal, or (iii) perform a stack neutral transition while reading an τ terminal. This closely relates the stack behaviour of a visibily pushdown automaton to the input word. In Babic and Rakamaric's model the input words are generated from channel contents. Thus, processes are visibly pushdown transducers.

Let us now develop two ways in which we can simplify the reasoning about ACPS. One concerns a simplification of the coverability problem, the other fixes a normal form for ACPS.

Simple Coverability. In the sequential setting of (ordinary) pushdown systems, the control-state reachability problem is of central interest. It asks, given a control-state q, if it is possible to reach a process-configuration (q, γ) where γ is some stack.

In the concurrent setting, we wish to know whether, given an ACPS and control states $q_1, \ldots, q_n$, do there exist stacks $\gamma_1, \ldots, \gamma_n$ and channel contents Γ such that the configuration $(q_1, \gamma_1) \parallel \cdots \parallel (q_n, \gamma_n) \parallel \Pi \lhd \Gamma$ is reachable for some parallel composition of processes Π. It is immediate that we can phrase this question as a coverability problem of a restricted form.

Decision Problem (Simple Coverability). Let $\mathcal{P}$ be an ACPS and $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ be a coverability query. We say the query $\mathcal{Q}$ is *simple*, if all processes (q, β) of Π and Π_0 satisfy $\beta \in \{\epsilon, A \mid A \in \mathcal{A}\}$. We refer to the coverability problem restricted to simple queries as *simple coverability*.

We give ourselves a bit more flexibility in the definition of simple coverability and allow the query to also contain processes of the form (q, A). This enables the definition to remain expressive enough for ACPS in normal form which we will formally define in the following. Note that simple coverability allows us to express not just control-state reachability queries but also mutual exclusion properties. *Prima facie* it appears that simple coverability is an easier problem. Indeed, we are motivated to define simple coverability by the fact that it simplifies many of our arguments. However, it turns out that for the choice of order $\leq_{ACPS}$ coverability and simple coverability are equivalent:

Lemma 1. Coverability and simple coverability for ACPS are polynomial-time interreducible.

Proof Idea. Given an ACPS $\mathcal{P}$, we note that we can add a polynomial number of rules and control states that enable each ACPS process $(q, A\,\beta)$ to process its own stack and test whether $\beta \geq_{\text{Hig}} \beta_{cov}$ for a given β_{cov}. The process can then record this in its local state q. We may

thus encode a coverability query $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ into $\mathcal{P}$ by allowing each process to non-deterministically switch to this processing mode. We then obtain a coverability query $(\mathcal{P}', \Pi_0 \lhd \Gamma_0, q_1^{cov} \parallel \cdots \parallel q_n^{cov} \lhd \Gamma)$ which is a yes instance if, and only if, $\mathcal{Q}$ is a yes-instance. Note that with a polynomial number of rules it is possible to set up the configuration $\Pi_0 \lhd \Gamma_0$ from a configuration $(q_0, \epsilon) \lhd \Gamma_0$. Hence, we obtain a simple coverability query in polynomial time.
For a full formal proof we refer the reader to Appendix B.1. □

Example 9. Consider the task server program in Figure 4.1 and the ACPS $\mathcal{P}$ implementing it in Figure 4.2. The erroneous configuration in which, simultaneously, the message ready is in the channel task_bag and the message despatcher_done is in the channel system can be formalised as the coverability problem

$$\mathcal{Q}_1 = (\mathcal{P}, (q_0, \mathsf{main}) \lhd \emptyset, \emptyset \lhd [\mathsf{task\_bag} \mapsto [\mathsf{ready}], \mathsf{system} \mapsto [\mathsf{despatcher\_done}]]).$$

Further, the situation in which two processes execute do_server and do_task concurrently can be expressed by the coverability query

$$\mathcal{Q}_2 = (\mathcal{P}, (q_0, \mathsf{main}) \lhd \emptyset, (q_0, \mathsf{do\_server}) \parallel (q_0, \mathsf{do\_task}) \lhd \emptyset).$$

Note that both $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are simple queries.

A Normal Form for ACPS. It is a well known fact (see Proposition 1) that control-states may be encoded in an enlarged stack-alphabet and that one may w.l.o.g. consider ACPS in *normal form*:

Definition 19. We say an ACPS $\mathcal{P}$ is in *normal form* if there is a distinguished control-state q_0 such that for all rules $(q, \beta) \xrightarrow{\lambda} (q', \beta')$ of $\mathcal{P}$ the following conditions are met: (i) $q = q' = q_0$, (ii) $\beta = A \in \mathcal{A}$, (iii) if $\lambda \neq \epsilon$ then $\beta' = \epsilon$, otherwise $\beta' \in \{\epsilon, B\,C : B, C \in \mathcal{A}\}$, and (iv) if $\lambda = \nu(q'', \beta'')$ then $q'' = q_0$, and $\beta'' \in \mathcal{A}$.

As in the case of ordinary pushdown systems, we do not loose expressivity by only considering ACPS in normal form:

Proposition 1. Given an ACPS $\mathcal{P}$, a simple coverability query $\mathcal{Q}$, and a $\Pi^0 \lhd \Gamma^0$ there exist an ACPS $\mathcal{F}_{\text{nf}}(\mathcal{P})$ in normal form, a simple coverability query $\mathcal{F}_{\text{nf}}(\mathcal{Q})$, and $\mathcal{F}_{\text{nf}}(\Pi^0 \lhd \Gamma^0)$ — all polynomial-time computable — such that: $\mathcal{Q}$ is a yes-instance if, and only if, $\mathcal{F}_{\text{nf}}(\mathcal{Q})$ is a yes-instance; and $\mathcal{P}$ is bounded (terminating) from $\Pi^0 \lhd \Gamma^0$ if, and only if, $\mathcal{F}_{\text{nf}}(\mathcal{P})$ is bounded (terminating respectively) from $\mathcal{F}_{\text{nf}}(\Pi^0 \lhd \Gamma^0)$.

Proof Idea. Since ACPS subsume ACPS in normal form one direction is trivial.

To reduce an ACPS to normal form we can essentially apply the same steps as we have in Section 2.4 to obtain a pushdown system in *Chomsky normal form*. We need to take care in the reduction to separate a communication-side effect of a rule from a possible stack action. The transformation is easily seen to be polynomially-time computable. As in the transformation in Section 2.4 a process state $(q, A_1 \cdots A_n)$ can be represented by a set of states of the shape

$(q_0, A_1^{q,q_1} A_2^{q_1,q_2} \cdots A_n^{q_{n-1},q_n} \Theta^{q_n,q_{n+1}})$ in the ACPS in normal form. It is easy to see that a co-universal powerset lifting of the transformed ACPS can weakly bisimulate the original ACPS. From this bisimulation, it is easy to see that simple coverability, boundedness, and termination polynomial-time interreduce.
A detailed proof of this proposition can be found in Appendix B.1. □

Henceforth, we shall elide the single control-state q_0 and write a rule r simply as $\beta \xrightarrow{\lambda} \beta'$. There is a well-known connection between pushdown systems and *context-free grammars* (CFG) (see Section 2.4). It is trivial to see that the transitions of a single ACPS process in normal form are (essentially) the left-most derivations of a CFG.

Let us fix an ACPS $\mathcal{P} = (\{q_0\}, \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R})$ in normal form. From $\mathcal{P}$ we obtain a CFG $\mathcal{G}(\mathcal{P}) = (\Sigma(\mathcal{P}), \mathcal{N}(\mathcal{P}), \mathcal{R}(\mathcal{P}))$ whose non-terminals $\mathcal{N}(\mathcal{P})$ are the stack alphabet $\mathcal{A}$, the set of terminals $\Sigma(\mathcal{P})$ is essentially Λ but for brevity we replace $\nu(q, A)$ by νA; the rules $\mathcal{R}(\mathcal{P})$ are obtained by the mapping:

$$A \xrightarrow{\nu A'} \epsilon \;\mapsto\; A \to \nu A' \qquad A \xrightarrow{c!m} \epsilon \;\mapsto\; A \to c\,!\,m$$
$$A \xrightarrow{\epsilon} \epsilon \;\mapsto\; A \to \epsilon \qquad A \xrightarrow{c?m} \epsilon \;\mapsto\; A \to c\,?\,m$$
$$A \xrightarrow{\epsilon} B\,C \;\mapsto\; A \to B\,C$$

Among other things, the CFG $\mathcal{G}(\mathcal{P})$ may be seen to formalise the set of sequences of concurrency actions that we can identify with each stack character. With each stack character A we associate the language $\mathcal{L}(A)$ of the non-terminal A generated by $\mathcal{G}(\mathcal{P})$, i.e. $\mathcal{L}(A) = \{\boldsymbol{\lambda} \in \Sigma^* : A \to^* \boldsymbol{\lambda}\}$.

4.2 Asynchronous Partially Communicating Pushdown Systems

In moving from the operational view of pushdown systems to context-free grammars, it is possible to exploit and to build upon known results on how CFGs interact with reorderings in the words they produce. For this purpose, we introduce a hybrid model, *asynchronous partially commutative pushdown systems*. Processes are modelled by a variant of context-free grammars which distinguish commutative and non-commutative concurrency actions. In our setting channels are unordered and a new process may start at its own leisure: we note that the precise execution order of concurrency side-effects such as send and spawn is immaterial. However, the sequencing of other side-effects, notably message retrieval which requires synchronisation, is observable.

Exploiting Commutativity

The use of an *independence relation* is a common technique to formalise such sensitivity to order. An independence relation I over a set U is a symmetric irreflexive relation over U. It induces a congruence relation $\simeq_I$ on U^* defined as the least equivalence relation $\mathcal{R}$ satisfying the following two conditions: (i) $(a, b) \in I \implies (a\,b, b\,a) \in \mathcal{R}$ and (ii) $(\mu, \mu') \in \mathcal{R} \implies \forall \nu_0, \nu_1 \in U^* : (\nu_0\,\mu\,\nu_1, \nu_0\,\mu'\,\nu_1) \in \mathcal{R}$. An element $u \in U$ is said to be *non-commutative wrt to* I if $(u, u') \notin I$ for all $u' \in U$, i.e. u does not commute with any other element. We denote the set of non-commutative elements by $U^{\neg\mathrm{com}}$. Similarly, we call a $u \in U$ *commutative wrt to* I if $(u, u') \in I$

for all $u' \in U \setminus U^{\neg\mathrm{com}}$, i.e. a commutative element commutes with all elements except non-commutative ones. We write U^{com} for the set of commutative elements in the following. An independence relation I that partitions U into commutative and non-commutative elements is said to be *unambiguous*.

Since receive is the only blocking concurrency action, we shall define an independence relation I that allows us to commute all concurrency actions *except* receive. Let Σ be an alphabet and $\Xi \subseteq \Sigma$. We define the independence relation over Σ generated by Ξ as

$$\mathit{IndRel}_{\Sigma}(\Xi) := \{(a, a'), (a', a) \mid a, a' \in \Xi, a \neq a'\}.$$

We can then specify the actions that we want to commute

$$\Sigma^{\mathrm{com}} := \Sigma(\mathcal{P}) \setminus \{c\,?\,m \mid c \in \mathit{Chan}, m \in \mathit{Msg}\}$$

which is captured by $\mathit{IndRel}_{\Sigma(\mathcal{P})}(\Sigma^{\mathrm{com}})$. The independence relation $\mathit{IndRel}_{\Sigma(\mathcal{P})}(\Sigma^{\mathrm{com}})$ is unambiguous and partitions $\Sigma(\mathcal{P})$ into the commutative (Σ^{com}) and non-commutative actions ($\Sigma^{\neg\mathrm{com}}$). Thus, $\mathit{IndRel}_{\Sigma(\mathcal{P})}(\Sigma^{\mathrm{com}})$ allows us to commute all concurrency actions *except* receive.

Given a CFG $\mathcal{G}(\Sigma(\mathcal{P}), \mathcal{N}, \mathcal{R})$, we want to define independence of non-terminals in a way that is consistent with the independence relation $\mathit{IndRel}_{\Sigma(\mathcal{P})}(\Sigma^{\mathrm{com}})$. Intuitively, we want a non-terminal to be commutative if it is productive and rewrites only to commutative symbols. This intuition can be captured as a fixpoint of a suitable monotone function $F : \mathscr{P}[\mathcal{N}] \to \mathscr{P}[\mathcal{N}]$. Let us write $\Sigma(w)$ (respectively $\mathcal{N}(w)$) for the set of terminal (respectively non-terminal) symbols that occur in the word w. For a subset $U \subseteq \mathcal{N}$, a non-terminal N is an element of the set $F(U)$ just if (i) $\mathcal{L}(N) \neq \emptyset$, and (ii) for each rule $N \to w$ the set inclusions $\Sigma(w) \subseteq \Sigma^{\mathrm{com}}$ and $\mathcal{N}(w) \subseteq U$ hold. We define $\mathcal{N}^{\mathrm{com}}$ as the greatest fixpoint of F. Similarly to the case of concurrency actions, this choice of commutative non-terminals gives rise to the independence relation $I(\mathcal{G})$:

$$I(\mathcal{G}) := \mathit{IndRel}_{\mathcal{N} \cup \Sigma(\mathcal{P})}(\Sigma^{\mathrm{com}} \cup \mathcal{N}^{\mathrm{com}}).$$

Again, the independence relation $I(\mathcal{G})$ yields a partition of $\mathcal{N} \cup \Sigma(\mathcal{P})$ into the commutative ($\Sigma^{\mathrm{com}} \cup \mathcal{N}^{\mathrm{com}}$) and non-commutative symbols $\Sigma^{\neg\mathrm{com}} \cup \mathcal{N}^{\neg\mathrm{com}}$. The set $\mathcal{N}^{\neg\mathrm{com}}$ denotes the set of non-commutative non-terminals and it is easy to see that $\mathcal{N}^{\neg\mathrm{com}} = \mathcal{N} \setminus \mathcal{N}^{\mathrm{com}}$.

This definition of $\mathcal{N}^{\mathrm{com}}$ and $\mathcal{N}^{\neg\mathrm{com}}$ captures our intuition. Commutative non-terminals are productive and only produce commutative symbols. Rewriting non-commutative non-terminals may block. For example, a non-terminal is an element of $\mathcal{N}^{\mathrm{com}}$ if it produces only finite words of sends and spawns (possibly infinitely many), or infinite words of sends and spawns including all their prefixes. By contrast, a non-terminal that produces a receive terminal or no finite word is categorised as non-commutative.

Example 10. Let us illustrate the definitions of $\mathcal{N}^{\mathrm{com}}$ and $\mathcal{N}^{\neg\mathrm{com}}$ on several concrete examples. Returning to the task server, its implementation $\mathcal{P}$, and the CFG $\mathcal{G}(\mathcal{P})$ (Figures 4.1 and 4.2), suppose we choose the definition of do_server that sends log messages on the system channel

$$\mathsf{do\_server}() \to \mathsf{system}\,!\,\mathsf{begin}, \mathsf{system}\,!\,\mathsf{end}.$$

and augment $\mathcal{P}$ with the rules

$$\mathsf{do\_server} \xrightarrow{\epsilon} \mathrm{L} \cdot \mathrm{L}', \qquad \mathrm{L} \xrightarrow{\mathsf{system!begin}} \epsilon, \qquad \mathrm{L}' \xrightarrow{\mathsf{system!end}} \epsilon. \qquad (\star)$$

It is thus easy to see that do_server is classified as commutative. This remains true in the case that do_server sends an arbitrary number of log messages:

$$\text{do\_server}() \to \textbf{case}\ (*)\ \textbf{of}\ \text{true} \to \text{system} \mathbin{!} \text{log\_msg}, \text{do\_server}();\quad \text{false} \to ()\ \textbf{end}.$$

which we may implement with the set of rules

$$\text{do\_server} \xrightarrow{\epsilon} \mathrm{L} \cdot \text{do\_server}, \qquad \text{do\_server} \xrightarrow{\epsilon} \epsilon, \qquad \mathrm{L} \xrightarrow{\text{system!log\_msg}} \epsilon.$$

Note that $\mathcal{L}(\text{do\_server}) = \{(\text{system} \mathbin{!} \text{log\_msg})^n : n \geq 0\}$ and hence the non-terminal do_server of $\mathcal{G}(\mathcal{P})$ is productive and it does not rewrite to any non-commutative terminals or non-terminals. However, suppose we have another procedure block. Suppose we give block one of the following two definitions:

$$\text{block}() \to \text{system} \mathbin{?} \text{unblock}. \tag{Block$_1$}$$

$$\text{block}() \to \text{system} \mathbin{!} \text{msg},\ \text{block}(). \tag{Block$_2$}$$

The definition Block$_1$ is easily implemented and we immediately see that the stack character block is classified as non-commutative. We can implement definition Block$_2$ with the ACPS rule:

$$\text{block} \xrightarrow{\text{system!msg}} \text{block}.$$

Hence, block only produces the infinite sequence of sends $(\text{system} \mathbin{!} \text{msg})^\omega$. Clearly, block is a non-terminating procedure. It should not be possible to commute past such a non-terminating computation. Indeed, $\text{block} \notin \mathcal{N}^{\text{com}}$ since $\mathcal{L}(\text{block}) = \emptyset$. Further, let us we modify the last definition of do_server to

$$\text{do\_server}() \to \textbf{case}\ (*)\ \textbf{of}\ 1 \to \text{system} \mathbin{!} \text{log\_msg}, \text{do\_server}();\quad 2 \to ();\quad \_ \to \text{block}()\ \textbf{end}.$$

which adds an additional ACPS rule $\text{do\_server} \xrightarrow{\epsilon} \text{block}$ to the rules of ($\star$). In this setting, invoking do_server *may* block and it is thus not safe to commute past it. The definition of $\mathcal{N}^{\text{com}}$ rules out do_server as commutative because $\mathcal{G}(\mathcal{P})$ can rewrite do_server to block — a non-commutative non-terminal.

A CFG endowed with an independence relation on $\mathcal{N} \cup \Sigma$ is called a *partially commutative context-free grammar* (PCCFG) [CFL09]. Let us recall their definition:

Definition. Let Σ be an alphabet of terminal symbols and I an independence relation over Σ. A *partially commutative context-free grammar* (PCCFG) is a quadruple $\mathcal{G} = (\Sigma, I, \mathcal{N}, \mathcal{R})$ where $(\Sigma, \mathcal{N}, \mathcal{R})$ is a CFG.

Note the derivation relation of a PCCFG is defined over the quotient induced by $\simeq_I$, so the words generated are congruence classes under $\simeq_I$. Hence, the (leftmost) derivation relation $\to$ of a PCCFG $\mathcal{G}$ is a binary relation over $(\Sigma \cup \mathcal{N})^* / \simeq_I$ defined as $X\,\alpha \to \beta\,\alpha$ if $X \to \beta$ is a rule.

Asynchronous partially commutative pushdown systems (APCPS) is a class of PCCFG that are defined over the alphabet $\Sigma(\mathcal{P})$ and endowed with $I(\mathcal{G})$. The independence relation $I(\mathcal{G})$ formalises the sensitivity to sequencing that arises due to the nature of unordered message passing. Even though APCPS originate from a grammar, they induce a transition system semantics that is suitable for the analysis of asynchronous concurrent pushdown systems.

Definition 20. An *asynchronous partially commutative pushdown system* (APCPS) is a PCCFG $\mathcal{G} = (\Sigma(\mathcal{P}), I(\mathcal{G}), \mathcal{N}, \mathcal{R})$.

In particular endowing $\mathcal{G}(\mathcal{P})$ with $I(\mathcal{G}(\mathcal{P}))$ yields an APCPS. Given an APCPS $\mathcal{G}$ the *standard semantics* gives rise to an infinite-state transition system $\mathcal{G}_{\text{std}} = (\mathit{State}_{\text{std}}, \rightarrow_{\mathcal{G}_{\text{std}}})$ very similar to an ACPS. The main difference is that processes are equivalence classes induced by $\simeq_{I(\mathcal{G})}$. To illustrate the similarity we will work with representatives of equivalence classes.

Standard Semantics. A process γ is a word $\delta\beta X_1\beta_1 \cdots X_n\beta_n$ (modulo $\simeq_{I(\mathcal{G})}$) in the standard semantics, where $X_i \in \mathcal{N}^{\neg\text{com}}$, $\beta, \beta_i \in \mathcal{N}^{\text{com}*}$, and $\delta \in \Sigma \cup \mathcal{N} \cup \{\epsilon\}$. We denote the set of processes of the standard semantics by *Procs*. The standard semantics of an APCPS is the transition system $\mathcal{G}_{\text{std}}$ over $\mathit{State}_{\text{std}} = \mathbb{M}[\mathit{Procs}] \times \mathit{Chans}$ where the transition relation $\rightarrow_{\mathcal{G}_{\text{std}}}$ is defined in Table 4.1 and $\mathit{Chans} = \mathit{Chan} \rightarrow \mathbb{M}[\mathit{Msg}]$. Processes change state by performing a leftmost CFG derivation of $\mathcal{G}$ until an action appears in the leftmost position (Rules (R-1) and (R-2)). Note that the side condition (†) separates the case of pushing a commutative non-terminal (Rule (R-2)) from the general case (Rule (R-1)). In the standard semantics, this separation has no consequence. However, in the following we make use of this separation to handle a push of a commutative non-terminal in a different way, and so we introduce (†) here to facilitate the comparison. Once Rules (R-1) and (R-2) have exposed a concurrency action in the leftmost position, its type determines a concurrency side-effect that interacts with the rest of the configuration (Rules (R-3)–(R-5)) causing the action to be consumed and enabling further leftmost reductions of $\mathcal{G}$. Let us demonstrate the standard semantics on an example.

Example 11. To illustrate the standard semantics of APCPS, let us consider the task server example again. Let $\mathcal{G}(\mathcal{P})$ be the APCPS induced by the ACPS $\mathcal{P}$ of Figure 4.2 together with the first definition ($\star$) of do_server in Example 10. The APCPS $\mathcal{G}(\mathcal{P})$ gives rise to the run below which is almost the same as $\mathcal{P}$'s, except that the side-effect task_bag ! init appears as part of the server process' local state:

$$\begin{aligned}
\text{main} \lhd \emptyset &\rightarrow^*_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \text{init_despatcher do_server post_task } \mathrm{L}_4\, \mathrm{L}_8 \parallel \text{despatcher} \lhd \emptyset \\
&\rightarrow_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \mathrm{L}_{12:1}\, \mathrm{L}_{12:2} \text{ do_server post_task } \mathrm{L}_4\, \mathrm{L}_8 \parallel \text{despatcher} \lhd \emptyset \\
&\rightarrow_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \text{task_bag ! init } \mathrm{L}_{12:2} \text{ do_server post_task } \mathrm{L}_4\, \mathrm{L}_8 \parallel \text{despatcher} \lhd \emptyset \\
&\rightarrow_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \mathrm{L}_{12:2} \text{ do_server post_task } \mathrm{L}_4\, \mathrm{L}_8 \parallel \text{despatcher} \lhd [\text{task_bag} \mapsto [\text{init}]].
\end{aligned}$$

Extending the run until the server process can execute the do_server procedure, we can observe that commutativity allows the sending of the message end before message begin has been sent to the channel system.

$$\begin{aligned}
\text{main} \lhd \emptyset &\rightarrow^*_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \text{do_server post_task } \mathrm{L}_4\, \mathrm{L}_8 \parallel \mathrm{L}_{15:2}\, \mathrm{L}_{16:1}\, \mathrm{L}_{16:2} \text{ do_task } \mathrm{L}_{18} \lhd \emptyset \\
&\rightarrow_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \mathrm{L}\, \mathrm{L}' \text{ post_task } \mathrm{L}_4\, \mathrm{L}_8 \parallel \mathrm{L}_{15:2}\, \mathrm{L}_{16:1}\, \mathrm{L}_{16:2} \text{ do_task } \mathrm{L}_{18} \lhd \emptyset
\end{aligned}$$

We can now use the independence relation to see that

$$\mathrm{L}\, \mathrm{L}' \text{ post\_task } \mathrm{L}_4\, \mathrm{L}_8 \simeq_{I(\mathcal{G}(\mathcal{P}))} \mathrm{L}'\, \mathrm{L} \text{ post\_task } \mathrm{L}_4\, \mathrm{L}_8$$

Standard semantics	
(R-1):	$A\,\gamma \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \beta\,\gamma \parallel \Pi \lhd \Gamma$
(R-2):	$A\,\gamma \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} B\,C\,\gamma \parallel \Pi \lhd \Gamma$
(R-3):	$(c\,?\,m)\,\gamma \parallel \Pi \lhd ([m] \oplus q)^c \oplus \Gamma \to_{\mathcal{G}_{\text{std}}} \gamma \parallel \Pi \lhd q^c \oplus \Gamma$
(R-4):	$(c\,!\,m)\,\gamma \parallel \Pi \lhd q^c \oplus \Gamma \to_{\mathcal{G}_{\text{std}}} \gamma \parallel \Pi \lhd ([m] \oplus q)^c \oplus \Gamma$
(R-5):	$(\nu X)\,\gamma \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \gamma \parallel X \parallel \Pi \lhd \Gamma$

Let $(\dagger)$ be a condition on a $\beta \in \mathcal{N}^*$: $\beta = B\,C$ and C is commutative.
Rule (R-1) has a side condition: $A \to \beta \in \mathcal{G}$ and β does *not* satisfy $(\dagger)$.
Rule (R-2) has a side condition: $A \to B\,C \in \mathcal{G}$, $B\,C$ satisfies $(\dagger)$, and $C \to^* w$.

Table 4.1: Standard transition semantics for APCPS.

and hence we can continue the path as follows:

$$\begin{aligned}
\mathsf{main} \lhd \emptyset &\to^*_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \mathrm{L}'\,\mathrm{L}\,\mathsf{post_task}\,\mathrm{L}_4\,\mathrm{L}_8 \parallel \mathrm{L}_{15:2}\,\mathrm{L}_{16:1}\,\mathrm{L}_{16:2}\,\mathsf{do_task}\,\mathrm{L}_{18} \lhd \emptyset \\
&\to_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \mathsf{system}\,!\,\mathsf{end}\,\mathrm{L}\,\mathsf{post_task}\,\mathrm{L}_4\,\mathrm{L}_8 \parallel \mathrm{L}_{15:2}\,\mathrm{L}_{16:1}\,\mathrm{L}_{16:2}\,\mathsf{do_task}\,\mathrm{L}_{18} \lhd \emptyset \\
&\to_{\mathcal{G}(\mathcal{P})_{\text{std}}} \mathrm{L}_{22:3} \parallel \mathrm{L}\,\mathsf{post_task}\,\mathrm{L}_4\,\mathrm{L}_8 \parallel \mathrm{L}_{15:2}\,\mathrm{L}_{16:1}\,\mathrm{L}_{16:2}\,\mathsf{do_task}\,\mathrm{L}_{18} \lhd [\mathsf{system} \mapsto [\mathsf{end}]].
\end{aligned}$$

Let us augment the standard semantics with a preorder to yield a PSTS. We order processes $\delta\,\pi_0 \leq_{\mathit{Procs}} \delta\,\pi_1$ just if there exist π'_0 and π'_1 such that $\delta\,\pi'_0 \leq_{\text{Hig}} \delta\,\pi'_1$ and both $\delta\,\pi_i \simeq_{I(\mathcal{G})} \delta\,\pi'_i$. We lift $\leq_{\mathit{Procs}}$ to a preorder $\leq_{\mathcal{G}_{\text{std}}}$ on configurations, using the multiset and function extension similarly to the ACPS case. Thus, we obtain a PSTS $(\mathit{State}_{\text{std}}, \to_{\mathcal{G}_{\text{std}}}, \leq_{\mathcal{G}_{\text{std}}})$ and hence coverability, boundedness, and termination are well-defined. As in the ACPS setting, we introduce simple coverability. We say an APCPS coverability query $(\mathcal{G}_{\text{std}}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ is simple if $\pi \simeq_{I(\mathcal{G})} A \in \mathcal{N}$ for all π in Π and Π_0. We are now in a position to define the shaped stack constraint for APCPS and ACPS.

Definition 21 (The Shaped Stack Constraint). Let $K \in \mathbb{N}$, $\beta_i \in \mathcal{N}^{\text{com}*}$, $X_i \in \mathcal{N}^{\neg\text{com}}$ and $\delta \in \mathcal{N}$. We say an ACPS process π such that

$$\pi = \delta\,\beta_0\,X_1\,\beta_1 \cdots X_n\beta_n$$

is *K-shaped* if $n \leq K$. Similarly, we say that an APCPS process $\bar{\pi}$ such that

$$\bar{\pi} \simeq_{I(\mathcal{G})} \bar{\delta}\,\beta\,X_1\,\beta_1 \cdots X_n\beta_n,$$

where $\bar{\delta} \in \mathcal{N} \cup \Sigma \cup \{\epsilon\}$, is *$K$-shaped* if $n \leq K$. An ACPS (APCPS) configuration $\Pi \lhd \Gamma$ is K-shaped if all processes in Π are K-shaped and we say an ACPS $\mathcal{P}$ in normal form (an APCPS $\mathcal{G}$) satisfies the *K-shaped stack constraint from* $\Pi_0 \lhd \Gamma_0$ just if all reachable configurations from $\Pi_0 \lhd \Gamma_0$ are K-shaped. For an arbitrary ACPS $\mathcal{P}$, we say that $\mathcal{P}$ satisfies the *K-shaped stack constraint from* $\Pi_0 \lhd \Gamma_0$ if $\mathcal{F}_{\text{nf}}(\mathcal{P})$[1] satisfies the K-shaped stack constraint from $\mathcal{F}_{\text{nf}}(\Pi_0 \lhd \Gamma_0)$. If $\Pi_0 \lhd \Gamma_0$ is clear from the context, we omit it and say the ACPS $\mathcal{P}$ (the APCPS $\mathcal{G}$) is *K-shaped* or simply *shaped*, if K is immaterial.

[1] See Proposition 1 for $\mathcal{F}_{\text{nf}}(\mathcal{P})$ and $\mathcal{F}_{\text{nf}}(\Pi_0 \lhd \Gamma_0)$.

Intuitively, the shaped stack constraint requires that, at all times, at most an *a priori* fixed number K of non-commutative non-terminals K may reside in the stack. Because the restriction does not apply to commutative non-terminals, stacks can grow to arbitrary heights.

Remark 2. To simplify some of our arguments, we make an additional assumption. Suppose $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ is a simple coverability query for a K-shaped ACPS $\mathcal{P}$ in normal form and $\Pi = A_1 \parallel \cdots \parallel A_n$. Our arguments are simplified if we assume that all A_i are non-commutative wrt to $I(\mathcal{G}(\mathcal{P}))$. This is a reasonable assumption since there is often a natural choice of a program point such as a receive or non-commutative procedure call for a simple coverability query. However, we may also transform $\mathcal{P}$ in polynomial-time to ensure that A_i are non-commutative wrt to $I(\mathcal{G}(\mathcal{P}))$. We can add a control-state to $\mathcal{P}$ that counts how many non-commutative stack-characters are on a process' stack, i.e. a process state looks as follows: $(i, \delta\, \beta_0 X_1 \cdots X_i \beta_i)$. We then add a stack character A'_j for each $j \in \langle n \rangle$ and make it non-commutative using a rule $A'_j \xrightarrow{c_{\text{cov}}?m_{\text{cov}}} \epsilon$. The channel c_{cov} and the message m_{cov} are fresh and therefore will block A'_j. Adding for each rule $(i, A) \xrightarrow{\epsilon} (i', A_j\, B)$ and $(i, A) \xrightarrow{\epsilon} (i, B\, A_j)$ the rules $(i, A) \xrightarrow{\epsilon} (i', A'_j\, B)$ and $(i, A) \xrightarrow{\epsilon} (i+1, B\, A'_j)$ allows a non-deterministic choice of whether to use the commutative A_i or the non-commutative A'_i. We can continually apply this for the non-terminal A if it becomes non-commutative unintentionally. This leads to at most a doubling of $\mathcal{N}$ and the rules administering the counter i. Note, that the counter may go from $i = 0$ to $i = K + 1$ where the additional non-commutative non-terminal may come from a A'_j. We may then eliminate the control-state as prescribed by Proposition 1. The control-state counter enforces that the transformed ACPS will be at most $K + 1$ shaped and polynomial-time computable from $\mathcal{P}$. Hence, we assume in the following w.l.o.g. that the A_i are all non-commutative.

Let us now turn to relating simple coverability of ACPS in normal form and APCPS. When given a simple coverability query $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ for an ACPS in normal form $\mathcal{P}$ we may analyse $\mathcal{Q}' = (\mathcal{G}(\mathcal{P}), \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ a simple coverability query for the APCPS $\mathcal{G}(\mathcal{P})$ instead; the reduction preserves the shape constraint:

Proposition 2. $\mathcal{Q}$ is a yes-instance, if and only if, $\mathcal{Q}'$ is a yes-instance. Hence simple coverability for ACPS and APCPS polynomial-time interreduce. Further, $\mathcal{P}$ is K-shaped from $\Pi_0 \lhd \Gamma_0$ if, and only if, $\mathcal{G}(\mathcal{P})$ is K-shaped from $\Pi_0 \lhd \Gamma_0$.

Proof Idea. We exploit the fact that we may accelerate any transitions with commutative side-effects (send, spawn, ϵ) and that we may delay blocking/non-commutative transitions until the last possible point. This means that we can synchronise reductions of $\mathcal{P}$ and $\mathcal{G}(\mathcal{P})$ at configurations where all processes have non-commutative non-terminals as a head symbol. This gives us a weak bisimulation result that we can use to show that the simple coverability queries $\mathcal{Q}$ and $\mathcal{Q}'$ are equivalent. For a detailed proof see Appendix B.2. □

Deciding coverability, boundedness, or termination on the standard semantics looks daunting. Even a K-shaped process is infinite state, follows a stack discipline, and may synchronise with an unbounded number of other processes. Since APCPS subsume ordinary concurrent pushdown systems, simple coverability is undecidable in general for the standard semantics. However,

Alternative semantics	
(R′-1):	$A\,M\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \beta\,M\,\gamma' \parallel \Pi' \lhd \Gamma$
(R′-2):	$A\,M\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} B\,(\mathbb{M}(w) \oplus M)\,\gamma' \parallel \Pi' \lhd \Gamma$
(R′-3):	$(c\,?\,m)\,\gamma' \parallel \Pi' \lhd ([m] \oplus q)^c \oplus \Gamma \to_{\mathcal{G}_{\text{alt}}} \gamma' \parallel \Pi' \lhd q^c \oplus \Gamma$
(R′-4):	$(c\,!\,m)\,\gamma' \parallel \Pi' \lhd q^c \oplus \Gamma \to_{\mathcal{G}_{\text{alt}}} \gamma' \parallel \Pi' \lhd ([m] \oplus q)^c \oplus \Gamma$
(R′-5):	$(\nu X)\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \gamma' \parallel X \parallel \Pi' \lhd \Gamma$
(R′-6):	$M\,X\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} X\,\gamma' \parallel \Pi' \parallel \Pi(M) \lhd \Gamma \oplus \Gamma(M)$
(R′-7):	$M'\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} M''\,\gamma' \parallel \Pi' \parallel \Pi(M') \lhd \Gamma \oplus \Gamma(M')$

Rule (R′-2) has a side condition: $A \to B\,C \in \mathcal{G}$, $B\,C$ satisfies (†), and $C \to^* w$.
Rule (R′-1) has a side condition: $A \to \beta \in \mathcal{G}$ and β does *not* satisfy (†).
In rule (R′-6) we require the multiset $M \in \mathbb{M}[\Sigma^{\text{com}}]$ whereas in rule (R′-7) M' is an element of $\mathbb{M}[\Sigma^{\text{com}} \cup \mathcal{N}] \setminus (\mathbb{M}[\Sigma^{\text{com}}] \cup \mathbb{M}[\mathcal{N}])$ and $M'' = M' \restriction \mathcal{N}$.
Further, $\Pi(M) := \|_{A \in \mathcal{N}}\|_1^{M(\nu A)} A$, and $\Gamma(M) := \bigoplus_{c \in Chan} \bigoplus_{m \in Msg} [c \mapsto [m^{M(c!m)}]]$.

Table 4.2: The alternative transition semantics for APCPS.

the independence relation $\simeq_{I(\mathcal{G})}$ enables a simplification which is formalised in the *alternative semantics* which allows us to exploit the shaped stack constraint.

4.3 An Alternative Semantics

In this section, we present an alternative semantics for APCPS which exploits commutativity to simplify the state space. The key idea is to summarise the effects of the commutative non-terminals. In the alternative semantics, rather than keeping track of the contents of the stack, we precompute the actions of those non-terminals that produce only *commutative* side-effects, i.e. sends and spawns, and store them in *summaries* on the stack. The non-commutative non-terminals, which are left on the stack, then act as separators for the summaries of commutative side-effects. As soon as the top non-terminal on the stack is popped, which may be triggered by a concurrency action, the summary just below it is unlocked. The cached actions are made effective instantaneously. This is enough to ensure a precise correspondence between e.g. simple coverability for APCPS in the standard and alternative semantics.

Alternative Semantics. In the alternative semantics a process, which we denote by γ', has the shape $\delta\,M\,X_1\,M_1 \cdots X_n M_n$, for some $X_i \in \mathcal{N}^{\neg\text{com}}$, $M, M_i \in \mathbb{M}[\mathcal{N} \cup \Sigma^{\text{com}}]$, and $\delta \in \Sigma \cup \mathcal{N} \cup \{\epsilon\}$, and is said to be K-shaped if $n \leq K$. We denote the set of such processes by $Procs_{\text{alt}}$. Then, the alternative concurrent semantics of an APCPS is a transition system $\mathcal{G}_{\text{alt}} = (State_{\text{alt}}, \to_{\mathcal{G}_{\text{alt}}})$ over elements of $State_{\text{alt}} = \mathbb{M}[Procs_{\text{alt}}] \times Chans$. We abbreviate a set of alternative processes running in parallel as Π'.

Table 4.2 shows the transition rules for the alternative semantics. Most transition rules of the alternative semantics, barring the different shape of processes, are essentially the same as the standard semantics (Rules (R′-1), (R′-3)–(R′-5)). The rules that are different implement and manage the summary of commutative non-terminals. Rule (R′-2) executes a rule $A \to B\,C$ by

precomputing the actions w of the commutative non-terminal C and inserting w's *Parikh image* $\mathbb{M}(w)$ into the summary M. This is the counterpart of pushing a commutative non-terminal in the alternative semantics. As we mentioned in the standard semantics the side condition (†) separates this case and handles it differently in the alternative semantics. Note that a table comparing the standard and alternative semantics side-by-side is available in the appendix (Table B.1).

The pop counterparts of Rule (R′-2) are defined by Rules (R′-6) and (R′-7). They ensure that the precomputed actions are rendered effective at the appropriate moment. Rule (R′-6) is applicable when the precomputed summary M contains exclusively commutative actions. Such a summary denotes a sequence of commutative non-terminals whose computation terminates and generates concurrency actions. Rule (R′-7), on the other hand, handles the case where the summary M' contains non-terminals. An interpretation of such a summary is a partial computation of a sequence of commutative non-terminals. In this case, rule (R′-7) dispatches all commutative actions and then blocks. It is necessary to consider this case since not all non-terminals have terminating computations. Thus, rule (R′-2) may non-deterministically decide to abandon the precomputation of actions.

Example 12. To illustrate the alternative semantics of APCPS, let us consider the task server in the same setting as in Example 11. Note that APCPS $\mathcal{G}(\mathcal{P})$ is not in Chomsky normal form but it is trivial to see how it can be transformed and how this enables the run given below:

$$\begin{aligned}
\mathsf{main} \lhd \emptyset &\to^*_{\mathcal{G}_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{server} \parallel \mathsf{despatcher} \lhd \emptyset \\
&\to_{\mathcal{G}_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{init_despatcher}\ [\mathsf{system} \,!\, \mathsf{begin}, \mathsf{system} \,!\, \mathsf{end}]\ \mathsf{post_task}\,\mathrm{L}_4\ [\mathsf{task_bag} \,!\, \mathsf{stop}] \\
&\qquad\qquad \parallel \mathsf{despatcher} \lhd \emptyset.
\end{aligned}$$

The last step is possible since $\mathcal{G}(\mathcal{P})$ can rewrite do_server to system ! begin · system ! end and L_8 to task_bag ! stop. If we continue on this path progressing both the server and despatcher processes until the former process has completely executed init_despatcher we observe how the summary [system ! begin, system ! end] is made effective instantaneously:

$$\begin{aligned}
\mathsf{main} \lhd \emptyset &\to^*_{\mathcal{G}_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{init_despatcher}\ [\mathsf{system} \,!\, \mathsf{begin}, \mathsf{system} \,!\, \mathsf{end}]\ \mathsf{post_task}\,\mathrm{L}_4\ [\mathsf{task_bag} \,!\, \mathsf{stop}] \\
&\qquad\qquad \parallel \mathrm{L}_{15:1}\ [\mathsf{task_bag} \,!\, \mathsf{ready}]\ \mathrm{L}_{16:1}\ [\mathsf{task_bag} \,!\, \mathsf{ok}]\ \mathsf{do_task}\,\mathrm{L}_{18} \lhd \emptyset \\
&\to^*_{\mathcal{G}_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel [\mathsf{system} \,!\, \mathsf{begin}, \mathsf{system} \,!\, \mathsf{end}]\ \mathsf{post_task}\,\mathrm{L}_4\ [\mathsf{task_bag} \,!\, \mathsf{stop}] \\
&\qquad\qquad \parallel \mathrm{L}_{16:1}\ [\mathsf{task_bag} \,!\, \mathsf{ok}]\ \mathsf{do_task}\,\mathrm{L}_{18} \lhd \emptyset \\
&\to_{\mathcal{G}_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{post_task}\,\mathrm{L}_4\ [\mathsf{task_bag} \,!\, \mathsf{stop}] \\
&\qquad\qquad \parallel \mathrm{L}_{16:1}\ [\mathsf{task_bag} \,!\, \mathsf{ok}]\ \mathsf{do_task}\,\mathrm{L}_{18} \lhd [\mathsf{system} \mapsto [\mathsf{begin}, \mathsf{end}]].
\end{aligned}$$

Again, we can turn $\mathcal{G}_{\mathrm{alt}}$ into a PSTS $(\mathit{State}_{\mathrm{alt}}, \to_{\mathcal{G}_{\mathrm{alt}}}, \leq_{\mathcal{G}_{\mathrm{alt}}})$ by endowing it with a preorder $\leq_{\mathcal{G}_{\mathrm{alt}}}$. We order elements of $\mathbb{M}[\Sigma \cup \mathcal{N}]$ with the usual multiset ordering $\leq_{\mathbb{M}[\Sigma\cup\mathcal{N}]}$ and elements of $\mathit{Procs}_{\mathrm{alt}}$ are ordered $\delta\gamma' \leq_{\mathit{Procs}_{\mathrm{alt}}} \delta\gamma''$ by the Higman extension on $\Sigma \cup \mathcal{N} \cup \mathbb{M}[\Sigma \cup \mathcal{N}]$ which is lifted to configurations as before. We say a coverability query $(\mathcal{G}_{\mathrm{alt}}, \Pi_0 \lhd \Gamma_0, \Pi' \lhd \Gamma)$ for the alternative semantics is *simple* if $\pi' = A \in \mathcal{N}$ for all $\pi' \in \Pi'$ and Π_0.

A Connection between the Standard and the Alternative Semantics

In this section, we illustrate the connection between the standard and alternative semantics. Since the alternative semantics non-deterministically precomputes summaries, a configuration of the standard semantics is represented by a set of configurations in the alternative semantics. In the following we make this representation precise and show that the alternative semantics maintains this representation on the level of sets. Note that the connection between the standard and alternative semantics is independent of the shaped stack constraint. The connection holds for all APCPS. However, we only know how to exploit the alternative semantics for decision properties when the shaped constraint is satisfied.

Representing APCPS Configurations. The state of a process of the standard semantics is a sequence of non-terminals that is possibly headed by a concurrency action. We can handle sequences of commutative non-terminals with arbitrary length by computing a summary. This is formalised by the following representation function:

$$\mathcal{F}_{Sum}(C_1 \cdots C_n) = \left\{ \bigoplus_{i=1}^{n} \mathbb{M}(w_i) \,\middle|\, C_i \to^* w_i, w_i \in (\Sigma \cup \mathcal{N})^* /_{\simeq_I} \right\} \text{ for } C_1 \cdots C_n \in \mathcal{N}^{\text{com}*}.$$

We can then lift $\mathcal{F}_{Sum}(-)$ to arbitrary process states and configurations. To reduce the notational burden we will overload $\mathcal{F}_{Sum}(-)$ on these domains. Hence, an APCPS process state is represented by the following set of process states in the alternative semantics:

$$\mathcal{F}_{Sum}(\delta\, \beta_0\, X_1\, \beta_1 \cdots X_n \beta_n) = \{\delta\, M_0\, X_1\, M_1 \cdots X_n\, M_n \mid M_i \in \mathcal{F}_{Sum}(\beta_i), i \leq n\}\,,$$

where $X_i \in \mathcal{N}^{\neg\text{com}}$, $\beta_i \in \mathcal{N}^{\text{com}*}$, and $\delta \in \mathcal{N} \cup \Sigma \cup \{\epsilon\}$. Further, a representation of configurations of the standard semantics is obtained by lifting the function $\mathcal{F}_{Sum}(-)$ to configurations:

$$\mathcal{F}_{Sum}(\pi_1 \parallel \cdots \parallel \pi_n \lhd \Gamma) = \{\bar{\pi}_1 \parallel \cdots \parallel \bar{\pi}_n \lhd \Gamma \mid \bar{\pi}_i \in \mathcal{F}_{Sum}(\pi_i), i \in \langle n \rangle\}\,.$$

Example 13. Let us consider the task server example again in the same setting as in Example 11 in order to explain the definition $\mathcal{F}_{Sum}(-)$. Below we show the result of applying $\mathcal{F}_{Sum}(-)$ on the configuration in which the server and despatcher have been spawned and the non-terminal server has just been rewritten.

$$\mathcal{F}_{Sum}(\mathrm{L}_{22:3} \parallel \text{init\_despatcher do\_server post\_task } \mathrm{L}_4\, \mathrm{L}_8 \parallel \text{despatcher} \lhd \emptyset)$$

$$= \left\{ \begin{array}{l} \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\text{do\_server}]\ \text{post\_task } \mathrm{L}_4\ [\mathrm{L}_8] \parallel \text{despatcher} \lhd \emptyset, \\ \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\text{do\_server}]\ \text{post\_task } \mathrm{L}_4\ [\text{task\_bag!stop}] \parallel \text{despatcher} \lhd \emptyset, \\ \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\text{system!begin}, \mathrm{L}']\ \text{post\_task } \mathrm{L}_4\ [\mathrm{L}_8] \parallel \text{despatcher} \lhd \emptyset, \\ \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\text{system!begin}, \mathrm{L}']\ \text{post\_task } \mathrm{L}_4\ [\text{task\_bag!stop}] \parallel \text{despatcher} \lhd \emptyset, \\ \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\mathrm{L}, \text{system!end}]\ \text{post\_task } \mathrm{L}_4\ [\mathrm{L}_8] \parallel \text{despatcher} \lhd \emptyset, \\ \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\mathrm{L}, \text{system!end}]\ \text{post\_task } \mathrm{L}_4\ [\text{task\_bag!stop}] \parallel \text{despatcher} \lhd \emptyset, \\ \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\text{system!begin}, \text{system!end}]\ \text{post\_task } \mathrm{L}_4\ [\mathrm{L}_8] \parallel \text{despatcher} \lhd \emptyset, \\ \mathrm{L}_{22:3} \parallel \text{init\_despatcher } [\text{system!begin}, \text{system!end}]\ \text{post\_task } \mathrm{L}_4\ [\text{task\_bag!stop}] \parallel \text{despatcher} \lhd \emptyset \end{array} \right\}$$

Lifting the Alternative Semantics to Sets. To illustrate the connection between the standard and alternative semantics, we employ a co-universal powerset lifting onto the alternative semantics. Hence, we get the transition system $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}] = (\mathscr{P}[\mathit{State}_{\text{alt}}], \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]})$. To remind ourselves this means that:

$$S \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]} S' \text{ if, and only if, } \forall \Pi' \lhd \Gamma' \in S'.\exists \Pi \lhd \Gamma \in S.\Pi \lhd \Gamma \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi \lhd \Gamma.$$

Let us explain the intuition behind a co-universal powerset-lifting. Suppose we have transition systems $\mathcal{S}$ and $\mathcal{S}'$. And suppose a set of $\mathcal{S}'$-configurations $\mathcal{F}(s)$ represents a $\mathcal{S}$-configuration s. Each $\bar{s} \in \mathcal{F}(s)$ represents a *guess* of how to represent s. If we can show that the transition systems have the lockstep property

$$s \rightarrow_{\mathcal{S}} s' \text{ if, and only if, } \mathcal{F}(s) \rightarrow_{\mathscr{P}^{\mathsf{A}}[\mathcal{S}']} \mathcal{F}(s'),$$

we may say, informally, that the transition relation can accurately guess a representation set of s' from $\mathcal{F}(s)$. More concretely, if $\bar{s} \in \mathcal{F}(s)$ is a guess to represent s, then the transition relation $\rightarrow_{\mathcal{S}'}$ can produce an accurate guess $\bar{s}'$ for s' that is consistent with the guess $\bar{s}$, or the guess $\bar{s}$ is ruled out to be inconsistent. Clearly, the lockstep property gives a bisimulation of $\mathcal{S}$ and $\mathscr{P}^{\mathsf{A}}[\mathcal{S}']$. In the case of the standard and alternative semantics, we are not fortunate enough to have such a strong connection. The above lockstep synchronisation is only maintained in a weaker sense.

Establishing a Bisimulation. First, let us demonstrate the easy direction of the bisimulation. It is easy to verify that the transitions of the alternative semantics can be followed (almost) in lockstep by the standard semantics. We observe that the precomputations in the alternative semantics predetermine choices when it comes to unlocking a commutative part of the stack. It is easy to see that the standard semantics can produce the precomputed effects, albeit in more steps.

Lemma 2. Suppose $\bar{\Pi} \lhd \Gamma \in \mathcal{F}_{\mathit{Sum}}(\Pi \lhd \Gamma)$ such that $\bar{\Pi} \lhd \Gamma \rightarrow_{\mathcal{G}_{\text{alt}}} \bar{\Pi}' \lhd \Gamma'$ for some $\bar{\Pi}' \lhd \Gamma'$. Then, there exists a configuration $\Pi' \lhd \Gamma'$ such that $\Pi \lhd \Gamma \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$ and $\bar{\Pi}' \lhd \Gamma' \in \mathcal{F}_{\mathit{Sum}}(\Pi' \lhd \Gamma')$.

Proof Idea. A straightforward case analysis. See Appendix B.3 for a full proof. □

This means that the standard semantics *weakly simulates* the alternative semantics for APCPS. Owing to the nature of precomputations and caches, it is more difficult to relate transitions of the standard semantics to those of the alternative semantics. Nevertheless, if we consider $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$ we can establish a simulation. However, the simulation is weaker. We are only able to guarantee that we can synchronise the simulation at configurations of a specific shape. Let us define the set of configurations in which every process state starts with a non-commutative non-terminal:

$$\mathit{Sync} = \{\Pi \lhd \Gamma \mid \forall \pi \in \Pi.\pi = X_1\,\beta_1 \cdots X_n\beta_n, X_i \in \mathcal{N}^{\neg\text{com}}, \beta_i \in \mathcal{N}^{\text{com}*}\}\,.$$

We can then define a relation on the configurations of $\mathcal{G}_{\text{std}}$ and $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$:

$$\mathcal{R} = \{(\Pi \lhd \Gamma, \mathcal{F}_{\mathit{Sum}}(\Pi \lhd \Gamma)) \mid \Pi \lhd \Gamma \in \mathit{Sync}\}\,.$$

If we label the transitions of $\mathcal{G}_{\text{std}}$ and $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$ appropriately, the relation $\mathcal{R}$ becomes a weak bisimulation. If we choose to label transitions by configurations in *Sync*, we allow enough flexibility in the transition relation to let $\mathcal{R}$ be a weak bisimulation while we ensure that $\mathcal{G}_{\text{std}}$ and $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$ synchronise on configurations in *Sync*. Thus, we label a transition $\Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$ by $\Pi' \lhd \Gamma'$ if $\Pi' \lhd \Gamma' \in \mathit{Sync}$, otherwise the label is ϵ. Similarly for $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$, we label a transition $S \to_{\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]} S'$ with $\Pi' \lhd \Gamma'$ if $S' = \mathcal{F}_{\mathit{Sum}}(\Pi' \lhd \Gamma')$ and $\Pi' \lhd \Gamma' \in \mathit{Sync}$, otherwise the label is also ϵ. This labelling then allows us to conclude:

Proposition 3. $\mathcal{R}$ is a weak bisimulation for $\mathcal{G}_{\text{std}}$ and $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$.

Proof Idea. One direction easily follows from Lemma 2. The other direction can be established by first observing that a non-ϵ labelled step in $\mathcal{G}_{\text{std}}$ $(\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}{}^{*}_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma')$ can be maximally "stretched" $(\Pi \lhd \Gamma \xrightarrow{\Pi_1 \lhd \Gamma_1}{}^{*}_{\mathcal{G}_{\text{std}}} \cdots \xrightarrow{\Pi_i \lhd \Gamma_i}{}^{*}_{\mathcal{G}_{\text{std}}} \cdots \xrightarrow{\Pi' \lhd \Gamma'}{}^{*}_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma')$, i.e. we may reorder independent transitions in such a way that we expose the maximal number of labels in *Sync*. This can be achieved by accelerating any transitions with commutative side-effects (send, spawn, ϵ) and delaying blocking/non-commutative transitions until the last possible point. We can then assume w.l.o.g. that we are considering a non-ϵ labelled step $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}{}^{*}_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$, that cannot be reordered to divide it and expose further labels. In this setting, we can see that only one process is active along $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}{}^{*}_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$. This process either (i) rewrites its top non-commutative terminal while possibly adding commutative non-terminals, (ii) adds a new non-commutative non-terminals to the top, or (iii) pops a non-commutative non-terminal, unlocks a sequence of commutative non-terminals, and completely executes them. The first two cases are easily simulated by the alternative semantics. The simulation of the last case relies on the existence of a precomputation that correctly guessed the complete execution of the sequence of commutative non-terminals. The set representing the configuration of the standard semantics ensures that this precomputation is available. See Appendix B.3 for a detailed formal proof. □

Correspondence of Decision Problems. This bisimulation result is enough to ensure that we can use the alternative semantics to answer decision questions on the APCPS $\mathcal{G}$. First, we establish the equivalence of simple coverability for the standard and alternative semantics.

Theorem 7 (Equivalence of Simple Coverability). *Suppose we have the simple coverability queries $Q_{std} = (\mathcal{G}_{std}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ and $Q_{alt} = (\mathcal{G}_{alt}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$. Then, the query Q_{std} is a yes-instance for simple coverabililty for the standard semantics if, and only if, Q_{alt} is a yes-instance for simple coverability for the alternative semantics.*

Proof Idea. For one direction we can exploit Lemma 2. For the other, it is an easy exercise to exploit the bisimulation result to obtain a covering path in the alternative semantics from one in the standard semantics. See Appendix B.3 for a detailed formal proof. □

Two Conjectures for Boundedness and Termination. The alternative semantics changes the order of evaluation due to precomputations. Moreover, precomputations collapse potentially unbounded derivations of a commutative non-terminal into a single step. Thus, relating the boundedness and termination problems is more difficult. However, we may restrict ourselves to subclasses of APCPS to simplify our reasoning.

In the following, let us present an argument of why we believe it is possible to answer termination and boundedness questions posed on the standard semantics using the alternative semantics. We summarise our argument by two conjectures for which we provide proof sketches.

Let us call an APCPS $\mathcal{G}$ *termination-bounded* if no commutative non-terminals C of $\mathcal{G}$ has an infinite derivation. In this setting, we conjecture that we can use the alternative semantics to answer a termination question.

Conjecture 1 (Equivalence of Termination). *Suppose the APCPS $\mathcal{G}$ is termination-bounded. Then there is an infinite path from $\Pi_0 \lhd \Gamma_0$ in $\mathcal{G}_{std}$ if, and only if, there is an infinite path from $\bar{\Pi}_0 \lhd \Gamma_0$ in $\mathcal{G}_{alt}$ where $\bar{\Pi}_0 \lhd \Gamma_0 \in \mathcal{F}_{Sum}(\Pi_0 \lhd \Gamma_0)$.*

Proof Sketch. Again, if there is an infinite path in the alternative semantics we can exploit Lemma 2. If, on the other hand, there is an infinite path in the standard semantics, we may assume that it is going through infinitely many configurations in *Sync* (possibly after reordering independent transitions). Otherwise, we would obtain a contradiction to the fact that $\mathcal{G}$ is termination-bounded. The bisimulation result then gives us an infinite path in the alternative semantics. □

We can now build on this conjecture to reason about termination of general APCPS. In order to check termination of an APCPS $\mathcal{G}$, we may perform a simple static analysis on the commutative non-terminals. If there is a commutative non-terminal with an infinite derivation, we may check using a coverability query if it can ever execute. If yes, we obtain an infinite path, otherwise, it can safely be eliminated. Hence, we obtain an infinite path witnessing non-termination or we arrive at a termination-bounded APCPS. Therefore, if the above conjecture holds true, then we can reason about termination using the alternative semantics.

It is possible to apply similar reasoning to the boundedness problem. Let us call an APCPS $\mathcal{G}$ *rewrite-bounded* if all commutative non-terminals C of $\mathcal{G}$ rewrite to a finite set of words, i.e. the set $\{w \in (\mathcal{N} \cup \Sigma)^* \mid C \to^* w\}$ is finite for all C. As in the case of termination, we conjecture that we can establish a connection between the standard and alternative semantics.

Conjecture 2 (Equivalence of Boundedness). *Suppose we have an APCPS $\mathcal{G}$ that is rewrite-bounded. Then, the set $Reach_{\mathcal{G}_{std}}(\Pi_0 \lhd \Gamma_0)$ is infinite if, and only if, the set $Reach_{\mathcal{G}_{alt}}(\bar{\Pi}_0 \lhd \Gamma_0)$ is infinite for some $\bar{\Pi}_0 \lhd \Gamma_0 \in \mathcal{F}_{Sum}(\Pi_0 \lhd \Gamma_0)$.*

Proof Sketch. Since $\mathcal{G}$ is rewrite-bounded, if $Reach_{\mathcal{G}_{\text{alt}}}(\bar{\Pi}_0 \lhd \Gamma_0)$ is infinite we can conclude using Lemma 2 that $Reach_{\mathcal{G}_{\text{std}}}(\Pi_0 \lhd \Gamma_0)$ is infinite. This follows from the fact that if $\mathcal{G}$ is rewrite-bounded for each $\Pi \lhd \Gamma$ the set $\mathcal{F}_{Sum}(\Pi \lhd \Gamma)$ is finite. On the other hand, if $Reach_{\mathcal{G}_{\text{std}}}(\Pi_0 \lhd \Gamma_0)$ is an infinite set, then $Reach_{\mathcal{G}_{\text{std}}}(\Pi_0 \lhd \Gamma_0) \cap Sync$ must be an infinite set, since $\mathcal{G}$ is rewrite-bounded. Thus, the bisimulation result is sufficient to imply that $Reach_{\mathcal{G}_{\text{alt}}}(\bar{\Pi}_0 \lhd \Gamma_0)$ is an infinite set. □

In a similar fashion to the termination case, we build on this conjecture. We conjecture that a boundedness question on a general APCPS $\mathcal{G}$ in the standard semantics can be answered by a combination of coverability and boundedness in the alternative semantics. We can detect commutative non-terminals of $\mathcal{G}$ that violate rewrite-boundedness by a simple static analysis on $\mathcal{G}$

and check whether they can execute by a simple coverability query. If there is such a non-terminal that can execute, then clearly we have a no-instance of the boundedness problem. Otherwise, such non-terminals may be safely removed. In this case the truth of the above conjecture implies that we can use the alternative semantics to decide the boundedness problem.

4.4 Shaped APCPS are Well-Structured

In this section, we show that the alternative semantics for shaped APCPS gives rise to a *well-structured transition system* (WSTS). This allows us to transfer the good decision properties of WSTS (Section 2.3) to the alternative semantics for shaped APCPS. Thus, we obtain e.g. that coverability is decidable. Note that due to Rule (R′-2) the alternative semantics gives rise to a potentially infinite branching transition system. This complicates the application of some results from the WSTS literature. Let us recall a few definitions. Let $\leq$ be a well-quasi-order (WQO) over a set S, we say $\leq$ is a *well-partial-order* (WPO) if $\leq$ is a partial order. A *well-structured transition system* (WSTS) is a PSTS $\mathcal{S} = (S, \to_{\mathcal{S}}, \leq_{\mathcal{S}})$ such that $\leq$ is a WQO and $\to_{\mathcal{S}}$ is monotone with respect to $\leq_{\mathcal{S}}$ [FS01]. We say a WSTS $\mathcal{S}$ is *strict* if $\to_{\mathcal{S}}$ is strictly monotone with respect to $\leq_{\mathcal{S}}$ and say a $\mathcal{S}$ is *post-effective* if the function $f(s) = |post_{\mathcal{S}}(s)|$ is effectively computable, where we define the set of one-step successors by $post_{\mathcal{S}}(s) = \{s' \mid s \to_{\mathcal{S}} s'\}$. Coverability is decidable for finitely or infinitely branching WSTS if $\uparrow Pred(\uparrow\{s\})$ is effectively computable [FS01; Abd+00], while under the same assumption termination is decidable for finitely branching WSTS [FS01]. For infinitely branching WSTS that are post-effective and strict and for which $\leq$ is a WPO, boundedness is decidable [BFM14].

An Analysis of the State Space

In the process of proving a PSTS to be a WSTS, an important step is to expose a WQO. For this purpose, let us analyse the state space of the alternative semantics. Let $\mathcal{G}$ be a K-shaped APCPS and $\mathcal{G}_{\text{alt}}$ be the transition system over $State_{\text{alt}}$ induced by the alternative semantics.

Let us first focus on summaries. We can classify summaries into three types: complete summaries, partial summaries, and degenerate summaries. Complete summaries are multisets over commutative concurrency actions. They are complete in the sense that they represent the complete execution of a sequence of commutative non-terminals. Partial summaries, on the other hand, represent a partial execution of a sequence of commutative non-terminals and hence contain non-terminals. We can define the following sets to formalise these classifications:

$$\begin{aligned} CompleteSummary &:= \mathbb{M}[\Sigma^{\text{com}}] & PartialSummary &:= \mathbb{M}[\Sigma^{\text{com}} \cup \mathcal{N}^{\text{com}}] \\ DegenSummary &:= \mathbb{M}[\mathcal{N}^{\text{com}}] & Summary &:= CompleteSummary + PartialSummary \end{aligned}$$

There are three rules of the alternative semantics Rules (R′-2), (R′-6), (R′-7) that interact with summaries. Note that Rule (R′-6) only applies to complete summaries, while Rule (R′-7) applies only to partial summaries and turns them into degenerate summaries. Degenerate summaries contain only non-terminals and force a process to block. No rule applies to a process that carries a degenerate summary at its head.

We can now use this categorisation to define other sets that describe part of a process' local state in the alternative semantics:

$$\begin{aligned} \mathit{Stack}^{\leq k} &:= (\mathcal{N}^{\neg \mathrm{com}} \cdot \mathit{Summary})^{\leq k} \\ \mathit{DelayedControl} &:= \mathit{CompleteSummary} + \mathit{PartialSummary} + \mathit{DegenSummary} \\ \mathit{NormalControl} &:= (\mathcal{N} \cdot \mathit{Summary}) + (\Sigma \cdot \mathcal{N} \cdot \mathit{Summary}) + (\Sigma \cdot \mathit{Summary}) \\ \mathit{ControlState} &:= \mathit{NormalControl} + \mathit{DelayedControl} \\ \mathit{Procs}_{\mathrm{alt}} &= \sum_{k \leq K} \mathit{ControlState} \cdot \mathit{Stack}^{\leq k} \end{aligned}$$

Decomposing the state space of the alternative semantics in this fashion makes it easy to apply known composition results for WQO (Section 2.3) and expose a well-order.

A Well-Quasi-Order for the Alternative Semantics

Our goal is to establish that the order $\leq_{\mathcal{G}_{\mathrm{alt}}}$ is a WQO for $\mathit{State}_{\mathrm{alt}}$. This is the first step to showing the alternative semantics for shaped APCPS gives rise to a WSTS. We achieve this by constructing a WQO by composition which is isomorphic to $\leq_{\mathcal{G}_{\mathrm{alt}}}$.

The sets *CompleteSummary*, *DegenSummary*, *PartialSummary*, and $\mathbb{M}[\mathit{Msg}]$ are multisets and ordered by multiset inclusion which is a well-quasi-order. Since the set of channel names *Chan* is a finite and $\mathit{Chans} = \mathit{Chan} \to \mathbb{M}[\mathit{Msg}] \cong \mathbb{M}[\mathit{Msg}]^{|\mathit{Chan}|}$ we obtain a well-quasi-order for *Chans* using a generalisation of Dickson's lemma. We then compose the WQO of *CompleteSummary* and *PartialSummary* to obtain a WQO $\leq_{\mathit{Summary}} := \leq_{\mathit{CompleteSummary}} + \leq_{\mathit{PartialSummary}}$ for *Summary*. For each $j \in \{1 \dots k\}$ we define

$$X_1\, M_1\, X_2\, M_2 \cdots X_j\, M_j \leq_{\mathit{Stack}^{\leq k}} X_1\, M_1'\, X_2\, M_2' \cdots X_j\, M_j' \quad \text{iff} \quad \forall i : M_i \leq_{\mathit{Summary}} M_i'$$

which gives a well-quasi-order for $\mathit{Stack}^{\leq k}$. We obtain a WQO for *DelayedControl* by composing the WQOs of *CompleteSummary*, *DegenSummary*, and *PartialSummary*:

$$\leq_{\mathit{DelayedControl}} := \leq_{\mathit{CompleteSummary}} + \leq_{\mathit{DegenSummary}} + \leq_{\mathit{PartialSummary}} .$$

Since Σ and $\mathcal{N}$ are finite sets, $(\Sigma, =_\Sigma)$ and $(\mathcal{N}, =_\mathcal{N})$ are WQOs. Thus, we obtain a WQO for *NormalControl* by composition:

$$\leq_{\mathit{NormalControl}} := \left(=_\Sigma \cdot \leq_{\mathit{Summary}}\right) + \left(=_\Sigma \cdot =_\mathcal{N} \cdot \leq_{\mathit{Summary}}\right) + \left(=_\mathcal{N} \cdot \leq_{\mathit{Summary}}\right) .$$

Similarly, we can construct WQOs for *ControlState* and $\mathit{Procs}_{\mathrm{alt}}$ by composition:

$$\begin{aligned} \leq_{\mathit{ControlState}} &:= \leq_{\mathit{NormalControl}} + \leq_{\mathit{DelayedControl}}, \\ \leq_{\mathit{Procs}_{\mathrm{alt}}} &:= \sum_{k \leq K} \leq_{\mathit{ControlState}} \cdot \leq_{\mathit{Stack}^{\leq k}} . \end{aligned}$$

As a last step, we make use of the fact that the multi-set extension of a WQO remains a WQO (Section 2.3, (WQO-d)) to construct a WQO for $\mathbb{M}\left[\mathit{Procs}_{\mathrm{alt}}\right]$. This allows us to define a WQO for

$State_{\text{alt}}$ by $\leq_{State_{\text{alt}}} := \leq_{\mathbb{M}[Procs_{\text{alt}}]} \times \leq_{Chans}$. Clearly, the following preorder isomorphism holds: $(State_{\text{alt}}, \leq_{State_{\text{alt}}}) \cong (State_{\text{alt}}, \leq_{\mathcal{G}_{\text{alt}}})$. Hence, we know that $\leq_{\mathcal{G}_{\text{alt}}}$ is in fact a WQO and we can use $\leq_{State_{\text{alt}}}$ and $\leq_{\mathcal{G}_{\text{alt}}}$ interchangeably. Further, we note that $\leq_{\mathcal{G}_{\text{alt}}}$ satisfies anti-symmetry and hence is a well-partial-order.

To transfer the good decision properties of WSTS for shaped APCPS, it remains to show that $\rightarrow_{\mathcal{G}_{\text{alt}}}$ is strictly monotonic, post-effective, and $\uparrow Pred(\uparrow\{\gamma\})$ is computable.

Lemma 3 (Monotonicity). The transition relation $\rightarrow_{\mathcal{G}_{\text{alt}}}$ is strictly monotone with respect to the well-order $\leq_{\mathcal{G}_{\text{alt}}}$.

Proof Idea. Monotonicity follows from a straightforward case analysis on the rules of the alternative semantics. A detailed proof is available in Appendix B.4. □

Since the transition relation of the alternative semantics $\rightarrow_{\mathcal{G}_{\text{alt}}}$ is strictly monotonic, the alternative semantics for shaped APCPS gives rise to a strict post-effective WSTS.

Corollary 4. The pre-structured transition system $\mathcal{G}_{\text{alt}} = \left(State_{\text{alt}}, \rightarrow_{\mathcal{G}_{\text{alt}}}, \leq_{\mathcal{G}_{\text{alt}}}\right)$ is a strict WSTS, $\uparrow Pred(\uparrow\{\gamma\})$ is computable, and $\mathcal{G}_{\text{alt}}$ is post-effective.

Proof. To see that $\uparrow Pred(\uparrow\{\gamma\})$ is computable is mostly trivial. Only predecessors generated by Rule (R′-2) are not immediately obvious. Given $M' \in Summary$, we observe that it is enough to be able to compute the set

$$Pre_{M'} := \uparrow\{(C, M) \mid C \in \mathcal{N}^{\text{com}}, C \rightarrow^* w, M'' = M \oplus \mathbb{M}(w), M' \leq_{\mathbb{M}} M''\}.$$

Since C is a commutative non-terminal, the derivation $C \rightarrow^* w$ can be seen as a computation of a commutative context-free grammar (CCFG) for which an encoding into Petri nets has been shown by Ganty and Majumdar [GM12] (see Section 2.4). Their encoding builds on work by Esparza [Esp97] modelling CCFG in Petri nets. Their translation leverages a recent result [Esp+11]: every word of a CCFG has a *bounded-index* derivation. A budget counter constrains the Petri net encoding of a CCFG to respect boundedness of index; termination of a CCFG computation can be detected by a transition that is only enabled when the full budget is available. This result allows us to compute the set $Pre_{M'}$ using a backwards coverability algorithm for Petri nets.

We can use a similar argument to show that $\mathcal{G}_{\text{alt}}$ is post-effective. Only successors generated by Rule (R′-2) may pose a problem. It is enough to show that we can compute the cardinality of the set

$$Post_C := \{w \mid C \rightarrow^* w\}.$$

Clearly, we can use Ganty and Majumdar's encoding and the decidability of boundedness on Petri nets to check if $|Post_C| = \infty$. If $|Post_C| < \infty$, we can enumerate all the words in $Post_C$, since it is generated by a CFG [Döm00]. Note that since $|Post_C| < \infty$ the enumeration is guaranteed to stop before we reach $\mathcal{G}$'s pumping length. Hence, by counting the enumerated words we can compute $|Post_C|$. Therefore, $\mathcal{G}_{\text{alt}}$ is post-effective. □

Now that we have established that $\mathcal{G}_{\text{alt}}$ is a strict post-effective WSTS, we can apply the results of the WSTS toolbox to obtain decision results for the alternative semantics of shaped APCPS. Note that the alternative semantics for both rewrite-bounded and termination-bounded APCPS are clearly finitely branching, and thus termination is decidable.

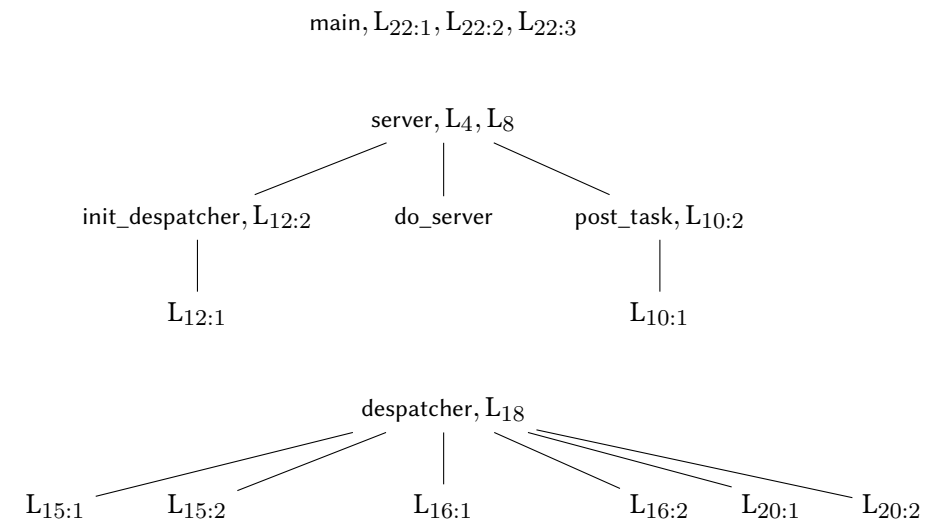


Figure 4.5: A variant of a Hasse diagram depicting a preorder $\geq_{\text{shape}}$. The preorder is a witness that the task server example satisfies the shaped constraint. Nodes are sets of $\geq_{\text{shape}}$-equivalent elements.

Theorem 8. *Coverability and boundedness are decidable for the alternative semantics of APCPS that satisfy the shaped stack constraint. Hence, coverability is decidable for the standard semantics of shaped APCPS. Further, termination is decidable on the alternative semantics for termination-bounded and rewrite-bounded APCPS that satisfy the shaped stack constraint.*

This result together with the conjectures on the connection of the termination and boundedness problem between the standard and alternative semantics for termination-bounded and rewrite-bounded APCPS (Conjectures 1 and 2) leads us to believe that both termination and boundedness are also decidable for the standard semantics for shaped APCPS.

Conjecture 3. *Boundedness and termination are decidable for the standard semantics of shaped APCPS.*

At this point, we can summarise the implication of the results so far. The shaped stack constraint implies decidability of coverability for ACPS/APCPS. On the alternative semantics, boundedness is decidable for shaped APCPS and termination is decidable for rewrite- and termination-bounded shaped APCPS. We conjecture that the latter two results can be extended to the standard semantics and to shaped ACPS.

4.5 A Syntactic Sufficient Condition and Two Examples

An APCPS $\mathcal{G}$ satisfies the K-shaped stack constraint from $\Pi_0 \lhd \Gamma_0$ if for all reachable configurations $\Pi \lhd \Gamma$, i.e. $\Pi \lhd \Gamma \in \mathit{Reach}_{\mathcal{G}_{\text{std}}}(\Pi_0 \lhd \Gamma_0)$, every process π of Π has less than K non-commutative non-terminals on its stack. Since the definition of the shaped stack constraint involves the set of reachable terms $\mathit{Reach}_{\mathcal{G}_{\text{std}}}(\Pi_0 \lhd \Gamma_0)$, it is considered a semantic property of

$\mathcal{G}$. Hence, the question whether for an arbitrary APCPS (or ACPS) $\mathcal{G}$ there exists a K such that $\mathcal{G}$ satisfies the K-shaped stack constraint is in general undecidable. However, there is a simple, easy-to-check *syntactic* sufficient condition. We omit the easy proof.

Proposition 4. Let $\mathcal{G}$ be an APCPS. If there is a preorder $\geq_{\text{shape}}$ on $\mathcal{N}$ such that for every rule $A \to \beta$ and $B \in \mathcal{N}(\beta)$: (i) $A \geq_{\text{shape}} B$, and (ii) $\exists C \in \mathcal{N}^{\neg\text{com}} : A \to B\,C$ is a rule of $\mathcal{G}$ $\implies$ $A >_{\text{shape}} B$, then $\mathcal{G}$ is a shaped APCPS from any configuration.

Intuitively, Proposition 4 says that if no non-terminals A can ever be rewritten to $A\gamma$ such that γ contains a non-commutative non-terminal, then $\mathcal{G}$ satisfies the shaped stack restriction. This condition appears natural and readily satisfied by recursive programs humans write. To illustrate its usefulness let us apply Proposition 4 to two examples.

Example 14. Let us consider the task server example of Figure 4.2 again. Figure 4.5 shows a diagram of a preorder $\geq_{\text{shape}}$ on the stack characters/non-terminals of the task server program. Note that the depicted preorder satisfies the conditions of Proposition 4. Proposition 4 is phrased for APCPS/ACPS in normal form. However, if we transform every rule of the form $A \to A_1 \cdots A_n$ to rules $A \to A_1\,A'_1, A'_1 \to A_2 A'_2, \cdots A'_{n-2} \to A_{n-1}\,A_n$, it is easy to see how to apply Proposition 4. Note that if A_k is the last non-commutative non-terminal in the sequence $A_1 \cdots A_n$, Proposition 4 allows us to set $A =_{\text{shape}} A_k =_{\text{shape}} A_{k+1} =_{\text{shape}} \cdots =_{\text{shape}} A_n$. This explains why we can set $\mathsf{main} =_{\text{shape}} \mathrm{L}_{22:1} =_{\text{shape}} \mathrm{L}_{22:2} =_{\text{shape}} \mathrm{L}_{22:3}$, $\mathsf{server} =_{\text{shape}} \mathrm{L}_4 =_{\text{shape}} \mathrm{L}_8$, and $\mathsf{despatcher} =_{\text{shape}} \mathrm{L}_{18}$.

Example 15. In Figure 4.6, we give an example program implementing a simple replicated workers pattern. It consists of a distributor process that initially spawns a number of workers, sets up a single shared resource, and distributes one task per worker over a one-to-many channel.

Each worker runs a task-processing loop. Upon reception of a task, the worker recursively decomposes it, which involves communicating with the shared resource at each step. Note that the communication of each worker with the resource is protected by a lock. For the worker, the decomposition has two possible outcomes: (i) the task is partially solved, generating one subtask and an intermediate result or (ii) the task is broken down into one subtask and one new distributable task. In case (i) the worker recursively solves the subtask and combines the result with the intermediate result. In case (ii) the worker recursively solves the subtask and subsequently dispatches the newly generated distributable task before returning. When a worker has finished processing a task, it relays the result to the server and awaits a new task to process. We have left the implementation of the functions `decompose_task` and `combine` open. For the purpose of this example, we only assume that they do not perform any concurrency actions, but they may be recursive functions.

Note that the call stacks of both the distributor and the workers may reach arbitrary heights, and communication actions may be performed by a process at any stage of the computation, regardless of stack height. For example, the worker sends and receives messages at every decomposition, and each recursive call increases the height of the call stack.

```
1  main() → setup_network(),
2           redistribute () .
3
4  setup_network() →
5    spawn(worker),
6    case (*) of
7      true  → setup_network();
8      false →
9        spawn(res_start(init )),
10       toResource ! isReady,
11       toDistributor ? ready;
12   end,
13   toWorkers ! task.
14
15 redistribute () →
16   case (*) of
17     true  → toDistributor ? redist (Task),
18             toWorkers ! Task;
19     false → toDistributor ? result (Result),
20             print (Result);
21   end,
22   redistribute () .
23
24 % Resource
25 res_start (S) =
26   fun() → toDistributor ! ready,
27           resource(S)
28   end.
29
30 resource(S) →
31   toResource ? lock_req,
32   toWorkers ! locked,
33   resource_locked(S).
34
35 resource_locked(S) →
36   case (*) of
37     1 → toResource ? unlock_req,
38         resource(S);
39     2 → toResource ? getState,
40         toWorkers ! state (S),
41         resource_locked(S);
42     _ → toResource ? update(X),
43         resource_locked(X)
44   end.
```

```
45 % Worker
46 worker() →
47   toWorkers ? Task,
48   result = do_task(Task),
49   toDistributor ! result ,
50   worker().
51
52
53 do_task(Task) →
54   case decompose(Task) of
55     local (Task', Int_result ) →
56       Result = do_task(Task'),
57       Result' =
58         combine(Result, Int_result )
59       return Result';
60     redist (Task', Task'') →
61       Result = do_task(Task'),
62       toDistributor ! Redist(Task'') ,
63       return Result;
64   end.
65
66 combine(res, res')      → ...
67
68
69 decompose(Task)→
70   lock(toResource),
71   toResource ! getState,
72   ?label ("critical "),
73   toWorkers ? state (State),
74   (Result, Update) = decompose_task(Task, State),
75   toResource ! update(Update),
76   unlock(toResource),
77   return Result.
78
79 decompose_task(Task,State) → ...
80
81
82 lock(C) →
83   C ! lock_req,
84   toWorkers ? locked.
85
86 unlock(C) → C ! unlock_req.
```

Figure 4.6: A resource, a task distributor, and a worker that recursively solves tasks and shares its workload.

An interesting verification question for this example program is whether the locking mechanism for the shared resource guarantees exclusive access to the shared resource for each worker process in its critical section.

Since the replicated worker program is first-order, it is easy to see that we can model it as an ACPS. We omit the straightforward description of the ACPS implementing this program. In Figure 4.7 we present a diagram for a preorder that can be derived for this program. A non-terminal L_{i-j} models the code of line numbers i to j. We have omitted non-terminals L_{i-j} from the diagram in Figure 4.7 whose ordering is trivially seen. Note that non-tail recursive calls are potentially problematic. In the example, the recursive call to setup_network() in the definition of setup_network is non-tail recursive, but only places a send action (L_{13}) onto the stack, thus causing no harm. Note that in the partial order we may thus set $\text{setup\_network} =_{\text{shape}} L_{6-12} =_{\text{shape}} L_{13}$.

The only other non-tail recursive calls occur in do_task: the call to decompose_task poses no

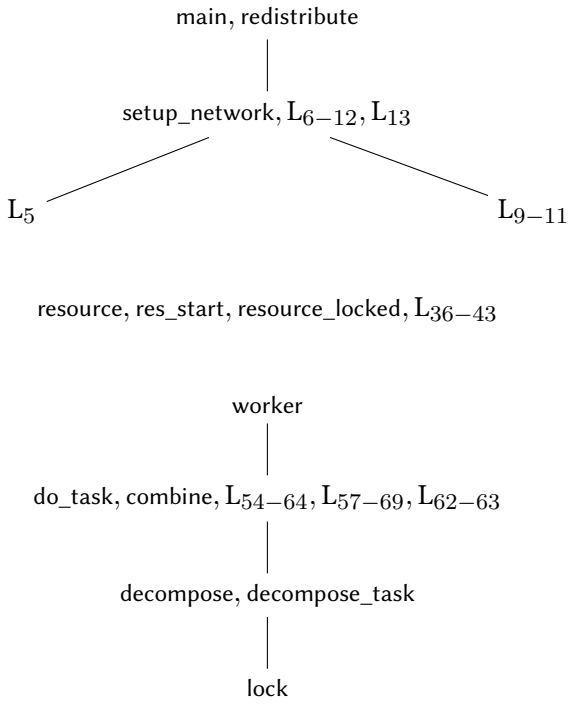


Figure 4.7: A variant of a Hasse diagram depicting a preorder $\geq_{\text{shape}}$ witnessing that the replicated worker example satisfies the shaped constraint. Note, some line numbers for which the orderings are straightforward are omitted. Nodes are sets of $\geq_{\text{shape}}$-equivalent elements.

threat since decompose_task does not invoke do_task again. The two recursive calls to do_task which either place procedure calls without concurrent actions (L_{57-69}) or send actions (L_{62-63}) on the stack are harmless. The preorder $\geq_{\text{shape}}$ shows that they can all be set $=_{\text{shape}}$-equivalent to do_task. Therefore, Proposition 4 tells us that the replicated worker program satisfies the shaped stack constraint.

We now proceed to give an account of the literature that is related to our work.

4.6 Related Work

In this section, we give an overview of the literature on decidability results for asynchronously communicating pushdown systems and other concurrent pushdown systems.

Asynchronously Communicating Pushdown Systems. Mukund et al. were the first to investigate dynamic networks of finite state processes that communicate asynchronously via a fixed finite number of unordered channels [Muk+98b; Muk+98a], i.e. ACS. They identified the equivalence of ACS and Petri nets and investigated regularity of ACS [Muk+98b]. Further, Mukund et al. studied asynchronous protocols that are robust with respect to the unordered nature of channels [Muk+98a]. In an influential paper [SV06] in 2006, Sen and Viswanathan showed that safety

verification is decidable for recursive programs with asynchronous atomic procedure calls. They introduced the empty-stack constraint on ACPS to model programs that are written in task-based asynchronous programming style which are often referred to as *asynchronous programs*. Sen and Viswanathan observed the applicability of Parikh's Theorem [Par66] and showed that recursive procedures in asynchronous programs can be accurately modelled by regular multiset automata which give rise to WSTSs. Further, Sen and Viswanathan established EXPSPACE-hardness of control-state reachability for asynchronous programs by reduction from the halting problem of *net programs* [Esp98]. Building on this, Jhala and Majumdar [JM07] developed an extension for asynchronous programs in BLAST [Hen+02], a highly competitive on-the-fly software model-checker employing CEGAR. The state space that is explored by Jhala and Majumdar's algorithm contains counters for pending asynchronous procedure calls. The exploration is reminiscent of the *expand, enlarge, and check* (EEC) [GRB04] algorithm scheme. Both an under-approximation (bounding counters at k) and an over-approximation (counters are widened to ∞ at $k+1$) are explored for successively larger k. The under- and over-approximations are guaranteed to converge for large enough k. Despite the EXPSPACE-hardness of the problem, Jhala and Majumdar observed that their algorithm only explored state spaces and k that are sufficiently small on realistic benchmarks and that their verification procedure thus performs well in practise. Liveness properties of asynchronous programs were extensively studied by Ganty et al. [GMR09]. They presented an encoding into Petri nets that embeds Parikh automata into the net that model recursive procedures. Decision results on Petri nets are then applicable and can be exploited to decide boundedness, fair termination, and fair non-starvation for asynchronous programs. Further, Ganty et al. showed how fair-termination and fair-starvation for Petri-nets can be reduced to the satisfiability problem for Presburger arithmetic. However, the best known constructions of Parikh automata are exponential [Esp+11]. Thus, Ganty and Majumdar improved on their construction [GM12] by combining an encoding of CCFG into Petri nets [Esp97], due to Esparza, with recent results [Esp+11] on Parikh's Theorem. In [Esp+11], Esparza et al. show that for every CCFG $\mathcal{G}$ there exists a $k \geq 1$, polynomial in the size of $\mathcal{G}$, such that the entire language of $\mathcal{G}$ can be generated by derivations of index k. Ganty and Majumdar exploited this result to obtain a compact encoding of CCFGs into Petri nets (see Section 2.4). Their encoding is linear in the size of the CCFG [GM12] and thus yields the polynomial-time interreducibility of a variety of verification problems for asynchronous programs to decision problems on Petri nets. Thus, safety verification, boundedness, and termination are EXPSPACE-complete, while configuration-reachability, fair-termination, and model-checking liveness properties are polynomial-time equivalent to Petri net reachability. Moreover, Ganty and Majumdar studied decision properties of various extension of asynchronous programs that allow capabilities such as cancellation of, or tests of absence of, asynchronous procedure calls. Ganty and Majumdar show that the former extensions are equivalent to Petri nets with reset arcs and Petri nets with zero tests, respectively [GM12]. Further extensions that allow the modelling of real-world asynchronous task scheduling systems such as *Grand Central Dispatch*, which is available on *MacOSX*, *iOS*, and *FreeBSD*, were investigated by Geeraerts et al. [GHR13]. Emmi et al. studied under-approximation techniques for asynchronous programs with prioritized task-buffers [ELQ12]. Their method is based on sequentialisation, bounds context-switches, and task-reorderings to obtain a practical and competitive analysis. A general-class of under-approximation techniques for asynchronous programs based on sequentialisation is investigated in [EOT14]. Given a suitable scheduler that bounds

context-switches or synchronisations, Emmi et al. encode an asynchronous program into a sequential program which enables mature program analyses. Many non-trivial extensions of Sen and Viswanathan's model have been proposed that allow e.g. the dynamic creation of task buffers [EGM14] or WQO stack alphabets [CO13]. An extension proposed by Chadha and Viswanathan is possibly the most general [CV07]. It applies to any computational model for which reachability is decidable. The chosen model is augmented with a well-quasi-ordered auxiliary storage such as counters. Chadha and Viswanathan then formulate an analogue of the empty-stack restriction for such WQO auxiliary storages. Their results imply that algorithmic verification for e.g. concurrent higher-order (collapsible) pushdown systems that satisfy the empty-stack constraint is possible. Nevertheless, the empty-stack constraint remains a key restriction for the models that have been investigated.

The literature on ACPS is part of the vast body of work on concurrent pushdown systems. Numerous classes with decidable verification problems have been discovered. We give a brief overview of the literature in the following.

Communicating Pushdown Systems. Heußner et al. [Heu+10] studied a restriction on pushdown processes that communicate asynchronously via FIFO channels: a process may send a message only when its stack is empty, while message retrieval is unconstrained. Parallel flow graph systems are a model of fork-join parallelism and can be captured by the process algebra PA. Building on results by Lugiez and Schnoebelen [LS98] that the reachability set of a regular set remains regular for PA-terms, Esparza and Podelski present an algorithm for computing the forward- and backward-reachability sets for parallel flow graph systems that is linear in the size of the program [EP00]. Dynamic pushdown networks (DPN) where investigated by Bouajjani et al. [BMOT05]. DPN model dynamic networks of pushdown processes that can create new processes which may communicate results back to a parent process in a restricted fashion. Bouajjani et al. showed that DPN's predecessor operator pre^* preserves regularity and thus they obtain a polynomial-time backwards-reachability algorithm [BMOT05]. Recent work by Bouajjani and Emmi investigates several classes of recursively parallel programs [BE12a]. Pushdown processes may create new processes and await the termination of a child process' computation. Communication is hierarchical and limited to the point of creation and termination. A process may communicate values to a newly created child process. Upon termination, a child process may return a value to its parent. Bouajjani and Emmi identify this model as Expspace-hard and provide a 2Exptime algorithm for control-state reachability. Their algorithm is based on a reduction to coverability of BVAS [Dem+09]. The concurrency model is rich and subsumes one subclass that can model asynchronous values via futures (control-state reachability in P) and includes another subclass that corresponds to asynchronous programs. However, a combination of futures and asynchronous programs lies outside the decidable fragments proposed by Bouajjani and Emmi. In particular, modelling processes that synchronise on a shared resource, which can be modelled by shaped ACPS, is not possible in their model. Several other communicating pushdown systems have been explored such as visibly pushdown automata that communicate over FIFO-queues [BR13] and pushdown systems communicating over locks [Kah09a].

Verification techniques such as *thread-modular model-checking* that over-approximate correctness properties of concurrent pushdown systems have been studied [FQ03; Hen+03]. Under-

approximation techniques such as *context-bounded* [QR05], *phase-bounded* [BE12b], and *scope-bounded* [TN11] analyses have been explored (see Section 4.1). The phase-bounded restriction is reminiscent of the *reversal-bounded* restriction on counter-machines [Iba78]. Counters are classified as *incrementing* or *decrementing* and a change in classification is called a *reversal*. In reversal-bounded model-checking only paths are considered along which the number of reversals does not exceed an *a priori* fixed bound. The reversal-bounded restriction has also been studied in the context of pushdown systems [HL11; HL12]. The *ordered* restriction [ABH08] on a concurrent pushdown system requires a linear order $\leq$ on its finite number of processes. A pushdown process may push onto its stack at any time while it may only perform a pop if it is the least process (with respect to $\leq$) that has a non-empty stack. In this line of the literature, a unifying under-approximation technique for *multi pushdown systems* (MPDS), i.e. concurrent pushdown systems with a fixed finite number of processes and shared state, has been proposed by Madhusudan and Parlato called *bounded tree-width* model-checking [MP11] . In this setting, bounded tree-width model-checking subsumes ordered, context-bounded, phase-bounded, and scope-bounded model-checking. Instead of paths, a MPDS generates run graphs which can be interpreted as paths with annotation edges which justify e.g. corresponding pushes and pops. The tree-width of such a graph measures its similarity to a tree. Madhusudan and Parlato show that, given a MPDS $\mathcal{P}$ and a bound on tree-width B, MSO is decidable on $\mathcal{P}$'s run graphs with tree-width less than B. Bounded tree-width model-checking has also been studied in the context of MPDS that communicate over FIFO channels, satisfy the empty-stack constraint, and have a restricted communication topology [MP11; Heu12]. Following Madhusudan and Parlato, Cyriac et al. propose *split-width* [CGK12] as an alternative graph measure which is equi-expressive with tree-width. In the setting of MPDS with shared state, they show that model-checking restricted to paths satisfying combinations of either (i) *scope-boundedness or phase-boundedness*, or (ii) *scope-boundedness or ordered* are instances of bounded split-width model-checking and are thus decidable. In further work, Aiswarya et al. extended bounded split-width model-checking to MPDS which communicate over FIFO channels [AGK14]. The bounded split-width restriction yields decidability of reachability, MSO, and model-checking temporal logic. While the properties that can be decided using the under-approximation techniques of bounded tree/split-width are impressive, it is important to note that an *a priori* bound of the number of processes is necessary. Moreover, when MPDS with FIFO channels and no restricted communication topology are considered the split/tree-width can easily become unbounded [MP11].

Pattern-based verification has been introduced by Kahlon in [Kah09b]. Pattern-based verification considers synchronisation traces of concurrent pushdown systems that lie within a regular language that is *bounded*. A bounded regular language is characterised by a regular expression of the form $w_1^* \cdots w_n^*$ where all w_i are words over an alphabet Σ. In pattern-based verification, a family of context-free grammars $\mathcal{G}_i$ model the synchronisation traces of a number of concurrent recursive programs. A bounded regular language B is chosen to specify the pattern of synchronisation traces which are to be considered for verification. The procedure is then based on Kahlon's result that the intersection-emptiness problem, i.e. $\bigcap_{i=1}^{k} L(\mathcal{G}_i) \cap B \stackrel{?}{=} \emptyset$, is decidable for bounded regular B. Esparza and Ganty investigated the complexity of this intersection emptiness problem and showed it to be NP-complete [EG11; EGP14]. The decision procedure presented by Esparza and Ganty is based on an application of Parikh's theorem. Esparza and

Ganty construct an existential Presburger formula that is satisfiable if, and only if, the intersection is empty. They exploit the fact that Parikh images of context-free languages are semi-linear sets which are definable in existential Presburger arithmetic. Subsequently, an effective CEGAR method [Lon+12] has been developed using the pattern-based verification approach.

Partially Commutative Context-Free Grammars (PCCFG). Czerwinski et al. introduced partially commutative context-free grammars as a study in process algebra [CFL09]. They proved that bisimulation is NP-complete for BPC, a class of processes where the sequential composition of certain processes is commutative. Bisimulation is defined on the traces of BPC processes. The process algebra BPC subsumes both BPA [BK84] and BPP [Esp97]. Note that synchronisation between processes (which transforms PCCFG to APCPS) is not a feature considered by Czerwinski et al. In further work, Czerwinski et al. investigated partially commutative context-free languages [CL12] and their word reachability problem, which is NP-complete [CHL12].

4.7 Conclusion

Summary. In this chapter, we have introduced asynchronously communicating pushdown systems (ACPS) as an extension of ACS. We have shown that for the purposes of studying coverability, boundedness, and termination of ACPS we may consider ACPS in normal form. Further, we have illustrated how unordered message passing in ACPS naturally gives rise to the notion of commutativity and an independence relation. Exploiting the latter, we have demonstrated how a concurrent system generated by an ACPS can be modelled by asynchronous partially commutative pushdown systems (APCPS). We have introduced two operational semantics for APCPS, a standard and an alternative one. A major contribution of ours is a proof that the standard and alternative semantics of APCPS are tightly connected. This connection can be formalised as a weak bisimulation between the standard semantics and the co-universal powerset lifting of the alternative semantics of an APCPS. This result allows us to conclude that we can reduce a coverability decision question on the standard semantics to its counterpart in the alternative semantics. Another major contribution is the introduction of a novel class of ACPS which is identified by the shaped stack constraint. We have shown that for APCPS that satisfy the shaped stack constraint the alternative semantics becomes a strict post-effective WSTS which is well-partially-ordered yielding the decidability for coverability and boundedness. Further, for termination- and rewrite-bounded shaped APCPS, the alternative semantics is finitely branching and hence termination is decidable. Hence, our reductions imply that coverability is decidable for shaped ACPS. Moreover, we have presented an easy-to-check syntactic condition on ACPS/APCPS from which the shaped stack constraint can be deduced. Lastly, we have presented an overview of the literature on decidable classes of concurrent pushdown systems and compared our work on shaped ACPS to other formalisms.

Open Problems and Future Directions. A number of open problems remain concerning the decision problems boundedness and termination.

- A missing link for boundedness and termination lies at present between ACPS in normal and the standard semantics of APCPS. We conjecture that it is possible to exhibit a polyno-

mial time interreduction. However, it seems a more complicated and tighter bisimulation argument than we use in Proposition 2 is required to establish a reduction.

- Another open question is whether we can decide termination and boundedness questions on the alternative semantics instead of the standard semantics. We believe that this is possible for the termination-bounded and rewrite-bounded classes of APCPS. We further believe that the proof sketches of Conjectures 1 and 2 can be expanded to formal proofs.

Positive answers to these two questions would imply that decision questions of boundedness and termination for shaped ACPS can be reduced to the alternative semantics for APCPS that are rewrite- and termination-bounded resp. (Conjecture 3) and hence they would be decidable.

A Perspective on Applications of Shaped ACPS. Let us briefly comment on possible applications of shaped ACPS. The empty-stack constraint fits nicely with the atomicity requirement of procedure calls in asynchronous programs. The only synchronisations an atomic procedure call needs to make are immediately before and after its execution which may thus be performed with an empty call-stack. The shaped stack constraint enables a relaxation of the atomicity requirement: it allows non-trivial synchronisations during the execution of a procedure call. This capability allows us to model the worker processes in Example 15 by shaped ACPS. Such programs can be obtained from a suitable abstract interpretation of Erlang programs. As we have mentioned at the beginning of this chapter, we consider it an interesting research problem to extend Soter's framework [DKO12; DKO13] to produce shaped ACPS. Further, the development of practical algorithms for shaped ACPS is of natural interest.

Another interesting research direction is to consider extensions to asynchronous programs that can be modelled by shaped ACPS. An attractive extension for asynchronous programs is to weaken the atomicity requirements on procedures and allow an asynchronous procedure call to return a value *asynchronously*. Such an asynchronous value is often called a *future* or *promise*. Futures that can interact freely with other processes cannot be sequentialised in the general case. Further, they require synchronisation between the caller and the callee, which can be modelled by ACPS provided the shaped stack constraint remains satisfied. Futures have been investigated in the context of recursive programs with hierarchical and thus restricted communication [BE12a]. Another recent work [Boe+12] considers futures in a lock-based concurrency model and studies tail-recursive processes. We believe that the shaped stack restriction could be useful to extend the scope of algorithmic verification techniques beyond the models that have been considered in the literature.

In the next chapter, we investigate the computational complexity of coverability for ACPS that satisfy the shaped stack constraint. The WSTS toolbox from which decidability of coverability can be deduced unfortunately only allows the deduction of coarse complexity results. To overcome this obstacle, we introduce a non-trivial extension of Petri nets, *nets with nested coloured tokens* (NNCT), which enjoys a tight connection with APCPS. Our complexity results on NNCT imply that coverability for shaped ACPS/APCPS is Tower-complete and that boundedness and termination on the alternative semantics is Tower-hard.

Chapter 5

Nets with Nested Coloured Tokens

and a Complexity Analysis of Shaped ACPS

The work presented in this chapter is joint work with Luke Ong and is unpublished at the time of writing [KO14].

In this chapter, we investigate the complexity of key decision problems for shaped ACPS/ APCPS. For ACPS that satisfy the empty-stack constraint, Ganty and Majumdar established a tight connection with Petri nets [GM12]. Thanks to this connection central decision problems for verification, such as coverability, termination, and boundedness, are polynomial-time interreducible to their respective Petri net counterparts. This result is highly encouraging. However, it is non-trivial to find a suitable model that is closely related to shaped ACPS/APCPS. As we have seen in the previous chapter, the shaped stack constraint captures a larger class of ACPS than the empty-stack restriction. If we focus on the alternative semantics of APCPS, we notice three capabilities that appear to lie outside the scope of Petri nets. A process carries summaries which are multisets — unbounded in size — of commutative concurrency actions. A parallel composition of processes thus has a *nested multiset structure* (C_1) in the alternative semantics. Further, a summary may be *added to* (C_2) or *dispatched* (C_3) in one step. Is there an extension to Petri nets that can support these capabilities? We seek inspiration in *nested Petri nets* [LS99] and *Petri nets with transfer arcs* [Cia94] to define an extension of Petri nets that corresponds to shaped ACPS/APCPS. Nested Petri nets give rise to configurations that feature arbitrarily nested multisets, while the capability to dispatch a summary in one step is reminiscent of a transfer transition. Note that both nested Petri nets and Petri nets with transfer arcs have non-primitive recursive coverability problems. For our purposes, it is thus important to augment Petri nets with just enough expressivity (and no more) to capture the alternative semantics. For example, only singly nested multiset structures occur in the alternative semantics. Further, except for a dispatch, it is not possible to remove a single element from the multiset constituting a summary. This means a process cannot (directly) justify a transition by the contents of its summaries. Fortunately, these two facts are sufficient to avoid non-primitive complexities. We introduce a non-trivial extension of Petri nets that is designed to capture the alternative semantics of APCPS: *nets with nested coloured tokens* (NNCT). As the name suggests NNCT feature, among ordinary Petri net tokens, *complex tokens* that carry *coloured tokens*. Transitions may inject coloured tokens into a complex token; or transfer certain

coloured tokens—those whose colour is from a specified set of *active colours*—from a complex token to predefined places. We establish a connection between the alternative semantics of K-shaped APCPS and NNCT. Therefore, coverability of shaped ACPS/APCPS Exptime-interreduces to coverability of NNCT. Further, we show that we can reduce simple coverability, boundedness, and termination for a subclass of NNCT to their respective decision problems for shaped APCPS. Building on this connection, we establish that coverability for NNCT is Tower-complete, and termination and boundedness are decidable and Tower-hard. This implies Tower-completeness of coverability for shaped ACPS/APCPS and Tower-hardness for boundedness and termination on the alternative semantics.

In order to motivate the design of NNCT in more detail, let us highlight capabilities (C_1)–(C_3) of the alternative semantics and explain why an extension of Petri nets is required to model them.

A Closer Look at the Alternative Semantics

Let us examine the three features (C_1)–(C_3) of the alternative semantics in more detail. We will demonstrate which requirements they set on a suitable Petri net extension. For this purpose, we revisit the task server example presented in Chapter 4.

Example 16. Let us consider a setting of the task server example in which we have a function definition $\mathsf{do\_server}() \to ()$. and where $\mathcal{G}(\mathcal{P})$ is amended appropriately. It is easy to verify that we can reach the following configuration in the alternative semantics:

$$\begin{aligned}\mathsf{main} \lhd \emptyset &\to^*_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{server} \parallel \mathsf{despatcher} \lhd \emptyset \\ &\to_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{init\_despatcher}\ \mathsf{post\_task}\ \mathrm{L}_4\ [\mathsf{task\_bag} \,!\, \mathsf{stop}] \parallel \mathsf{despatcher} \lhd \emptyset\end{aligned}$$

Omitting empty mulitsets $\emptyset$, we can see that the state of the server process is an element of $(\mathcal{N}^{\neg\mathrm{com}} \times \mathbb{M}[\Sigma^{\mathrm{com}}])^3$, i.e. its state consists in part of multisets such as $[\mathsf{task\_bag} \,!\, \mathsf{stop}]$. If we write $\Pi \lhd \emptyset$ for the last configuration, it is immediate that Π is a multiset of such process states. Thus, the nested multiset structure (capability (C_1)) becomes visible in Π. This illustrates that a single process in the alternative semantics is infinite-state, since a summary may contain concurrency actions of an arbitrary number.

A process can interact with a summary via Rule (R'-2) and add the result of a precomputation to it (capability (C_2)):

$$\begin{aligned}\mathsf{main} \lhd \emptyset &\to^*_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{init\_despatcher}\ \mathsf{post\_task}\ \mathrm{L}_4\ [\mathsf{task\_bag} \,!\, \mathsf{stop}] \parallel \mathsf{despatcher} \lhd \emptyset \\ &\to^*_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{server}\ [\mathsf{task\_bag} \,!\, \mathsf{stop}] \parallel \mathsf{despatcher} \lhd \emptyset \\ &\to_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} \mathrm{L}_{22:3} \parallel \mathsf{init\_despatcher}\ \mathsf{post\_task}\ \mathrm{L}_4\ [\mathsf{task\_bag} \,!\, \mathsf{stop}, \mathsf{task\_bag} \,!\, \mathsf{stop}] \parallel \mathsf{despatcher} \lhd \emptyset.\end{aligned}$$

The summary $[\mathsf{task\_bag} \,!\, \mathsf{stop}]$ of L_8 remains when the server process recurses. However, when server is rewritten, the new summary $[\mathsf{task\_bag} \,!\, \mathsf{stop}]$ of L_8 is added to yield the summary

$$[\mathsf{task\_bag} \,!\, \mathsf{stop}, \mathsf{task\_bag} \,!\, \mathsf{stop}]$$

of $\mathrm{L}_8\,\mathrm{L}_8$. We observe that a single multiset/summary represents a potentially growing number of stack characters L_8. Note that there is no mechanism to *remove* a single, specified concurrency action from a summary. We can only add elements to a summary.

However, a process may dispatch the concurrency actions of a summary using Rules (R′-6) and (R′-7) (capability (C_3)). As far as the process is concerned, the dispatch is oblivious of the summary's content. For illustration, let us observe how the stop message sent by the main process unlocks the summary of $\mathrm{L}_8\,\mathrm{L}_8$.

$$\begin{aligned}
\mathsf{main} \lhd \emptyset &\to^*_{\mathcal{G}(\mathcal{P})_{\mathsf{alt}}} \mathsf{init_despatcher}\ \mathsf{post_task}\ \mathrm{L}_4\ [\mathsf{task_bag}!\mathsf{stop}, \mathsf{task_bag}!\mathsf{stop}] \parallel \mathsf{despatcher} \parallel \mathrm{L}_{22:3} \lhd \emptyset \\
&\to^*_{\mathcal{G}(\mathcal{P})_{\mathsf{alt}}} \mathrm{L}_4\ [\mathsf{task_bag} \,!\, \mathsf{stop}, \mathsf{task_bag} \,!\, \mathsf{stop}] \parallel \mathsf{despatcher} \parallel \mathrm{L}_{22:3} \lhd \emptyset \\
&\to^*_{\mathcal{G}(\mathcal{P})_{\mathsf{alt}}} \mathrm{L}_4\ [\mathsf{task_bag} \,!\, \mathsf{stop}, \mathsf{task_bag} \,!\, \mathsf{stop}] \parallel \mathsf{despatcher} \lhd [\mathsf{system} \mapsto [\mathsf{stop}]] \\
&\to^*_{\mathcal{G}(\mathcal{P})_{\mathsf{alt}}} [\mathsf{task_bag} \,!\, \mathsf{stop}, \mathsf{task_bag} \,!\, \mathsf{stop}] \parallel \mathsf{despatcher} \lhd \emptyset \\
&\to^*_{\mathcal{G}(\mathcal{P})_{\mathsf{alt}}} \mathsf{despatcher} \lhd [\mathsf{task_bag} \mapsto [\mathsf{stop}^2]]\ .
\end{aligned}$$

The fact that a configuration in the alternative semantics features a nested multiset structure poses a difficulty. In a Petri net we may think of a configuration or *marking* as a distribution of tokens over a set of places P, i.e. a configuration is a mapping of type $P \to \mathbb{M}[\{\bullet\}]$. As we can see, the definition admits no nesting of multisets. A similar difficulty arises in the definition of *nested Petri nets* [LS99]. Configurations of nested Petri nets describe markings where a token can be a whole (nested) Petri net. The concept of a U-marking allows us to generalise the definition of markings. Suppose U is a set of *token types*, then a U-marking is a distribution of U-tokens (i.e. elements of U) over a set of places P, i.e. a mapping of type:

$$P \to \mathbb{M}[U].$$

Hence, Petri nets have $\{\bullet\}$-markings, nested Petri nets have *nPN*-markings where *nPN* is the set of nested Petri nets. For our purposes, nested Petri nets are too expressive. Nested Petri nets allow *horizontal* and *vertical* synchronisations of tokens.

Example 17. To illustrate the difference, let us briefly give an example of a nested Petri net (see Figure 5.1). Suppose we have the nested Petri net $\mathcal{N}$ which has two places p_1 and p_2. Further, let us assume the tokens of $\mathcal{N}$ are drawn from the same ordinary Petri net $\mathcal{N}_I$ with possibly different markings. The outer net $\mathcal{N}$ has two transitions: the transition t_1 takes a token, i.e. $\mathcal{N}_I$-marking, from p_1 to p_2, the transition t_2 is guarded by the label l and also takes a token from p_1 to p_2. The *internal* Petri net $\mathcal{N}_I$ has two places p_1' and p_2', a transition t_1' which takes a token from p_1' to p_2', and a transition t_2' guarded by the label $\bar{l}$ which also takes a token from p_1' to p_2'. The transition t_2 and t_2' are meant to fire together and perform a vertical synchronisation. This is indicated by the label l and its counterpart $\bar{l}$. Let u be a marking of $\mathcal{N}_I$ that places one token in p_1', i.e. $u = [p_1' \mapsto [\bullet]]$, and let u' place a token in place p_2', i.e. $u' = [p_2' \mapsto [\bullet]]$. Placing u into place p_1 of $\mathcal{N}$, it is then possible to perform the following vertical synchronisation in one step:

$$[p_1 \mapsto [u]] \to_{\mathcal{N}} [p_2 \mapsto [u']]$$

using both transitions t_2 and t_2'. However, for this particular nested Petri net it is also possible to reach the same configuration along two horizontal transitions:

$$[p_1 \mapsto [u]] \to_{\mathcal{N}} [p_1 \mapsto [u']] \to_{\mathcal{N}} [p_2 \mapsto [u']]$$

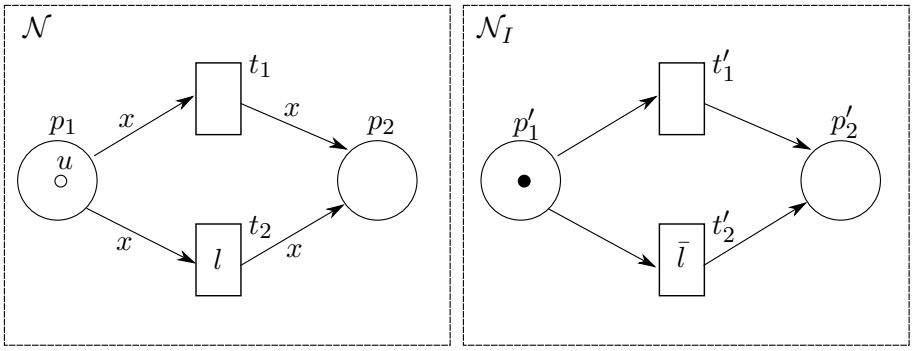


Figure 5.1: An example nested Petri net. On the right we display the configuration u and on the left we display $[p_1 \mapsto [u]]$. Note that arcs are labelled with variables to indicate where a token selected by an input arc is moved to by an output arc.

firing first t'_1 and then t_1. Note that the vertical transition involving t_2 and t'_2 is not possible at $[p_1 \mapsto [u']]$ since t'_2 is not enabled in u'.

Vertical synchronisations are powerful and easily allow nested Petri nets to simulate Petri nets with reset arcs [LS99]. Hence, reachability and boundedness are undecidable, and coverability is non-primitive recursive for nested Petri nets [LS99].

Fortunately, we do not require the full expressive power of vertical synchronisations. Let us return to the task server example and show how we can model a process of the alternative semantics as a U-marking for a suitable U.

Towards Nets with Nested Coloured Tokens. Consider the following state of the server process $\pi = \mathrm{L}_4\,[\mathsf{task\_bag} \,!\, \mathsf{stop}, \mathsf{task\_bag} \,!\, \mathsf{stop}]$. The process state π consists in part of the multiset

$$M := [\mathsf{task\_bag} \,!\, \mathsf{stop}, \mathsf{task\_bag} \,!\, \mathsf{stop}]$$

over elements of Σ^{com}. Suppose we have a set of places that contains a place L_4, it is easy to see that we can represent the state of π by the $\mathbb{M}[\Sigma^{\mathrm{com}}]$-marking

$$[\mathrm{L}_4 \mapsto [[\mathsf{task\_bag} \,!\, \mathsf{stop}, \mathsf{task\_bag} \,!\, \mathsf{stop}]]]\,,$$

i.e. M, which is a $\mathbb{M}[\Sigma^{\mathrm{com}}]$-token, is in place L_4. Suppose further we use a similar representation of channels as in the ACS case, i.e. for each channel c and message m there is an ordinary Petri net place (c, m). Consider the following path of π

$$\begin{aligned}\mathrm{L}_4\,M \lhd [\mathsf{system} \mapsto \mathsf{stop}] &\to_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} [\mathsf{task\_bag} \,!\, \mathsf{stop}, \mathsf{task\_bag} \,!\, \mathsf{stop}] \lhd \emptyset \\ &\to_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} \epsilon \lhd [\mathsf{task\_bag} \mapsto [\mathsf{stop}, \mathsf{stop}]]\end{aligned}$$

and let us imagine a Petri net extension that can generate the following transitions involving $(\mathbb{M}[\Sigma^{\mathrm{com}}] \cup \{\bullet\})$-markings:

$$\begin{aligned}[\mathrm{L}_4 \mapsto [M]\,, (\mathsf{system}, \mathsf{stop}) \mapsto [\bullet]] &\to [\epsilon \mapsto [[\mathsf{task\_bag} \,!\, \mathsf{stop}, \mathsf{task\_bag} \,!\, \mathsf{stop}]]] \\ &\to [\epsilon \mapsto [\emptyset]\,, (\mathsf{task\_bag}, \mathsf{stop}) \mapsto [\bullet, \bullet]]\,.\end{aligned}$$

The last step is reminiscent of a transfer transition. For each element $\mathsf{task\_bag} \,!\, \mathsf{stop}$ contained in M an ordinary Petri net token $\bullet$ is added to the place $(\mathsf{task\_bag}, \mathsf{stop})$. In this setting here, a mapping of type $\Sigma^{\mathrm{com}} \to \mathit{Chan} \times \mathit{Msg}$ can specify an assignment that tells us to which place an $a \in \Sigma^{\mathrm{com}}$ is *transferred* to. Hence, we observe that we require a Petri net extension that supports $(\mathbb{M}[\Sigma^{\mathrm{com}}] \cup \{\bullet\})$-markings and transfers that can be characterised by a mapping of type $\Sigma^{\mathrm{com}} \to \mathit{Chan} \times \mathit{Msg}$ to model the alternative semantics.

Unfortunately, representing a process state as a $\mathbb{M}[\Sigma^{\mathrm{com}}]$-token is not quite sufficient. Let us consider another example to illustrate how we can augment the requirements for our Petri net extension. Suppose we have a different state of the server process:

$$\pi' = [\mathsf{system} \,!\, \mathsf{begin}, \mathsf{system} \,!\, \mathsf{end}]\ \mathsf{post\_task}\, \emptyset\, \mathrm{L}_4\, [\mathsf{task\_bag}!\mathsf{stop}]\,.$$

In this case, the process state π' is comprised of three multisets over Σ^{com}. However, a simplification is possible. We can accurately represent these three multisets in a single multiset over an enlarged carrier set. Let us create three copies of Σ^{com}: $\Sigma^{\mathrm{com}}_0, \Sigma^{\mathrm{com}}_1$, and Σ^{com}_2 and write a_i for an element $a_i \in \Sigma^{\mathrm{com}}_i$ where $a \in \Sigma^{\mathrm{com}}$. Then, we can represent the process state π' as a $\mathbb{M}[\bigcup_{i=0}^{2} \Sigma^{\mathrm{com}}_i]$-token in a place representing the sequence of non-commutative non-terminals $\mathsf{post\_task}\, \mathrm{L}_4$:

$$[\mathsf{post\_task}\, \mathrm{L}_4 \mapsto [[(\mathsf{system} \,!\, \mathsf{begin})_0\,, (\mathsf{system} \,!\, \mathsf{end})_0\,, (\mathsf{task\_bag} \,!\, \mathsf{stop})_2]]]\,.$$

In this representation, modelling dispatch transition of the alternative semantics works slightly differently. Suppose we have the transition:

$$\pi' \lhd \emptyset \to_{\mathcal{G}(\mathcal{P})_{\mathrm{alt}}} \mathsf{post\_task}\, \emptyset\, \mathrm{L}_4\, [\mathsf{task\_bag}!\mathsf{stop}] \lhd [\mathsf{system} \mapsto [\mathsf{begin}, \mathsf{end}]].$$

Using the representation of π' as a $\mathbb{M}[\bigcup_{i=0}^{2} \Sigma^{\mathrm{com}}_i]$-token we can model this transition as a *'partial'* transfer where only elements of Σ^{com}_0 are involved:

$$\begin{aligned}[\mathsf{post\_task}\, \mathrm{L}_4 \mapsto [[(\mathsf{system} \,!\, \mathsf{begin})_0\,, (\mathsf{system} \,!\, \mathsf{end})_0\,, (\mathsf{task\_bag} \,!\, \mathsf{stop})_2]]] \\ \to [\mathsf{post\_task}\, \mathrm{L}_4 \mapsto [[(\mathsf{task\_bag} \,!\, \mathsf{stop})_2]]\,, (\mathsf{system}, \mathsf{begin}) \mapsto [\bullet]\,, (\mathsf{system}, \mathsf{end}) \mapsto [\bullet]]\,.\end{aligned}$$

Note that in this transition for each element $(c \,!\, \mathit{msg})_0 \in \Sigma^{\mathrm{com}}_0$ we place a $\bullet$-token in place (c, msg). We can make this assignment precise again with a mapping ζ of type

$$\zeta : \textstyle\bigcup_{i=0}^{2} \Sigma^{\mathrm{com}}_i \to \mathit{Chan} \times \mathit{Msg}$$

that is defined by $\zeta((c \,!\, \mathit{msg})_i) = (c, \mathit{msg})$. The partial transfer can then be indicated by considering Σ^{com}_0 as an *active* subset of $\bigcup_{i=0}^{2} \Sigma^{\mathrm{com}}_i$ along this transfer transition. Thus, if we have a $\mathbb{M}[\bigcup_{i=0}^{2} \Sigma^{\mathrm{com}}_i]$-token m and a partial transfer transition t specified by an active subset P, then P tells us whether an element $a_i \in m$ will be transferred along t, while ζ tells us where to transfer a_i. Hence, we can augment the requirements to the Petri net extension we seek. If we abstract away the particulars of this example, we can imagine elements of $\bigcup_{i=0}^{2} \Sigma^{\mathrm{com}}_i$ being represented by different *coloured tokens*, i.e. there is one colour for each element of $\bigcup_{i=0}^{2} \Sigma^{\mathrm{com}}_i$. Let us call such a set of colours *Col*. We would then expect a suitable Petri net extension to give rise to configurations of type:

$$(P \to \mathbb{M}[\mathbb{M}[\mathit{Col}]]) + \left(P' \to \mathbb{M}[\{\bullet\}]\right)$$

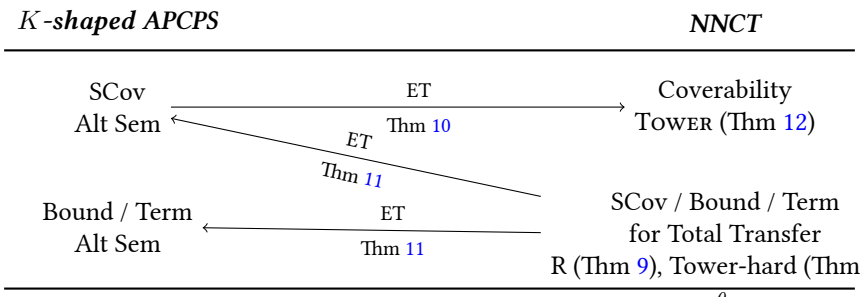


Figure 5.2: A map of the technical results presented in this chapter. "$P_1 \xrightarrow{\theta} P_2$" means that (decision problem) P_1 θ-reduces to P_2. *Legend: ET= exponential time, R= decidable (recursive), Alt Sem= alternative semantics, SCov= simple coverability, Bound= boundedness, and Term= termination.*

i.e. $\mathbb{M}[Col] \cup \{\bullet\}$-markings where we have a dedicated set of places P' for *simple* $\{\bullet\}$-tokens (representing channels) and a set of places P for *complex* $\mathbb{M}[Col]$-tokens representing processes[1]. This allows us to model capability (C_1). To model capability (C_2), the Petri net extension needs to allow for transitions that can insert a coloured token into a complex token, i.e. given a colour $c \in Col$ and a $\mathbb{M}[Col]$-token m the transition can produce $m \oplus [c]$. Further, to model the kind of partial transfers that we have presented we need a mapping $\zeta : Col \to P'$ and a set of *active colours* associated with each such partial transfer transition. This is enough to model capability (C_3). These are the basic ingredients of *nets with nested coloured tokens*.

Note, that these requirements illustrate that we do not need the full expressivity that nested Petri nets offer. We only require a singly nested multiset structure whereas nested Petri nets support an *a priori* fixed but arbitrary nested multiset structure. A further difference with respect to nested Petri nets is that we do not need the full power offered by vertical synchronisations. Moreover, the transfer transitions that we have described appear constrained when compared to Petri nets with transfer arcs. This seems to be an explanation of why the non-primitive recursive complexities of nested Petri nets and Petri nets with transfer arcs can be avoided in shaped ACPS/APCPS and NNCT.

Before we move on to give a formal presentation of NNCT, let us give a brief outline of the material that we present in the remainder of this chapter with the help of Figure 5.2.

A Look Ahead. In Section 5.1, we formally introduce nets with nested coloured tokens (Definition 22). We show that NNCT are strict WSTS and thus coverability, termination, and boundedness are decidable (Theorem 9). We then present how to encode the alternative semantics of a shaped APCPS into an NNCT (Section 5.2). This encoding yields an Exptime-reduction of coverability for shaped APCPS to NNCT coverability (Theorem 10). In order to transfer a lower bound of these decision problems on NNCT, we turn to *total transfer* NNCT which are a subclass of NNCT. In this subclass colours are always active. We show how a total transfer NNCT can be simulated by a shaped APCPS. This yields an Exptime-reduction from simple coverability, termination, and boundedness to their counterparts for shaped APCPS (Theorem 11). Thus, we focus on establishing the complexity of coverability, boundedness, and termination for NNCT.

[1]Note that there is an isomorphism of $\mathbb{M}[Col] \cong Col \to \mathbb{M}[\{\bullet\}]$ which allows us to view a multiset of coloured tokens as a distribution of $\bullet$-tokens over the set of colours *Col*.

We prove that coverability for NNCT is Tower-complete. To our knowledge, NNCT is the first extension of Petri nets, belonging to the class of nets with an infinite set of token types, that is proven to have primitive recursive coverability. To prove Tower-membership of coverability for NNCT (Theorem 12), we devise a geometrically inspired version of the Rackoff technique [Rac78], which was originally used to prove an Expspace upper bound for coverability of Petri nets (Section 5.3). We obtain non-elementary lower bounds and thus Tower-hardness of simple coverability, boundedness, and termination by adapting a Stockmeyer/Lipton ruler construction [Sto74; Lip76] to NNCT (Theorem 13).

In the next section we introduce NNCT, illustrate their operational semantics by an example, and show that they give rise to strict WSTS.

5.1 Nets with Nested Coloured Tokens

As we have seen in the previous section, the alternative operational semantics of APCPS has capabilities that appear to be beyond the expressive power of Petri nets. Configurations of the alternative semantics feature a nested multiset structure (capability (C_1)). Rules (R′-6) and (R′-7) seem to require a feature that allows the transfer or *ejection* of elements from inside a nested multiset (capability (C_3)). Fortunately, we know from the literature [Esp97; GM12; Esp+11] that computing the summary of a commutative non-terminal, as performed by Rule (R′-2), can be achieved by a Petri net (see Section 2.4). To model transitions generated by Rule (R′-2) thus only requires the capability to *inject* elements into a nested multiset (capability (C_2)).

Inspired by *nested Petri nets* [LS99], which give rise to configurations of nested structures of multisets, we introduce *nets with nested coloured tokens* which feature multisets of multisets and vertical transfers. NNCT are designed with the necessary features to implement the alternative semantics for APCPS, while also allowing APCPS to simulate NNCT. Let us first define NNCT:

Definition 22. A *net with nested coloured tokens* (NNCT) is a quintuple $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$ where $\mathcal{P}^S$, $\mathcal{P}^C$, $\mathcal{P}^{col}$, and $\mathcal{R}$ are respectively finite sets of *simple places*, *complex places*, *colours*, and *rules*; and the *colour mapping* ζ is a function from $\mathcal{P}^{col}$ to $\mathcal{P}^S$.

Markings. To define the set of configurations that a NNCT generates, we employ the concept of U-markings. Let us first define the set of *complex tokens* $\mathcal{M}^{colour}$ to be the set of markings over $\mathcal{P}^{col}$:

$$\mathcal{M}^{colour} := \left\{ m \mid m : \mathcal{P}^{col} \to \mathbb{M}[\{\bullet\}] \right\}$$

We also view a complex token, i.e. an element of $\mathcal{M}^{colour}$, as a multiset of *coloured tokens*. Note that the isomorphism $\mathcal{M}^{colour} \cong \mathbb{M}[\mathcal{P}^{col}]$ illustrates how we can view a complex token both as a $\{\bullet\}$-marking over places $\mathcal{P}^{col}$ and as a multiset of coloured tokens. It is then easy to specify markings over simple places and complex places as

$$\mathcal{M}^{simple} := \left\{ m \mid m : \mathcal{P}^S \to \mathbb{M}[\{\bullet\}] \right\}, \qquad \mathcal{M}^{complex} := \left\{ m \mid m : \mathcal{P}^C \to \mathbb{M}[\mathcal{M}^{colour}] \right\}.$$

Thus, a marking for $\mathcal{P}^C$ fills each complex place with a multiset of complex tokens. Similarly, a marking for $\mathcal{P}^S$ is a distribution of *simple tokens*, i.e. $\bullet$-tokens, over the simple places. A *configuration* of a NNCT is the disjoint union of a marking over simple places and a marking over complex places:

$$\mathit{Config} := \left\{ m + m' \mid m \in \mathcal{M}^{simple}, m' \in \mathcal{M}^{complex} \right\}.$$

Rules. We partition the set of rules $\mathcal{R}$ into *SimpleRules*, *ComplexRules*, and *TransferRules*.
A rule $r \in$ *SimpleRules* is a pair $r = (I, O)$ such that $I, O \in$ *Config* and I satisfies the following condition:

$$I(p)(m) = 0 \text{ if both } p \in \mathcal{P}^C \text{ and } m \neq \mathbf{0}.$$

As a reminder, the map $\mathbf{0}$ maps any argument x to the empty multiset. Hence, if p is a complex place, then $I(p)(m)$ may only be positive if m is the empty complex token $\mathbf{0}$.
A rule $r \in$ *ComplexRules* is a pair $r = ((p, I), (p', c, O))$ where p, p' are complex places $(p, p' \in \mathcal{P}^C)$, c is a complex token ($c \in \mathcal{M}^{colour}$), and I, O are elements of the set of simple configurations *Config*$_S$ ($I, O \in$ *Config*$_S$) which is defined by

$$\textit{Config}_S := \{m + \mathbf{0} \mid m \in \mathcal{M}^{simple}, \mathbf{0} \in \mathcal{M}^{complex}\}.$$

A rule $r \in$ *TransferRules* is a pair $((p, I), (p', P, O))$ such that p, p' are complex places $(p, p' \in \mathcal{P}^C)$, I, O are simple configurations ($I, O \in$ *Config*$_S$), and $P \subseteq \mathcal{P}^{col}$ is a set of *active colours* such that the following condition is met: $\zeta \restriction P : P \to \zeta(P)$ is bijective, i.e. ζ^{-1} is well-defined on $\zeta(P)$.
Given a rule $r \in \mathcal{R}$, we sometimes need to refer to the mappings I, O in which case we write I_r and O_r.
Operational Semantics. The operational semantics of NNCT is defined by three transition rules corresponding to *SimpleRules*, *ComplexRules*, and *TransferRules*. Let $s \in$ *Config*.

(R1) Suppose $r = (I, O) \in$ *SimpleRules*. We say rule r is *enabled* at s, if $s = s_0 \oplus I$ for some $s_0 \in$ *Config*. In this case, there is a transition $s \xrightarrow{r}_{\mathcal{N}} s'$ using rule r where $s' := s_0 \oplus O$.

(R2) Suppose $r = ((p, I), (p', c, O)) \in$ *ComplexRules*. Rule r is said to be enabled at s if there exists a $s_0 \in$ *Config* such that $s = s_0 \oplus I \oplus [p \mapsto [m]]$. Then, there is a transition $s \xrightarrow{r}_{\mathcal{N}} s'$ using rule r where $s' = s_0 \oplus O \oplus [p' \mapsto [m \oplus c]]$.

(R3) Suppose $r = ((p, I), (p', P, O)) \in$ *TransferRules*. Then, we say rule r is *enabled* at s, just if there exists a $s_0 \in$ *Config* such that $s = s_0 \oplus I \oplus [p \mapsto [m]]$. In this case, there is a transition $s \xrightarrow{r}_{\mathcal{N}} s'$ using rule r such that

$$s' = s_0 \oplus O \oplus \left(m_P \circ \zeta^{-1}\right) \oplus \left[p' \mapsto \left[m_{\overline{P}}\right]\right]$$

where $m_P = m \restriction P$ and $m_{\overline{P}} = m \restriction (\mathcal{P}^{col} \setminus P)$.

A simple rule may insert new complex tokens into complex places, since $I, O \in$ *Config*; it may also remove empty complex tokens. A complex rule removes a complex token m from a complex place p and inserts a new complex token $m \oplus c$ to a complex place p'. A set of active colours $P \subseteq \mathcal{P}^{col}$ is associated to each transfer rule, which removes a complex token m from a complex place p, and inserts into p' the complex token $m_{\overline{P}}$ which is obtained from m *less* all tokens with an active colour c. For each such $c \in P$, these tokens are transferred instead to the simple place $\zeta(c)$ which corresponds to a *multiset-addition* of $m_P \circ \zeta^{-1}$.

We observe that NNCT fulfil the requirements we set out in the previous section. NNCT give rise to configurations with complex tokens that provide a nested multiset structure (capability (C_1)), allow to inject coloured tokens into complex tokens (capability (C_2)) using a complex rule, and feature transfer transitions (capability (C_3)). Before we illustrate the operational semantics of NNCT on an example, let us define a subclass of NNCT that restricts transfer transitions.

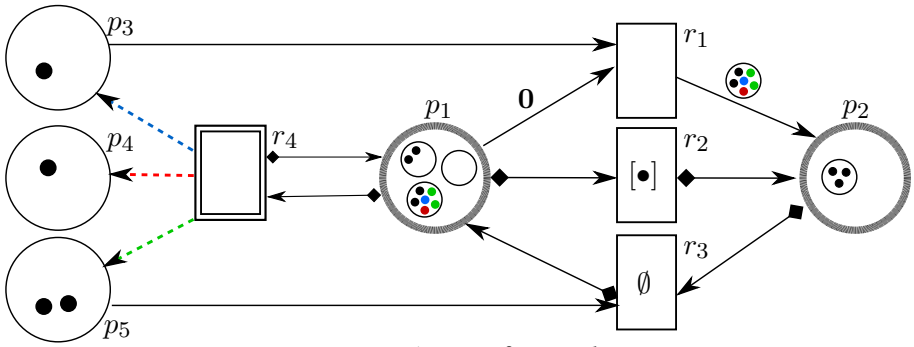


Figure 5.3: The net of Example 18

Total Transfer NNCT. Let $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$ be an NNCT. Suppose r is a transfer rule such that $r = ((p, I), (p', P, O))$ and $P = \mathcal{P}^{col}$, then we say r is a *total* transfer rule. If all transfer rules of $\mathcal{N}$ are total then we say $\mathcal{N}$ is a *total transfer* NNCT.

To relate total transfer NNCT to shaped APCPS, we may think of them as modelling APCPS processes that require *at most one* non-empty summary. This feels like a significant restriction. However, it turns out that total transfer NNCT are sufficient to obtain a lower bound on the complexities of coverability, termination, and boundedness for NNCT.

Before we move on to investigate the decision properties of NNCT, let us present an example of NNCT to illustrate the operational semantics in detail.

Example 18. We define a NNCT $\mathcal{N}$ with complex places $\mathcal{P}^C = \{p_1, p_2\}$, a set of simple places $\mathcal{P}^S = \{p_3, p_4, p_5\}$, colours $\mathcal{P}^{col} = \{\mathit{red}, \mathit{green}, \mathit{blue}, \mathit{black}\}$, and colour mapping:

$$\zeta = [\mathit{red} \mapsto p_4, \mathit{green} \mapsto p_5, \mathit{blue} \mapsto p_3] .$$

The rule $r_1 = (\{p_1 \mapsto [\mathbf{0}], p_3 \mapsto [\bullet]\}, \{p_2 \mapsto [m_1]\})$ is a simple rule that can create the complex token $m_1 = \{\mathit{black} \mapsto [\bullet^2], \mathit{blue} \mapsto [\bullet], \mathit{green} \mapsto [\bullet^2], \mathit{red} \mapsto [\bullet]\}$. The complex rules are:

$$r_2 = ((p_1, \emptyset), (p_2, \{\mathit{black} \mapsto [\bullet]\}, \emptyset)), \qquad r_3 = ((p_2, \{p_5 \mapsto [\bullet]\}), (p_1, \emptyset, \emptyset))$$

The transfer rule r_4 is defined by $r_4 = ((p_1, \emptyset), (p_1, \{\mathit{red}, \mathit{blue}, \mathit{green}\}, \emptyset))$. We note that $\mathcal{N}$ is *not* a total transfer NNCT. We can graphically represent NNCT similarly to Petri nets. In Figure 5.3 we can see a configuration of $\mathcal{N}$. The complex place p_1 contains the complex token m_1, the empty complex token $\mathbf{0}$ (displayed in p_1 as an empty circle), and the complex token $m_2 = \{\mathit{black} \mapsto [\bullet^2]\}$. The complex token $m_3 = \{\mathit{black} \mapsto [\bullet^3]\}$ is located in complex place p_2. We graphically distinguish transfer rules by using double edged boxes and we display ζ and the set of active colours using dashed arrows. We indicate the origin and destination for complex tokens moved by complex and transfer rules with additional arcs that have a ◆-end. Complex rules are displayed as a box labelled with the colour marking c to inject. Simple rules may have arcs from complex places labelled with $\mathbf{0}$ (indicating the removal of the empty complex token) and arcs to complex places that are labelled with the complex token to be added. With the help of Figure 5.3, we will illustrate the operational semantics. Simple rule r_1 removes an empty complex token from p_1 and a simple token from p_3, and adds the complex token m_1 to p_2. The

complex rule r_2 non-deterministically selects a complex token e.g. m_2 and moves it to p_2 while inserting one *black*-coloured token into m_2, i.e. m_2 becomes m_3. A complex rule may move a complex token, e.g. rule r_3 non-deterministically selects a complex token from p_2 and moves it to p_1 while consuming a simple token from p_5. Let us imagine for a moment that the simple places p_3, p_4, and p_5 are all empty. Rule r_4 non-deterministically selects a complex token in p_1, m_1 say, and distributes its *blue*, *green*, and *red*-coloured tokens as indicated by the coloured dashed arrows to the simple places p_3, p_4, and p_5 and turns them into simple (black) tokens. The result is that p_3 and p_4 contain a simple token each and p_5 two as displayed in Figure 5.3; the token m_1, less its *blue*, *green*, and *red*-coloured tokens, remains in place p_1 and becomes m_2.

We now focus on the decision properties of NNCT. In the following, we define an order on NNCT configurations to obtain a PSTS, and we show this yields a strict WSTS.

Decision Properties of NNCT. It turns out that NNCT enjoy good model-checking properties. This is thanks to NNCT being strict WSTS. Let us briefly recall the definition of WSTS [FS01]: A PSTS $\mathcal{S}$ is a transition system equipped with a preorder $\leq_{\mathcal{S}}$. Suppose $\leq_{\mathcal{S}}$ is a *well-quasi-order* (WQO) over a set S, then a *well-structured transition system* (WSTS) is a PSTS $\mathcal{S} = (S, \to_{\mathcal{S}}, \leq_{\mathcal{S}})$ such that $\to_{\mathcal{S}}$ is monotone with respect to $\leq_{\mathcal{S}}$. A WSTS $\mathcal{S}$ is said to be *strict*, if $\to_{\mathcal{S}}$ is strictly monotone with respect to $\leq_{\mathcal{S}}$ (see Sections 2.1 and 2.3).

First, let us turn NNCT into PSTS. For this purpose, we equip *Config* and $\mathcal{M}^{colour}$ with preorders. NNCT allow the removal of empty complex tokens via simple rules. Thus, we need to define slightly different preorders on *Config* and $\mathcal{M}^{colour}$ than we would expect. This is necessary to ensure that NNCT are strict WSTS. Let $m, m' \in \mathcal{M}^{colour}$ be two complex tokens. We define $m \leq_{\mathcal{M}^{colour}} m'$ if for all $p^{col} \in \mathcal{P}^{col}$ we have either $0 = |m(p^{col})| = |m'(p^{col})|$, or $0 < |m(p^{col})| \leq |m'(p^{col})|$. Hence, the ordering $m \leq_{\mathcal{M}^{colour}} m'$ ensures that if m contains no tokens of colour p^{col}, then m' also contains no p^{col}-coloured tokens. This implies that if m is the empty complex token (i.e. $m = \mathbf{0}$), then also $m' = \mathbf{0}$. We can now lift this ordering to NNCT configurations using the standard multiset extension. For configurations $s, s' \in \mathit{Config}$, we define $s \leq_{\mathit{Config}} s'$ if for all $p \in \mathcal{P}^S$ we have $s(p) \leq_{\mathbb{M}[\{\bullet\}]} s(p')$ and for all $p' \in \mathcal{P}^C$ we have $s(p') \leq_{\mathbb{M}[\mathcal{M}^{colour}]} s(p')$. As we have mentioned before, the preorder $\leq_{\mathcal{M}^{colour}}$ is non-standard yet it remains a WQO:

Lemma 4. $(\mathcal{M}^{colour}, \leq_{\mathcal{M}^{colour}})$ and $(\mathit{Config}, \leq_{\mathit{Config}})$ are WQO.

Proof Idea. To see that $(\mathcal{M}^{colour}, \leq_{\mathcal{M}^{colour}})$ is a WQO, we decompose $\mathcal{M}^{colour}$ into a disjoint union of sets that are easily seen to be well-quasi-ordered and give rise to $\leq_{\mathcal{M}^{colour}}$. We define a family of subsets $\mathcal{M}(P)$ of $\mathcal{M}^{colour}$ parametrised by subsets P of $\mathcal{P}^{col}$. An element M of $\mathcal{M}(P)$ does not contain any token of colours $p \in P$ and at least one p'-coloured token for each colour $p' \notin P$. It is easy to see that the isomorphism $\mathcal{M}^{colour} \cong \sum_{P \subseteq \mathcal{P}^{col}} \mathcal{M}(P)$ holds and that $\sum_{P \subseteq \mathcal{P}^{col}} \mathcal{M}(P)$ is a finite disjoint union. If we restrict the normal multiset order to elements of $\mathcal{M}(P)$ it is clear that we obtain a WQO for $\mathcal{M}(P)$ and this order coincides with $\leq_{\mathcal{M}^{colour}}$ on elements of $\mathcal{M}(P)$. Thus, we observe that $\leq_{\mathcal{M}^{colour}}$ arises by composing the orders on $\mathcal{M}(P)$ as a finite disjoint union. Hence, $\leq_{\mathcal{M}^{colour}}$ is a WQO. It is then a routine exercise to lift the well-orderedness of $\leq_{\mathcal{M}^{colour}}$ to $\leq_{\mathit{Config}}$.
For a formal proof we refer the reader to Appendix C.1. □

Thanks to the refined order $\leq_{\mathcal{M}^{colour}}$, the transition relation $\rightarrow_{\mathcal{N}}$ of a NNCT $\mathcal{N}$ is strictly monotone with respect to $\leq_{Config}$. Thus, we can conclude:

Proposition 5. Given an NNCT $\mathcal{N}$, $(Config, \rightarrow_{\mathcal{N}}, \leq_{Config})$ is a strict WSTS.

Proof Idea. The fact $\rightarrow_{\mathcal{N}}$ is strictly monotone is mainly trivial. The only case that requires care is that of a simple rule that can remove an empty complex token. Note that the order $\leq_{Config}$ ensures that for two configurations s and s' such that $s <_{Config} s'$ the number of empty complex tokens in s for a complex place p is strictly less than in s', i.e. $|s(p)(\mathbf{0})| < |s'(p)(\mathbf{0})|$. Hence, if we remove an empty complex token from s in complex place p we can remove an empty complex token from s' in place p maintaining the strict ordering.
For a full formal proof see Appendix C.1. □

It is straightforward to see that the set $\uparrow Pred(\uparrow\{s\})$ is effectively computable and the WQO $\leq_{Config}$ is decidable. Since NNCT are strict WSTS this implies that the coverability, termination, and boundedness problems are decidable [FS01]:

Theorem 9. *Coverability, termination, and boundedness are decidable for NNCTs.*

As in the case of ACPS and APCPS, we consider a simplification of the coverability problem: *simple coverability*. A NNCT coverability query $\mathcal{Q} = (\mathcal{N}, s_0, s_{\text{cov}})$ is said to be *simple* if s_0 and s_{cov} do *not* contain complex tokens, i.e. $s_0(p) = s_{\text{cov}}(p) = \emptyset$ for all $p \in \mathcal{P}^C$.

NNCT are able to implement the alternative semantics of APCPS and *vice versa* by construction. In the next section, we illustrate how the alternative semantics of shaped APCPS can be modelled as an NNCT and how this yields an Exptime-reduction for simple coverability. Further, we show how a total transfer NNCT can be encoded into APCPS. This encoding yields Exptime-reductions from NNCT simple coverability, boundedness, and termination to their shaped APCPS counterparts.

5.2 NNCT and the Alternative Semantics of Shaped APCPS

In this section, we demonstrate how the alternative semantics of a shaped APCPS can be encoded into an NNCT and *vice versa*. We first concentrate on the direction from shaped APCPS to NNCT. The NNCT that we obtain weakly bisimulates the given shaped APCPS. The encoding is exponential in the size of the shaped APCPS and yields an Exptime-reduction of coverability from shaped APCPS to NNCT. We represent processes as complex tokens and channel contents as simple tokens. We use a slightly modified CCFG widget *à la* Ganty and Majumdar [GM12] and the capability to inject coloured tokens into complex tokens to implement Rule (R′-2). The dispatch of summaries (Rules (R′-6) and (R′-7)) is modelled by transfer rules.

Reducing Simple Coverability from Shaped APCPS to NNCT

In the interest of readability, we simplify part of the encoding and reduce the notational burden. We start with a demonstration of how a configuration of the alternative semantics is represented in a NNCT. Then, we explain how we can implement the alternative semantics in the rules of an NNCT so that this representation is maintained.

Let us fix a K-shaped APCPS $\mathcal{G} = (\Sigma, I, \mathcal{N}, \mathcal{R})$ and a simple coverability query

$$\mathcal{Q}_{\mathcal{G}} = (\mathcal{G}_{\text{alt}}, \Pi_0 \lhd \Gamma_0, A_1^{\text{cov}} \parallel \cdots \parallel A_n^{\text{cov}} \lhd \Gamma^{\text{cov}})$$

throughout this section. The encoding will yield an NNCT $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$ and a simple coverability query $\mathcal{Q}$ for $\mathcal{N}$. We start with an account on how we employ $\mathcal{N}$'s places to capture a configuration of the alternative semantics.

Representing Configurations. Suppose we have a K-shaped process π that we can write as

$$\pi = \delta\, M_0\, X_1\, M_1 \cdots X_n\, M_n,$$

where the X_i are non-commutative non-terminals ($X_i \in \mathcal{N}^{\neg\text{com}}$), $n \leq K$, the summaries $M_i \in \mathbb{M}[\Sigma^{\text{com}}]$, and $\delta \in \mathcal{N} \cup \Sigma \cup \{\epsilon\}$. Following our explanations at the start of this chapter, we want to represent the state of this process as the following configuration:

$$\left[p_{\delta\, X_1\, X_2 \cdots\, X_n} \mapsto \left[M_0' \oplus M_1' \oplus \cdots \oplus M_n'\right]\right]$$

where the multisets M_i' are tagged versions of the M_i. For this purpose, $\mathcal{N}$ has a complex place $p_{\delta\, X_1 \cdots\, X_n}$ for each possible choice of $X_1, \ldots, X_n$, and δ. Note that the number of choices is bounded by $O((|\Sigma| + |\mathcal{N}|) \times |\mathcal{N}^{\neg\text{com}}|^K)$. As before, we want to represent the multisets $M_0, M_1, \ldots, M_n$ by one multisets. We know that $\mathcal{G}$ is K-shaped. So for any given reachable process we need to represent at most $K+1$ summaries. Hence, we can create $K+1$ copies of Σ^{com}: $\Sigma_0^{\text{com}}, \Sigma_1^{\text{com}}, \ldots, \Sigma_K^{\text{com}}$. We tag elements a of Σ^{com} with an index i (denoted by a_i) that creates a link to the summaries of a process. Then, a multiset $M \in \mathbb{M}[\bigcup_{i=0}^{K} \Sigma_i^{\text{com}}]$ can represent the multisets $M_0, M_1, \ldots, M_n$ by interpreting elements of e.g. $\Sigma_{K-(n-i)}^{\text{com}}$ as the contents of M_i, i.e.

$$M \restriction \Sigma_{K-(n-i)}^{\text{com}} = [a_{K-(n-i)} \mapsto M_i(a) \mid a \in \Sigma^{\text{com}}].$$

For the above configuration, $M = M_0' \oplus \cdots M_n'$. Thus, $\mathcal{N}$ has $(K+1) \times |\Sigma^{\text{com}}|$ colours: one colour for each element of $\bigcup_{i=0}^{K} \Sigma_i^{\text{com}}$. In the full proof of Theorem 10 (see Appendix C.2), we have more than one complex place $p_{\delta\, X_1\, X_2 \cdots\, X_n}$. These places encode additional information that we require to correctly model partial summaries and cover two edge cases. To improve readability, we omit these technical details here. However, we want to stress that a process π is represented by a *set* of configurations $\mathcal{F}(\pi)$ and that we may think of the configuration

$$\left[p_{\delta\, X_1\, X_2 \cdots\, X_n} \mapsto \left[M_0' \oplus M_1' \oplus \cdots \oplus M_n'\right]\right]$$

as representative of the configurations in $\mathcal{F}(\pi)$. This representation easily lifts to parallel compositions of processes. Suppose we have $\Pi = \pi_1 \parallel \cdots \parallel \pi_n$ then Π is represented by a set of configurations $\mathcal{F}(\Pi)$. The set $\mathcal{F}(\Pi)$ consists of configurations $m_1 \oplus \cdots \oplus m_n$ where each $m_i \in \mathcal{F}(\pi_i)$. We represent channels in the familiar way. For each *msg* $\in$ *Msg* and *c* $\in$ *Chan*, we introduce a simple place $p_{(c,msg)}$. The contents of a set of channels Γ is then represented in the usual way by the configuration $\mathcal{F}(\Gamma)$: for each occurrence of message *msg* in channel *c* there is one simple token in the place $p_{(c,msg)}$ in configuration $\mathcal{F}(\Gamma)$. Thus, we represent a configuration $\Pi \lhd \Gamma$ of $\mathcal{G}_{\text{alt}}$ by a set of configurations $\mathcal{F}(\Pi \lhd \Gamma)$. The set $\mathcal{F}(\Pi \lhd \Gamma)$ consists of configurations $M \oplus \mathcal{F}(\Gamma)$ where M is a representative for Π, i.e. $M \in \mathcal{F}(\Pi)$.

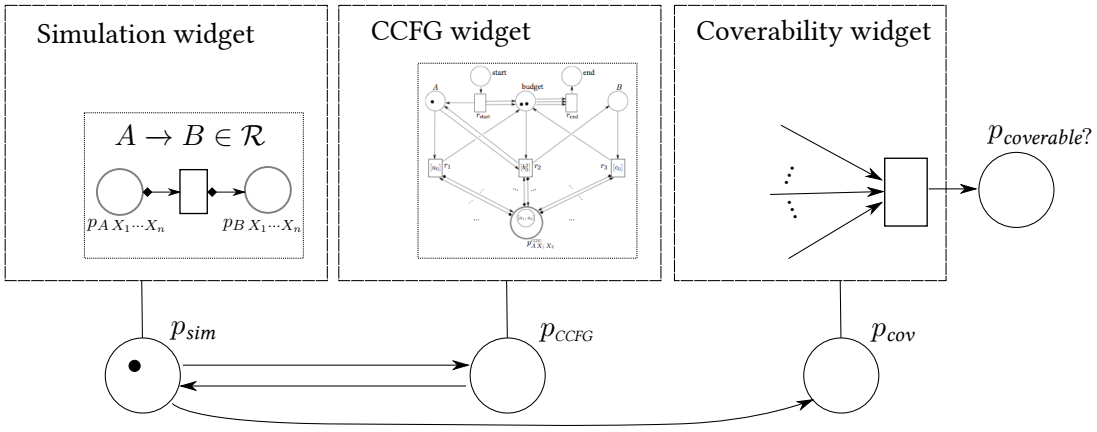


Figure 5.4: A schematic visualisation of the NNCT $\mathcal{N}$. The net $\mathcal{N}$ is subdivided in three parts. The *simulation widget* simulates the Rules (R′-1), (R′-3)–(R′-7), the *CCFG widget* simulates Rule (R′-2), and the *coverability widget* processes a coverability query.

The Colour Mapping. It remains to specify the colour mapping ζ. As we would expect $\mathcal{N}$'s colour mapping ζ maps a send $(c\,!\,msg)_i \in \Sigma_i^{\text{com}}$ to the simple place $p_{(c,msg)}$. However, Σ^{com} also contains spawns νX. To correctly model dispatch transition (Rules (R′-6) and (R′-7)), we need to ensure that a spawn νX leads to the creation of a complex token. Since in NNCT a transfer rule can only take coloured tokens to simple places, we introduce for each $\nu X \in \Sigma^{\text{com}}$ a simple place $p_{\nu X}$. The net $\mathcal{N}$ then has a simple transition that removes a single token from place $p_{\nu X}$ and adds an empty complex token to the complex place p_X. Thus, we can complete the definition of the colour mapping ζ by assigning each tagged spawn $(\nu X)_i \in \Sigma_i^{\text{com}}$ to the simple place $p_{\nu X}$. Note that by taking Σ_i^{com} as a set of active colours we ensure that the map $\zeta \restriction \Sigma_i^{\text{com}}$ is a bijection. We now move on to describe how we derive the rules (and other auxiliary places) of $\mathcal{N}$.

An Encoding of the Alternative Semantics. Let us illustrate how we encode the alternative semantics of $\mathcal{G}$ into the rules of $\mathcal{N}$. In Figure 5.4 we display a schematic overview of $\mathcal{N}$. We can think of the $\mathcal{N}$ as subdivided into three widgets. The *simulation widget* simulates the Rules (R′-1), (R′-3)–(R′-7). The *CCFG widget* is an augmented version of Ganty and Majumdar's encoding tailored for NNCT. We use the CCFG widget to simulate Rule (R′-2). The *Coverability widget* processes the simple coverability query $Q_{\mathcal{G}}$. If this processing is successful a single simple token is placed into the dedicated simple place $p_{coverable?}$. Hence, the simple coverability query for $\mathcal{N}$ may test whether the configuration that places a simple token in place $p_{coverable?}$ is coverable. We associate with each widget a *'mode'* place, i.e. the simple places p_{sim}, p_{CCFG}, and p_{cov} are linked with the widgets *simulation*, *CCFG*, and *coverability* respectively. A rule of a widget is only enabled if the widget's associated mode place contains a token. Our encoding ensures that there is only one token in the places p_{sim}, p_{CCFG}, and p_{cov} at any point in time. The arrows in Figure 5.4 indicate that if the simulation widget is active, it is possible to activate the CCFG widget (to simulate Rule (R′-2)). It is also possible to abandon the simulation and activate the coverability widget. On the other hand, it is only possible to return to the simulation widget from the CCFG

widget. In the following, we focus on each widget in turn.

The Simulation Widget. The simulation widget is mainly straightforward. It consists of the complex places $p_{\delta\, X_1\, X_2 \cdots\, X_n}$ to represent process states, the simple places $p_{(c,msg)}$ to represent channel contents, and $p_{\nu X}$ to keep track of spawns. Let us demonstrate how the simulation widget encodes a rule $A \to \beta \in \mathcal{G}$ such that (R′-1) applies. For each place $p_{A\, X_1\, X_2 \cdots\, X_n}$ there is a complex rule $((p_{A\, X_1\, X_2 \cdots\, X_n}, \emptyset), (p_{\beta\, X_1\, X_2 \cdots\, X_n}, \emptyset, \emptyset))$ which enables the following transition

$$[p_{A\, X_1\, X_2 \cdots\, X_n} \mapsto [M]] \oplus s \to_{\mathcal{N}} \left[p_{\beta\, X_1\, X_2 \cdots\, X_n} \mapsto [M]\right] \oplus s.$$

Note that if $\beta = B\, C$ and C is non-commutative then of course we require $n < K$. Further, let us illustrate how (R′-3) is encoded. Suppose we have a receive $c\,?\,msg$, then there is a complex rule $((p_{c?msg\, X_1\, X_2 \cdots\, X_n}, [(c, msg) \mapsto [\bullet]]), (p_{\epsilon\, X_1\, X_2 \cdots\, X_n}, \emptyset, \emptyset))$. This rule enables transitions of the form:

$$\left[p_{c?msg\, X_1\, X_2 \cdots\, X_n} \mapsto [M]\right] \oplus [(c, msg) \mapsto [\bullet]] \oplus s \to_{\mathcal{N}} [p_{\epsilon\, X_1\, X_2 \cdots\, X_n} \mapsto [M]] \oplus s.$$

Rules (R′-4) and (R′-5) are similarly encoded. Lastly, we show how a transfer rule implements the dispatch of a complete summary (Rule (R′-6)). For each place $p_{\epsilon\, X_1\, X_2 \cdots\, X_n}$ we have a transfer rule $((p_{\epsilon\, X_1\, X_2 \cdots\, X_n}, \emptyset), (p_{X_1\, X_2 \cdots\, X_n}, \Sigma^{\text{com}}_{K-n}, \emptyset))$ that enables transitions of the form:

$$\begin{aligned}\left[p_{\epsilon\, X_1\, X_2 \cdots\, X_n} \mapsto \left[M'_0 \oplus M'_1 \oplus \cdots M'_n\right]\right] \oplus s \\ \to_{\mathcal{N}} \left[p_{\epsilon\, X_1\, X_2 \cdots\, X_n} \mapsto \left[M'_1 \oplus \cdots M'_n\right]\right] \oplus s \\ \oplus \left[p_{(c,m)} \mapsto \left[\bullet^{M'_0((c!m)_{K-n})}\right], p_{\nu X} \mapsto \left[\bullet^{M'_0((\nu X)_{K-n})}\right]\right].\end{aligned}$$

Note that $\Sigma^{\text{com}}_{K-n}$ is the colour of the topmost summary M_0. The implementation of Rule (R′-7) is analogous. The simulation widget encodes all transition rules of the alternative semantics except Rule (R′-2). The latter rule is simulated by the CCFG widget.

The CCFG Widget. The CCFG widget is a slightly altered version of Ganty and Majumdar's encoding [GM12] (see Section 2.4). It simulates the computation of a CCFG derivation $C \to^* w$. In contrast to Ganty and Majumdar's encoding which produces the Parikh image $\mathbb{M}(w)$ in a set of ordinary Petri net places, our encoding carefully inserts $\mathbb{M}(w)$ into a selected complex token. For this purpose, we use a copy of each complex place $p_{\delta\, X_1\, X_2 \cdots\, X_n}$ that is part of the simulation widget. We denote this place by $p^{CCFG}_{\delta\, X_1\, X_2 \cdots\, X_n}$ and it is part of the CCFG widget. The CCFG widget is activated to simulate a transition generated by Rule (R′-2) such as the following:

$$A\, M_0\, X_1\, M_1 \cdots\, X_n\, M_n \parallel \Pi \lhd \Gamma \to_{\mathcal{G}(\mathcal{P})_{\text{alt}}} B\, M_0 \oplus \mathbb{M}(w)\, X_1\, M_1 \cdots\, X_n\, M_n \parallel \Pi \lhd \Gamma$$

where $A \to B\, C$ in $\mathcal{G}$ with C commutative and $C \to^* w \in \Sigma^{\text{com}}$. To simulate this transition, we have a complex rule that takes a complex token from place $p_{A\, X_1\, X_2 \cdots\, X_n}$ to $p^{CCFG}_{B\, X_1\, X_2 \cdots\, X_n}$ while changing the *'mode'* of the simulation to the CCFG widget. Note that our encoding maintains the invariant that the complex places of the CCFG widget are empty while the simulation widget is active and contains exactly one complex token while the CCFG widget is active. Once the CCFG widget has completed, the complex token is moved from $p^{CCFG}_{B\, X_1\, X_2 \cdots\, X_n}$ to $p_{B\, X_1\, X_2 \cdots\, X_n}$ and the

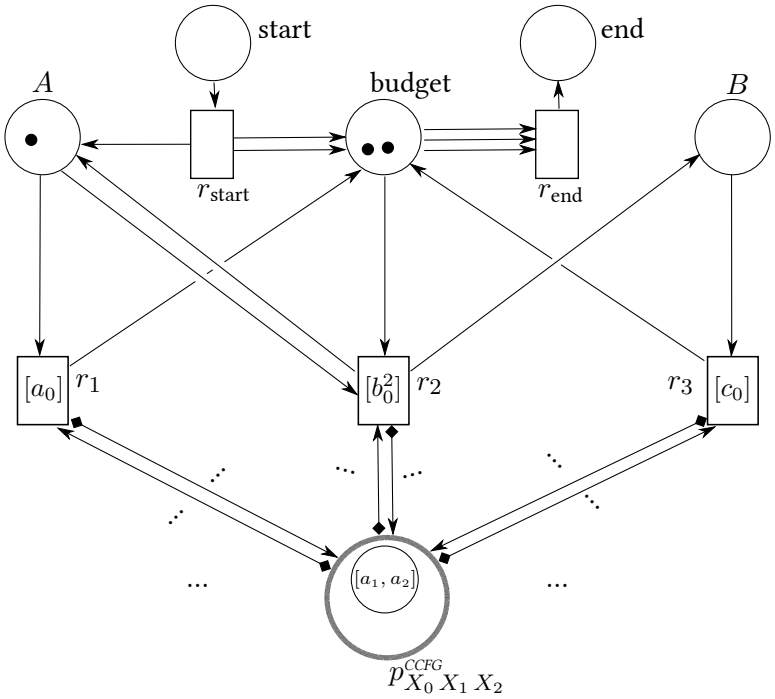


Figure 5.5: The CCFG widget *à la* Ganty and Majumdar of the CCFG $\mathcal{G}$. This example is taken from [Esp95] and modified to present Ganty and Majumdar's encoding.

'mode' is changed back to the simulation widget. Hence, we simulate the above transition by the sequence of transitions of the following shape:

$$\begin{aligned}
&\left[p_{A\,X_1\,X_2\cdots X_n} \mapsto M'_0 \oplus M'_1 \oplus \cdots \oplus M'_n\right] \oplus s \oplus [p_{sim} \mapsto [\bullet]] \\
&\quad \to_{\mathcal{N}} \left[p^{CCFG}_{B\,X_1\,X_2\cdots X_n} \mapsto M'_0 \oplus M'_1 \oplus \cdots \oplus M'_n\right] \oplus s \oplus [p_{CCFG} \mapsto [\bullet]] \\
&\quad \to^*_{\mathcal{N}} \left[p^{CCFG}_{B\,X_1\,X_2\cdots X_n} \mapsto M'_0 \oplus M'_1 \oplus \cdots \oplus M'_n \oplus M'\right] \oplus s \oplus [p_{CCFG} \mapsto [\bullet]] \\
&\quad \to_{\mathcal{N}} \left[p_{B\,X_1\,X_2\cdots X_n} \mapsto M'_0 \oplus M'_1 \oplus \cdots \oplus M'_n \oplus M'\right] \oplus s \oplus [p_{sim} \mapsto [\bullet]]
\end{aligned}$$

where M' is the correctly tagged version of $\mathbb{M}(w)$ with respect to the top cache M_0. To show our alterations of Ganty and Majumdar's encoding, we believe it is best to use an example. In Figure 5.5 we reuse Example 4 of Section 2.4. We concentrate on two commutative non-terminals A and B and the three rules

$$r_1 :\ A \to a, \quad r_2 :\ A \to b\,A\,b\,B, \quad r_3 :\ B \to c$$

We show (part of) our encoding in Figure 5.5. Let us assume that $K = 2$. Note the complex token $[a_1, a_2]$ in place $p^{CCFG}_{X_0 X_1 X_2}$. It represents an intermediate state APCPS process $X_0 \emptyset X_1\,[a_1]\,X_2\,[a_2]$ involved in a Rule (R′-2) transition. For example, its pre-state may be $X'_0 \emptyset X_1\,[a_1]\,X_2\,[a_2]$ and there is a rule $X'_0 \to X_0\,A$ in $\mathcal{G}$. The CCFG widget is virtually the same as Ganty and Majumdar's widget. Let us briefly point out the differences. In our version, rules r_1, r_2, and r_3 are implemented as a collection of complex rules. Each rule of the collection, synchronises with one

of the complex places in the CCFG widget. For example, rule r_1 produces the terminal a in the CCFG $\mathcal{G}$. In our encoding, the place in which the complex token is located determines which copy of a, i.e. $a_i \in \Sigma_i^{\text{com}}$, is injected. In the displayed case, we have a token in $p^{CCFG}_{X_0\, X_1\, X_2}$ and hence the correct copy of Σ^{com} is Σ_0^{com}. Since this synchronised injection of coloured tokens is the only modification of Ganty and Majumdar's widget we ensure that transitions generated by Rule (R′-2) are correctly simulated[2]. It remains for us to explain the *coverability widget*.

Coverability Widget. The coverability widget encodes the simple coverability query $\mathcal{Q}_{\mathcal{G}}$. This widget is necessary since $\mathcal{Q}_{\mathcal{G}}$ may ask for the coverability of process states that are encoded as complex tokens. However, in NNCT a simple coverability query may only involve simple tokens. Thus, for each non-terminal A_i^{cov} that appears in the coverability query $\mathcal{Q}_{\mathcal{G}}$, there is a simple place $p^{cov}_{A_i}$ in the coverability widget. The rules are then fairly straightforward. For each complex place p that encodes a process state that covers A_i^{cov}, we have a complex rule that moves a complex token from p to an auxiliary complex place p_{aux} while placing a simple token into $p^{cov}_{A_i}$. Such rules are only enabled when the net is in coverability widget *'mode'*. There is one rule r^{cov} that removes a simple token from $p^{cov}_{A_i}$ for each A_i^{cov}. Further, to test that the channel contents Γ^{cov} are covered, the rule r^{cov} removes appropriate tokens from simple places and places a simple token into the place $p_{coverable?}$. Hence, we can use a simple coverability query on $\mathcal{N}$ to decide whether $\mathcal{Q}_{\mathcal{G}}$ is a yes-instance.

The Size of $\mathcal{N}$. The resulting NNCT $\mathcal{N}$ has $O(n \cdot |\Sigma||\mathcal{N}|)$ simple places, $O((|\mathcal{N}||\Sigma|)^{O(K)})$ complex places, $O(K \cdot |\Sigma|)$ colours, and $O(n \cdot |\mathcal{R}| \cdot (|\mathcal{N}||\Sigma|)^{O(K)})$ rules. Thus, we can compute $\mathcal{N}$ in Exptime.

A Bisimulation and a Reduction of Simple Coverability. We now give an account of the relationship between $\mathcal{N}$ and $\mathcal{G}_{\text{alt}}$. Since we are representing a configuration of the alternative semantics by a set of NNCT configurations, we have to move to a co-universal powerset lifting of $\mathcal{N}$,i.e. $\mathscr{P}^{\mathsf{A}}[\mathcal{N}]$. Then, there is a weak bisimulation between $\mathscr{P}^{\mathsf{A}}[\mathcal{N}]$ and $\mathcal{G}_{\text{alt}}$. The bisimulation is given by the relation

$$\mathcal{R} = \{(\Pi \lhd \Gamma, \mathcal{F}(\Pi \lhd \Gamma)) \mid \Pi \lhd \Gamma \in \mathit{State}_{\text{alt}}\}\,.$$

As in the case of the bisimulation between the alternative and standard semantics, we need to label the transitions systems appropriately. We label a transition $\Pi \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \Pi' \lhd \Gamma'$ by $\Pi' \lhd \Gamma'$. For a transition $S \to_{\mathscr{P}^{\mathsf{A}}[\mathcal{N}]} S'$ we attach a label $\Pi' \lhd \Gamma'$ just if $S' = \mathcal{F}(\Pi' \lhd \Gamma')$. Otherwise the label is ϵ. With this labelling it is then straightforward to prove the following lemma.

Lemma 5. The relation $\mathcal{R}$ is a weak bisimulation for $\mathscr{P}^{\mathsf{A}}[\mathcal{N}]$ and $\mathcal{G}_{\text{alt}}$.

Equipped with this lemma, it is easily deduced that we can decide simple coverability queries on $\mathcal{G}_{\text{alt}}$ using the Exptime-computable $\mathcal{N}$.

[2]For completeness, we would like to add that our CCFG widget also allows premature abortion of the CCFG computation and may move the complex token to correctly represent partial summaries. We omit these technical details here; they can be found in full in Appendix C.2.

Theorem 10. *Simple coverability for K-shaped APCPS in the alternative semantics* Exptime*-time reduces to simple coverability for NNCT.*

The full proof of the bisimulation lemma, the above theorem, and a detailed account of the construction of $\mathcal{N}$ can be found in Appendix C.2.

The reduction of the termination and boundedness problem for shaped APCPS in the alternative semantics to NNCT is more intricate. The CCFG widget may introduce non-terminating runs in the precompuation of a summary that do not arise in $\mathcal{G}_{\text{alt}}$. It may also add infinitely many intermediate configurations. However, we believe that if we restrict ourselves to termination- and rewrite-bounded shaped APCPS, we can use $\mathcal{N}$ to decide termination and boundedness problems. Again we conjecture that using a preprocessing step that involves a combination of a polynomial-time static analysis on $\mathcal{G}$ and coverability (see end of Section 4.3) we can answer boundedness and termination questions for $\mathcal{G}_{\text{alt}}$ on $\mathcal{N}$. We summarise the above in the following conjecture

Conjecture 4. *Boundedness (termination) for shaped APCPS on the alternative semantics* Exptime*-reduces to a decision problem pair consisting of boundedness (termination resp.) and coverability for NNCT.*

In Section 5.4, we show that NNCT simple coverability, boundedness, and termination are Tower-hard. These lower bounds can be transferred to the alternative semantics. The fact that the reduction from $\mathcal{G}$ to $\mathcal{N}$ is exponential is thus immaterial for establishing the complexities of the former decision problems.

In the next section, we show how to model a total transfer NNCT in a shaped APCPS and demonstrate Exptime-reductions from NNCT to APCPS.

From Total-Transfer NNCT to Shaped APCPS

In this section, we demonstrate how we can construct a shaped APCPS $\mathcal{G}$ that simulates a given total-transfer NNCT $\mathcal{N}$. We use channels to encode the contents of simple places and represent the state of complex tokens by processes that carry a single summary. This encoding yields an Exptime-reduction from simple coverability, boundedness, and termination for NNCT to their respective counterparts for shaped APCPS. Hence, let us fix a total transfer NNCT $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$ for the remainder of this section. We begin with a brief account of how we represent a configuration of $\mathcal{N}$ as a configuration of $\mathcal{G}_{\text{alt}}$.

Representing NNCT Configurations. We represent a configuration s of $\mathcal{N}$ in the following way. For each simple place $p \in \mathcal{P}^S$ we introduce a channel c_p in $\mathcal{G}$. The channels c_p contain only one type of message that we write as $\bullet$. We represent the contents of the simple place p, i.e. $s(p)$, by placing a message $\bullet$ into c_p for each simple token in p. This specifies a channel mapping $\mathcal{F}^\Gamma(s) \in \mathit{Chans}$ such that

$$\mathcal{F}^\Gamma(s) = \left[c_p \mapsto \left[\bullet^{|s(p)|} \right] \mid p \in \mathcal{P}^S \right].$$

Since $\mathcal{N}$ is a total transfer NNCT, we know that we can associate to each colour $col \in \mathcal{P}^{col}$ a simple place $\zeta(col)$. So let us represent a complex token $M \in \mathbb{M}[\mathcal{P}^{col}]$, where we can w.l.o.g assume that $M = [col_1, \ldots, col_n]$, by the summary $\mathcal{F}^{sum}(M)$ which is

$$\mathcal{F}^{sum}(M) = \left[p_{\zeta(col_1)} \,!\, \bullet, \ldots, p_{\zeta(col_n)} \,!\, \bullet\right].$$

For each complex place p we introduce a non-commutative non-terminal N^p into $\mathcal{G}$ together with a dedicated non-commutative non-terminal $N^?$. Using these non-terminals we represent a complex token M in complex place p, i.e. a marking $[p \mapsto [M]]$, by a process $N^p\, \mathcal{F}^{sum}(M)\, N^?$ which we denote by $\mathcal{F}^{\pi}([p \mapsto [M]])$. There is one edge case that we need to consider where $M = \emptyset$. In a total transfer NNCT, a complex token can only be empty as a result of being created empty or being emptied by a transfer transition. Hence, we can keep track of this information by tagging the non-terminal N^p, i.e. we introduce the non-commutative non-terminal $N^p_\emptyset$ that captures this edge case. Hence, $\mathcal{F}^{\pi}([p \mapsto [\emptyset]]) = N^p_\emptyset\, \emptyset\, N^?$.

Further, we introduce a control process that executes the rules of $\mathcal{N}$. $\mathcal{G}$ has a special non-terminal N^{sim} that is reserved for the control process. Thus, a configuration s, which we may decompose as

$$s = s_{complex} + s_{simple} \qquad \text{where} \quad s_{complex} = [p_1 \mapsto [M_1]] \oplus \cdots \oplus [p_n \mapsto [M_n]]\,,$$

is represented by the APCPS configuration $\mathcal{F}(s)$ defined by

$$\mathcal{F}(s) = N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple})$$
$$\text{where } \mathcal{F}^{\Pi}(s_{complex}) = \mathcal{F}^{\pi}([p_1 \mapsto [M_1]]) \parallel \cdots \parallel \mathcal{F}^{\pi}([p_n \mapsto [M_n]]).$$

Let us elaborate on how the control process manipulates the state of processes and channels in order to execute the rules of $\mathcal{N}$.

The Control Process. It is the responsibility of the control process to execute the rules of $\mathcal{N}$. It achieves this in several steps and with the cooperation of the processes that encode complex tokens. For each rule $r \in \mathcal{N}$ there is a non-commutative non-terminal N^r that specifies the actions of the control process. We introduce a rule $N^{sim} \to N^r\, N^{sim}$ that lets the control process non-deterministically select a rule r to execute.

$$N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple}) \to_{\mathcal{G}_{\text{alt}}} N^r\, N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple})$$

The implementation of each rule r by the non-terminal N^r follows a similar scheme whether $r \in$ *SimpleRules*, *ComplexRules*, or *TransferRules*. Each rule r has mappings I_r and O_r whose effect on simple places are implemented by non-commutative non-terminals N^r_I and N^r_O. The effect of r on complex tokens is implemented by a non-commutative non-terminal N^r_C. So for each $r \in \mathcal{R}$ we add a rule $N^r \to N^r_I\, N^r_O\, N^r_C$. When the control process executes N^r_I, it extracts messages from channels simulating the removal of simple tokens as prescribed by I_r. Similarly, N^r_O sends messages to update the contents of channels as determined by O_r.

$$\begin{aligned} N^r\, N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) &\lhd \mathcal{F}^{\Gamma}(s_{simple}) \\ &\to_{\mathcal{G}_{\text{alt}}} N^r_I\, N^r_O\, N^r_C\, N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple}) \\ &\to^*_{\mathcal{G}_{\text{alt}}} N^r_O\, N^r_C\, N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r) \\ &\to^*_{\mathcal{G}_{\text{alt}}} N^r_C\, N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r \oplus O_r). \end{aligned}$$

Note that there is an exponential blow-up introduced by the rules implementing N_I^r and N_O^r. The mappings I_r an O_r can be compactly represented by an array with binary representations of the numbers $I_r(p)$ and $O_r(p)$. The implementations of N_I^r and N_O^r in Chomsky normal form represent the numbers $I_r(p)$ in unary and hence lead to the introduction of an exponential number of non-terminals.

Let us now turn to a high level account of the implementations of N_C^r. The effect of executing N_C^r depends on the type of rule r. The control process communicates with the processes that represent complex tokens over a number of administrative channels. For each complex place p there are two channels c'_p and $c'_{(p,\emptyset)}$. The first is designated for communication to all processes representing complex tokens in place p. The second is meant for the subset of processes representing empty complex tokens in place p. Additionally, there is a channel $c'_?$ that is reserved for communications to processes that are in state $N^?$.

A simple rule r may remove empty complex tokens and create new complex tokens. We encode this in N_C^r as follows. For each removal of an empty complex token in the complex place p, the control process sends a message *remove* to channel $c'_{(p,\emptyset)}$ which is received by a process π in state $N_\emptyset^p\,\emptyset\,N^?$. The process π sends an acknowledgement *ok* on $c'_{(p,\emptyset)}$ and pops off its head non-terminal making the empty summary effective. Thus, the process π arrives in state $N^?$ and is the only process in this state. After receiving the acknowledgement, the control process sends another message *remove* on channel $c'_?$ and waits for another acknowledgement. Receiving this message the π process responds with *ok* on $c'_?$, pops off $N^?$, and terminates by rewriting to ϵ. To create a complex token M in the complex place p, the control process executes a spawn $\nu N^{(p,M)}$ and waits for an acknowledgement on c'_p. The non-terminal $N^{(p,M)}$ sends this acknowledgement and rewrites to $N^p\,N_M^{sum}\,N^?$. Note that N_M^{sum} is a commutative non-terminal that produces one word w such that $\mathbb{M}(w) = \mathcal{F}^{sum}(M)$. Hence, executing N_C^r for a simple rule r leads to the desired run:

$$\begin{aligned} N_C^r\,N^{sim} \parallel\ & \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r \oplus O_r) \\ & \rightarrow^*_{\mathcal{G}_{\text{alt}}} N^{sim} \parallel\ \mathcal{F}^{\Pi}(s_{complex} \ominus I_r \oplus O_r) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r \oplus O_r). \end{aligned}$$

The implementation of N_C^r for a complex rule r is slightly simpler. In this case r may move a complex token M from a complex place p to another p' while injecting a multiset of colours M'. For this purpose, we let the control process send a message $move_{p',M'}$ to channel c'_p and wait for an acknowledgement on $c'_{p'}$. A process π in state $N^p\,\mathcal{F}^{sum}(M)\,N^?$ may receive this message and rewrite its top non-terminal to $N^{p'}\,N_{M'}^{sum}$ while sending the acknowledgement on $c'_{p'}$. Thus, π arrives in the state $N^{p'}\,\mathcal{F}^{sum}(M') \oplus \mathcal{F}^{sum}(M)\,N^? = N^p\,\mathcal{F}^{sum}(M' \oplus M)\,N^?$. Note that if $M = \emptyset$ then, of course, π can mimick the same synchronisation from non-terminal $N_\emptyset^p$, but ends up in state $N^{p'}\,\mathcal{F}^{sum}(M')\,N^?$. Hence, we obtain the desired run:

$$\begin{aligned} N_C^r\,N^{sim} \parallel\ & \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r \oplus O_r) \\ & \rightarrow^*_{\mathcal{G}_{\text{alt}}} N^{sim} \parallel\ \mathcal{F}^{\Pi}(s_{complex} \ominus [p \mapsto [M]] \oplus [p' \mapsto \left[M \oplus M'\right]]) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r \oplus O_r). \end{aligned}$$

The implementation of N_C^r for a total transfer rule r follows a similar synchronisation scheme. Suppose r takes a complex token M from complex place p to p' while ejecting its entire contents.

The control process simulates this by sending a message *transfer* to channel c'_p and waits for acknowledgement on $c'_?$. A process π representing M, i.e. in state $N^p\,\mathcal{F}^{sum}(M)\,N^?$, receives this message, sends an acknowledgement on $c'_?$, pops N^p, and makes the summary $\mathcal{F}^{sum}(M)$ effective. Thus, π arrives in state $N^?$. On receiving the acknowledgement the control process synchronises with π once more which leads to π rewriting to $N^{p'}_{\emptyset}\,\mathcal{F}^{sum}(\emptyset)\,N^?$. Note that if $M = \emptyset$ then this synchronisation can again be mimicked from the nonterminal $N^p_{\emptyset}$. Thus, we obtain the following run as desired:

$$N^r_C\,N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex}) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r \oplus O_r)$$
$$\to^*_{\mathcal{G}_{\text{alt}}} N^{sim} \parallel \mathcal{F}^{\Pi}(s_{complex} \ominus [p \mapsto [M]] \oplus [p' \mapsto [\emptyset]]) \lhd \mathcal{F}^{\Gamma}(s_{simple} \ominus I_r \oplus O_r \oplus (M \circ \zeta^{-1})).$$

For a full and detailed description of $\mathcal{G}$, we refer the reader to Appendix C.2. Let us now move on and focus on the relationship between the transition systems generated by $\mathcal{N}$ and $\mathcal{G}$.

A Bisimulation and a Reduction. The relationship between $\mathcal{N}$ and the alternative semantics $\mathcal{G}$ is slightly simpler than in our encoding of shaped APCPS into NNCT. We can represent each configuration of $\mathcal{N}$ by a single configuration of $\mathcal{G}_{\text{alt}}$. Thus, no powerset lifting is necessary. The transition system $\mathcal{G}_{\text{alt}}$ can replicate each step of $\mathcal{N}$ using a number of steps, i.e. $\mathcal{G}_{\text{alt}}$ weakly bisimulates $\mathcal{N}$. The *weak* bisimulation is due to the fact that the control process takes several steps to execute one rule of $\mathcal{N}$. We make the weak bisimulation explicit with the following relation:

$$\mathcal{R} = \{(s, \mathcal{F}(s)) \mid s \in \mathit{Config}\}\,.$$

In order to make sense of a bisimulation, we need to label the transition systems $\mathcal{N}$ and $\mathcal{G}_{\text{alt}}$. We follow the familiar scheme and label the transition $s \to_{\mathcal{N}} s'$ by s'. A transition of $\mathcal{G}_{\text{alt}}$ $\Pi \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \Pi' \lhd \Gamma'$ is then labelled by s' just if $\mathcal{F}(s') = \Pi' \lhd \Gamma'$. Otherwise we label it with ϵ. With this labelling we can establish the following lemma.

Lemma 6. The relation $\mathcal{R}$ is a weak bisimulation for $\mathcal{N}$ and $\mathcal{G}_{\text{alt}}$.

It is not difficult to see that the control process does not add any infinite paths and introduces at most finitely more reachable configurations. In fact $\mathcal{G}$ is easily seen to be both rewrite- and termination-bounded. Thus, we can answer simple coverability, boundedness, and termination questions for $\mathcal{N}$ on $\mathcal{G}$. Note that $\mathcal{G}$ is EXPTIME-computable from $\mathcal{N}$ and that it is easy to verify that $\mathcal{G}$ is 4-shaped.

Theorem 11. *Simple coverability, boundedness, and termination for a total-transfer NNCT* EXPTIME *reduces to simple coverability, boundedness, and termination respectively for 4-shaped APCPS in the alternative semantics.*

A full detailed proof of the above lemma and Theorem 11 together with a complete description of $\mathcal{G}$ can be found in Appendix C.2. In the next two sections, we show that coverability for NNCT is a TOWER-complete problem and that boundedness and termination are TOWER-hard for NNCT. As a reminder, the notion of reduction used in TOWER are ELEMENTARY-time reductions which give rise to TOWER-complete problems. The EXPTIME reductions of Theorem 10 and Theorem 11 are thus sufficient to infer that (simple) coverability for APCPS is TOWER-complete and that boundedness and termination are TOWER-hard.

5.3 Upper Bound: A Nested Rackoff Argument.

In this section, we focus on establishing an upper bound to the complexity of NNCT coverability. An effective algorithm for deciding coverability is already available to us since NNCT are WSTS. However, it is difficult to obtain upper bounds from the WSTS coverability algorithm. Thus, we turn to a technique that has been frequently applied in the Petri net literature. First, let us fix a NNCT $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$ with a coverability query $(\mathcal{N}, s_0, s_{\text{cov}})$ throughout this section. We enumerate $\mathcal{N}$'s simple places $\mathcal{P}^S = \{p_{(1)}, \ldots, p_{(n_S)}\}$, complex places $\mathcal{P}^C = \{p'_{(1)}, \ldots, p'_{(n_C)}\}$, and colours $\mathcal{P}^{col} = \{p^{col}_{(1)}, \ldots, p^{col}_{(n_{col})}\}$. Our proof of Tower-membership for NNCT coverability is inspired by Rackoff's method [Rac78]. Originally used to establish an Expspace upper bound for *vector addition systems* (VAS), Rackoff's method has recently seen an increase in popularity [Dem+09; BS11b; BFP12; Dem13; LS14; Ler11a; Ler11b; LPS13]. The Rackoff method constructs a bound on the *covering radius* of a given target state s_{cov}, which is the greatest *covering distance* to s_{cov} from an arbitrary state s. The covering distance is the length of the shortest covering path. Typically the covering radius can then be used to establish a bound $\mathcal{B}$ on the space required for a machine representation of any state along a covering path for s_{cov}. Depending on the flavour of VAS, it is then easy to see that a $\mathcal{B}$-space bounded non-deterministic/alternating Turing machine can find a covering path, if there is one, and reject a path if its length exceeds the covering radius.

Definition 23 (Covering distance, radius, and diameter). Suppose $\mathcal{S} = (S, \to_{\mathcal{S}}, \leq_{\mathcal{S}})$ is a PSTS. Let us recall the definition of a *covering path*. A path $\boldsymbol{s} \in S^*$ is said to be *covering* for s' from s if $s' \leq_{\mathcal{S}} \boldsymbol{s}(|\boldsymbol{s}|)$ and $\boldsymbol{s}(1) = s$. We define $dist_{\mathcal{S}}(s, s')$, the *covering distance* between s and s' in $\mathcal{S}$, as follows: if there exists a covering path for s' from s in $\mathcal{S}$ then

$$dist_{\mathcal{S}}(s, s') := \min \left\{ |\boldsymbol{s}| \mid \boldsymbol{s} \text{ covering path for } s' \text{ from } s \right\};$$

otherwise set $dist_{\mathcal{S}}(s, s') := 0$. The maximum covering distance for s' from an arbitrary s then gives rise to the *covering radius* $\rho_{\mathcal{S}}(s')$. Formally, we define the covering radius $\rho_{\mathcal{S}}(s')$ by

$$\rho_{\mathcal{S}}(s') := \sup \left\{ dist_{\mathcal{S}}(s, s') \mid s \in S \right\}.$$

Another geometrically inspired notion is the *covering diameter* of a set. Given a set of pairs $S' \subseteq S \times S$, the *covering diameter* of S', which we denote by $d_{\mathcal{S}}(S')$, is the maximum covering distance between any pair in S', i.e

$$d_{\mathcal{S}}(S') := \sup \left\{ dist_{\mathcal{S}}(s, s') \mid (s, s') \in S' \right\}.$$

Note that the definition of *dist* ensures that the covering distance between any pair of configurations is finite. However, *prima facie* the covering radius or diameter may be infinite. With the Rackoff argument we can establish that both are always finite for NNCT.

A Relativised Problem. In order to determine a bound on the covering radius, a more general, relativised problem is considered. Suppose the contents of the last $(n - i)$ places are *ignored* and the remaining i places are *not ignored*, how can we bound the covering radius ρ_i for s_{cov}?

We follow an elegant reformulation of this relativised argument by Bonnet et al. where places are assigned *'unbounded'* contents. In order to ignore the contents of simple places in $\mathcal{N}$, simple markings are relaxed: define $\mathcal{M}^{simple^\infty} = \mathcal{P}^S \to \mathbb{M}^\infty[\{\bullet\}]$ where we write $\mathbb{M}^\infty[U]$ for multisets over elements in U with possibly infinite multiplicity, i.e. the set of functions $U \to \mathbb{N}^\infty$. Configurations with possibly infinite simple markings can be defined as

$$\mathit{Config}^\infty = \{m + m' \mid m \in \mathcal{M}^{simple^\infty}, m' \in \mathcal{M}^{complex}\}.$$

We extend the transition relation $\to_{\mathcal{N}}$ in the obvious way to Config^∞. Further, for each $i \leq n_S$ we define a new transition relation $\to_{\mathcal{N}_i}$ that formalises that we ignore the last $n_S - i$ simple places. Formally, we have a transition $s \to_{\mathcal{N}_i} s'$ just if $s \to_{\mathcal{N}} s'$ and $s(p_{(j)})(\bullet) = s'(p_{(j)})(\bullet) = \infty$ for all $i < j \leq n_S$. Writing $\mathcal{N}_i = (\mathit{Config}^\infty, \to_{\mathcal{N}_i}, \leq_{\mathit{Config}})$ the relativised problem is to determine the covering radius $\rho_{\mathcal{N}_i}(s_{\mathrm{cov}})$.

The Rackoff Recurrence Relation. The core argument underlying the general Rackoff method establishes a recurrence relation that relates ρ_{i+1} to ρ_i. This is achieved by a careful analysis of the size or *norm* of configurations that occur along covering paths for s_{cov}. In general, we say $\|-\|$ is a *norm* for a set S if $\|-\|$ is a function from S to $\mathbb{N}^\infty$. We extend $\|-\|$ to a norm $\|-\|^*$ on sequences S^* in the usual way:

$$\|\boldsymbol{s}\|^* = \max\{\|\boldsymbol{s}\| : 1 \leq i \leq |\boldsymbol{s}|\}.$$

The general Rackoff method introduces a family of norms $(\|-\|_i)_{i=0}^n$ where each $\|-\|_i$ ignores the contents of the last $(n-i)$ places. The intention is to use the norm $\|-\|_{i+1}$ in establishing the recurrence relation between ρ_{i+1} and ρ_i. For this purpose, two facts are established:

(Fact$_1$) There exists a number $B(\rho_i)$ that is an upper bound on how many tokens can be removed in a path of length ρ_i.

(Fact$_2$) For any covering path $\boldsymbol{s}$ for s_{cov} one of two cases apply.

- Case$_1$: $\|\boldsymbol{s}\|^*_{i+1} < B(\rho_i)$, i.e. the configurations along $\boldsymbol{s}$ have a $\|-\|_{i+1}$ norm that is bounded by $B(\rho_i)$.
- Case$_2$: The path $\boldsymbol{s}$ can be split at a *pivot configuration* s_p, i.e. $\boldsymbol{s} = \boldsymbol{s}_1 \cdot s_p \cdot \boldsymbol{s}_2$ such that the following two properties are satisfied: (Property$_1$) $\|\boldsymbol{s}_1\|^*_{i+1} < B(\rho_i)$ and (Property$_2$) one *not* ignored place of s_p, the $i+1$'s say, contains more than $B(\rho_i)$ tokens.

A consequence of Property$_2$ is that we may ignore the contents of the $i+1$'s place for any covering path for s_{cov} from s_p. Thus, we can replace $\boldsymbol{s}_2$ by a path $\boldsymbol{s}'_2$ with length at most ρ_i. We note that Property$_1$ is satisfied both by paths $\boldsymbol{s}$ to which Case$_1$ applies and paths $\boldsymbol{s}_1$ that arise in Case$_2$. A path that satisfies Property$_1$ may be replaced by a path of length no more than the covering diameter d_{i+1} of the set of configuration pairs

$$\{(\boldsymbol{s}(1), \boldsymbol{s}(|\boldsymbol{s}|)) : \boldsymbol{s} \text{ path}, \|\boldsymbol{s}\|^*_{i+1} < B(\rho_i)\}$$

with respect to the transition system that does not ignore the first $i+1$ places. This yields *the Rackoff recurrence relation* $\rho_{i+1} \leq d_{i+1} + \rho_i$. The problem is thus reduced to bounding the

covering diameter of a set of configurations of bounded norm. Note that the Rackoff recurrence relation establishes a connection between the covering radius and a notion of size or norm of configurations. From a high-level perspective, we may thus say that it describes a connection between two different notions of distance: the covering distance and a distance induced by the norm.

It is challenging to apply Rackoff's method to NNCT, as opposed to the usual VAS setting. We are forced to adjust the above argument in a few technical details, while we remain faithful to the high-level argument. In the setting of NNCT, we introduce two families of norms $\|-\|_{\mathcal{N}_i}$ and $\|-\|_{\mathcal{N}_i;C}$ on Config^∞. The norm $\|-\|_{\mathcal{N}_i;C}$ ignores the simple places $p_{(i+1)}, \dots, p_{(n_S)}$. The norm $\|-\|_{\mathcal{N}_i}$ also ignores coloured tokens and complex places, i.e. for $s \in \mathit{Config}^\infty$ we define

$$\|s\|_{\mathcal{N}_i} = \max\left\{|s(p_{(j)})| : j \leq i\right\} \text{ and}$$
$$\|s\|_{\mathcal{N}_i;C} = \max\left(\left\{|s(p'_{(j)})|, \max_{m \in s(p'_{(j)})} \|m\|_{\mathit{col}} \;\middle|\; j \in \langle n_C \rangle\right\} \cup \left\{\|s\|_{\mathcal{N}_i}\right\}\right).$$

Further, the norm $\|-\|_{\mathit{col}}$ on complex tokens is defined by $\|m\|_{\mathit{col}} = \max\{|m(p^{\mathit{col}}_{(j)})| : j \in \langle n_{\mathit{col}} \rangle\}$. In the following section, we show that in the setting of NNCT appropriate versions of Fact_1 and Fact_2 hold which leads us to the desired Rackoff recurrence relation.

Establishing the Rackoff Recurrence Relation

As a first step, we turn to Fact_1. Let us define a number R that helps us bound the number of tokens that may be removed from any place along a single transition:

$$R := \max\left(\left\{|I_r(p)|, |O_r(p)| \mid r \in \mathcal{R}, p \in \mathcal{P}^S \cup \mathcal{P}^C\right\} \cup \{1\}\right).$$

The definition of R bounds the number of tokens that can be removed or added along a single transition. Thus, we know that if a place contains more than R tokens it cannot disable any rule. We define another bound R' as the maximal norm of s_{cov} or new complex tokens plus R:

$$R' := R+1+\max(\Xi \cup \{\|s_{\text{cov}}\|_{\mathcal{N}_{n_S};C}\}) \text{ where } \Xi := \left\{\|c\|_{\mathit{col}} \mid c \in O_r(p'), p' \in \mathcal{P}^C, r \in \mathit{SimpleRules}\right\}$$

In contrast to R, the number R' is large enough to ensure that if a place p contains more than R' tokens then after applying any rule r there will be more tokens in p than in $s_{\text{cov}}(p)$. Hence, we can show that an appropriate version of Fact_1 holds in our setting.

Lemma 7. Suppose $\boldsymbol{s}$ is a covering path for s_{cov} in $\mathcal{N}_{i+1}$ and $|\boldsymbol{s}(1)(p_{(i+1)})| \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$. Then there exists $\boldsymbol{s}'$ a covering path for s_{cov} in $\mathcal{N}_{i+1}$ such that $\boldsymbol{s}'(1) = \boldsymbol{s}(1)$ and $|\boldsymbol{s}'| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Let us now address Fact_2. We can establish that covering paths fall into two categories.

Lemma 8. For all covering paths $\boldsymbol{s}$ for s_{cov} in $\mathcal{N}_{i+1}$ one of two cases applies:
(C_1) $\|\boldsymbol{s}\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$; or
(C_2) $\boldsymbol{s} = \boldsymbol{s}_1 \cdot s_p \cdot \boldsymbol{s}_2$ such that $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$ and $\|s_p\|_{\mathcal{N}_{i+1}} \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Unfortunately, in the NNCT setting we need stronger guarantees on the pivot configuration s_p. It is necessary for us to require that there exists a "small" predecessor $s_{p'}$ of s_p, i.e. $\|s_{p'}\|_{\mathcal{N}_{i+1};C} < R' \cdot (\rho_{\mathcal{N}_i}(s_{\text{cov}}) + 1)$ and $s_{p'} \to_{\mathcal{N}_{i+1}} s_p$, that is covered by $\boldsymbol{s}_1$. Of course, this does not hold for all pivot configurations. However, given a covering path satisfying case (C_2) we can construct a pivot with this property. The construction exploits Property$_2$. We replace $\boldsymbol{s}_2$ with a path $\boldsymbol{s}'_2$ satisfying $|\boldsymbol{s}'_2| < \rho_{\mathcal{N}_i}(s_{\text{cov}})$ and we make two observations. Suppose we have a path $\boldsymbol{s}_0$ of length L then (O_1) along $\boldsymbol{s}_0$ only L complex tokens can be moved/removed; and (O_2) for any complex token m occurring in $\boldsymbol{s}_0(1)$ at most $L \cdot R'$ carried coloured tokens of a given colour can be ejected and removed along $\boldsymbol{s}_0$. Hence it is possible to eliminate any superfluous complex and coloured tokens in $\boldsymbol{s}_0(1)$ without affecting the validity of the path. We summarise these observations in the following lemma:

Lemma 9. Suppose $\boldsymbol{s}$ is a covering path for s_{cov} in $\mathcal{N}_{i+1}$. If $L \in \mathbb{N}$ such that $|\boldsymbol{s}| \leq L$ and $\|\boldsymbol{s}(1)\|_{\mathcal{N}_{i+1}} < L \cdot R'$ then there exists a covering path $\boldsymbol{s}''$ for s_{cov} in $\mathcal{N}_{i+1}$ such that $\boldsymbol{s}''(1) \leq_{\mathit{Config}} \boldsymbol{s}(1)$, $\|\boldsymbol{s}''(1)\|_{\mathcal{N}_{i+1};C} < L \cdot R'$, and $|\boldsymbol{s}''| \leq L$.

We use Lemma 9 as a tool to strengthen Lemma 8. This yields an appropriate version of Fact$_2$ for the NNCT setting. It is strong enough to help us establish the Rackoff recurrence relation.

Corollary 5. For all covering paths $\boldsymbol{s}$ for s_{cov} in $\mathcal{N}_{i+1}$ either
(C'_1) $\|\boldsymbol{s}\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$; or
(C'_2) there exist paths $\boldsymbol{s}_1$ and $s_{p'} \cdot \boldsymbol{s}_2$ such that $\boldsymbol{s}_1$ is a covering path for $s_{p'}$, $\boldsymbol{s}_1(1) = \boldsymbol{s}(1)$, and $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$; further $s_{p'} \cdot \boldsymbol{s}_2$ is a covering path for s_{cov} and $|\boldsymbol{s}'_2| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$; and $\|s_{p'}\|_{\mathcal{N}_{i+1};C} \leq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

The final piece to establish the Rackoff recurrence relation is the set of configuration pairs that capture the paths that satisfy Property$_1$. To simplify notation, let us write $B_i = R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$. We denote the set of configuration-pairs (s, s') for which there exists a covering path from s for s' of norm less than B by $S_{(i,B)}$. Formally, we define the set $S_{(i,B)}$ by

$$S_{(i,B)} = \{(\boldsymbol{s}(1), s') : \boldsymbol{s} \text{ path}, \|\boldsymbol{s}\|^*_{\mathcal{N}_i} < B, \|s'\|_{\mathcal{N}_i;C} \leq B, s' \leq_{\mathit{Config}} \boldsymbol{s}(|\boldsymbol{s}|)\}.$$

Inspecting case (C'_2), we notice that the pair $(\boldsymbol{s}_1(1), s_{p'}) \in S_{(i,B_i)}$. Hence, it is immediate that we can find a path $\boldsymbol{s}'_1$ of length at most $d_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right)$ to replace $\boldsymbol{s}_1$. A similar argument applies to paths $\boldsymbol{s}$ of case (C'_1). We are thus in a position to establish the Rackoff recurrence relation for NNCT:

Proposition 6. (i) $\rho_{\mathcal{N}_0}(s_{\text{cov}}) \leq d_{\mathcal{N}_0}\left(S_{(0,R')}\right)$ and (ii) $\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) \leq d_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right) + \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Establishing the Rackoff recurrence relation is a significant step in finding a bound on the covering radius. The problem is now much simpler. We can focus on establishing a bound on the covering diameter of the set $S_{(i,B)}$. Note that a bound on d_i in Rackoff's original argument is trivially established. In the VAS case, the analogue of $S_{(i,B)}$ is generated by the set of paths that have norm $\|-\|^*_i$ less than B. Given such a VAS-path $\boldsymbol{s}$ with $\|\boldsymbol{s}\|^*_i < B$, i.e. all configurations along $\boldsymbol{s}$ have counters bounded by B, we can eliminate all simple loops. This yields a path $\boldsymbol{s}'$ of

length at most B^i, since no configuration in $\boldsymbol{s}'$ is repeated [Rac78]. Establishing a bound on $\not{d}_i$ is also considered an easier part in other applications of the Rackoff method [Dem+09; BS11b; BFP12; Dem13; LS14]. In the case of NNCT finding a bound on $\not{d}_{\mathcal{N}_i}\left(S_{(i,B)}\right)$ is non-trivial.

The Covering Diameter of Bounded Paths

Our strategy to establish a bound $\not{d}_{\mathcal{N}_i}\left(S_{(i,B)}\right)$ is as follows. We first uncover a *sub-transition system* $\mathcal{T}$ of $\mathcal{N}_i$ that we may think of as *generating* the configuration pairs in $S_{(i,B)}$. Inspecting $\mathcal{T}$ yields that we can obtain a Petri net $\mathcal{V}$ such that a sub-transition system $\mathcal{T}'$ of $\mathcal{V}$ can bisimulate $\mathcal{T}$. This bisimulation allows us reason about the paths of $\mathcal{V}$ rather than $\mathcal{N}_i$. Further, we can apply results from the literature on covering radii to $\mathcal{V}$ and transfer bounds to $\mathcal{N}_i$. This yields a bound on $\not{d}_{\mathcal{N}_i}\left(S_{(i,B)}\right)$.

Normed PSTS. It is our goal to derive a transition system from $\mathcal{N}_i$ that generates exactly the paths of bounded norm. We achieve this in a general setting by appealing once more to geometric intuition. Suppose $(S, \to_{\mathcal{S}}, \leq_{\mathcal{S}})$ is a PSTS and $\|-\|$ is a norm for S then we say

$$\mathcal{S} = (S, \to_{\mathcal{S}}, \leq_{\mathcal{S}}, \|-\|)$$

is a *normed PSTS*. Given a normed PSTS $\mathcal{S} = (S, \to_{\mathcal{S}}, \leq_{\mathcal{S}}, \|-\|)$ and a *norm-radius* $\rho \in \mathbb{N}^\infty$ we define a normed sub-PSTS $\mathbb{B}_\rho[\mathcal{S}]$ — the *ball* of norm-radius ρ in $\mathcal{S}$ — by

$$\mathbb{B}_\rho[\mathcal{S}] = \left(\mathbb{B}_\rho[S], \to_{\mathbb{B}_\rho[\mathcal{S}]}, \leq_{\mathcal{S}}, \|-\|\right)$$

where $\mathbb{B}_\rho[S] := \{s \in S \mid \|s\| < B\}$, and $\to_{\mathbb{B}_\rho[\mathcal{S}]} := \to_{\mathcal{S}} \cap (\mathbb{B}_\rho[S])^2$. Note that by construction all $\mathbb{B}_\rho[\mathcal{S}]$-paths $\boldsymbol{s}$ satisfy $\|\boldsymbol{s}\|^* < \rho$. Attaching the norm $\|-\|_{\mathcal{N}_i}$ to $\mathcal{N}_i$, we then obtain a normed PSTS $(\mathit{Config}^\infty, \to_{\mathcal{N}_i}, \leq_{\mathit{Config}}, \|-\|_{\mathcal{N}_i})$ to which we also refer to as $\mathcal{N}_i$. The transition system $\mathbb{B}_B[\mathcal{N}_i]$ produces the paths that generate the set $S_{(i,B)}$, in the sense that we may characterise $S_{(i,B)}$ by the equation:

$$S_{(i,B)} = \left\{(\boldsymbol{s}(1), s') \mid \boldsymbol{s} \text{ covering path for } s' \text{ in } \mathbb{B}_B[\mathcal{N}_i], \left\|s'\right\|_{\mathcal{N}_i;C} \leq B\right\}.$$

A Counter Abstraction on $\mathbb{B}_B[\mathcal{N}_i]$. Let us now investigate the paths generated by $\mathbb{B}_B[\mathcal{N}_i]$. Suppose we have a path $\boldsymbol{s}$ in $\mathbb{B}_B[\mathcal{N}_i]$. A complex token m that appears along $\boldsymbol{s}$ cannot carry more than B tokens of a particular colour *col* if m's *col*-coloured tokens are ejected on the path $\boldsymbol{s}$. Otherwise, we could find more than B simple tokens in a configuration along $\boldsymbol{s}$ violating $\|\boldsymbol{s}\|^*_{\mathcal{N}_i} < B$. If m's *col*-coloured tokens are *not* ejected on the path $\boldsymbol{s}$ then they play no rôle in enabling the path. This observation gives us a stepping stone for an abstraction. Along paths of $\mathbb{B}_B[\mathcal{N}_i]$, we may represent a complex token m as a mapping $\hat{m}$ of type $\mathcal{P}^{col} \to \{0, 1, \ldots, B-1, \infty\}$. Note that there are only finitely many possible mappings of this type. We can thus apply a counter abstraction on $\mathbb{B}_B[\mathcal{N}_i]$ which keeps a counter for each pair $(p', \hat{m})$ representing the number of abstract complex tokens $\hat{m}$ in complex place p'. This counter abstraction yields a Petri net. Petri nets are an instance of *strongly increasing affine nets* [BFP12] for which convenient results for

bounds on covering paths exist [BFP12]. We first recall the definition of (strongly increasing) affine nets [FMP04] and Petri nets.

Definition (Petri net, affine net). An *affine net* (AN) is a tuple $\mathcal{V} = (d, F)$ where $d \in \mathbb{N}^+$ and $F \subseteq \mathbb{N}^{d\times d} \times \mathbb{Z}^d \times \mathbb{N}^d$. For $s, s' \in \mathbb{N}^d$ and $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) \in F$ we have $s \xrightarrow{(\boldsymbol{A},\boldsymbol{B},\boldsymbol{C})}_{\mathcal{V}} s'$ just if $s \geq \boldsymbol{C}$, $s' = \boldsymbol{A}\, s + \boldsymbol{B}$ and $s' \in \mathbb{N}^d$. We say an AN $\mathcal{V} = (d, F)$ is a *Petri net* (strongly increasing) just if for all $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) \in F$, $\boldsymbol{A} = id$ ($\boldsymbol{A} \geq id$ resp.) where id is the identity matrix. If $\mathcal{V}$ is a Petri net, we treat F as $F \subseteq \mathbb{Z}^d \times \mathbb{N}^d$.

Representing Configurations of $\mathbb{B}_B[\mathcal{N}_i]$. Before we define the counter abstraction Petri net $\mathcal{V}_{i,B}$, let us define an abstraction function from NNCT configurations to configurations of $\mathcal{V}_{i,B}$. Configurations of $\mathcal{V}_{i,B}$ have dimension $\widehat{d}_{i,B} := i + (B+1)((n_C+1)(B+1)^{n_{col}-1} - 1)$, i.e. are elements of $\mathbb{N}^{\widehat{d}_{i,B}}$. Note that we need the first i places to represent the contents of $\mathbb{B}_B[\mathcal{N}_i]$'s i simple places. Further, we require the counters $i+1, i+2, \ldots, \widehat{d}_{i,B}$ to represent the pairs $(p', \hat{m})$. For concreteness, we assign a counter $i + \theta$ to each pair $(p', \hat{m})$ based on a $(B+1)$-ary representation. If $p' = p^{col}_{(j_{n_{col}})}$ and $\hat{m} = [p^{col}_{(1)} \mapsto j_0, p^{col}_{(2)} \mapsto j_1, \cdots, p^{col}_{(j_{n_{col}})} \mapsto j_{n_{col}-1}]$, then we define $\theta = \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k$ and assign counter $i + \theta$ to represent the pair. In the counter abstraction $\mathcal{V}_{i,B}$, rather than seeing an abstract complex token as a mapping of type $\mathcal{P}^{col} \to \{0, 1, \ldots, B-1, \infty\}$ we view it as a mapping of type $\mathcal{P}^{col} \to \{0, 1, \ldots, B-1, B\}$. This is may under-approximate the number of coloured tokens in a complex token, but only once it reaches B. This is necessary to correctly implement transfer transitions as ordinary Petri net transitions. Even though it slightly complicates the definition of a counter abstraction function α, it remains the same in spirit. The definition of the counter abstraction function $\alpha_{i,B} : \mathit{Config}^\infty \to \mathbb{N}^{\widehat{d}_{i,B}}$ below makes this representation formal for configurations in Config^∞:

$$\alpha_{i,B}(s)(j) := |s(p_{(j)})| \text{ for } 1 \leq j \leq i \text{ and}$$
$$\alpha_{i,B}(s)(i+\theta) := \sum_{m^{col} \in \gamma^{col}_B(j_0,\ldots,j_{n_{col}-1})} s\left(p'_{(j_{n_{col}})}\right)(m^{col})$$

where $j_0, \ldots, j_{n_{col}}$ range over $1 \leq j_{n_{col}} \leq n_C$ and $0 \leq j_k \leq B$, $k = 0, \ldots, n_{col} - 1$ and determine $\theta = \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k$. Further, we write

$$\gamma^{col}_B(j_0, \ldots, j_{n_{col}-1}) = \left\{ m \in \mathcal{M}^{colour} \mid \min(m(p^{col}_{(k)}), B) = j_{k-1} \right\}$$

for the set of *concrete* complex tokens that are represented by an abstract complex token

$$[p^{col}_{(1)} \mapsto j_0, p^{col}_{(2)} \mapsto j_1, \cdots, p^{col}_{(j_{n_{col}})} \mapsto j_{n_{col}-1}].$$

It turns out that $\alpha_{i,B}$ satisfies a *'linearity'* property. This makes $\alpha_{i,B}$ easier to use and simplifies many of our arguments.

Lemma 10. Let $i \leq n_S$, $B \in \mathbb{N}$. Suppose $s, s' \in \mathit{Config}^\infty$ such that both $\|s\|_{\mathcal{N}_i}, \|s'\|_{\mathcal{N}_i} < \infty$, then $\alpha_{i,B}(s \oplus s') = \alpha_{i,B}(s) + \alpha_{i,B}(s')$ and if $s \ominus s' \in \mathit{Config}^\infty$ then $\alpha_{i,B}(s \ominus s') = \alpha_{i,B}(s) - \alpha_{i,B}(s')$.

The Petri Net $\mathcal{V}_{i,B}$. As a next step, we define the Petri net $\mathcal{V}_{i,B}$. It is our intention to show there is a tight (length preserving) correspondence between $\mathbb{B}_B[\mathcal{N}_i]$-paths and paths in $\mathcal{V}_{i,B}$. We define the counter abstraction Petri net $\mathcal{V}_{i,B} = (\widehat{d}_{i,B}, \hat{F}_{i,B})$ where the set of rules $\hat{F}_{i,B}$ is derived from $\mathcal{R}$ by case analysis:

- For $r = (I, O) \in$ *SimpleRules*, add $\widehat{r} := (\alpha_{i,B}(O \ominus I), \alpha_{i,B}(I))$ to $\hat{F}_{i,B}$.
- For $r = ((p, I), (p', c, O)) \in$ *ComplexRules*, add for all $m \in \mathcal{M}^{colour}$ such that $\|m\|_{col} \leq B$ the rule $\widehat{r}_m := (\widehat{r}'_m, d_m)$ to $\hat{F}_{i,B}$ where

$$\begin{aligned} \widehat{r}'_m &= \alpha_{i,B}\left(O \oplus \left[p' \mapsto [m \oplus c]\right] \ominus (I \oplus [p \mapsto [m]])\right) \\ d_m &= \alpha_{i,B}\left(I \oplus [p \mapsto [m]]\right). \end{aligned}$$

- For $r = ((p, I), (p', P, O)) \in$ *TransferRules*, we derive a set of rules $\widehat{r}_m$ parametrised by the set of $m \in \mathcal{M}^{colour}$ such that $\|m\|_{col} \leq B$ and $\max_{p \in P}(|m(p)|) < B$. We define the rule $\widehat{r}_m := (\widehat{r}'_m, d_m)$ and add them to $\hat{F}_{i,B}$ where $m_P = m \restriction P$, $m_{\overline{P}} = m \restriction (\mathcal{P}^{col} \setminus P)$, and

$$\begin{aligned} \widehat{r}'_m &= \alpha_{i,B}(O \oplus \left(m_P \circ \zeta^{-1}\right) \oplus \left[p' \mapsto \left[m_{\overline{P}}\right]\right] \ominus (I \oplus [p \mapsto [m]])), \\ d_m &= \alpha_{i,B}\left(I \oplus [p \mapsto [m]]\right). \end{aligned}$$

While we can simply apply $\alpha_{i,B}$ to obtain a corresponding rule $\hat{r}$ in $\hat{F}_{i,B}$ for a simple rule r of $\mathcal{N}$, the situation is different for complex and transfer rules. To simulate the latter two, we need to manipulate counters that represent complex tokens. Since we only need to consider a finite number of abstract complex tokens, we can introduce an appropriate abstract complex or transfer rule for each such complex token. Let us illustrate the derivation of $\mathcal{V}_{i,B}$ on a simple example.

Example 19. Suppose we have a NNCT $\mathcal{N}$ with a simple place p, a complex place p', two colours *red* and *black*, and a colour mapping ζ that maps both *red* and *black* to p. Further, let us assume that $\mathcal{N}$ has two rules: a complex rule r and a transfer rule r' such that

$$r = ((p', \emptyset), (p', [\mathit{black} \mapsto [\bullet]], \emptyset)), \qquad r' = ((p', \emptyset), (p', \{\mathit{black}\}, \emptyset)),$$

i.e. both rules r and r' take a complex token from p' to p'; while doing so r injects a black token and r' transfers all black tokens. Suppose we take the bound $B = 2$ and consider the counter abstraction Petri net $\mathcal{V}_{1,2}$. First, let us focus on the configuration s with 5 simple tokens in p and a complex token m in p'. The complex token m carries one *red* and 10 *black* tokens, i.e.

$$s = \left[p \mapsto \left[\bullet^5\right], p' \mapsto [m]\right] \text{ where } m = \left[\mathit{red} \mapsto [\bullet], \mathit{black} \mapsto \left[\bullet^{10}\right]\right].$$

The abstraction function $\alpha_{1,2}$ determines a counter $j_{p', red=1, black \geq 2}$ that represents complex tokens in p' with *red* $= 1$ and *black* ≥ 2. The configuration $\alpha_{1,2}(s)$ of the counter abstraction Petri net $\mathcal{V}_{1,2}$ representing s then looks as follows:

$$\left[j_p \mapsto 5, j_{p', red=1, black \geq 2} \mapsto 1\right].$$

The rules of $\mathcal{V}_{1,2}$ maintain this representation. For example, let us consider the complex rule r. In $\mathcal{V}_{1,2}$, we obtain a family of rules for r. One of them, $\hat{r}_{m_0}$, applies in the above configuration and

removes a token from $j_{p',\mathit{red}=1,\mathit{black}\geq 2}$ and adds a token to $j_{p',\mathit{red}=1,\mathit{black}\geq 2}$. Since r injects a *black* coloured token and $j_{p',\mathit{red}=1,\mathit{black}\geq 2}$ represents complex token with more than 2 black coloured tokens, firing $\hat{r}_{m_0}$ does not change the configuration. Consider now a different complex token $m' = [\mathit{red} \mapsto [\bullet], \mathit{black} \mapsto [\bullet]]$ and the transfer transition r'. In $\mathcal{V}_{1,2}$, r' leads to the rule $\hat{r}'_{m'}$ which removes a token from $j_{p',\mathit{red}=1,\mathit{black}=1}$, adds one token to $j_{p',\mathit{red}=1,\mathit{black}=0}$, and adds one token to j_p, i.e.

$$\hat{r}'_{m'} = \left(\left[j_p \mapsto 1, j_{p',\mathit{red}=1,\mathit{black}=0} \mapsto 1, j_{p',\mathit{red}=1,\mathit{black}=1} \mapsto -1\right], \left[j_{p',\mathit{red}=1,\mathit{black}=1} \mapsto 1\right]\right).$$

Remedying the Coverability Discrepancy. Coverability instances for $\mathcal{V}_{i,B}$ and $\mathbb{B}_B[\mathcal{N}_i]$ are unfortunately not isomorphic. The reason is that the order on $\mathbb{N}^{\widehat{d}_{i,B}}$ is ignorant of complex tokens. Consider the above example again. Clearly $m' \leq_{\mathcal{M}^{\mathit{colour}}} m$, but in $\mathbb{N}^{\widehat{d}_{i,B}}$ the configurations

$$\hat{s}_0 = \left[j_{p',\mathit{red}=1,\mathit{black}=1} \mapsto 1\right] \text{ and } \hat{s}_1 = \left[j_{p',\mathit{red}=1,\mathit{black}\geq 2} \mapsto 1\right]$$

are incomparable. To remedy this, we add rules to $\mathcal{V}_{i,B}$ that allow $\hat{s}_1$ to transition to $\hat{s}_0$. This is reminiscent of introducing *'lossy'* steps. Thus, we obtain the Petri net

$$\mathcal{V}^{\leq}_{i,B} = \left(\widehat{d}_{i,B}, \hat{F}_{i,B} \cup \hat{F}^{\leq}_{i,B}\right).$$

Let us write $s_{p,m} := [p \mapsto [m]]$ for $p \in \mathcal{P}^C$, $m \in \mathcal{M}^{\mathit{colour}}$, and define

$$\hat{F}^{\leq}_{i,B} = \left\{ \left(\alpha_{i,B}(s_{p',m'}) - \alpha_{i,B}(s_{p',m}), \alpha_{i,B}(s_{p',m'})\right) \;\middle|\; \begin{array}{l} m <_{\mathcal{M}^{\mathit{colour}}} m' \in \mathcal{M}^{\mathit{colour}}, \\ \|m\|_{\mathit{col}}, \left\|m'\right\|_{\mathit{col}} \leq B, p' \in \mathcal{P}^C \end{array} \right\}$$

Remark. Even though the Petri nets $\mathcal{V}_{i,B}$ and $\mathcal{V}^{\leq}_{i,B}$ have a vast number of rules, the maximal entry in any rule is tightly controlled: $\max(R, B) \geq \max\{\widehat{r}(i) \mid \widehat{r} \in \hat{F}_{i,B} \cup \hat{F}^{\leq}_{i,B}\}$. We aim to exploit a result by Bonnet et al. that shows that covering radii in SIAN are only sensitive to the maximal entry in any rule instead of the cardinality of the rule set.

Establishing a Bisimulation between $\mathbb{B}_B[\mathcal{N}_i]$ and $\mathbb{B}_B[\mathcal{V}_{i,B}]$. We now set out to show the connection between $\mathcal{V}_{i,B}$ and $\mathcal{N}_i$. First, we need to add a norm to $\mathcal{V}_{i,B}$: define

$$\|v\|_{\mathcal{V}_{i,B}} = \max_{j \in \langle i \rangle}(|v(j)|).$$

Hence, the transition system $\mathbb{B}_B[\mathcal{V}_{i,B}]$, the ball of norm-radius B, is well-defined for $\mathcal{V}_{i,B}$. The technical lemma below shows that we can follow transitions $\mathbb{B}_B[\mathcal{N}_i]$ in lock-step in $\mathcal{V}_{i,B}$ and *vice versa* under appropriate conditions.

Lemma 11. Suppose $s \in \mathit{Config}^\infty$, $i \leq n_S$, $B \in \mathbb{N}$ such that $|s(p_{(j)})| = \infty$ for all $i < j \leq n_S$.
(i) If $s \to_{\mathbb{B}_B[\mathcal{N}_i]} s'$ then we have $\alpha_{i,B}(s) \to_{\mathcal{V}_{i,B}} \alpha_{i,B}(s')$.
(ii) If $\alpha_{i,B}(s) \to_{\mathcal{V}_{i,B}} \alpha_{i,B}(s) + r$ then there exists an $s' \in \mathit{Config}^\infty$ such that $\alpha_{i,B}(s') = \alpha_{i,B}(s) + r$ and $s \to_{\mathcal{N}_i} s'$.

In order to make sense of a bisimulation, let us temporarily label the transition systems relations of $\mathcal{N}_i$ and $\mathcal{V}_{i,B}$. We label transitions of $\mathcal{N}_i$ by $s \xrightarrow{s'}_{\mathcal{N}_i} s'$ if $s \to_{\mathcal{N}_i} s'$. We label the transition $v \to_{\mathcal{V}_{i,B}} v'$ of $\mathcal{V}_{i,B}$ by s' if $\alpha_{i,B}(s') = v'$ and by ϵ otherwise.

Proposition 7. The relation $\{(s, \alpha_{i,B}(s)) \mid s \in \mathbb{B}_B[\mathit{Config}^\infty], |s(p_{(j)})| = \infty, i < j\}$ is a bisimulation between the labelled transition systems $\mathbb{B}_B[\mathcal{N}_i]$ and $\mathbb{B}_B[\mathcal{V}_{i,B}]$.

A Connection between $\mathcal{N}_i$, $\mathcal{V}_{i,B}$, and $\mathcal{V}^{\leq}_{i,B}$. Unfortunately, the bisimulation between $\mathbb{B}_B[\mathcal{N}_i]$ and $\mathbb{B}_B[\mathcal{V}_{i,B}]$ is not quite enought to obtain a bound on the covering diameter $d_{\mathcal{N}_i}\left(S_{(i,B)}\right)$. The reason is that the covering paths of $\mathbb{B}_B[\mathcal{N}_i]$ do not necessarily give rise to covering paths in $\mathbb{B}_B[\mathcal{V}_{i,B}]$ because of the order discrepancy. Hence, we need to extend the relationship between $\mathcal{V}_{i,B}$ and $\mathcal{N}_i$ to $\mathcal{V}^{\leq}_{i,B}$. For this purpose, we temporarily relabel the transition relations of $\mathcal{V}_{i,B}$, $\mathcal{V}^{\leq}_{i,B}$, and $\mathcal{N}_i$ so that we can establish a set of simulation results. We label all transitions of $\mathcal{V}_{i,B}$ and $\mathcal{N}_i$ with the distinguished label $\sim$. Transitions of $\mathcal{V}^{\leq}_{i,B}$ are labelled to indicate which rule set was used. We label the transition $v \to_{\mathcal{V}^{\leq}_{i,B}} v'$ with ϵ if it uses a rule $r \in \hat{F}^{\leq}_{i,B}$; if the used rule r is an element of $\hat{F}_{i,B}$ then we label the transition with $\sim$. This labelling allows us to prove several simulations:

Proposition 8. The relation $\{(\alpha_{i,B}(s), \alpha_{i,B}(s)) \mid s \in \mathit{Config}^\infty\}$ is a simulation relation for $\mathcal{V}_{i,B}$ and $\mathcal{V}^{\leq}_{i,B}$. The relation $\{(\alpha_{i,B}(s), s') \mid s \leq_{\mathit{Config}} s', s, s' \in \mathit{Config}^\infty, |s(p_{(j)})| = \infty, i < j\}$ is a weak simulation for $\mathcal{V}^{\leq}_{i,B}$ and $\mathcal{N}_i$.

Exploiting the (Bi-)Simulations for a Bound on the Covering Radius. We can now exploit the bisimulation established in Proposition 7 and the simulations of Proposition 8 to relate the covering distances induces by $\mathcal{N}_i$ and $\mathcal{V}^{\leq}_{i,B}$. Since (i) $\mathbb{B}_B[\mathcal{V}_{i,B}]$ bisimulates $\mathbb{B}_B[\mathcal{N}_i]$, (ii) $\mathcal{V}^{\leq}_{i,B}$ simulates $\mathcal{V}_{i,B}$, and (iii) $\mathcal{N}_i$ simulates $\mathcal{V}^{\leq}_{i,B}$, we can take an arbitrary covering path $\boldsymbol{s}$ for a configuration pair in $S_{i,B}$ and obtain a path $\hat{\boldsymbol{s}}_0$ in $\mathcal{V}_{i,B}$ (using (i)) and extend it to a covering path $\hat{\boldsymbol{s}}$ that corresponds to $\boldsymbol{s}$ in $\mathcal{V}^{\leq}_{i,B}$ (using (ii)). If we find a suitable, shorter covering path $\hat{\boldsymbol{s}}'$ in $\mathcal{V}^{\leq}_{i,B}$, we can replicate it in $\mathcal{N}_i$ using (iii). Hence, we can show the following covering distance inequality:

Corollary 6. Let $i \leq n_S$, $B \in \mathbb{N}$. For all $(s, s') \in S_{i,B}$ we have

$$\mathit{dist}_{\mathcal{N}_i}(s, s') \leq \mathit{dist}_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s), \alpha_{i,B}(s')).$$

This result immediately gives us a way to bound the covering diameter $d_{\mathcal{N}_i}\left(S_{(i,B)}\right)$ by the greatest covering radius in $\mathcal{V}^{\leq}_{i,B}$ for configurations that have bounded norm:

Corollary 7. Let $i \leq n_S$, $B \in \mathbb{N}$. Then

$$d_{\mathcal{N}_i}\left(S_{(i,B)}\right) \leq \sup\left\{\rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s')) : \left\|s'\right\|_{\mathcal{N}_i;C} \leq B\right\}.$$

Since the covering radii on the right-hand-side involve the Petri net $\mathcal{V}^{\leq}_{i,B}$, we can appeal to a result by Bonnet et al. for bounds on their cover radii.

Lemma 12 ([BFP12, Lemma 12]). Let $\mathcal{V} = (d, F)$ be a strongly increasing AN, M_{cov} be the marking to be covered, and $R_{\mathcal{V}} = \max\{|\mathbf{A}(i)(j)|, |\mathbf{B}(i)|, |\mathbf{C}(i)| : (\mathbf{A}, \mathbf{B}, \mathbf{C}) \in F, 1 \leq i, j \leq d\}$. If we let $R'_{\mathcal{V}} = \max(\{M_{\text{cov}}(p) \mid p \in P\} \cup \{R_{\mathcal{V}}\})$ then $\rho_{\mathcal{V}}(M_{\text{cov}}) \leq (6R_{\mathcal{V}}R'_{\mathcal{V}})^{(d+1)!}$.

Applying this lemma on the inequality above immediately gives us a concrete bound on the covering diameter of $S_{(i,B)}$:

Corollary 8. For $i \leq n_S$ and $B \in \mathbb{N}$ we have

$$d_{\mathcal{N}_i}\left(S_{(i,B)}\right) \leq \left(6 \max\{R, B, 1\} \max\{R', B\}\right)^{\left(\widehat{d}_{i,B}+1\right)!}.$$

Solving the Rackoff Recurrence Relation. We have finally obtained a bound on the covering diameter of $S_{(i,B)}$. With a simple induction we can now exploit the Rackoff recurrence relation we have established earlier. This yields the desired bound on the covering radius for s_{cov}.

Theorem 12. *For all $i \leq n_S$ the following three inequalties hold:*

$$\begin{aligned}
\rho_{\mathcal{N}_0}(s_{cov}) &\leq 2 \uparrow\uparrow 2 \operatorname{slog}(48 n_C n_S n_{col} R'), \\
\rho_{\mathcal{N}_{i+1}}(s_{cov}) &\leq 2^{2^{(\max\{\rho_{\mathcal{N}_i}(s_{cov}),2\})^{48 n_{col} n_S n_C R'}}}, \text{ and} \\
\rho_{\mathcal{N}}(s_{cov}) &\leq 2 \uparrow\uparrow 2n_S + 2 \operatorname{slog}(48(n_S+1) n_{col} n_S n_C R').
\end{aligned}$$

Equipped with this bound on the covering radius $\rho_{\mathcal{N}}(s_{\text{cov}})$, we can follow the usual Rackoff argument. The bound gives us an upper limit on how much space we require to represent a NNCT configuration in a non-deterministic Turing machine that guesses a covering path. This immediately yields Tower-membership of NNCT coverability.

Corollary 9. Coverability for NNCTs is decidable and in Tower.

Note that the full and detailed proofs of all lemmas, propositions, corollaries, and theorems can be found in Appendix C.3. Now that we have established Tower-membership of NNCT coverability we move on to obtaining lower bounds.

5.4 A Lower Bound

In this section, we show that simple coverability, boundedness, and termination for total-transfer NNCT is Tower-hard. Our hardness proof uses a variant of Stockmeyer's yardstick construction [Sto74]. We show that we can encode a bounded counter machine in NNCT which is constructed inductively. Given $n \geq 1$, we construct the *yardstick* counters $c_1, \ldots, c_n$: for each i the counter c_i is bounded by $2 \uparrow\uparrow i$. Each counter c_i can be incremented, decremented, and tested for zero. Furthermore, operations of the counter c_{i+1} are implemented using operations of the counters $c_1, \ldots, c_i$. Following Lazic et al. [Laz+07], we present our proof using pseudo code

rather than explicit NNCT rules, which we believe is clearer and more readable. In any case, it is straightforward, if tedious, to derive the corresponding NNCT. The type of yardstick construction we use is reminiscent of Lipton's Expspace-hardness proof for coverability and reachability for VAS [Lip76]. A $2 \uparrow\uparrow i$-bounded counter c is represented by two places: p_c and its *complement place* $\overline{p}_c$. A valuation $v(c)$ of c is represented by p_c containing $v(c)$ tokens and $\overline{p}_c$ containing $2 \uparrow\uparrow i - v(c)$ tokens. Notice that the places p_c and $\overline{p}_c$ maintain the invariant that the number of tokens they carry sum up to $2 \uparrow\uparrow i$ at all times. An increment (decrement) of c can then be implemented by adding a token to p_c ($\overline{p}_c$ resp.) and removing one from $\overline{p}_c$ (p_c resp.). This yields implementations for c++ and c−−. In order to implement a zero test *iszero*(c), an additional $2 \uparrow\uparrow i$-bounded counter s_i is maintained. On the invocation of *iszero*(c) a non-deterministic number k of tokens is removed from $\overline{p}_c$ and k is added to s_i, which is assumed to be 0 at the invocation of *iszero*(c). Then, the operation *ismax&reset*$(-)$ is applied to s_i. The operation *ismax&reset*$(-)$ performs a decrement of precisely $2 \uparrow\uparrow i$ on s_i and blocks if s_i has a smaller value. This of course means that *ismax&reset*$(-)$ can only succeed if $c = 0$ to begin with and $k = 2 \uparrow\uparrow i$. This construction scheme involving the operation *ismax&reset*$(-)$ is originally due to Lipton [Lip76]. An excellent account of the technique and the ideas underlying the construction by Esparza can be found in [Esp98]. Lipton's scheme has also been been applied to obtain Tower-hardness of coverability for VAS with one stack (SVAS) [Laz13] and branching VAS with states (BVASS) [LS14]. Our construction of *ismax&reset*$(-)$ is similar to Lazic's proof for SVAS. Lazic uses the $2 \uparrow\uparrow i$-bounded counters to enumerate all possible stacks over a binary alphabet of height $2 \uparrow\uparrow i$ while decrementing the given counter one-by-one. This yields a guaranteed decrement of $2 \uparrow\uparrow (i+1)$. In the case of NNCT, stacks are not available. However, we can encode arrays into complex tokens in a yardstick fashion and, instead of enumerating stacks, we enumerate all binary arrays of length $2 \uparrow\uparrow i$.

Theorem 13. *Simple coverability, boundedness, and termination for total-transfer NNCT is Tower-hard.*

Proof. We can deduce Tower-hardness by showing that given a deterministic bounded two-counter machine $\mathcal{M}$ of size n with counters that are $2 \uparrow\uparrow n$-bounded, we can construct an NNCT $\mathcal{N}_\mathcal{M}$ in polynomial-time that weakly bisimulates $\mathcal{M}$. The machine $\mathcal{M}$ can use the following operations: x++, x−−, *reset*(x), *iszero*(x), and *ismax*(x) for each counter x.

The NNCT $\mathcal{N}_\mathcal{M}$. Each simulation state of $\mathcal{N}_\mathcal{M}$ represents a valuation v of $6n + 2$ *active* and *inactive* counters, and n arrays. In addition to the counters x, y of $\mathcal{M}$, the NNCT $\mathcal{N}_\mathcal{M}$ simulates the auxiliary counters s_i, p_i, p'_i, c_i, c'_i, and an auxiliary array a_i for each $i \leq n$. To simplify notation, let us write C_i for the set of counters $\{s_i, p_i, p'_i, c_i, c'_i\}$ when $i < n$, and C_n for the set of counters $\{s_n, p_n, p'_n, c_n, c'_n, x, y\}$. Each active counter $d \in C_i$ is $2 \uparrow\uparrow i$-bounded, each inactive counter has an undefined value. For each i the array a_i has length exactly $(2 \uparrow\uparrow i) + 1$ and carries values $a_i(j) \in \{0, 1, 2\}$ for $j \leq 2 \uparrow\uparrow i$. The NNCT $\mathcal{N}_\mathcal{M}$ has two simple places $p[d]$ and $\overline{p}[d]$ for each counter $d \in C_i$, three complex places $p[a_i]$, $\overline{p}[a_i]$, and $p[\mathit{aux}_i]$, and two colours $p[j_i]$ and $\overline{p}[j_i]$ for each array a_i. Further, $\mathcal{N}_\mathcal{M}$ has a complex *'sink'* place $p[\mathit{disc}]$. The colour mapping ζ of $\mathcal{N}_\mathcal{M}$ maps $p[j_i]$ to $p[p'_i]$ and $\overline{p}[j_i]$ to $\overline{p}[p'_i]$. Lastly, $\mathcal{N}_\mathcal{M}$ has a number (polynomial in n) of simple places encoding the control of $\mathcal{M}$ and the internal control of $\mathcal{N}_\mathcal{M}$. Further, $\mathcal{N}_\mathcal{M}$'s transfer rules are all total, and hence $\mathcal{N}_\mathcal{M}$ is a total-transfer NNCT.

Move a complex-token from $\overline{p}[a_i]$ *to* $p[\mathit{aux}_i]$*;*
deactivate(p'_i);
Eject the contents of a complex-token m *in* $p[\mathit{aux}_i]$ *and its remains into* $p[\mathit{disc}]$*;*
isequal(p_i, p'_i); *reset*(p'_i);
Create an empty complex-token in $p[\mathit{aux}_i]$*;*
while $(p_i \neq p'_i)$ **do**
  Inject a $p[j_i]$*-coloured token into a complex token in* $p[\mathit{aux}_i]$*;*
  p'_i++;
while $(\neg(\mathit{ismax}(p'_i)))$ **do**
  Inject a $\overline{p}[j_i]$*-coloured token into a complex token in* $p[\mathit{aux}_i]$*;*
  p'_i++;
Move a complex-token from $p[\mathit{aux}_i]$ *to* $p[a_i]$*;*
reset(p'_i);

Figure 5.6: The implementation of $a_i(p_i)$++. Swapping $p[a_i]$ for $\overline{p}[a_i]$ in Line 1 and $\overline{p}[a_i]$ for $p[a_i]$ in Line 12 yields an implementation of $a_i(p_i)$−−.

Representing a Valuation. A valuation v is represented by a configuration s as follows:

- For each i and $d \in C_i$, if d is active then there are exactly $v(d)$ •-tokens in $p[d]$ and $2 \uparrow\uparrow i - v(d)$ •-tokens in $\overline{p}[d]$.
- For each i and $d \in C_i$, if d is inactive then both places $p[d], \overline{p}[d]$ are empty.
- For each i the array a_i is represented as follows. For each $k \leq 2 \uparrow\uparrow i$, let $m_{(i,k)}$ be a complex token such that $m_{(i,k)}$ contains exclusively tokens of colours $p[j_i]$ and $\overline{p}[j_i]$: k tokens of colour $p[j_i]$ and $2 \uparrow\uparrow i - k$ tokens of colour $\overline{p}[j_i]$. Then, to represent a_i, there are exactly $v(a_i(k))$ complex tokens $m_{(i,k)}$ in $p[a_i]$ and $2 - v(a_i(k))$ complex tokens $m_{(i,k)}$ in $\overline{p}[a_i]$.
- For each i the place $p[\mathit{aux}_i]$ is empty.

Reducing from $\mathcal{M}$'s Halting Problem. The question whether $\mathcal{M}$ halts, i.e. whether $\mathcal{M}$ reaches a halting control state from its initial state, can then be answered by performing a simple coverability query on $\mathcal{N}_\mathcal{M}$'s simple places encoding $\mathcal{M}$'s finite control. Assuming that only $\mathcal{M}$'s halting control states have no successors, $\mathcal{M}$'s halting problem also reduces to the termination problem for $\mathcal{N}_\mathcal{M}$. Augmenting $\mathcal{N}_\mathcal{M}$ with an additional simple place that is incremented with every transition shows that the halting problem for $\mathcal{M}$ reduces to deciding boundedness of $\mathcal{N}_\mathcal{M}$.

Auxilliary Operations. In addition to $\mathcal{M}$'s operations, we implement further instructions to simplify and improve readability. The NNCT $\mathcal{N}_\mathcal{M}$ will simulate *activate*(d), *deactivate*(d) for $d \in \bigcup_{i \in \langle n \rangle} C_i$, and operations *reset*$(d)$, *iszero*$(d)$, and *ismax*$(d)$ for $d \in \bigcup_{i \in \langle n \rangle} C_i \setminus \{s_i\}$. Further for each $i \in \langle n \rangle$, we implement *ismax*&*reset*(s_i) and the counter-specialised operations:

isequal(p_i, p'_i), $a_i(p_i)$++, $a_i(p_i)$−−, *reset*$(a_i(p_i))$, *iszero*$(a_i(p_i))$, *ismax*$(a_i(p_i))$, *activate*(a_i).

All above operations are only guaranteed to succeed if the counters in question are active at the start of the operation.

An Inductive Construction. The counters in C_1 are 2-bounded so implementing operations on them is trivial. For $i < n$, operations on a_i are simulated using s_i, p_i, p'_i, c_i, c'_i and operations

on s_{i+1}, p_{i+1}, p'_{i+1}, c_{i+1}, c'_{i+1} are simulated using operations on p_i, p'_i, c_i, c'_i, and a_i. In the following, we give an account of how to implement a number of selected operations. The implementation of the remainder is either analogous or straightforward. We refer the reader to Appendix C.4 for a complete version of the construction. A cornerstone of our encoding scheme is the capability to represent and operate on arrays. They are a key ingredient of our implementation of the operation *ismax&reset*(s_{i+1}) which is central in the Lipton-style construction. Thus, we start with the array operations.

(i) In Figure 5.6, we show how to implement $a_i(p_i)$++. The operation $a_i(p_i)$++ can only succeed if p_i and p'_i are active.

Suppose $a_i(p_i)$++ is executed in a configuration s that represents valuation v and p_i, p'_i are active. If $v(a_i(v(p_i))) < 2$, then we know there exists a complex token $m_{(i,v(p_i))}$ in $\overline{p}[a_i]$ and all complex tokens in $\overline{p}[a_i]$ are of the form $m_{(i,k)}$ for some $k \leq 2 \Uparrow i$. The move of one $m_{(i,k)}$ complex token from $\overline{p}[a_i]$ to $p[aux_i]$ results in $p[aux_i]$ containing just $m_{(i,k)}$, since by assumption $p[aux_i]$ was empty before. After deactivating p'_i, we know that both $p[p'_i]$ and $\overline{p}[p'_i]$ are empty. Ejecting the contents of $m_{(i,k)}$ removes $m_{(i,k)}$ from $p[aux_i]$, inserts k $\bullet$-tokens into $p[p'_i]$, $2 \Uparrow i - k$ $\bullet$-tokens into $\overline{p}[p'_i]$, and places the remaining (now empty) complex token into $p[disc]$. We can see that, disregarding a_i, the configuration we have reached represents a partial valuation v' that sets $v'(p'_i) = k$ and $v'(p_i) = v(p_i)$. After executing *isequal*(p_i, p'_i), the simulation only succeeds if $k = v(p_i)$. Hence, $\overline{p}[a_i]$ now contains $2 - (v(a_i(v(p_i))) + 1)$ complex tokens $m_{(i,v(p_i))}$ and the same number of other complex tokens $m_{(i,k)}$. The two following while loops carefully inject $p[j_i]$-coloured tokens and $\overline{p}[j_i]$-coloured tokens into the newly created token at $p[aux_i]$ to yield a new $m_{(i,v(p_i))}$ located in $p[aux_i]$ which we move to $p[a_i]$. Thus, $p[a_i]$ now contains $v(a_i(v(p_i)))+1$ complex tokens $m_{(i,v(p_i))}$ and the same number of other complex tokens $m_{(i,k)}$, as before. The configuration s' that we have reached represents a valuation v'' such that $v''(a_i(j)) = v(a_i(j))$ for all $0 \leq j \leq 2 \Uparrow i$ and $j \neq v(p_i)$, and $v''(a_i(v(p_i))) = v(a_i(v(p_i))) + 1$, as desired.

Considering the case that $v(a_i(v(p_i))) = 2$ initially, we see that the simulation either blocks while attempting to move a complex token $m_{(i,k)}$ from $\overline{p}[a_i]$ to $p[aux_i]$ or the simulation blocks on the execution of *isequal*(p_i, p'_i), since it is impossible to obtain $k = v(p_i)$. By swapping $p[a_i]$ and $\overline{p}[a_i]$ in Figure 5.6, we obtain an implementation of $a_i(p_i)$--.

(ii) In Figure 5.7, we show how to implement the operation *activate*(a_i) which can only succeed if both counters p_i and p'_i are active.

The interior for-loop simply repeats its body twice and can be considered syntactic sugar. Suppose *activate*(a_i) is invoked in a configuration s such that in s the places $p[a_i]$ and $\overline{p}[a_i]$ are empty. We can then follow our analysis in the second part of $a_i(p_i)$++'s implementation to see that the interior for-loop places two complex tokens $m_{i,k}$ into $\overline{p}[a_i]$ for all $k \leq 2 \Uparrow i$. Hence, when the outer for-loop terminates, we have reached a configuration that represents a valuation v such that $v(a_i(k)) = 0$ for all k.

(iii) We now turn to the implementation of *ismax&reset*(s_{i+1}). Note that we assume implementations of *reset*$(a_i(p_i))$, *iszero*$(a_i(p_i))$, and *ismax*$(a_i(p_i))$ here. Since we know how to implement $a_i(p_i)$++ and $a_i(p_i)$-- and since the entries in a_i are all 2-bounded, their implementations are straightforward and can be found in Appendix C.4. The following shows how to implement *ismax&reset*(s_{i+1}).

for $p_i := 0$ **to** $2 \Uparrow i$ **do**
  for $z := 0$ **to** 1 **do**
    reset(p_i');
    Create an empty complex-token in $p[aux_i]$;
    while $(p_i \neq p_i')$ **do**
      Inject a $p[j_i]$*-coloured token into a complex token in* $p[aux_i]$;
      p_i'++;
    while $(\neg(ismax(p_i')))$ **do**
      Inject a $\overline{p}[j_i]$*-coloured token into a complex token in* $p[aux_i]$;
      p_i'++;
    Move a complex-token from $p[aux_i]$ *to* $\overline{p}[a_i]$;

Figure 5.7: The implementation of *activate*(a_i).

for $p_i := 0$ **to** $2 \Uparrow i$ **do** (*reset*$(a_i(p_i))$;)
while (*iszero*$(a_i(2 \Uparrow i))$) **do**
  s_{i+1}--; *reset*(p_i); $a_i(p_i)$++;
  while *ismax*$(a_i(p_i))$ **do**
    $a_i(p_i)$--; p_i++; $a_i(p_i)$++;

We know that after the for-loop, a_i is the array such that $a_i(x) = 0$ for all $x \leq 2 \Uparrow i$. The array a_i is meant to be binary representation of a number between 0 and $2 \Uparrow (i+1)$. This number is initially 0 and we can see that the outer while loop performs a long addition of 1 for each iteration. If $v(a_i(v(p_i))) = 2$, then $v(p_i)$ is an index representing a carry bit in the long addition computation. For each number represented by a_i we perform s_{i+1}--. Hence, if initially $v(s_{i+1}) = 2 \Uparrow (i+1)$ then, after performing *ismax&reset*(s_{i+1}), we reach a configuration that represents a valuation v' such that $v'(s_{i+1}) = 0$. However, if $v(s_{i+1}) < 2 \Uparrow (i+1)$ holds initially, then after $v(s_{i+1})$ iterations the resulting valuation v' would set $v'(s_{i+1}) = 0$ and a_i would represent the number $v(s_{i+1})$. Since $v(s_{i+1}) < 2 \Uparrow (i+1)$, this implies that $a_i(2 \Uparrow i) = 0$, and hence the body of the outer while loop is executed again leading to an invocation of s_{i+1}-- which will block. Hence, *ismax&reset*(s_{i+1}) will block when executed in a configuration representing v such that $v(s_{i+1}) < 2 \Uparrow (i+1)$.
(iv) The following shows how to implement *iszero*(d) for $d \in \{p_{i+1}, p_{i+1}', c_{i+1}, c_{i+1}'\}$ and in the case that $i+1 = n$: $d = x$ or y. The precondition for *iszero*(d) is that s_{i+1} is 0.

while $(*)$ **do** (d++; s_{i+1}++;); *ismax&reset*(s_{i+1});
while $(*)$ **do** (d--; s_{i+1}++;); *ismax&reset*(s_{i+1})

Note that the only operations that modify s_{i+1} are *iszero*(d) and *ismax&reset*(s_{i+1}). Further, *ismax&reset*(s_{i+1}) is only invoked by *iszero*(d). Thus, except when executing *iszero*(d) we maintain that the counter s_{i+1} is 0.

If *ismax&reset*(s_{i+1}) is successfully executed after the first while loop, it is clear that the latter must have incremented s_{i+1} to $2 \Uparrow (i+1)$ and hence also d. Since d is $2 \Uparrow (i+1)$-bounded this implies that d had to be 0 to begin with and is valued $2 \Uparrow (i+1)$ after *ismax&reset*(s_{i+1})'s first invocation. The rest of the implementation then guarantees that d is decremented to 0 again and that also s_{i+1} is guaranteed to be 0 after the successful execution of *ismax&reset*(s_{i+1}). Hence,

iszero(d) succeeds only if started in a configuration that values d as 0.
A detailed description of how to implement the remaining operations can be found in Appendix C.4.

Lastly, in order to set up $\mathcal{N}_{\mathcal{M}}$ to simulate the initial configuration of $\mathcal{M}$ from its initial state, we set $\mathcal{N}_{\mathcal{M}}$ to execute *activate*$(-)$ for all counters and arrays in turn, in order of the bound size and length. From this point on, $\mathcal{N}_{\mathcal{M}}$ weakly bisimulates $\mathcal{M}$. Thus, we can reduce the halting problem for $\mathcal{M}$ to simple coverability, termination, or boundedness of $\mathcal{N}_{\mathcal{M}}$.

□

Theorem 13 tells us that simple coverability, boundedness, and termination is Tower-hard for total-transfer NNCT. Since total transfer NNCT are a subclass of NNCT, these hardness results carry over to NNCT. Further, Theorem 13 implies that coverability of NNCT is Tower-complete, and thus the upper bound that we established in Section 5.3 can be considered asymptotically optimal. Another implication of Theorem 13 is the Tower-hardness of (simple) coverability, boundedness, and termination in the alternative semantics for shaped APCPS. This follows from Theorem 11. Focussing just on coverability, we can deduce that for shaped APCPS simple coverability is Tower-complete (via Theorem 7) and that for shaped ACPS coverability is Tower-complete (using Proposition 2, Proposition 1, and Lemma 1). We summarise these results in the following theorems.

Theorem 14. *NNCT coverability is Tower-complete.*

Theorem 15. *Coverability for shaped ACPS is Tower-complete.*

In the next section, we give an overview of related work.

5.5 Related Work

In this chapter, we have introduced NNCT, shown that its coverability, boundedness, and termination problems are decidable, and established complexity results: coverability is Tower-complete, boundedness and termination are Tower-hard. Since NNCT are a Petri net extension, the literature on complexity results for similar extensions is related to our work.

Extensions of Petri nets. Coverability, boundedness, and termination are central decision problems in the vast literature of Petri net extensions. These decision problems are Expspace-complete for Petri nets. However, any non-trivial extension, such as *reset arcs*, *transfer arcs*, or *affine transitions*, typically gives rise to extraordinary increases in complexity or even the loss of decidability. For example, coverability and termination for Petri nets with reset or transfer arcs is decidable [DFS98] and Ack-complete [Sch02; Sch10; Fig+11]. However, boundedness is undecidable for reset Petri nets [DFS98], yet decidable for Petri nets with transfer arcs. *Affine nets* [FMP04] inherit the complexity and decision results for coverability, termination, and boundedness from reset nets (see also Section 2.3).

While reset, transfer, and affine nets provide extensions to the transition functions available in Petri nets, another class of extensions considers richer token types. *Nested Petri nets* [LS99] are a prominent example of the class of nets with an infinite set of token types. They may appear

closely related to NNCT, they are however much more expressive. Nested Petri nets are WSTS and so coverability and termination are decidable. However, it is easy to encode Petri nets with reset arcs into a nested Petri net. Thus, coverability and termination are Ackermann-hard and boundedness is undecidable [LS99]. Nested Petri nets allow arbitrary nesting of tokens, and synchronisation can take place across nested layers of a token. By contrast, internal synchronisation is not possible in NNCT: coloured tokens can only be ejected to simple places and then inspected (see start of Chapter 5). Our proof of the upper bound exploits this fact and it seems to explain the Tower-membership of NNCT coverability.

Data nets [Laz+07] are another member of the family of nets with arbitrary token types. Data nets allow tokens to be drawn from an arbitrary linearly ordered set. The selection of tokens that are involved in affine transitions of data nets can be constrained by a boolean formula over the linear order. Thus, data nets are a very general model. Recently, it has been discovered that coverability and termination for data nets and a subclass, Petri data nets (PDN), are in fact $\mathbf{F}_{\omega^{\omega^\omega}}$-complete [HSS12]. By contrast, Tower $= \mathbf{F}_3$ in the hierarchy of *fast-growing complexity classes* [Sch13] (see Section 2.2). In PDN, only Petri net-style transitions are allowed, i.e., addition and subtraction of tokens are allowed but not multiplication (transfer, reset, or copying). A more restricted subclass of Petri data nets studied by Lazic et al. gives rise to a Tower-hard coverability problem, namely, *unordered* Petri data nets (UPDN) which features an equality check on tokens. Lazic et al. show that coverability, termination, and boundedness are all Tower-hard, but no upper-bound is available. Unfortunately, the equality check on tokens is a big hurdle and makes it unclear whether coverability of UPDN reduces to coverability of NNCT which is why we opted for a Tower-hardness proof from first principles. Adding the ability to create fresh tokens to UPDNs yields ν-Petri nets (ν-PN). In ν-PN tokens can be seen to carry names on which transitions can perform name matching — coverability is Ack-hard, i.e. non-primitive recursive [RVFE11].

Rackoff Technique. Our proof of Tower-membership for NNCT coverability employs the *Rackoff technique*. Originally introduced to show Expspace-membership of coverability and boundedness for VAS [Rac78], the Rackoff technique has lately become popular.

Expspace Upper Bounds on Petri Nets. Rackoff-style arguments have recently been successfully applied to obtain answers to a range of open complexity questions in the Petri net literature. Demri showed that selective unboundedness can be decided in Expspace for VASS [Dem10; Dem13]. Reversal-boundedness, place-boundedness, and regularity can be captured by selective unboundedness. Thus, Demri provided Expspace upper-bounds for these decision problems. In [BS11b] Blockelet and Schmitz give an Expspace upper bound for model-checking *eventually increasing, existential CTL formulæ with Presburger path constraints* (eiPrECTL) on VAS. The logic eiPrECTL captures many properties that are decidable using the Karp-Miller tree such as coverability, boundedness, place-boundedness, and regularity. Another contribution to model-checking VAS is due to Leroux et al. [LPS13]. They obtain an Expspace upper bound on model-checking a logic that can express properties on the trace language of a Petri net. Leroux et al. leverage their result to show that the context-freeness problem for Petri nets is in Expspace. In [Ler11b], Leroux considers the reversible reachability problem for VAS. A pair of configurations (s, s') is *reversible reachable* if s' is reachable from s and *vice versa*. Leroux obtains an Expspace upper

bound for the reversible reachability problem by bounding the length of reversible paths using the Rackoff-method.

The Rackoff Technique on Petri Net Extensions. The Rackoff technique has also been generalised to apply to Petri net extensions such as SIAN, branching VAS (BVAS), and alternating BVAS with states (ABVASS). Bonnet et al. extended the Rackoff technique to SIAN to obtain Expspace upper bounds for coverability and boundedness [BFP12]. SIAN are a subclass of affine nets that prohibit reset, transfer, or copying like transitions. Another extension was presented by Demri et al. [Dem+09]. They considered coverability and boundedness for BVAS and obtained AltExpspace upper bounds for coverability and boundedness. In addition to nondeterministic transitions, BVAS feature *branching rules* that e.g. allow transitions $v \to \{v', v''\}$ such that $v = v' \oplus v''$. Hence, BVAS generate run trees. Extending BVAS with states and allowing *alternating rules* yields the model ABVASS. An alternating rule enables transitions of the form $(q, v) \to \{(q', v), (q'', v)\}$, i.e. in contrast to branching transitions where v is nondeterministically distributed along the two branches of the run tree, in an alternating transition v is passed to both branches. The coverability problem for ABVASS was investigated by Lazic and Schmitz. Using a Rackoff-style argument, they obtained a Tower upper bound [LS14]. Further, they found that already for BVASS the coverability problems is Tower-hard. Thus, coverability is Tower-complete for ABVASS. Lazic and Schmitz's work on ABVASS is of interest since both NNCT coverability and ABVASS coverability are Tower-complete. It is not obvious how to interreduce the coverability problems for NNCT and ABVASS. In particular, it is hard to see how one can simulate along a path in an ABVASS (of a fixed dimension) the ability of NNCT to model arbitrarily many complex tokens, carrying an unbounded number of coloured tokens, that can interact via ejection with other complex tokens. In the other direction, there is no clear counterpart to the tree-like runs of ABVASS in NNCT.

Stockmeyer/Lipton Yardstick Construction. The method of reducing from the parametrised halting problem of a $2 \uparrow\uparrow n$ bounded counter machine to establish Tower-hardness goes back to Stockmeyer. To establish such a reduction to the equivalence problem for star-free regular expressions (SFEq), he introduced the Stockmeyer yardstick construction [Sto74]. Lipton used a similar inductive construction from a 2^{2^n} bounded counter machine to obtain Expspace lower bounds for VAS reachability, coverability, boundedness, and termination. Lipton's construction is specialised and relies on the capability to implement a *ismax&reset*$(-)$ operation that performs a guaranteed 2^{2^n} decrement on a bounded counter. An excellent account of the Lipton construction is due to Esparza [Esp98]. Esparza shows how to model *net programs* in Petri nets, which are simple counter programs with linearly ordered procedures. The presentation of the Lipton construction as a *net program* is remarkably simple and instructive, and has inspired the presentation of several hardness results [SV06; AG11; Laz+07; Laz13] including ours.

The Lipton construction has recently been extended to yield Tower-hardness for coverability of BVASS and *VAS with one stack* (SVAS). Lazic shows that reachability and coverability are Tower-hard for SVAS in [Laz13]. The decidability of coverability and reachability is still an open question. Interestingly, recent work by Leroux et al. on SVAS has shown that boundedness and termination are decidable for SVAS and that the problem lies in Hack [LPS14]. Lazic's construction of the *ismax&reset*$(-)$ operation in SVAS is possibly to closest to ours for NNCT. Lazic's

implementation of *ismax&reset*$(-)$ enumerates all stacks over a binary alphabet of height $2 \uparrow\uparrow i$ to ensure a $2 \uparrow\uparrow (i+1)$ decrement. Our construction follows a similar strategy. We encode arrays into complex tokens and enumerate all binary arrays of length $2 \uparrow\uparrow i$ to obtain an implementation of *ismax&reset*$(-)$.

Lazic and Schmitz's Tower-hardness proof of coverability for BVASS [LS14] exploits branching transitions to implement an *ismax&reset*$(-)$ operation for $2 \uparrow\uparrow n$ bounded counters. Given that coverability for BVAS lies in AltExpspace, this is surprising at first sight. However, the combination of branching rules and control states is powerful enough to force the halving of a counter, i.e. transitions of the form $(q, v' \oplus v') \to \{(q', v'), (q', v')\}$. This is exploited in Lazic and Schmitz's construction of the *ismax&reset*$(-)$ operation.

CCFG Encodings into Petri Nets. For completeness, we would like to mention Esparza's encoding of CCFG into Petri nets [Esp97] and Ganty and Majumdar's CCFG widget [GM12] here again (see Sections 2.4 and 4.6). Ganty and Majumdar's encoding is an important ingredient in showing that verification problems for asynchronous programs, or ACPS that satisfy the empty-stack constraint, are polynomial-time interreducible to decision problems on Petri nets. We use a slightly modified version of Ganty and Majumdar's CCFG widget to implement Rule (R′-2) of the alternative semantics of APCPS in NNCT. This allows us to reduce coverability for shaped ACPS to coverability of NNCT and thus yields a Tower upper bound for the former.

In the next section, we present a brief summary of our contributions and the technical material that we have presented in this chapter.

5.6 Conclusion

Let us give a brief summary of this chapter's contents, before we list our contributions, and elaborate on future directions.

Summary. In this chapter, we have introduced NNCT as a vehicle to analyse the computational complexity of coverability, boundedness, and termination for shaped ACPS/APCPS. NNCT feature simple and complex tokens. The latter carry coloured tokens which may be injected into and ejected from complex tokens. We show that NNCT are strict WSTS and thus enjoy good model-checking properties. Configurations of NNCT have a nested multiset structure due to complex tokens, and it is possible to add elements to and transfer partial contents from a nested multiset. We exploit these capabilities of NNCT to show that the alternative semantics of shaped APCPS can be implemented in NNCT. Further, we identify total transfer NNCT, a rich subclass of NNCT, that can model shaped APCPS. We then turn to the computational complexity of NNCT coverability, boundedness, and termination. We give a geometrically inspired version of the Rackoff argument to establish Tower membership of NNCT coverability. We then present a reduction from the parametrised halting problem of $2 \uparrow\uparrow n$ bounded counter machines to simple coverability, boundedness, and termination of total transfer NNCT. This yields a non-elementary lower bound for the former decision problems.

Contributions. Let us summarise the technical contributions that we present in this chapter:

- NNCT are strict WSTS (Proposition 5), and NNCT coverabililty is Tower-complete (Theorem 14).
- Boundedness and termination for NNCT are decidable (Theorem 9) and Tower-hard (Theorem 13).
- To our knowledge, NNCT is the first extension of Petri nets—belonging to the class of nets with an infinite set of token types—that is proven to have primitive recursive coverability.
- Coverability of shaped ACPS/APCPS Exptime-reduces to simple coverability for NNCT (Theorem 10) and is Tower-complete (Theorem 15).
- Simple coverability, boundedness, and termination Exptime-reduce to their respective counterparts on the alternative semantics of shaped APCPS (Theorem 11). The latter are thus Tower-hard.

Our analysis of the coverability problem for NNCT has a surprising implication. The bound K on the number of non-commutative non-terminals in the K-shaped constraint is not the expensive resource. Inspecting the simulating NNCT $\mathcal{N}$ that we construct in the proof of Theorem 10, we see that K influences only the number of colours n_{col} and complex places n_C of $\mathcal{N}$. Combining this with the result of Theorem 9, we see that coverability is decidable in space bounded by an exponential tower of height $O(n_S + \mathrm{slog}(n_S \cdot n_{col} \cdot n_C))$ where n_S is the number of simple places of $\mathcal{N}$ which is independent of K.

Future Directions. We leave the upper bounds for boundedness and termination for NNCT as open problems. We believe that our arguments of Section 5.3 can be refined to obtain Tower membership of NNCT boundedness and termination. Once complexity results for boundedness and termination are available, it would be interesting to investigate whether our reduction from shaped APCPS to NNCT can be amended to yield reductions for boundedness and termination. In combination, such results would constitute a step towards obtaining decision and complexity results for boundedness and termination for shaped ACPS.

Chapter 6

Conclusion

In this dissertation, we have presented research contributions to the verification of asynchronous concurrency. We have focussed in particular on concurrent programs employing asynchronous message passing written in the programming language Erlang. Erlang offers a range of expressive features that give rise to many distinct Turing powerful fragments and hence its verification poses a difficult challenge. Previous research has concentrated on under-approximation [FS07; LS04] and semi-automatic techniques [Art+98; AEP04; Nys03], or restrictive fragments [Huc99; AD99; MW97]. We approach the problem of verifying Erlang programs with the following strategy: (i) identify a suitable decidable computational model as a target for abstraction, and (ii) develop a procedure that extracts a sound model from an input Erlang program, using abstract interpretation [CC77], which allows us to verify properties conservatively.

ACS and an Abstract Interpretation of λ_{Actor}. In Chapter 3 we have applied this approach with actor communicating systems (ACS) as the abstract model. ACS model a dynamic network of finite-state processes that communicate over a fixed, finite number of unordered, unbounded channels, and are equivalent to Petri nets. Thus, ACS are a suitable target model for an abstract interpretation of λ_{Actor}, which is an idealised yet expressive fragment of Erlang. We generate a sound ACS by exploring the state space of an abstract operational semantics of a given λ_{Actor} program and subsequently use it to verify coverability properties conservatively. The abstract operational semantics of λ_{Actor} is obtained by an application of the abstracting abstract machines (AAM) methodology [HM11; HM12] in the novel setting of message passing concurrency. We have implemented this procedure in our tool Soter. Soter accepts an Erlang program as input and uses a Petri net coverability tool to model-check the generated ACS. To our knowledge, Soter is the first verification tool for Erlang that makes use of infinite-state model-checking.

We have evaluated Soter on a collection of Erlang programs and our empirical results are encouraging. Even though coverability for ACS is Expspace-complete, Soter is able to verify a range of interesting and non-trivial benchmarks (50–550 LOC) in a matter of seconds (<6s). In particular, we find that our worklist exploration algorithm and the ACS simplification procedure are effective in providing speed ups and reductions in the problem size. Thus, we believe that infinite-state verification can be successfully applied to the verification of Erlang.

However, we find practical and theoretical limitations of our verification procedure that re-

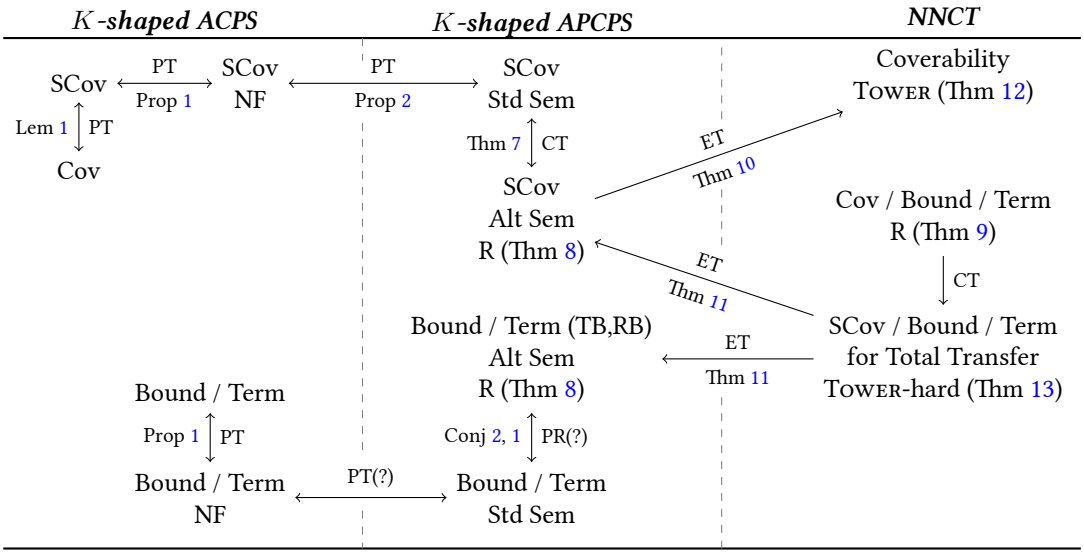


Figure 6.1: A map of the technical results presented in Chapters 4 and 5. "$P_1 \xrightarrow{\theta} P_2$" means that (decision problem) P_1 θ-reduces to P_2. *Legend: PT= polynomial time, CT= constant time, ET= exponential time, PR= primitive recursive time, R= decidable (recursive), NF= normal form, TB= termination bounded, RB= rewrite bounded, Std Sem= standard semantics, Alt Sem= alternative semantics, (?)= conjecture, Cov= coverability, SCov= simple coverability, Bound= boundedness, and Term= termination.*

quire further work. In order to apply Soter to real-world code bases, the support for Erlang features, such as I/O, arithmetic, built-in data types, dynamic type checks, receive timeouts, and fault-tolerance primitives needs to be improved. Furthermore, limitations of a theoretical nature are imposed by the expressivity of ACS. In particular, ACS cannot model recursive processes, higher-order control-flow, or name mobility.

ACPS and the Shaped Stack Constraint. In this dissertation, we have chosen to improve the precision of our verification procedure with respect to recursion. Thus, we ask the question: Can we find an extension of ACS that (i) has good algorithmic properties, (ii) can model (restricted) pushdown processes, and (iii) is expressive enough to model a wide range of recursive function definitions?

Such an extension of ACS would be a subclass of *asynchronously communicating pushdown systems* (ACPS). Recent work by Vardoulakis and Shivers [VS10] in a sequential setting has shown that the AAM methodology can be applied to yield a pushdown system while significantly improving the precision of the abstraction. Equipped with a subclass of ACPS that fulfils the above requirements, we believe it is possible to extend Vardoulakis and Shivers's work to λ_{Actor} and the concurrent setting. However, an expressive subclass of ACPS with good model-checking properties is, of course, of independent and wider interest.

We have conducted a theoretical study of ACPS and we present our findings in Chapters 4 and 5. Figure 6.1 displays a summary of the technical results that we have obtained. We have shown that we can simplify our reasoning on ACPS in two ways. Instead of the coverability

problem, we can consider *simple coverability* which restricts the form of coverability queries (Lemma 1). Further, we can restrict ourselves to ACPS in *normal form* for the purposes of deciding simple coverability, boundedness, and termination (Proposition 1). In the setting of unordered message passing, we make the observation that the relative order of some concurrency actions, such as message send and spawn, is immaterial and that we can classify concurrency actions as *commutative* or *non-commutative*. We formalise this sensitivity to order using an independence relation, lift it to stack-characters, and introduce the hybrid model *asynchronous partially commutative pushdown systems* (APCPS). One the one hand, APCPS induces a concurrent system which is suitable for the analysis of ACPS. On the other, APCPS give rise to processes that are modelled by a variant of context-free grammars which distinguish commutative and non-commutative concurrency actions which is less standard. However, we show that there is a polynomial-time interreduction between simple coverability for ACPS and APCPS (Proposition 2). Thus, we focus on APCPS and observe that it is possible to exploit commutativity to simplify the standard semantics of APCPS. This yields an alternative semantics for APCPS which precomputes summaries of commutative stack characters' concurrency effects. For the purposes of deciding simple coverability, the standard and alternative semantics of APCPS are equivalent (Theorem 7). The alternative semantics suggests a natural semantic restriction: *the shaped stack constraint* (Definition 21). The shaped stack constraint limits the number of non-commutative stack characters that may reside in the stack of any APCPS process to an *a priori* fixed bound while commutative stack characters are unconstrained. We find that the alternative semantics for an APCPS that satisfies the shaped stack constraint gives rise to a strict *well-structured transition system* (WSTS) [Fin87] and thus e.g. coverability is decidable (Theorem 8). Further, the shaped stack constraint can be recovered on ACPS in normal form (Proposition 2) and thus we can deduce that coverability is decidable for shaped ACPS, i.e ACPS that satisfy the shaped stack constraint. Hence, we find that shaped ACPS can model (restricted) pushdown processes, a wide range of recursive function definitions, and are significantly more expressive than other models that have been considered in the literature. In particular, shaped ACPS strictly subsume ACPS that satisfy the empty-stack constraint [SV06]. The latter are a popular decidable model that has emerged in the verification literature on *asynchronous programs*. More concretely, we show that

- coverability for ACPS polynomial time interreduces with simple coverability for APCPS,
- there is a weak bisimulation between the standard semantics and the co-universal powerset lifting of the alternative semantics of an APCPS and thus simple coverability queries are equivalent between the standard and alternative semantics,
- the alternative semantics for APCPS that satisfy the shaped stack constraint gives rise to an infinitely-branching strict post-effective WSTS that is well-partially-ordered and hence, coverability and boundedness is decidable, and
- there is an easy-to-check syntactic condition on ACPS/APCPS from which the shaped constraint can be deduced (Proposition 4).

Therefore, ACPS that satisfy the shaped stack constraint appear to be a suitable decidable model. However, the following question arises: What price do we incur for the expressivity provided by shaped ACPS in terms of higher worst-case complexity?

Nets with Nested Coloured Tokens. In order to find an answer to this question, we have decided to investigate the complexity properties of shaped ACPS. We present our results in Chapter 5. Encouraged by a connection between Petri nets and ACPS that satisfy the empty-stack constraint [GM12], we introduce nets with nested coloured tokens (NNCT), a novel extension of Petri nets. NNCT feature simple and complex tokens. Complex tokens carry coloured tokens which may be injected into and ejected from complex tokens by transitions of the net. NNCT are designed to capture the essential features that are required to implement the alternative semantics of shaped APCPS (Theorem 10). Further, shaped APCPS can simulate total transfer NNCT (Theorem 11), a subclass of NNCT. Equipped with these two results, we focus on a study of NNCT. We find that NNCT give rise to strict WSTS and thus coverability, boundedness, and termination are decidable (Theorem 9). In order to obtain precise complexity results, we turn to a proof technique originally used by Rackoff to establish Expspace membership of coverability for Petri nets [Rac78]. We rephrase Rackoff's argument using geometric intuition and adapt it to obtain Tower membership of NNCT coverability. Tower is a complexity class recently introduced by Schmitz for non-elementary decision problems [Sch13]. We show that this upper bound for NNCT coverability is asymptotically optimal by exhibiting a polynomial-time reduction from the parametrised halting problem of $2 \uparrow\uparrow n$ bounded counter machines to simple coverability, boundedness, and termination of total transfer NNCT. Our proof is based on a Stockmeyer/Lipton [Sto74; Lip76] construction. This result implies that NNCT coverability is Tower-complete and that boundedness and termination are Tower-hard. Moreover, this result allows us to deduce that coverability for shaped ACPS is Tower-complete. In summary, we establish that

- NNCT are strict WSTS, and thus coverability, boundedness, and termination are deciable,
- NNCT coverability is Tower-complete,
- boundedness and termination for NNCT are Tower-hard,
- to our knowledge, NNCT is the first extension of Petri nets—belonging to the class of nets with an infinite set of token types—that is proven to have primitive recursive coverability,
- simple coverability for shaped APCPS on the alternative semantics Exptime-reduces to NNCT coverability,
- simple coverability, boundedness, and termination for NNCT Exptime-reduces to simple coverability, boundedness, and termination for shaped APCPS,
- hence, coverability for shaped ACPS is Tower-complete.

6.1 Further Research Directions

There a number of practical and theoretical research directions that address the limitations of our verification procedure for Erlang and improve the scope of Soter.

Support for a Larger Fragment of Erlang in Soter. In order to make Soter applicable to real-word Erlang programs, it is necessary to improve the support for more of Erlang's language features. λActor does not capture the module system, exception handling, arithmetic primitives, built-in data types, and I/O. Further, Soter does not handle fault-tolerance features, process re-

gisters, type guards, and timeouts at present. We believe the design of our abstract interpretation can be adapted to support these features and thus make Soter applicable to real-world Erlang code more readily.

Extensions of ACS. Beside the lack of support for recursion, there are two other theoretical limitations of ACS that require us to use substantial control-flow abstractions. ACS are not able to model private mailboxes of processes nor higher-order control-flow. Recently, Meyer has isolated a rich decidable fragment of the π-calculus called *depth-bounded* [Mey08]. This fragment may provide a basis for an extension of ACS that can model the link between a process and its private mailbox as well as name mobility (see also Emanuele D'Osualdo's forthcoming DPhil thesis). Concerning higher-order control flow, a general result by Chadha and Viswanathan [CV07] implies that coverability for concurrent higher-order collapsible pushdown systems [Hag+08] that satisfy the empty-stack constraint is decidable. Higher-order collapsible pushdown systems are a decidable model for higher-order functional programs with finite-data and thus Chadha and Viswanathan's result can be used to obtain an extension of ACS that can model higher-order control-flow. We believe that it is an interesting research question to investigate whether this result on the higher-order case can be extended to the shaped stack constraint.

Boundedness and Termination There are a number of open theoretical questions with respect to the boundedness and termination problems for shaped ACPS and NNCT.

NNCT. We know that for NNCT boundedness and termination are decidable and Tower-hard. However, we leave the problem of establishing an upper bound for these two decision problems open. We believe that our version of Rackoff's argument for NNCT coverability can be adapted to obtain Tower upper bounds. A difficulty lies in establishing a Rackoff recurrence relation for paths that witness unboundedness or non-termination.

Shaped ACPS. If Tower upper bounds were obtained for these two decision problems, it would open up the opportunity to find a reduction of boundedness and termination from the alternative semantics of shaped APCPS to NNCT. For suitable subclasses of shaped APCPS we may be able to exploit the encoding of the alternative semantics into NNCT that we present in Section 5.2. Such subclasses need to ensure that the CCFG widget (Section 2.4, Section 5.2) does not introduce infinitely many reachable intermediate configurations or paths of infinite length. Rewrite-bounded and termination-bounded APCPS that satisfy the shaped stack constraint are candidate subclasses that may enable such a reduction. As we explain at the end of Section 4.3, the former two subclasses are also likely to be suitable subclasses for which the standard and alternative semantics are equivalent with respect to boundedness and termination (Conjectures 1 and 2). Since we can establish whether a shaped APCPS is rewrite-bounded or termination-bounded using simple static analysis and coverability, we believe that Tower-completeness for boundedness and termination on the standard semantics can be established for shaped APCPS provided that NNCT boundedness and termination can be shown to be Tower-complete (Conjecture 4).

We further believe that, equipped with Tower-completeness results for termination and boundedness for shaped APCPS, it is possible to show that boundedness and termination for shaped ACPS are Tower-complete. The missing link at present lies between shaped ACPS in

normal form and shaped APCPS. It may be possible to adapt the bisimulation result that underlies Proposition 2 to also obtain an interreduction of the boundedness and termination problems. This would yield a more complete picture of the decision and complexity properties for shaped ACPS.

Practical Algorithms for and Applications of Shaped ACPS. In our opinion, there are three further interesting research directions. We believe that the development of a verification algorithm for shaped ACPS that is practical, i.e. performs well on non-degenerate programs written by real programmers, is an important and non-trivial research problem. While the Tower complexity of coverability for shaped ACPS is high, we are encouraged to pursue this direction by recent advances in the design of *practical* algorithms for verification problems of comparable or higher worst-case complexity, such as *higher-order model checking* [RNO14] (Tower-complete) and *safety verification of concurrent C programs with broadcast* [KKW12] (Ack-hard). Another promising research direction is to extend our verification procedure for Erlang programs to yield shaped ACPS. This could be achieved by extending Vardoulakis and Shivers's CFA2 [VS10] to λ_{Actor} and abstracting degenerate non-tail recursive function definitions while letting shaped ACPS model benign ones. Another interesting application of shaped ACPS could be to extend decision and complexity results on programs with asynchronous atomic procedure calls [SV06], which can be modelled by ACPS that satisfy the empty stack constraint. We believe it is possible to weaken the atomicity requirement on asynchronous procedure calls using shaped ACPS. This could allow the modelling of asynchronous calls that return values asynchronously via *futures* or *promises*. Futures have been studied in the tail-recursive case [Boe+12] and in the setting of hierarchical communication [BE12a] in which processes cannot synchronise freely. We believe that a larger decidable class of asynchronous programs with futures could be captured by shaped ACPS.

Bibliography

[AJ93] Parosh Aziz Abdulla and Bengt Jonsson. 'Verifying Programs with Unreliable Channels'. In: *Proceedings of the 8th Annual Symposium on Logic in Computer Science (LICS 1993)*. IEEE Computer Society, 1993, pp. 160–170 (cited on page 27).

[AM09] Parosh Aziz Abdulla and Richard Mayr. 'Minimal Cost Reachability/Coverability in Priced Timed Petri Nets'. In: *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2009)*. Ed. by Luca de Alfaro. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 348–363. URL: http://dx.doi.org/10.1007/978-3-642-00596-1_25 (cited on page 25).

[Abd+96] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 'General Decidability Theorems for Infinite-State Systems'. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*. IEEE Computer Society, 1996, pp. 313–321. URL: http://doi.ieeecomputersociety.org/10.1109/LICS.1996.561359 (cited on page 26).

[Abd+00] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. 'Algorithmic Analysis of Programs with Well Quasi-ordered Domains'. In: *Information and Computation* 160.1-2 (2000), pp. 109–127 (cited on pages 26, 102).

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986 (cited on page 1).

[AGK14] Cyriac Aiswarya, Paul Gastin, and K. Narayan Kumar. 'Verifying Communicating Multi-pushdown Systems via Split-Width'. In: *Proceedings of the 12th International Symposium on Automated Technology for Verification and Analysis (ATVA 2014)*. Ed. by Franck Cassez and Jean-François Raskin. Vol. 8837. Lecture Notes in Computer Science. Springer, 2014, pp. 1–17. URL: http://dx.doi.org/10.1007/978-3-319-11936-6_1 (cited on page 111).

[AK76] Toshiro Araki and Tadao Kasami. 'Some Decision Problems Related to the Reachability Problem for Petri Nets'. In: *Theoretical Computer Science* 3.1 (1976), pp. 85–104 (cited on pages 23–25).

[Arm03] Joe Armstrong. 'Making Reliable Distributed Systems in the Presence of Software Errors'. PhD thesis. Royal Institute of Technology, 2003 (cited on page 38).

[Arm10] Joe Armstrong. 'Erlang'. In: *Communications of the ACM* 53.9 (2010), pp. 68–75. URL: http://doi.acm.org/10.1145/1810891.1810910 (cited on pages 1, 38).

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993, pp. 1–281 (cited on page 1).

[AD99] Thomas Arts and Mads Dam. 'Verifying a Distributed Database Lookup Manager Written in Erlang'. In: *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM 1999)*. Ed. by Jeannette M. Wing, Jim Woodcock, and Jim Davies. Vol. 1708. Lecture Notes in Computer Science. Springer, 1999, pp. 682–700. URL: http://dx.doi.org/10.1007/3-540-48119-2_38 (cited on pages 3, 71, 155).

[AEP04] Thomas Arts, Clara B. Earle, and Juan José Sánchez Penas. 'Translating Erlang to μCRL'. In: *Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD 2004)*. IEEE Computer Society, 2004, pp. 135–144. URL: http://doi.ieeecomputersociety.org/10.1109/CSD.2004.1309124 (cited on pages 3, 71, 155).

[AG99] Thomas Arts and Jürgen Giesl. 'Applying Rewriting Techniques to the Verification of Erlang Processes'. In: *Proceedings of the 8th EACSL Annual Conference on Computer Science Logic, (CSL 1999)*. Ed. by Jörg Flum and Mario Rodríguez-Artalejo. Vol. 1683. Lecture Notes in Computer Science. Springer, 1999, pp. 96–110. URL: http://dx.doi.org/10.1007/3-540-48168-0_8 (cited on page 71).

[Art+98] Thomas Arts, Mads Dam, Lars-Åke Fredlund, and Dilian Gurov. 'System Description: Verification of Distributed Erlang Programs'. In: *Proceedings of the 15th International Conference on Automated Deduction (CADE 1998)*. Ed. by Claude Kirchner and Hélène Kirchner. Vol. 1421. Lecture Notes in Computer Science. Springer, 1998, pp. 38–41 (cited on pages 3, 70, 155).

[ABH08] Mohamed Faouzi Atig, Benedikt Bollig, and Peter Habermehl. 'Emptiness of Multi-pushdown Automata Is 2ETIME-Complete'. In: *Proceedings of the 12th International Conference on Developments in Language Theory (DLT 2008)*. Ed. by Masami Ito and Masafumi Toyama. Vol. 5257. Lecture Notes in Computer Science. Springer, 2008, pp. 121–133. URL: http://dx.doi.org/10.1007/978-3-540-85780-8_9 (cited on page 111).

[AG11] Mohamed Faouzi Atig and Pierre Ganty. 'Approximating Petri Net Reachability Along Context-free Traces'. In: *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011)*. Ed. by Supratik Chakraborty and Amit Kumar. Vol. 13. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 152–163. URL: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2011.152 (cited on page 151).

[BR13] Domagoj Babic and Zvonimir Rakamaric. 'Asynchronously Communicating Visibly Pushdown Systems'. In: *Proceedings of the IFIP Joint International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE 2013)*. Ed. by Dirk Beyer and Michele Boreale. Vol. 7892. Lecture Notes in Computer Science. Springer,

2013, pp. 225–241. URL: http://dx.doi.org/10.1007/978-3-642-38592-6_16 (cited on pages 88, 110).

[BS11a] Thomas Ball and Mooly Sagiv, eds. *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011.

[BK84] Jan A. Bergstra and Jan Willem Klop. 'Process Algebra for Synchronous Communication'. In: *Information and Control* 60.1-3 (1984), pp. 109–137 (cited on pages 27, 112).

[Bes93] Eike Best, ed. *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*. Vol. 715. Lecture Notes in Computer Science. Springer, 1993.

[BS11b] Michel Blockelet and Sylvain Schmitz. 'Model Checking Coverability Graphs of Vector Addition Systems'. In: *Proceedings of the 36th International Symposium on Mathematical Foundations of Computer Science (MFCS 2011)*. Ed. by Filip Murlak and Piotr Sankowski. Vol. 6907. Lecture Notes in Computer Science. Springer, 2011, pp. 108–119 (cited on pages 24, 25, 135, 139, 150).

[BFM14] Michael Blondin, Alain Finkel, and Pierre McKenzie. 'Handling Infinitely Branching WSTS'. In: *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP 2014)*. Ed. by Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias. Vol. 8573. Lecture Notes in Computer Science. Springer, 2014, pp. 13–25. URL: http://dx.doi.org/10.1007/978-3-662-43951-7_2 (cited on page 102).

[Boe+12] Frank S. de Boer, Mario Bravetti, Immo Grabe, Matias David Lee, Martin Steffen, and Gianluigi Zavattaro. 'A Petri Net Based Analysis of Deadlocks for Active Objects and Futures'. In: *Proceedings of the 9th International Symposium on Formal Aspects of Component Software (FACS 2012)*. Ed. by Corina S. Pasareanu and Gwen Salaün. Vol. 7684. Lecture Notes in Computer Science. Springer, 2012, pp. 110–127 (cited on pages 113, 160).

[Bon11a] Rémi Bonnet. 'Decidability of LTL for Vector Addition Systems with One Zero-Test'. In: *Proceedings of the 5th International Workshop on Reachability Problems (RP 2011)*. Ed. by Giorgio Delzanno and Igor Potapov. Vol. 6945. Lecture Notes in Computer Science. Springer, 2011, pp. 85–95 (cited on page 25).

[Bon11b] Rémi Bonnet. 'The Reachability Problem for Vector Addition System with One Zero-Test'. In: *Proceedings of the 36th International Symposium on Mathematical Foundations of Computer Science (MFCS 2011)*. Ed. by Filip Murlak and Piotr Sankowski. Vol. 6907. Lecture Notes in Computer Science. Springer, 2011, pp. 145–157 (cited on page 25).

[BFP12] Rémi Bonnet, Alain Finkel, and M. Praveen. 'Extending the Rackoff technique to Affine nets'. In: *Proceedings of the 32nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'12)*. Ed. by Deepak D'Souza, Jaikumar Radhakrishnan, and Kavitha Telikepalli. Leibniz International Proceedings in Informatics. Hyderabad, India: Leibniz-Zentrum für Informatik, Dec. 2012 (cited on pages 25, 135, 139, 140, 144, 151, 268).

[Bon+10] Rémi Bonnet, Alain Finkel, Jérôme Leroux, and Marc Zeitoun. 'Place-Boundedness for Vector Addition Systems with One Zero-Test'. In: *FSTTCS*. Ed. by Kamal Lodaya and Meena Mahajan. Vol. 8. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010, pp. 192–203 (cited on page 25).

[BE12a] Ahmed Bouajjani and Michael Emmi. 'Analysis of Recursively Parallel Programs'. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 203–214 (cited on pages 110, 113, 160).

[BE12b] Ahmed Bouajjani and Michael Emmi. 'Bounded Phase Analysis of Message-Passing Programs'. In: *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*. Ed. by Cormac Flanagan and Barbara König. Vol. 7214. Lecture Notes in Computer Science. Springer, 2012, pp. 451–465 (cited on pages 87, 111).

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 'Reachability Analysis of Pushdown Automata: Application to Model-Checking'. In: *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR 1997)*. Ed. by Antoni W. Mazurkiewicz and Józef Winkowski. Vol. 1243. Lecture Notes in Computer Science. Springer, 1997, pp. 135–150 (cited on page 28).

[BM99] Ahmed Bouajjani and Richard Mayr. 'Model Checking Lossy Vector Addition Systems'. In: *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1999)*. Ed. by Christoph Meinel and Sophie Tison. Vol. 1563. Lecture Notes in Computer Science. Springer, 1999, pp. 323–333 (cited on page 27).

[BMOT05] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. 'Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems'. In: *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR 2005)*. Ed. by Martín Abadi and Luca de Alfaro. Vol. 3653. Lecture Notes in Computer Science. Springer, 2005, pp. 473–487 (cited on pages 86, 110).

[BZ83] Daniel Brand and Pitro Zafiropulo. 'On Communicating Finite-State Machines'. In: *Journal of the ACM (JACM)* 30.2 (1983), pp. 323–342 (cited on pages 2, 11, 27).

[Büc64] Richard J. Büchi. 'Regular Canonical Systems'. In: *Archiv für mathematische Logik und Grundlagenforschung* 6.3-4 (1964), pp. 91–111. URL: http://dx.doi.org/10.1007/BF01969548 (cited on page 28).

[Cad] *CADP*. URL: http://www.inrialpes.fr/vasy/cadp/ (cited on page 71).

[CO13] Xiaojuan Cai and Mizuhito Ogawa. 'Well-Structured Pushdown Systems'. In: *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR 2013)*. Ed. by Pedro R. D'Argenio and Hernán C. Melgratti. Vol. 8052. Lecture Notes in Computer Science. Springer, 2013, pp. 121–136 (cited on page 110).

[Car01] Richard Carlsson. 'An Introduction to Core Erlang'. In: *Proceedings of the 2001 Workshop on Erlang (Erlang Workshop 2001)*. 2001 (cited on pages 2, 6).

[Cau88] Didier Caucal. *Récritures suffixes de mots*. Rapport de recherche RR-0871. INRIA, 1988. URL: http://hal.inria.fr/inria-00075683 (cited on page 28).

[CT09] Francesca Cesarini and Simon Thompson. *Erlang Programming - A Concurrent Approach to Software Development*. O'Reilly, 2009. URL: http://www.oreilly.de/catalog/9780596518189/index.html (cited on page 1).

[CV07] Rohit Chadha and Mahesh Viswanathan. 'Decidability Results for Well-Structured Transition Systems with Auxiliary Storage'. In: *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR 2007)*. Ed. by Luís Caires and Vasco Thudichum Vasconcelos. Vol. 4703. Lecture Notes in Computer Science. Springer, 2007, pp. 136–150 (cited on pages 76, 110, 159).

[CS10] Maria Christakis and Konstantinos F. Sagonas. 'Static Detection of Race Conditions in Erlang'. In: *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*. Ed. by Manuel Carro and Ricardo Peña. Vol. 5937. Lecture Notes in Computer Science. Springer, 2010, pp. 119–133. URL: http://dx.doi.org/10.1007/978-3-642-11503-5_11 (cited on page 72).

[CS11] Maria Christakis and Konstantinos F. Sagonas. 'Detection of Asynchronous Message Passing Errors Using Static Analysis'. In: *Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages (PADL 2011)*. Ed. by Ricardo Rocha and John Launchbury. Vol. 6539. Lecture Notes in Computer Science. Springer, 2011, pp. 5–18. URL: http://dx.doi.org/10.1007/978-3-642-18378-2_3 (cited on page 72).

[CHM93] Søren Christensen, Yoram Hirshfeld, and Faron Moller. 'Bisimulation Equivalence is Decidable for Basic Parallel Processes'. In: *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR 1993)*. Ed. by Eike Best. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 143–157 (cited on page 27).

[Cia94] Gianfranco Ciardo. 'Petri Nets with Marking-Dependent Ar Cardinality: Properties and Analysis'. In: *Proceedings of the 15th International Conference on Application and Theory of Petri Nets (APN 1994)*. Ed. by Robert Valette. Vol. 815. Lecture Notes in Computer Science. Springer, 1994, pp. 179–198 (cited on pages 23, 24, 115).

[CS05] Koen Claessen and Hans Svensson. 'A Semantics for Distributed Erlang'. In: *Proceedings of the ACM SIGPLAN Workshop on Erlang (Erlang Workshop 2005)*. Ed. by Konstantinos F. Sagonas and Joe Armstrong. ACM, 2005, pp. 78–87. URL: http://doi.acm.org/10.1145/1088361.1088376 (cited on pages 38, 71).

[Col95] Christopher Colby. 'Analyzing the Communication Topology of Concurrent Programs'. In: *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 1995)*. Ed. by Neil D. Jones. ACM Press, 1995, pp. 202–213. URL: http://doi.acm.org/10.1145/215465.215592 (cited on page 73).

[CMO14] Conrad Cotton-Barratt, Andrzej S. Murawski, and C.-H. Luke Ong. 'Weak and Nested Class Memory Automata'. In: *CoRR* abs/1409.1136 (2014). URL: http://arxiv.org/abs/1409.1136 (cited on page 27).

[CC77] Patrick Cousot and Radhia Cousot. 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'. In: *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages (POPL 1977)*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252 (cited on pages 3, 35, 42, 43, 155).

[CC79] Patrick Cousot and Radhia Cousot. 'Systematic Design of Program Analysis Frameworks'. In: *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages (POPL 1979)*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen. ACM Press, 1979, pp. 269–282 (cited on page 44).

[CGK12] Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. 'MSO Decidability of Multi-Pushdown Systems via Split-Width'. In: *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR 2012)*. Ed. by Maciej Koutny and Irek Ulidowski. Vol. 7454. Lecture Notes in Computer Science. Springer, 2012, pp. 547–561. URL: http://dx.doi.org/10.1007/978-3-642-32940-1_38 (cited on page 111).

[CFL09] Wojciech Czerwinski, Sibylle B. Fröschle, and Slawomir Lasota. 'Partially-Commutative Context-Free Processes'. In: *Proceedings of the 20th International Conference on Concurrency Theory (CONCUR 2009)*. Ed. by Mario Bravetti and Gianluigi Zavattaro. Vol. 5710. Lecture Notes in Computer Science. Springer, 2009, pp. 259–273 (cited on pages 34, 92, 112).

[CHL12] Wojciech Czerwinski, Piotr Hofman, and Slawomir Lasota. 'Reachability Problem for Weak Multi-Pushdown Automata'. In: *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR 2012)*. Ed. by Maciej Koutny and Irek Ulidowski. Vol. 7454. Lecture Notes in Computer Science. Springer, 2012, pp. 53–68 (cited on page 112).

[CL12] Wojciech Czerwinski and Slawomir Lasota. 'Partially-Commutative Context-Free Languages'. In: *Proceedings of the Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics (EXPRESS/SOS 2012)*. Ed. by Bas Luttik and Michel A. Reniers. Vol. 89. EPTCS. 2012, pp. 35–48 (cited on page 112).

[DM13] Pedro R. D'Argenio and Hernán C. Melgratti, eds. *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*. Vol. 8052. Lecture Notes in Computer Science. Springer, 2013.

[DKO13] Emanuele D'Osualdo, Jonathan Kochems, and C.-H. Luke Ong. 'Automatic Verification of Erlang-Style Concurrency'. In: *Proceedings of the 20th International Symposium on Static Analysis (SAS 2013)*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 454–476 (cited on pages 35, 113).

[DKO12] Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. 'Soter: An Automatic Safety Verifier for Erlang'. In: *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions (AGERE! 2012)*. Ed. by Gul A. Agha, Rafael H. Bordini, Assaf Marron, and Alessandro Ricci. ACM, 2012, pp. 137–140. URL: http://doi.acm.org/10.1145/2414639.2414658 (cited on pages 35, 113).

[Dem10] Stéphane Demri. 'On Selective Unboundedness of VASS'. In: *Proceedings of the 12th International Workshop on Verification of Infinite-State Systems (INFINITY 2010)*. Ed. by Yu-Fang Chen and Ahmed Rezine. Vol. 39. EPTCS. 2010, pp. 1–15. URL: http://dx.doi.org/10.4204/EPTCS.39.1 (cited on pages 11, 150).

[Dem13] Stéphane Demri. 'On Selective Unboundedness of VASS'. In: *Journal of Computer and System Sciences* 79.5 (2013), pp. 689–713 (cited on pages 24, 135, 139, 150).

[Dem+09] Stéphane Demri, Marcin Jurdzinski, Oded Lachish, and Ranko Lazic. 'The Covering and Boundedness Problems for Branching Vector Addition Systems'. In: *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009)*. Ed. by Ravi Kannan and K. Narayan Kumar. Vol. 4. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009, pp. 181–192 (cited on pages 110, 135, 139, 151).

[Döm00] Pál Dömösi. 'Unusual Algorithms for Lexicographical Enumeration'. In: *Acta Cybern.* 14.3 (2000), pp. 461–468 (cited on page 104).

[DFS98] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. 'Reset Nets Between Decidability and Undecidability'. In: *Proceedings of the 25th International Colloquium Automata, Languages and Programming (ICALP 1998)*. Ed. by Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel. Vol. 1443. Lecture Notes in Computer Science. Springer, 1998, pp. 103–115 (cited on pages 25, 149).

[DKO11] Emanuele D'Osualdo, Jonathan Kochems, and C-H Luke Ong. *Verifying Erlang-Style Concurrency Automatically*. Tech. rep. Department of Computer Science, University of Oxford, 2011. URL: http://mjolnir.cs.ox.ac.uk/soter/cpmrs.pdf (cited on page 11).

[EMH10] Christopher Earl, Matthew Might, and David Van Horn. 'Pushdown Control-Flow Analysis of Higher-Order Programs'. In: *CoRR* abs/1007.4268 (2010) (cited on page 73).

[Ear+12] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 'Introspective Pushdown Analysis of Higher-Order Programs'. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*. Ed. by Peter Thiemann and Robby Bruce Findler. ACM, 2012, pp. 177–188 (cited on page 73).

[EF12] Clara B. Earle and Lars-Åke Fredlund. 'Verification of Timed Erlang Programs Using McErlang'. In: *Proceedings of the IFIP Joint International Conference on Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*. Ed. by Holger Giese and Grigore Rosu. Vol. 7273. Lecture Notes in Computer Science. Springer, 2012, pp. 251–267 (cited on pages 71, 72).

[EGM14] Michael Emmi, Pierre Ganty, and Rupak Majumdar. 'An Expressive yet Decidable Model of Asynchronous Event-Driven Programs'. Preprint. 2014 (cited on page 110).

[ELQ12] Michael Emmi, Akash Lal, and Shaz Qadeer. 'Asynchronous Programs with Prioritized Task-Buffers'. In: *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT FSE 2012)*. Ed. by Will Tracz, Martin P. Robillard, and Tevfik Bultan. ACM, 2012, p. 48 (cited on page 109).

[EOT14] Michael Emmi, Burcu Kulahcioglu Ozkan, and Serdar Tasiran. 'Exploiting Synchronization in the Analysis of Shared-Memory Asynchronous Programs'. In: *Proceedings of the 2014 International Symposium on Model Checking of Software (SPIN 2014)*. Ed. by Neha Rungta and Oksana Tkachuk. ACM, 2014, pp. 20–29. URL: http://doi.acm.org/10.1145/2632362.2632370 (cited on page 109).

[Esp94] Javier Esparza. 'On the Decidability of Model Checking for Several μ-calculi and Petri Nets'. In: *Proceedings of the 19th International Colloquium on Trees in Algebra and Programming (CAAP 1994)*. Ed. by Sophie Tison. Vol. 787. Lecture Notes in Computer Science. Springer, 1994, pp. 115–129 (cited on page 24).

[Esp95] Javier Esparza. 'Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes'. In: *Proceedings of the 10th International Symposium on Fundamentals of Computation Theory (FCT 1995)*. Ed. by Horst Reichel. Vol. 965. Lecture Notes in Computer Science. Springer, 1995, pp. 221–232 (cited on pages 31–33, 129).

[Esp97] Javier Esparza. 'Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes'. In: *Fundamenta Informaticae* 31.1 (1997), pp. 13–25 (cited on pages 27, 104, 109, 112, 121, 152, 231).

[Esp98] Javier Esparza. 'Decidability and Complexity of Petri Net Problems — An Introduction'. In: *Lectures on Petri Nets I: Basic Models*. Ed. by Wolfgang Reisig and Grzegorz Rozenberg. Vol. 1491. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 374–428 (cited on pages 23, 109, 145, 151).

[EG11] Javier Esparza and Pierre Ganty. 'Complexity of Pattern-Based Verification for Multithreaded Programs'. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 499–510 (cited on pages 86, 111).

[EGP14] Javier Esparza, Pierre Ganty, and Tomáš Poch. 'Pattern-Based Verification for Multithreaded Programs'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36.3 (Sept. 2014), 9:1–9:29. URL: http://doi.acm.org/10.1145/2629644 (cited on page 111).

[EN94] Javier Esparza and Mogens Nielsen. 'Decidability Issues for Petri Nets - a Survey'. In: *Bulletin of the EATCS* 52 (1994), pp. 244–262 (cited on page 24).

[EP00] Javier Esparza and Andreas Podelski. 'Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs'. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*. Ed. by Mark N. Wegman and Thomas W. Reps. ACM, 2000, pp. 1–11 (cited on page 110).

[Esp+11] Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. 'Parikhs Theorem: A Simple and Direct Automaton Construction'. In: *Information Processing Letters* 111.12 (2011), pp. 614–619 (cited on pages 30–33, 104, 109, 121, 231).

[Esp+14] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp Meyer, and Filip Niksic. 'An SMT-Based Approach to Coverability Analysis'. In: *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*. Ed. by Armin Biere and Roderick Bloem. Springer, 2014, pp. 603–619. URL: http://dx.doi.org/10.1007/978-3-319-08867-9_40 (cited on page 74).

[FF87] Matthias Felleisen and Daniel P. Friedman. 'A Calculus for Assignments in Higher-Order Languages'. In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (POPL 1987)*. ACM Press, 1987, pp. 314–325 (cited on page 42).

[Fer05] Jérôme Feret. 'Abstract Interpretation of Mobile Systems'. In: *Journal of Logic and Algebraic Programming* 63.1 (2005), 59–130 (cited on pages 73, 74).

[Fig+11] Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. 'Ackermannian and Primitive-Recursive Bounds with Dickson's Lemma'. In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science (LICS 2011)*. IEEE Computer Society, 2011, pp. 269–278 (cited on pages 25, 149).

[Fin87] Alain Finkel. 'A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems'. In: *Proceedings of the 14th International Colloquium on Automata, Languages and Programming (ICALP 1987)*. Ed. by Thomas Ottmann. Vol. 267. Lecture Notes in Computer Science. Springer, 1987, pp. 499–508 (cited on pages 11, 26, 157).

[Fin90] Alain Finkel. 'Reduction and Covering of Infinite Reachability Trees'. In: *Information and Computation* 89.2 (1990), pp. 144–179 (cited on page 25).

[FMP04] Alain Finkel, Pierre McKenzie, and Claudine Picaronny. 'A Well-Structured Framework for Analysing Petri Net Extensions'. In: *Information and Computation* 195.1-2 (2004), pp. 1–29 (cited on pages 23, 24, 140, 149).

[FS08] Alain Finkel and Arnaud Sangnier. 'Reversal-Bounded Counter Machines Revisited'. In: *Proceedings of the 33rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2008)*. Ed. by Edward Ochmanski and Jerzy Tyszkiewicz. Vol. 5162. Lecture Notes in Computer Science. Springer, 2008, pp. 323–334 (cited on page 25).

[FS98] Alain Finkel and Philippe Schnoebelen. 'Fundamental Structures in Well-Structured Infinite Transition Systems'. In: *Proceedings of the Third Latin American Symposium on Theoretical Informatics (LATIN 1998)*. Ed. by Claudio L. Lucchesi and Arnaldo V. Moura. Vol. 1380. Lecture Notes in Computer Science. Springer, 1998, pp. 102–118 (cited on page 26).

[FS01] Alain Finkel and Philippe Schnoebelen. 'Well-Structured Transition Systems Everywhere!' In: *Theoretical Computer Science* 256.1-2 (2001), pp. 63–92 (cited on pages 4, 26, 27, 102, 124, 125).

[FQ03] Cormac Flanagan and Shaz Qadeer. 'Thread-Modular Model Checking'. In: *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN 2003)*. Ed. by Thomas Ball and Sriram K. Rajamani. Vol. 2648. Lecture Notes in Computer Science. Springer, 2003, pp. 213–224 (cited on page 110).

[Fre01] Lars-Åke Fredlund. 'A Framework for Reasoning about Erlang Code'. PhD thesis. Stockholm, Sweden: Royal Institute of Technology, 2001 (cited on pages 37, 71, 73).

[FS07] Lars-Åke Fredlund and Hans Svensson. 'McErlang: A Model Checker for a Distributed Functional Programming Language'. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*. Ed. by Ralf Hinze and Norman Ramsey. ACM, 2007, pp. 125–136. URL: http://doi.acm.org/10.1145/1291151.1291171 (cited on pages 3, 8, 37, 38, 71, 155).

[GM12] Pierre Ganty and Rupak Majumdar. 'Algorithmic Verification of Asynchronous Programs'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34.1 (2012), p. 6 (cited on pages 5, 31–34, 76, 80, 81, 86, 104, 109, 115, 121, 125, 127–130, 152, 158, 231, 232, 236, 239).

[GMR09] Pierre Ganty, Rupak Majumdar, and Andrey Rybalchenko. 'Verifying Liveness for Asynchronous Programs'. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 102–113 (cited on pages 5, 80, 86, 109).

[Gan+07] Pierre Ganty, Cédric Meuter, Giorgio Delzanno, Gabriel Kalyon, Jean-François Raskin, and Laurent Van Begin. *Symbolic Data Structure for Sets of k-uples*. Tech. rep. 570. Université Libre de Bruxelles, 2007. URL: https://github.com/pierreganty/mist (cited on page 39).

[GPT06] Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. 'Static Safety for an Actor Dedicated Process Calculus by Abstract Interpretation'. In: *Proceedings of the 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*. Ed. by Roberto Gorrieri and Heike Wehrheim. Vol. 4037. Lecture Notes in Computer Science. Springer, 2006, pp. 78–92. URL: http://dx.doi.org/10.1007/11768869_8 (cited on page 73).

[GHR13] Gilles Geeraerts, Alexander Heußner, and Jean-François Raskin. 'Queue-Dispatch Asynchronous Systems'. In: *Proceedings of the 13th International Conference on Application of Concurrency to System Design (ACSD 2013)*. IEEE, 2013, pp. 150–159 (cited on page 109).

[GRB04] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. 'Expand, Enlarge, and Check: New Algorithms for the Coverability Problem of WSTS'. In: *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2004)*. Ed. by Kamal Lodaya and Meena Mahajan. Vol. 3328. Lecture Notes in Computer Science. Springer, 2004, pp. 287–298. URL: http://dx.doi.org/10.1007/978-3-540-30538-5_24 (cited on page 109).

[GA01] Jürgen Giesl and Thomas Arts. 'Verification of Erlang Processes by Dependency Pairs'. In: *Applicable Algebra in Engineering, Communication and Computing* 12.1/2 (2001), pp. 39–72 (cited on page 71).

[GD07] Qiang Guo and John Derrick. 'Verification of Timed Erlang/OTP Components Using the Process Algebra μCRL'. In: *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang (Erlang Workshop 2007)*. Ed. by Simon J. Thompson and Lars-Åke Fredlund. ACM, 2007, pp. 55–64 (cited on page 71).

[GDH08] Qiang Guo, John Derrick, and Csaba Hoch. 'Verifying Erlang Telecommunication Systems with the Process Algebra μCRL'. In: *Proceedings of the 28th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2008)*. Ed. by Kenji Suzuki, Teruo Higashino, Keiichi Yasumoto, and Khaled El-Fakih. Vol. 5048. Lecture Notes in Computer Science. Springer, 2008, pp. 201–217 (cited on page 71).

[Guo+10] Qiang Guo, John Derrick, Clara B. Earle, and Lars-Åke Fredlund. 'Model-Checking Erlang - A Comparison between EtomCRL2 and McErlang'. In: *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques (TAIC PART 2010)*. Ed. by Leonardo Bottaci and Gordon Fraser. Vol. 6303. Lecture Notes in Computer Science. Springer, 2010, pp. 23–38 (cited on page 71).

[Hab97] Peter Habermehl. 'On the Complexity of the Linear-Time μ-calculus for Petri-Nets'. In: *Proceedings of the 18th International Conference on Application and Theory of Petri Nets (ICATPN 1997)*. Ed. by Pierre Azéma and Gianfranco Balbo. Vol. 1248. Lecture Notes in Computer Science. Springer, 1997, pp. 102–116 (cited on page 24).

[HSS12] Serge Haddad, Sylvain Schmitz, and Philippe Schnoebelen. 'The Ordinal-Recursive Complexity of Timed-arc Petri Nets, Data Nets, and Other Enriched Nets'. In: *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science (LICS 2012)*. IEEE, 2012, pp. 355–364 (cited on page 150).

[HL11] Matthew Hague and Anthony W. Lin. 'Model Checking Recursive Programs with Numeric Data Types'. In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 743–759. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_60 (cited on page 111).

[HL12] Matthew Hague and Anthony W. Lin. 'Synchronisation- and Reversal-Bounded Analysis of Multithreaded Programs with Counters'. In: *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 260–276. URL: http://dx.doi.org/10.1007/978-3-642-31424-7_22 (cited on page 111).

[Hag+08] Matthew Hague, Andrzej S. Murawski, C.-H. Luke Ong, and Olivier Serre. 'Collapsible Pushdown Automata and Recursion Schemes'. In: *Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*. IEEE Computer Society, 2008, pp. 452–461 (cited on pages 29, 159).

[HJ94] Nevin Heintze and Joxan Jaffar. 'Set Constraints and Set-Based Analysis'. In: *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming (PPCP 1994)*. Ed. by Alan Borning. Vol. 874. Lecture Notes in Computer Science. Springer, 1994, pp. 281–298 (cited on pages 47, 72).

[Hen+02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 'Lazy Abstraction'. In: *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*. Ed. by John Launchbury and John C. Mitchell. ACM, 2002, pp. 58–70. URL: http://doi.acm.org/10.1145/503272.503279 (cited on page 109).

[Hen+03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. 'Thread-Modular Abstraction Refinement'. In: *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 262–274 (cited on page 110).

[Her+02] Frédéric Herbreteau, Franck Cassez, Alain Finkel, Olivier Roux, and Grégoire Sutre. 'Verification of Embedded Reactive Fiffo Systems'. In: *Proceedings of the 5th Latin American Symposium on Theoretical Informatics (LATIN 2002)*. Ed. by Sergio Rajsbaum. Vol. 2286. Lecture Notes in Computer Science. Springer, 2002, pp. 400–414 (cited on pages 11, 36).

[Heu12] Alexander Heußner. 'Model Checking Communicating Processes: Run Graphs, Graph Grammars, and MSO'. In: *ECEASST* 47 (2012). URL: http://journal.ub.tu-berlin.de/eceasst/article/view/725 (cited on page 111).

[Heu+10] Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. 'Reachability Analysis of Communicating Pushdown Systems'. In: *Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2010)*. Ed. by C.-H. Luke Ong. Vol. 6014. Lecture Notes in Computer Science. Springer, 2010, pp. 267–281 (cited on pages 86, 88, 110).

[Hoa78] C. A. R. Hoare. 'Communicating Sequential Processes'. In: *Communications of the ACM* 21.8 (1978), pp. 666–677 (cited on page 27).

[HP79] John E. Hopcroft and Jean-Jacques Pansiot. 'On the Reachability Problem for 5-Dimensional Vector Addition Systems'. In: *Theoretical Computer Science* 8 (1979), pp. 135–159 (cited on page 25).

[HM10] David Van Horn and Matthew Might. 'Abstracting Abstract Machines'. In: *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*. Ed. by Paul Hudak and Stephanie Weirich. ACM, 2010, pp. 51–62. URL: http://doi.acm.org/10.1145/1863543.1863553 (cited on page 62).

[HM11] David Van Horn and Matthew Might. 'Abstracting Abstract Machines: a Systematic Approach to Higher-Order Program Analysis'. In: *Communications of the ACM* 54.9 (2011), pp. 101–109 (cited on pages 3, 12, 35, 39, 73, 155).

[HM12] David Van Horn and Matthew Might. 'Systematic Abstraction of Abstract Machines'. In: *Journal of Functional Programming* 22.4-5 (2012), pp. 705–746 (cited on pages 3, 12, 35, 39, 42, 155).

[HRY91] Rodney R. Howell, Louis E. Rosier, and Hsu-Chun Yen. 'A Taxonomy of Fairness and Temporal Logic Problems for Petri Nets'. In: *Theoretical Computer Science* 82.2 (1991), pp. 341–372 (cited on page 24).

[Huc99] Frank Huch. 'Verification of Erlang Programs using Abstract Interpretation and Model Checking'. In: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP 1999)*. Ed. by Didier Rémi and Peter Lee. ACM, 1999, pp. 261–272. URL: http://doi.acm.org/10.1145/317636.317908 (cited on pages 2, 3, 71, 74, 155).

[Huc01] Frank Huch. 'Model Checking Erlang Programs - Abstracting the Context-Free Structure'. In: *Electronic Notes in Theoretical Computer Science* 55.3 (2001), pp. 304–321 (cited on page 71).

[Huc02] Frank Huch. 'Model Checking Erlang Programs - Abstracting Recursive Function Calls'. In: *Electronic Notes in Theoretical Computer Science* 64 (2002), pp. 195–219. URL: http://dx.doi.org/10.1016/S1571-0661(04)80351-7 (cited on page 71).

[Iba78] Oscar H. Ibarra. 'Reversal-Bounded Multicounter Machines and Their Decision Problems'. In: *Journal of the ACM (JACM)* 25.1 (1978), pp. 116–133. URL: http://doi.acm.org/10.1145/322047.322058 (cited on page 111).

[JW95] Suresh Jagannathan and Stephen Weeks. 'A Unified Treatment of Flow Analysis in Higher-Order Languages'. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1995)*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 393–407 (cited on page 47).

[JM07] Ranjit Jhala and Rupak Majumdar. 'Interprocedural Analysis of Asynchronous Programs'. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, 2007, pp. 339–350 (cited on pages 80, 86, 109).

[Joh+13] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 'Optimizing Abstract Abstract Machines'. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 443–454 (cited on pages 35, 39, 63, 65, 74).

[Jon81] Neil D. Jones. 'Flow Analysis of Lambda Expressions (Preliminary Version)'. In: *Proceedings of the 8th Colloquium on Automata, Languages and Programming (ICALP 1981)*. Ed. by Shimon Even and Oded Kariv. Vol. 115. Lecture Notes in Computer Science. Springer, 1981, pp. 114–128 (cited on page 47).

[JA07] Neil D. Jones and Nils Andersen. 'Flow Analysis of Lazy Higher-Order Functional Programs'. In: *Theoretical Computer Science* 375.1-3 (2007), pp. 120–136 (cited on page 54).

[Kah09a] Vineet Kahlon. 'Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of Pairwise CFL-Reachability for Threads Communicating via Locks'. In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science (LICS 2009)*. IEEE Computer Society, 2009, pp. 27–36 (cited on pages 86, 110).

[Kah09b] Vineet Kahlon. *Tractable Dataflow Analysis for Concurrent Programs via Bounded Languages*. Patent WO/2009/094439. July 2009. URL: https://www.google.com/patents/WO2009094439A1?cl=en (cited on page 111).

[KKW12] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 'Efficient Coverability Analysis by Proof Minimization'. In: *Proceedings of the 23rd International Conference on Concurrency Theory (CONCUR 2012)*. Ed. by Maciej Koutny and Irek Ulidowski. Vol. 7454. Lecture Notes in Computer Science. Springer, 2012, pp. 500–515 (cited on pages 5, 39, 61, 160).

[KM69] Richard M. Karp and Raymond E. Miller. 'Parallel Program Schemata'. In: *Journal of Computer and System Sciences* 3.2 (1969), pp. 147–195. URL: http://dx.doi.org/10.1016/S0022-0000(69)80011-5 (cited on pages 11, 24, 25).

[KK11] Joost-Pieter Katoen and Barbara König, eds. *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*. Vol. 6901. Lecture Notes in Computer Science. Springer, 2011.

[Kob97] Naoki Kobayashi. 'A Partially Deadlock-Free Typed Process Calculus'. In: *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS 1997)*. IEEE Computer Society, 1997, pp. 128–139 (cited on page 73).

[Kob00] Naoki Kobayashi. 'Type Systems for Concurrent Processes: From Deadlock-Freedom to Livelock-Freedom, Time-Boundedness'. In: *Proceedings of the IFIP International Conference on Theoretical Computer Science (IFIP TCS 2000)*. Ed. by Jan van Leeuwen, Osamu Watanabe, Masami Hagiya, Peter D. Mosses, and Takayasu Ito. Vol. 1872. Lecture Notes in Computer Science. Springer, 2000, pp. 365–389 (cited on page 73).

[Kob06] Naoki Kobayashi. 'A New Type System for Deadlock-Free Processes'. In: *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR 2006)*. Ed. by Christel Baier and Holger Hermanns. Vol. 4137. Lecture Notes in Computer Science. Springer, 2006, pp. 233–247 (cited on page 73).

[KNY95] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. 'Static Analysis of Communication for Asynchronous Concurrent Programming Languages'. In: *Proceedings of the 2nd International Symposium on Static Analysis (SAS 1995)*. Ed. by Alan Mycroft. Springer, 1995, pp. 225–242. URL: http://dx.doi.org/10.1007/3-540-60360-3_42 (cited on page 74).

[KO13] Jonathan Kochems and C.-H. Luke Ong. 'Safety Verification of Asynchronous Pushdown Systems with Shaped Stacks'. In: *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR 2013)*. Ed. by Pedro R. D'Argenio and Hernán C. Melgratti. Vol. 8052. Lecture Notes in Computer Science. Springer, 2013, pp. 288–302 (cited on page 79).

[KO14] Jonathan Kochems and C.-H. Luke Ong. 'Decidable Models of Recursive Asynchronous Concurrency'. In: *CoRR* abs/1410.8852 (2014). URL: http://arxiv.org/abs/1410.8852 (cited on page 115).

[Kos82] S. Rao Kosaraju. 'Decidability of Reachability in Vector Addition Systems (Preliminary Version)'. In: *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC 1982)*. Ed. by Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber. ACM, 1982, pp. 267–281 (cited on page 24).

[KU12] Maciej Koutny and Irek Ulidowski, eds. *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*. Vol. 7454. Lecture Notes in Computer Science. Springer, 2012.

[Lam77] Leslie Lamport. 'Proving the Correctness of Multiprocess Programs'. In: *IEEE Transactions on Software Engineering* 3.2 (1977), pp. 125–143 (cited on page 10).

[LSW98] Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, eds. *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. Vol. 1443. Lecture Notes in Computer Science. Springer, 1998.

[Lar09] Jim Larson. 'Erlang for Concurrent Programming'. In: *Communications of the ACM* 52.3 (2009), pp. 48–56 (cited on page 1).

[Laz13] Ranko Lazic. 'The Reachability Problem for Vector Addition Systems with a Stack Is Not Elementary'. In: *CoRR* abs/1310.1767 (2013) (cited on pages 145, 151).

[LS14] Ranko Lazic and Sylvain Schmitz. 'Non-Elementary Complexities for Branching VASS, MELL, and Extensions'. In: *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*. Ed. by Thomas A. Henzinger and Dale Miller. ACM, 2014, p. 61. URL: http://doi.acm.org/10.1145/2603088.2603129 (cited on pages 135, 139, 145, 151, 152).

[Laz+07] Ranko Lazic, Thomas Christopher Newcomb, Joël Ouaknine, A. W. Roscoe, and James Worrell. 'Nets with Tokens Which Carry Data'. In: *Proceedings of the 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*. Ed. by Jetty Kleijn and Alexandre Yakovlev. Vol. 4546. Lecture Notes in Computer Science. Springer, 2007, pp. 301–320 (cited on pages 144, 150, 151).

[Ler11a] Jérôme Leroux. 'Vector Addition System Reachability Problem: A Short Self-Contained Proof'. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 307–316 (cited on pages 24, 135).

[Ler11b] Jérôme Leroux. 'Vector Addition System Reversible Reachability Problem'. In: *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR 2011)*. Ed. by Joost-Pieter Katoen and Barbara König. Vol. 6901. Lecture Notes in Computer Science. Springer, 2011, pp. 327–341 (cited on pages 135, 150).

[LPS13] Jérôme Leroux, M. Praveen, and Grégoire Sutre. 'A Relational Trace Logic for Vector Addition Systems with Application to Context-Freeness'. In: *Proceedings of the 24th International Conference on Concurrency Theory (CONCUR 2013)*. Ed. by Pedro R. D'Argenio and Hernán C. Melgratti. Vol. 8052. Lecture Notes in Computer Science. Springer, 2013, pp. 137–151 (cited on pages 135, 150).

[LPS14] Jérôme Leroux, M. Praveen, and Grégoire Sutre. 'Hyper-Ackermannian bounds for pushdown vector addition systems'. In: *Proceedings of the Joint Meeting of the 23rd EACSL Annual Conference on Computer Science Logic and the 29th Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS 2014)*. Ed. by Thomas A. Henzinger and Dale Miller. ACM, 2014, p. 63. URL: http://doi.acm.org/10.1145/2603088.2603146 (cited on page 151).

[LS04] Tobias Lindahl and Konstantinos F. Sagonas. 'Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story'. In: *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS 2004)*. Ed. by Wei-Ngan Chin. Vol. 3302. Lecture Notes in Computer Science. Springer, 2004, pp. 91–106 (cited on pages 3, 72, 155).

[LS05] Tobias Lindahl and Konstantinos F. Sagonas. 'TypEr: a type annotator of Erlang code'. In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang (Erlang Workshop 2005)*. Ed. by Konstantinos F. Sagonas and Joe Armstrong. ACM, 2005, pp. 17–25. URL: http://doi.acm.org/10.1145/1088361.1088366 (cited on page 72).

[LS06] Tobias Lindahl and Konstantinos F. Sagonas. 'Practical type inference based on success typings'. In: *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2006)*. Ed. by Annalisa Bossi and Michael J. Maher. ACM, 2006, pp. 167–178. URL: http://doi.acm.org/10.1145/1140335.1140356 (cited on page 72).

[Lip76] Richard J. Lipton. *The Reachability Problem Requires Exponential Space*. Tech. rep. Department of Computer Science, Yale University, 1976. URL: http://cpsc.yale.edu/sites/default/files/files/tr63.pdf (cited on pages 5, 14, 24, 121, 145, 147, 151, 158, 273).

[LS99] Irina A. Lomazova and Philippe Schnoebelen. 'Some Decidability Results for Nested Petri Nets'. In: *Proceedings of the Ershov Memorial Conference*. Ed. by Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin. Vol. 1755. Lecture Notes in Computer Science. Springer, 1999, pp. 208–220 (cited on pages 115, 117, 118, 121, 149, 150).

[Lon+12] Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. 'Language-Theoretic Abstraction Refinement'. In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*. Ed. by Juan de Lara and Andrea Zisman. Vol. 7212. Lecture Notes in Computer Science. Springer, 2012, pp. 362–376 (cited on page 112).

[LS98] Denis Lugiez and Philippe Schnoebelen. 'The Regular Viewpoint on PA-Processes'. In: *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR 1998)*. Ed. by Davide Sangiorgi and Robert de Simone. Vol. 1466. Lecture Notes in Computer Science. Springer, 1998, pp. 50–66. URL: http://dx.doi.org/10.1007/BFb0055615 (cited on page 110).

[LW70] Martin H. Löb and Stanley S. Wainer. 'Hierarchies of Number-Theoretic Functions. I'. In: *Archiv für mathematische Logik und Grundlagenforschung* 13.1-2 (1970), pp. 39–51. URL: http://dx.doi.org/10.1007/BF01967649 (cited on page 21).

[MP11] P. Madhusudan and Gennaro Parlato. 'The Tree Width of Auxiliary Storage'. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 283–294. URL: http://doi.acm.org/10.1145/1926385.1926419 (cited on page 111).

[MW97] Simon Marlow and Philip Wadler. 'A Practical Subtyping System For Erlang'. In: *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*. Ed. by Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman. ACM, 1997, pp. 136–149. URL: http://doi.acm.org/10.1145/258948.258962 (cited on pages 3, 72, 155).

[Mas76] A. N. Maslov. 'Multilevel Stack Automata'. In: *Problemy Peredachi Informatsii* 12.1 (1976), pp. 55–62 (cited on page 28).

[May81] Ernst W. Mayr. 'An Algorithm for the General Petri Net Reachability Problem'. In: *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC 1981)*. ACM, 1981, pp. 238–246 (cited on page 24).

[May97] Richard Mayr. 'Combining Petri Nets and PA-Processes'. In: *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software (TACS 1997)*. Ed. by Martín Abadi and Takayasu Ito. Vol. 1281. Lecture Notes in Computer Science. Springer, 1997, pp. 547–561 (cited on page 27).

[May00] Richard Mayr. 'Process Rewrite Systems'. In: *Information and Computation* 156.1-2 (2000), pp. 264–286 (cited on page 27).

[Mey08] Roland Meyer. 'On Boundedness in Depth in the π-Calculus'. In: *Proceedings of the Fifth IFIP International Conference On Theoretical Computer Science (IFIP TCS 2008)*. Ed. by Giorgio Ausiello, Juhani Karhumäki, Giancarlo Mauri, and C.-H. Luke Ong. Springer, 2008, pp. 477–489. URL: http://dx.doi.org/10.1007/978-0-387-09680-3_32 (cited on pages 27, 76, 159).

[MS10] Roland Meyer and Tim Strazny. 'Petruchio: From Dynamic Networks to Nets'. In: *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010)*. Ed. by Tayssir Touili, Byron Cook, and Paul Jackson. Springer, 2010, pp. 175–179. URL: http://dx.doi.org/10.1007/978-3-642-14295-6_19 (cited on page 39).

[Mid12] Jan Midtgaard. 'Control-Flow Analysis of Functional Programs'. In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 10 (cited on page 47).

[MJ08] Jan Midtgaard and Thomas P. Jensen. 'A Calculational Approach to Control-Flow Analysis by Abstract Interpretation'. In: *Proceedings of the 15th International Symposium on Static Analysis (SAS 2008)*. Ed. by María Alpuente and Germán Vidal. Springer, 2008, pp. 347–362. URL: http://dx.doi.org/10.1007/978-3-540-69166-2_23 (cited on pages 46, 47).

[Mig10] Matthew Might. 'Abstract Interpreters for Free'. In: *Proceedings of the 17th International Symposium on Static Analysis (SAS 2010)*. Ed. by Radhia Cousot and Matthieu Martel. Vol. 6337. Lecture Notes in Computer Science. Springer, 2010, pp. 407–421 (cited on pages 35, 39, 40, 47).

[MH11] Matthew Might and David Van Horn. 'A Family of Abstract Interpretations for Static Analysis of Concurrent Higher-Order Programs'. In: *Proceedings of the 18th International Symposium on Static Analysis (SAS 2011)*. Ed. by Eran Yahav. Vol. 6887. Lecture Notes in Computer Science. Springer, 2011, pp. 180–197 (cited on pages 47, 73).

[MS06] Matthew Might and Olin Shivers. 'Improving Flow Analyses via GammaCFA: Abstract Garbage Collection and Counting'. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)*. Ed. by John H. Reppy and Julia L. Lawall. ACM, 2006, pp. 13–25 (cited on pages 47, 73).

[MS08] Matthew Might and Olin Shivers. 'Exploiting Reachability and Cardinality in Higher-Order Flow Analysis'. In: *Journal of Functional Programming* 18.5-6 (2008), pp. 821–864 (cited on pages 47, 73).

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. URL: http://dx.doi.org/10.1007/3-540-10235-3 (cited on page 27).

[MPW92] Robin Milner, Joachim Parrow, and David Walker. 'A Calculus of Mobile Processes'. In: *Information and Computation* 100.1 (1992), pp. 1–77 (cited on pages 11, 27).

[Min61] Marvin Minsky. 'Recursive Unsolvability of Post's Problem of 'Tag' and Other Topics in Theory of Turing Machines'. In: *Annals of Mathematics* 74 (3 1961), pp. 437–455 (cited on pages 11, 25).

[Muk+98a] Madhavan Mukund, K. Narayan Kumar, Jaikumar Radhakrishnan, and Milind A. Sohoni. 'Robust Asynchronous Protocols Are Finite-State'. In: *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP 1998)*. Ed. by Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel. Vol. 1443. Lecture Notes in Computer Science. Springer, 1998, pp. 188–199 (cited on pages 38, 108).

[Muk+98b] Madhavan Mukund, K. Narayan Kumar, Jaikumar Radhakrishnan, and Milind A. Sohoni. 'Towards a Characterisation of Finite-State Message-Passing Systems'. In: *Proceedings of the 4th Asian Computing Science Conference on Advances in Computing Science (ASIAN 1998)*. Ed. by Jieh Hsiang and Atsushi Ohori. Vol. 1538. Lecture Notes in Computer Science. Springer, 1998, pp. 282–299 (cited on pages 3, 35, 38, 108).

[MS85] David E. Muller and Paul E. Schupp. 'The Theory of Ends, Pushdown Automata, and Second-Order Logic'. In: *Theoretical Computer Science* 37 (1985), pp. 51–75 (cited on page 28).

[MS11] Filip Murlak and Piotr Sankowski, eds. *Mathematical Foundations of Computer Science 2011 - 36th International Symposium, MFCS 2011, Warsaw, Poland, August 22-26, 2011. Proceedings*. Vol. 6907. Lecture Notes in Computer Science. Springer, 2011.

[NN93] Flemming Nielson and Hanne Riis Nielson. 'From CML to Process Algebras (Extended Abstract)'. In: *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR 1993)*. Ed. by Eike Best. Vol. 715. Lecture Notes in Computer Science. Springer, 1993, pp. 493–508 (cited on page 73).

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999, pp. I–XXI, 1–450 (cited on pages 43, 63, 64, 72).

[Nys03] S. Nyström. 'A Soft-Typing System for Erlang'. In: *ACM Sigplan Erlang Workshop*. 2003, pp. 56–71. URL: http://doi.acm.org/10.1145/940880.940888 (cited on pages 3, 72, 155).

[Ong06] C.-H. Luke Ong. 'On Model-Checking Trees Generated by Higher-Order Recursion Schemes'. In: *Proceedings of the 21th IEEE Symposium on Logic in Computer Science (LICS 2006)*. IEEE Computer Society, 2006, pp. 81–90 (cited on page 29).

[Ong13] C.-H. Luke Ong. 'Recursion Schemes, Collapsible Pushdown Automata and Higher-Order Model Checking'. In: *Proceedings of the 7th International Conference on Language and Automata Theory and Applications (LATA 2013)*. Ed. by Adrian Horia Dediu, Carlos Martín-Vide, and Bianca Truthe. Vol. 7810. Lecture Notes in Computer Science. Springer, 2013, pp. 13–41 (cited on page 29).

[OR11] C.-H. Luke Ong and Steven James Ramsay. 'Verifying Higher-Order Functional Programs with Pattern-Matching Algebraic Data Types'. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 587–598 (cited on pages 2, 11, 36).

[Par66] Rohit Parikh. 'On Context-Free Languages'. In: *Journal of the ACM (JACM)* 13.4 (1966), pp. 570–581 (cited on pages 31, 109).

[Pet62] Carl Adam Petri. 'Kommunikation mit Automaten'. PhD thesis. Technische Universität Darmstadt, 1962 (cited on pages 5, 11, 23).

[Pik07] Rob Pike. *Concurrency and Message Passing Newsqueak*. Google Tech Talks: Advanced Topics in Programming Languages. 2007. URL: https://www.youtube.com/watch?v=hB05UFqOtFA (cited on pages 65–67, 74).

[QR05] Shaz Qadeer and Jakob Rehof. 'Context-Bounded Model Checking of Concurrent Software'. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 93–107 (cited on pages 87, 111).

[Rac78] Charles Rackoff. 'The Covering and Boundedness Problems for Vector Addition Systems'. In: *Theoretical Computer Science* 6 (1978), pp. 223–231. URL: http://dx.doi.org/10.1016/0304-3975(78)90036-1 (cited on pages iii, 5, 14, 24, 121, 135–139, 144, 150–152, 158, 159).

[Ram00] Ganesan Ramalingam. 'Context-Sensitive Synchronization-Sensitive Analysis Is Undecidable'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22.2 (2000), pp. 416–430 (cited on pages 2, 4, 11, 36, 76, 79, 81, 87).

[RNO14] Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. 'A Type-Directed Abstraction Refinement Approach to Higher-Order Model Checking'. In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014)*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 61–72. URL: http://doi.acm.org/10.1145/2535838.2535873 (cited on pages 5, 160).

[Rei08] Klaus Reinhardt. 'Reachability in Petri Nets with Inhibitor Arcs'. In: *Electronic Notes in Theoretical Computer Science* 223 (2008), pp. 239–264 (cited on pages 23, 25).

[RX07] John H. Reppy and Yingqi Xiao. 'Specialization of CML Message-Passing Primitives'. In: *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*. Ed. by Martin Hofmann and Matthias Felleisen. ACM, 2007, pp. 315–326. URL: http://doi.acm.org/10.1145/1190216.1190264 (cited on page 73).

[RVFE11] Fernando Rosa-Velardo and David de Frutos-Escrig. 'Decidability and Complexity of Petri Nets with Unordered Data'. In: *Theoretical Computer Science* 412.34 (2011), pp. 4439–4451 (cited on page 150).

[ST77] George S. Sacerdote and Richard L. Tenney. 'The Decidability of the Reachability Problem for Vector Addition Systems (Preliminary Version)'. In: *Proceedings of the 9th Annual ACM Symposium on Theory of Computing (STOC 1977)*. Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, 1977, pp. 61–76 (cited on page 24).

[SST13] Konstantinos F. Sagonas, Josep Silva, and Salvador Tamarit. 'Precise Explanation of Success Typing Errors'. In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM 2013)*. Ed. by Elvira Albert and Shin-Cheng Mu. ACM, 2013, pp. 33–42 (cited on page 72).

[Sav70] Walter J. Savitch. 'Relationships Between Nondeterministic and Deterministic Tape Complexities'. In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192 (cited on page 22).

[Sch13] Sylvain Schmitz. 'Complexity Hierarchies Beyond Elementary'. In: *CoRR* abs/1312.5686 (2013) (cited on pages iii, 5, 11, 21, 22, 150, 158).

[Sch02] Philippe Schnoebelen. 'Verifying Lossy Channel Systems Has Nonprimitive Recursive Complexity'. In: *Information Processing Letters* 83.5 (2002), pp. 251–261 (cited on pages 25, 149).

[Sch10] Philippe Schnoebelen. 'Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets'. In: *Proceedings of the 35th International Symposium on Mathematical Foundations of Computer Science (MFCS 2010)*. Ed. by Petr Hlinený and Antonín Kucera. Vol. 6281. Lecture Notes in Computer Science. Springer, 2010, pp. 616–628 (cited on pages 25, 149).

[SV06] Koushik Sen and Mahesh Viswanathan. 'Model Checking Multithreaded Programs with Asynchronous Atomic Methods'. In: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 300–314 (cited on pages 4, 12, 33, 76, 80, 86, 88, 108–110, 151, 157, 160).

[Shi88] Olin Shivers. 'Control-Flow Analysis in Scheme'. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1988)*. Ed. by Richard L. Wexelblat. ACM, 1988, pp. 164–174 (cited on page 47).

[Shi91] Olin Shivers. 'Control-Flow Analysis of Higher-Order Languages'. PhD thesis. Carnegie Mellon University, 1991 (cited on pages 36, 47, 49, 73).

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997 (cited on page 21).

[Sto74] Larry Joseph Stockmeyer. 'The Complexity of Decision Problems in Automata Theory and Logic'. PhD thesis. MIT, 1974 (cited on pages iii, 5, 14, 121, 144, 151, 158).

[SF07] Hans Svensson and Lars-Åke Fredlund. 'A More Accurate Semantics for Distributed Erlang'. In: *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang (Erlang Workshop 2007)*. Ed. by Simon J. Thompson and Lars-Åke Fredlund. ACM, 2007, pp. 43–54. URL: http://doi.acm.org/10.1145/1292520.1292528 (cited on page 38).

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. 'The Type and Effect Discipline'. In: *Information and Computation* 111.2 (1994), pp. 245–296 (cited on page 72).

[TN11] Salvatore La Torre and Margherita Napoli. 'Reachability of Multistack Pushdown Systems with Scope-Bounded Matching Relations'. In: *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR 2011)*. Ed. by Joost-Pieter Katoen and Barbara König. Vol. 6901. Lecture Notes in Computer Science. Springer, 2011, pp. 203–218 (cited on pages 87, 111).

[VVN81] Rüdiger Valk and Guy Vidal-Naquet. 'Petri Nets and Regular Languages'. In: *Journal of Computer and System Sciences* 23.3 (1981), pp. 299–325 (cited on page 25).

[Var91] Moshe Y. Vardi. 'Verification of Concurrent Programs: The Automata-Theoretic Framework'. In: *Annals of Pure and Applied Logic* 51.1-2 (1991), pp. 79–98. URL: http://dx.doi.org/10.1016/0168-0072(91)90066-U (cited on page 10).

[VS10] Dimitrios Vardoulakis and Olin Shivers. 'CFA2: A Context-Free Approach to Control-Flow Analysis'. In: *Proceedings of the 19th European Symposium on Programming Languages and Systems (ESOP 2010)*. Ed. by Andrew D. Gordon. Vol. 6012. Lecture Notes in Computer Science. Springer, 2010, pp. 570–589 (cited on pages 47, 73, 79, 156, 160).

[Ven96] Arnaud Venet. 'Abstract Interpretation of the π-Calculus'. In: *Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*. Ed. by Mads Dam. Springer, 1996, pp. 51–75. URL: http://dx.doi.org/10.1007/3-540-62503-8_3 (cited on page 73).

[Yos96] Nobuko Yoshida. 'Graph Types for Monadic Mobile Processes'. In: *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*. Ed. by Vijay Chandru and V. Vinay. Vol. 1180. Lecture Notes in Computer Science. Springer, 1996, pp. 371–386 (cited on page 73).

Index

Appendix A

Appendix for Chapter 3

A.1 Proofs for Section 3.4

Concrete and Abstract Match Function:

$$\begin{aligned}
&\text{match}_{\rho,\sigma}(p_i,(x,\rho')) = \text{match}_{\rho,\sigma}(p_i,\sigma(\rho'(x)))\\
&\text{match}_{\rho,\sigma}(x,d) = \{x \mapsto d\} \text{ if } x \notin \text{dom}(\rho)\\
&\text{match}_{\rho,\sigma}(x,d) = \{x \mapsto d\} \text{ if } \text{match}_{\rho',\sigma}(p',d) \neq \bot\\
&\quad \text{where } (p',\rho') = \sigma(\rho(x))\\
&\text{match}_{\rho,\sigma}(p,(t,\rho')) = \bigotimes_{1\le i\le n} \text{match}_{\rho,\sigma}(p_i,(t_i,\rho'))\\
&\quad \text{where} \qquad p = \mathsf{c}(p_1,\ldots,p_n)\\
&\qquad\qquad\quad t = \mathsf{c}(t_1,\ldots,t_n)\\
&\qquad\qquad \theta\otimes\theta' = \bot \qquad \text{if } \exists x\,.\,\theta(x)\neq\theta'(x)\\
&\qquad\qquad \theta\otimes\theta' = \theta\cup\theta' \quad \text{otherwise}\\
&\qquad\qquad \bigotimes_{\emptyset} = []\\
&\text{match}_{\rho,\sigma}(p,d) = \bot \qquad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
&\widehat{\text{match}}_{\hat\rho,\hat\sigma}(p_i,(x,\hat\rho')) = \bigcup_{\hat d\in\hat\sigma(\hat\rho'(x))} \widehat{\text{match}}_{\hat\rho,\hat\sigma}(p_i,\hat d\,)\\
&\widehat{\text{match}}_{\hat\rho,\hat\sigma}(x,d) = \{x\mapsto \hat d\,\}\\
&\widehat{\text{match}}_{\hat\rho,\hat\sigma}(p,(t,\hat\rho')) = \widehat{\bigotimes_{1\le i\le n}} \widehat{\text{match}}_{\hat\rho,\hat\sigma}(p_i,(t_i,\hat\rho'))\\
&\qquad \text{if } p = \mathsf{c}(p_1,\ldots,p_n) \text{ and}\\
&\qquad\quad t = \mathsf{c}(t_1,\ldots,t_n)\\
&\qquad \text{where } \widehat{\bigotimes}(\{\Theta_i \mid 1\le i\le n\}) = \left\{ \theta \;\middle|\; \begin{array}{l} \theta = \bigotimes_{1\le i\le n} \theta_i, \theta\neq\bot,\\ \theta_i\in\Theta_i, 1\le i\le n \end{array} \right\}\\
&\widehat{\text{match}}_{\hat\rho,\hat\sigma}(p,d) = \emptyset \qquad \text{otherwise}
\end{aligned}$$

Theorem 4 (Soundness of Analysis). *Given a sound abstraction of the basic domains, the relation* $\mathcal{R} = \{(s, \hat{s}) \mid \alpha_{State}(s) \leq_{\widehat{State}} \hat{s}\}$ *is a simulation relation between the concrete and the abstract operational semantics.*

Proof of Theorem 4. The proof is by a case analysis of the rule that defines the concrete transition $s \to_{\mathcal{P}} s'$. For each rule, the transition in the concrete system can be replicated in the abstract transition system using the abstract version of the rule, with the appropriate choice of abstract pid, the continuation from the abstract store, the message from the abstract mailbox, etc.
Let $s = \langle \pi, \mu, \sigma \rangle \to \langle \pi', \mu', \sigma' \rangle = s'$ and $\hat{s} = \langle \hat{\pi}, \hat{\mu}, \hat{\sigma} \rangle$ such that $\alpha_{State}(s) \leq \hat{s}$. Thus, we know the following inequalities hold: $\eta(\pi) \leq \hat{\pi}$, $\eta(\mu) \leq \hat{\mu}$, and $\eta(\sigma) \leq \hat{\sigma}$. We consider a number of rules for illustration.

Case: (**Send**).
We know that $s \to_{\mathcal{P}} s'$ using rule **Send**; we can thus assume

$$\pi(\iota) = \langle v, \rho, a, t \rangle \qquad \sigma(a) = \mathsf{Arg}_2\langle \ell, d, \iota', \_, c \rangle \qquad d = (\mathbf{send}, \_)$$

and for s'

$$\pi' = \pi[\iota \mapsto \langle v, \rho, c, t \rangle] \qquad \mu' = \mu[\iota' \mapsto \mathsf{enq}((v, \rho), \mu(\iota'))] \qquad \sigma' = \sigma.$$

As a first step we will examine u and show that $u \to_{\hat{\mathcal{P}}} u'$ for some u'. For $\hat{\pi}$ and $\hat{\sigma}$, writing $\hat{\iota} := \eta(\iota)$, we know that $\eta(\pi)(\hat{\iota}) \subseteq \hat{\pi}(\hat{\iota})$ and $\eta(\sigma)(\hat{a}) \subseteq \hat{\sigma}(\hat{a})$ from which we can infer:

$$\langle v, \hat{\rho}, \hat{a}, \hat{t} \rangle \in \hat{\pi}(\hat{\iota}) \qquad \mathsf{Arg}_2\langle \ell, \hat{d}, \hat{\iota}', \_, \hat{c} \rangle \in \hat{\sigma}(\hat{a})$$

where $\hat{\rho} = \eta(\rho)$, $\hat{a} = \eta(a)$, $\hat{d} = \eta(d)$, $\hat{t} = \eta(t)$, $\hat{\iota}' = \eta(\iota')$ and $\hat{c} = \eta(c)$. Hence rule **AbsSend** is applicable and we can set

$$\hat{\pi}' := \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\hat{q}\}] \qquad \hat{q} := \langle v, \hat{\rho}, \hat{c}, \hat{t} \rangle \qquad \hat{\mu}' := \hat{\mu}[\hat{\iota}' \mapsto \hat{\mathsf{enq}}((v, \hat{\rho}), \hat{\mu}(\hat{\iota}'))]$$
$$\hat{s}' := \langle \hat{\pi}', \hat{\mu}', \hat{\sigma} \rangle.$$

It follows from rule (**AbsSend**) that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$. Let us show that $\eta(\pi') \leq \hat{\pi}'$. Suppose $\hat{\iota}_0$ be an abstract pid. There are two cases.

(i) $\hat{\iota}_0 \neq \hat{\iota}$. Then $\hat{\pi}'(\hat{\iota}_0) = \hat{\pi}(\hat{\iota}_0) \supseteq \eta(\pi)(\hat{\iota}_0) = \eta(\pi')(\hat{\iota}_0)$. Otherwise

(ii) $\hat{\iota}_0 = \hat{\iota}$. Then $\hat{\pi}'(\hat{\iota}) = \hat{\pi}(\hat{\iota}) \cup \{\hat{q}\} \supseteq \{\hat{q}\} = \eta(\pi')(\hat{\iota})$.

Hence, we conclude $\eta(\pi') \leq \hat{\pi}'$. Further the sound basic domain abstraction ensures us that

$$\eta(\mu'(\iota')) = \eta(\mathsf{enq}((v, \rho), \mu(\iota'))) \leq \widehat{\mathsf{enq}}((v, \hat{\rho}), \hat{\mu}(\hat{\iota}')) = \hat{\mu}'(\hat{\iota}')$$

and hence we can conclude both $\eta(\mu') \leq \hat{\mu}'$ and $\alpha_{State}(s') \leq \hat{s}'$ which concludes this case.

Case: (**Receive**). In the concrete $s \to_{\mathcal{P}} s'$ using the **Receive**, hence we can make the following assumptions

$$\begin{aligned}
\pi(\iota) &= \langle \textbf{receive}\, p_1 \to e_1 \ldots p_n \to e_n\, \textbf{end}, \rho, a, t\rangle =: q \\
(i, \theta, \mathfrak{m}) &= \text{mmatch}(p_1 \ldots p_n, \mu(\iota), \rho, \sigma) \\
\theta &= [x_1 \mapsto d_1 \ldots x_k \mapsto d_k] \\
b_j &= \text{new}_{\text{va}}(\iota, x_j, \delta_j, q) \\
\delta_j &= \text{res}(\sigma, d_j)
\end{aligned}$$

and for state s'

$$\begin{aligned}
\pi' &= \pi[\iota \mapsto q'] \\
q' &= \langle e_i, \rho', a, t\rangle \\
\rho' &= \rho[x_1 \mapsto b_1 \ldots x_k \mapsto b_k] \\
\mu' &= \mu[\iota \mapsto \mathfrak{m}] \\
\sigma' &= \sigma[b_1 \mapsto d_1 \ldots b_k \mapsto d_k].
\end{aligned}$$

As a first step we will look at $\hat{s}$ to prove there there exists a $\hat{s}'$ such that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ using rule **AbsReceive**. Since $\eta(\pi) \leq \hat{\pi}$ we know that for $\hat{\rho} := \eta(\rho)$, $\hat{a} := \eta(a)$, $\hat{t} = \eta(t)$ and $\hat{\iota} := \eta(\iota)$ we can find

$$\hat{q} := \langle \textbf{receive}\, p_1 \to e_1 \ldots p_n \to e_n\, \textbf{end}, \hat{\rho}, \hat{a}, \hat{t}\,\rangle \in \hat{\pi}(\hat{\iota}).$$

Further, the set inequality $\eta(\mu(\iota)) \subseteq \hat{\mu}(\hat{\iota})$ holds as $\eta(\mu) \leq \hat{\mu}$. Since the instantiation of the basic domains is sound and $\eta(\sigma) \leq \hat{\sigma}$ we then know that

$$(i, \hat{\theta}, \hat{\mathfrak{m}}) \in \widehat{\text{mmatch}}(\vec{p}, \hat{\mu}(\hat{\iota}), \hat{\rho}, \hat{\sigma})$$

such that $\hat{\theta} = \eta(\theta)$ and $\hat{\mathfrak{m}} \geq \eta(\mathfrak{m})$. Turning to the substitution $\hat{\theta}$ we can see that

$$\hat{\theta} = [x_1 \mapsto \hat{d}_1 \ldots x_k \mapsto \hat{d}_k]$$

where $\hat{d}_i = \eta(d_i)$ for $1 \leq i \leq k$. Appealing to the sound basic domain instantiation once more, noting that $\eta(\sigma) \leq \hat{\sigma}$, yields that for $j = 1, \ldots, k$ we have $\hat{\delta}_j := \eta(\delta_j) \in \widehat{\text{res}}(\hat{\sigma}, \hat{d}_j)$; to obtain new abstract variable addresses we can now set $\hat{b}_j := \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_j, \hat{\delta}_j, \hat{q}\,)$. Rule **AbsReceive** is applicable now; we make the following definitions

$$\begin{aligned}
\hat{\pi}' &:= \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\hat{q}'\}] \\
\hat{q}' &:= \langle e_i, \hat{\rho}', \hat{a}, \hat{t}\,\rangle \\
\hat{\rho}' &:= \hat{\rho}[x_1 \mapsto \hat{b}_1 \ldots x_k \mapsto \hat{b}_k] \\
\hat{\mu}' &:= \hat{\mu}[\hat{\iota} \mapsto \hat{\mathfrak{m}}] \\
\hat{\sigma}' &:= \hat{\sigma} \sqcup [\hat{b}_1 \mapsto \hat{d}_1 \ldots \hat{b}_k \mapsto \hat{d}_k] \\
\hat{s}' &:= \langle \hat{\pi}', \hat{\mu}', \hat{\sigma}'\rangle
\end{aligned}$$

and observe that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$. It remains to show that $\alpha_{State}(s') \leq \hat{s}'$. Similar reasoning to the one applied in the (Send) case yields that $\eta(\pi')(\hat{\iota}_0) \subseteq \hat{\pi}'(\hat{\iota}_0)$ for all abstract pids $\hat{\iota}_0$ and that $\eta(\sigma')(\hat{a}) \subseteq \hat{\sigma}'(\hat{a})$ for all abstract addresses $\hat{a}$. Hence, we conclude $\eta(\pi') \leq \hat{\pi}'$ and $\eta(\sigma') \leq \hat{\sigma}'$. Since $\hat{\mathfrak{m}} \geq \eta(\mathfrak{m})$ we can similarly conclude that $\eta(\mu')(\hat{\iota}_0) \leq \hat{\mu}'(\hat{\iota}_0)$ for all $\hat{\iota}_0$ and thus $\eta(\mu') \leq \hat{\mu}'$. This completes the proof of this case.

Case: (Apply). Since $s \to_{\mathcal{P}} s'$ using rule Apply we can assume that

$$\begin{aligned}
\pi(\iota) &= \langle v, \rho, a, t\rangle =: q \\
\sigma(a) &= \mathsf{Arg}_n\langle \ell, d_0 \dots d_{n-1}, \rho', c\rangle := \kappa \\
\mathrm{arity}(\ell) &= n \\
d_0 &= (\textbf{fun}(x_1 \dots x_n) \to e, \rho_0) \\
d_n &= (v, \rho)
\end{aligned}$$

and for $i = 1, \dots, n$

$$\begin{aligned}
\delta_i &= \mathrm{res}(\sigma, d_i) \\
b_i &= \mathrm{new}_{\mathrm{va}}(\iota, x_i, \delta_i, q)
\end{aligned}$$

additionally for the successor state s'

$$\begin{aligned}
\pi' &= \pi[\iota \mapsto q'] \\
&\quad \text{where } q' := \langle e, \rho'[x_1 \to b_1 \dots x_n \to b_n], c, \mathrm{tick}(\ell, t)\rangle \\
\sigma' &= \sigma[b_1 \mapsto d_1 \dots b_n \mapsto d_n] \\
\mu' &= \mu
\end{aligned}$$

As a first step we will examine $\hat{s}$ and show that there exists a $\hat{s}'$ such that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ using rule AbsApply. Since $\eta(\pi) \leq \hat{\pi}$, it follows that

$$\hat{q} := \langle v, \eta(\rho), \eta(a), \eta(t)\rangle \in \hat{\pi}(\eta(\iota)).$$

Letting $\hat{\rho} := \eta(\rho)$, $\hat{a} := \eta(a)$, $\hat{t} := \eta(t)$ and $\hat{\iota} := \eta(\iota)$ we further obtain, as $\eta(\sigma) \leq \hat{\sigma}$, that $\mathsf{Arg}_n\langle \ell, \hat{d}_0 \dots \hat{d}_{n-1}, \hat{\rho}', \hat{c}\rangle \in \hat{\sigma}(\hat{a})$ where we write $\hat{d}_i := \eta(d_i)$ for $0 \leq i < n$, $\hat{\rho}' := \eta(\rho')$ and $\hat{c} := \eta(c)$. Inspecting $\hat{d}_0$ we see that $\hat{d}_0 = (\textbf{fun}(x_1 \dots x_n) \to e, \hat{\rho}_0)$ where $\hat{\rho}_0 = \eta(\rho_0)$. Taking $\hat{d}_n := (v, \hat{\rho})$ we obtain from our sound basic domain abstraction $\hat{\delta}_i := \eta(\delta_i) \in \widehat{\mathrm{res}}(\hat{\sigma}, \hat{d}_i)$ for $i = 1, \dots, n$ as $\eta(\sigma) \leq \hat{\sigma}$ and $\hat{d}_i = \eta(d_i)$. Turning to the abstract variable addresses we define $\hat{b}_i := \widehat{\mathrm{new}_{\mathrm{va}}}(\hat{\iota}, x_i, \hat{\delta}_i, \hat{q})$ for $1 \leq i \leq n$. Rule AbsApply is now applicable and we define

$$\begin{aligned}
\hat{\pi}' &:= \hat{\pi} \sqcup [\hat{\iota} \mapsto \{\hat{q}'\}] \\
\hat{q}' &:= \langle e, \hat{\rho}'[x_1 \to \hat{b}_1 \dots x_n \to \hat{b}_n], \hat{c}, \widehat{\mathrm{tick}}(\ell, \hat{t})\rangle \\
\hat{\sigma}' &:= \hat{\sigma} \sqcup [\hat{b}_1 \mapsto \{\hat{d}_1\} \dots \hat{b}_n \mapsto \{\hat{d}_n\}] \\
\hat{s}' &:= \langle \hat{\pi}', \hat{\mu}, \hat{\sigma}'\rangle.
\end{aligned}$$

It is clear from rule AbsApply that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$; it remains to show that $\alpha_{State}(s') \leq \hat{s}'$ to prove this case. First observe that since we have a sound basic domain abstraction we know $\eta(\mathrm{tick}(\ell, t)) =$

$\widehat{\text{tick}}(\ell, \eta(t)\ )$ thus we can conclude that $\eta(q') = \hat{q}'$. Hence it is a routine exercise to verify $\eta(\pi') \leq \hat{\pi}'$ and $\eta(\sigma') \leq \hat{\sigma}'$ which implies $\alpha_{State}(s') \leq \hat{s}'$. This completes the proof of this case.

Case: (Case). Since $s \to_{\mathcal{P}} s'$ using rule Case we can assume that

$$\begin{aligned}
\pi(\iota) &= \langle \textbf{case}\ x\ \textbf{of}\ p_1 \to e_1 \cdots p_n \to e_n\ \textbf{end}, \rho, a, t\rangle =: q\\
\theta &= \text{match}_{\rho,\sigma}(p_i, (x, \rho)),\\
\theta &= [x_1 \mapsto d_1 \ldots x_k \mapsto d_k],\\
b_m &:= \text{new}_{\text{va}}(\iota, x_m, \delta_m, q),\\
\delta_m &= \text{res}(\sigma, d_m)
\end{aligned}$$

and for $m \in \langle k \rangle$. Additionally for the successor state s'

$$\begin{aligned}
\pi' &= \pi[\iota \mapsto q']\ \text{ where } q' := \langle e_i, \rho', a, t\rangle\\
\rho' &:= \rho[x_j \mapsto b_j \mid j \in \langle k \rangle],\\
\sigma' &= \sigma[b_i \mapsto d_i \mid i \in \langle k \rangle],\\
\mu' &= \mu
\end{aligned}$$

Let us examine $\hat{s}$ and show that there exists $\hat{s}'$ such that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ using rule AbsCase. Since $\eta(\pi) \leq \hat{\pi}$, it follows that

$$\hat{q} := \langle \textbf{case}\ x\ \textbf{of}\ p_1 \to e_1 \cdots p_n \to e_n\ \textbf{end}, \eta(\rho), \eta(a), \eta(t)\rangle \in \hat{\pi}(\eta(\iota)).$$

Thus let us define $\hat{\rho} := \eta(\rho)$, $\hat{a} := \eta(a)$, $\hat{t} := \eta(t)$ and $\hat{\iota} := \eta(\iota)$. Inspecting the definition of $\widehat{\text{match}}$ and noting that $\eta(\sigma) \leq \hat{\sigma}$ we can infer that

$$\hat{\theta} \in \widehat{\text{match}}_{\hat{\rho},\hat{\sigma}}(p_i, (x, \hat{\rho}))\ \text{ and } \hat{\theta} = [x_1 \mapsto \hat{d}_1 \ldots x_k \mapsto \hat{d}_k]$$

where $\eta(d_j) = \hat{d}_j$ for all $j \in \langle k \rangle$. Thus, since we have have sound basic domain abstraction we can take

$$\begin{aligned}
\hat{\delta}_j &\in \widehat{\text{res}}(\hat{\sigma}, \hat{d}_j)\\
\hat{b}_j &:= \widehat{\text{new}_{\text{va}}}(\hat{\iota}, x_j, \hat{\delta}_j, \hat{\pi}(\hat{\iota})),\\
\hat{\rho}' &:= \hat{\rho}[x_j \mapsto \hat{b}_j \mid j \in \langle k \rangle],\\
\hat{\pi}' &= \hat{\pi} \sqcup [\iota \mapsto \{\hat{q}'\}]\ \text{ where } \hat{q}' := \langle e_i, \hat{\rho}', \hat{a}, \hat{t}\rangle\\
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{b}_j \mapsto \{\hat{d}_j\} \mid j \in \langle k \rangle]\\
\hat{s}' &:= \langle \hat{\pi}', \hat{\mu}', \hat{\sigma}'\rangle
\end{aligned}$$

where $\hat{\delta}_j = \eta(\delta_j)$ for all $j \in \langle k \rangle$. It is easy to see that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ using AbsCase; it remains to show that $\alpha_{State}(s') \leq \hat{s}'$ to prove this case. First observe that since we have a sound basic domain abstraction we can conclude that that $\eta(q') = \hat{q}'$. Hence it is a routine exercise to verify $\eta(\pi') \leq \hat{\pi}'$ and $\eta(\sigma') \leq \hat{\sigma}'$ which implies $\alpha_{State}(s') \leq \hat{s}'$. This completes the proof of this case. □

A.2 Proofs for Section 3.5

Terminology. Analogously to our remark in Section 3.5 on active components for rules **AbsR** in the abstract operational semantics it is possible to identify a similar pattern in the concrete operational semantics. Henceforth we will speak of the concrete active component (ι, q, q') of a rule **R** of the concrete operational semantics and we will say the abstract active component $(\hat{\iota}, \hat{q}, \hat{q}')$ of a rule **AbsR** of the abstract operational semantics where **AbsR** is the abstract counterpart of **R**. We will omit the adjectives abstract and concrete when there is no confusion.

Lemma 13. Suppose $s \to_{\mathcal{P}} s'$ using the concrete rule **R** with concrete active component (ι, q, q') and $\hat{s} \geq \alpha_{\mathit{State}}(s)$. Then $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ with $\hat{s}' \geq \alpha_{\mathit{State}}(s')$ using rule **AbsR** with abstract active component $(\eta(\iota), \eta(q), \eta(q'))$.

Proof. The claim follows from inspection of the proof of Theorem 4. □

Theorem 6 (Soundness of generated ACS). *For all choices of $\mathcal{I}$ and $\mathcal{D}_{msg}$, the relation $\mathcal{R} = \{(s, \Pi \lhd \Gamma) \mid s \in \mathit{State}, \alpha_{\mathit{acs}}(s) \leq \Pi \lhd \Gamma\}$ is a simulation for $\mathcal{P}$ and $\mathcal{A}_{\mathcal{P}}$.*

Proof of Theorem 6. Suppose $s \to_{\mathcal{P}} s'$ using rule **R** of the concrete operational semantics with active component (ι, q, q') and suppose $(s, \Pi \lhd \Gamma) \in \mathcal{R}$. Let us perform a case analysis on **R**.

- **R** = FunEval, ArgEval, Apply, LetRec, Case, or Vars. Take $\hat{s} = \alpha_{\mathit{State}}(s)$; Lemma 13 gives us that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ using abstract rule **AbsR** = AbsFunEval, AbsArgEval, AbsApply, AbsLetRec, AbsCase, or Vars respectively with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota} = \eta(\iota)$, $\hat{q} = \eta(q)$ and $\hat{q}' = \eta(q')$. It follows that we have the rule

$$\boldsymbol{r} := (\hat{\iota}, \hat{q}) \xrightarrow{\epsilon} (\hat{\iota}, \hat{q}') \in R.$$

  We note that $\alpha_{\text{acs}}(s) = (\hat{\iota}, \hat{q}) \parallel \Pi' \lhd \Gamma'$ and $\alpha_{\text{acs}}(s') = (\hat{\iota}, \hat{q}') \parallel \Pi' \lhd \Gamma'$ for some Π' and Γ'. Thus, $\alpha_{\text{acs}}(s) \to_{\mathcal{A}_{\mathcal{P}}} \alpha_{\text{acs}}(s')$ using rule $\boldsymbol{r}$. Since by assumption $\alpha_{\text{acs}}(s) \leq \Pi \lhd \Gamma$ the monotonicity of $\to_{\mathcal{A}_{\mathcal{P}}}$ implies that $\Pi \lhd \Gamma \to_{\mathcal{A}_{\mathcal{P}}} \Pi' \lhd \Gamma'$ such that $\alpha_{\text{acs}}(s') \leq \Pi' \lhd \Gamma'$. We can deduce that $(s', \Pi' \lhd \Gamma') \in \mathcal{R}$ which is what we wanted to prove.

- **R** = Receive. Let the message matched by mmatch and extracted from $\mu(\iota)$ be $d = (v, \rho')$ for pattern p_i. Let $\hat{s} = \alpha_{\mathit{State}}(s)$. Inspecting the proof of Theorem 6 we can deduce that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ using abstract rule AbsReceive with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota} = \eta(\iota)$, $\hat{q} = \eta(q)$ and $\hat{q}' = \eta(q')$ and where message $\hat{d} = (v, \hat{\rho}')$ with $\hat{\rho}' = \eta(\rho')$, is matched by $\widehat{\text{mmatch}}$ for pattern p_i. We note that $s = \langle \pi, \sigma, \mu \rangle$ and $\hat{s} = \langle \hat{\pi}, \hat{\sigma}, \hat{\mu} \rangle$ where $\hat{\pi} = \eta(\pi)$, $\hat{\sigma} = \eta(\sigma)$ and $\hat{\mu} = \eta(\mu)$.

  Since the message abstraction is a sound data abstraction we know that

$$\hat{m} := \eta(\text{res}(\sigma, d)) \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d}\,)$$

  and hence we have the rule $\boldsymbol{r} := (\hat{\iota}, \hat{q}) \xrightarrow{\hat{\iota}?\hat{m}} (\hat{\iota}, \hat{q}') \in R$.

  Since d is the message extracted from $\mu(\iota)$ and $\hat{m} = \alpha_{\text{msg}}(\text{res}(\sigma, d))$ we know that $\alpha_{\text{acs}}(s) = (\hat{\iota}, \hat{q}) \parallel \Pi_0 \lhd \Gamma_0 \oplus [\hat{\iota} \mapsto [\hat{m}]]$ and $\alpha_{\text{acs}}(s') = (\hat{\iota}, \hat{q}') \parallel \Pi_0 \lhd \Gamma_0$ for some Π_0 and Γ_0. Thus,

$\alpha_{\text{acs}}(s) \to_{\mathcal{A}_\mathcal{P}} \alpha_{\text{acs}}(s')$ using rule $\boldsymbol{r}$. Since by assumption $\alpha_{\text{acs}}(s) \leq \Pi \lhd \Gamma$ the monotonicity of $\to_{\mathcal{A}_\mathcal{P}}$ implies that $\Pi \lhd \Gamma \to_{\mathcal{A}_\mathcal{P}} \Pi' \lhd \Gamma'$ such that $\alpha_{\text{acs}}(s') \leq \Pi' \lhd \Gamma'$. We can deduce that $(s', \Pi' \lhd \Gamma') \in \mathcal{R}$ which is what we wanted to prove.

- **R** = **Send**. Using Lemma 13, with $\hat{s} = \alpha_{\text{cfa}}(s)$, gives $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ for the abstract rule **AbsSend** where $\hat{\iota} = \eta(\iota)$, $\hat{q} = \eta(q)$ and $\hat{q}' = \eta(q')$. Examining the concrete and abstract states we see $s = \langle \pi, \sigma, \mu \rangle$ and $\hat{s} = \langle \hat{\pi}, \hat{\sigma}, \hat{\mu} \rangle$ where $\hat{\pi} = \eta(\pi)$, $\hat{\sigma} = \eta(\sigma)$ and $\hat{\mu} = \eta(\mu)$. Let the pid of the recipient be ι' and let d be the value enqueued to ι''s mailbox $\mu(\iota')$; inspecting the proof of Theorem 4 the pid of the abstract recipient is $\hat{\iota}' := \eta(\iota')$ and the sent abstract value is $\hat{d} = \eta(d)$. Appealing to the soundness of the message abstraction we obtain

$$\hat{m} := \eta(\text{res}(\sigma, d)) \in \widehat{\text{res}}_{\text{msg}}(\hat{\sigma}, \hat{d}\,)$$

and hence we have the rule

$$\boldsymbol{r} := (\hat{\iota}, \hat{q}) \xrightarrow{\hat{\iota}' ! \hat{m}} (\hat{\iota}, \hat{q}') \in R.$$

We know that $\alpha_{\text{acs}}(s) = (\hat{\iota}, \hat{q}) \parallel \Pi_0 \lhd \Gamma_0$ and $\alpha_{\text{acs}}(s') = (\hat{\iota}, \hat{q}') \parallel \Pi_0 \lhd \Gamma_0 \oplus [\hat{\iota}' \mapsto [\hat{m}]]$ for some Π_0 and Γ_0. Thus, $\alpha_{\text{acs}}(s) \to_{\mathcal{A}_\mathcal{P}} \alpha_{\text{acs}}(s')$ using rule $\boldsymbol{r}$. Since by assumption $\alpha_{\text{acs}}(s) \leq \Pi \lhd \Gamma$ the monotonicity of $\to_{\mathcal{A}_\mathcal{P}}$ implies that $\Pi \lhd \Gamma \to_{\mathcal{A}_\mathcal{P}} \Pi' \lhd \Gamma'$ such that $\alpha_{\text{acs}}(s') \leq \Pi' \lhd \Gamma'$. We can deduce that $(s', \Pi' \lhd \Gamma') \in \mathcal{R}$ which is what we wanted to prove.

- **R** = **Spawn**. Take $\hat{s} = \alpha_{\text{cfa}}(s)$; Lemma 13 gives us that $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$ using abstract rule **AbsSpawn** with active component $(\hat{\iota}, \hat{q}, \hat{q}')$ where $\hat{\iota} = \eta(\iota)$, $\hat{q} = \eta(q)$ and $\hat{q}' = \eta(q')$. We note that $s = \langle \pi, \sigma, \mu \rangle$, $s' = \langle \pi', \sigma', \mu' \rangle$ and $\hat{s} = \langle \hat{\pi}, \hat{\sigma}, \hat{\mu} \rangle$ where $\hat{\pi} = \eta(\pi)$, $\hat{\sigma} = \eta(\sigma)$ and $\hat{\mu} = \eta(\mu)$. Further we can assume

$$\begin{array}{lll} \pi(\iota) = \langle \textbf{fun}() \to e, \rho, a, t \rangle & \sigma(a) = \text{Arg}_1\langle \ell, d, \rho', c \rangle & d = (\textbf{spawn}, \_) \\ \iota' := \text{new}_{\text{pid}}(\iota, \ell, \vartheta) & \pi'(\iota) = \langle \iota', \rho', c, t \rangle = q' & \\ \pi'(\iota') = \langle e, \rho, *, t_0 \rangle =: q'' & & \end{array}$$

Noting that we are replicating the step $s \to_{\mathcal{P}} s'$ in the abstract $\hat{s} \to_{\hat{\mathcal{P}}} \hat{s}'$, $\alpha_{\text{cfa}}(s) = \hat{s}$ and $\eta \circ \text{new}_{\text{pid}} = \widehat{\text{new}_{\text{pid}}} \circ \eta$ we can see that the new abstract pid created is $\hat{\iota}' = \eta(\iota')$ together with its process state $\hat{q}'' = \eta(q'')$. Hence we can conclude that we have the rule

$$\boldsymbol{r} := (\hat{\iota}, \hat{q}) \xrightarrow{\nu \hat{\iota}', \hat{q}''} (\hat{\iota}, \hat{q}) \in R.$$

We observe that $\alpha_{\text{acs}}(s) = (\hat{\iota}, \hat{q}) \parallel \Pi_0 \lhd \Gamma_0$ and $\alpha_{\text{acs}}(s') = (\hat{\iota}, \hat{q}') \parallel (\hat{\iota}', \hat{q}'') \parallel \Pi_0 \lhd \Gamma_0 \oplus [\hat{\iota}' \mapsto [\hat{m}]]$ for some Π_0 and Γ_0. Thus, $\alpha_{\text{acs}}(s) \to_{\mathcal{A}_\mathcal{P}} \alpha_{\text{acs}}(s')$ using rule $\boldsymbol{r}$. Since by assumption $\alpha_{\text{acs}}(s) \leq \Pi \lhd \Gamma$ the monotonicity of $\to_{\mathcal{A}_\mathcal{P}}$ implies that $\Pi \lhd \Gamma \to_{\mathcal{A}_\mathcal{P}} \Pi' \lhd \Gamma'$ such that $\alpha_{\text{acs}}(s') \leq \Pi' \lhd \Gamma'$. We can deduce that $(s', \Pi' \lhd \Gamma') \in \mathcal{R}$ which completes the proof of this case and the theorem.

□

Appendix B

Appendix for Chapter 4

B.1 Proofs for Section 4.1

Lemma 1. Coverability and simple coverability for ACPS polynomial-time interreduce.

Proof. The reduction from simple coverability to coverability of ACPS is trivial since the former is a subproblem of the latter. For the other direction suppose we have a coverability query $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi_{\text{cov}} \lhd \Gamma_{\text{cov}})$. Let us assume that $\mathcal{P} = (\mathcal{Q}, \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R})$, $\Pi_{\text{cov}} = (q_1, \beta_1) \parallel \cdots \parallel (q_n, \beta_n)$ and $\Pi_0 = (q_1^0, \beta_1^0) \parallel \cdots \parallel (q_m^0, \beta_m^0)$ where $q_i, q_j^0 \in \mathcal{Q}$ and $\beta_i, \beta_j^0 \in \mathcal{A}^*$ for $i \in \langle n \rangle, j \in \langle m \rangle$.
Let us define the following ACPS $\mathcal{P}' = (\mathcal{Q}', \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R}')$ where $\mathcal{Q}' = \mathcal{Q} \cup \{q'_{(i,A,\beta')}, q'_{(i,\epsilon)}, q^0_{(j,\beta'')} : \beta_i = A \cdot \beta \cdot \beta', \beta_j^0 = \beta'' \cdot \beta''', \beta_k = \epsilon, i, k \in \langle n \rangle, j \in \langle m \rangle\}$ and all control states $q'_{(i,A,\beta)}$, $q'_{(k,\epsilon)}$ $q^0_{(j,\beta)}$ are fresh and

$$\mathcal{R}' = \mathcal{R} \cup \left\{ \begin{array}{c} (q'_{(i,A,B\,\beta')}, B) \xrightarrow{\epsilon} (q'_{(i,A,\beta')}, \epsilon), \\ (q'_{(i,A,\beta'')}, B') \xrightarrow{\epsilon} (q'_{(i,\beta'')}, \epsilon), \\ (q_i, A) \xrightarrow{\epsilon} (q'_{(i,A,\beta'_i)}, \epsilon) \end{array} \middle| \begin{array}{l} \beta_i = A\,\beta'_i, \\ \beta'_i = \beta_0 \cdot B \cdot \beta', \\ \beta'_i = \beta_1 \cdot \beta'', \\ i \in \langle n \rangle \end{array} \right\}$$

$$\cup \left\{ \begin{array}{c} (q_k, \epsilon) \xrightarrow{\epsilon} (q'_{(k,\epsilon)}, \epsilon), \\ (q^0_{(j,\beta'''\cdot C)}, D_0) \xrightarrow{\epsilon} (q^0_{(j,\beta''')}, D_0\, C), \\ (q^0_{(j,\epsilon)}, D_0) \xrightarrow{\epsilon} (q_j^0, \epsilon) \end{array} \middle| \begin{array}{l} \beta_k = \epsilon, \\ \beta_j^0 = \beta''' \cdot C \cdot \beta'''' \\ k \in \langle n \rangle, j \in \langle m \rangle \end{array} \right\}$$

The ACPS $\mathcal{P}'$ essentially implements the query $\mathcal{Q}'$. The rules involving $q^0_{(j,\beta)}$ set up the start configuration with arbitrary stacks from a length one stack. Rules involving $q'_{(k,\epsilon)}$ and $q'_{(i,A,\epsilon)}$ essentially check that the coverability query is satisfied. In order to account for this we change the coverability query to $\mathcal{Q}' = (\mathcal{P}, \Pi'_0 \lhd \Gamma_0, \Pi'_{\text{cov}} \lhd \Gamma_{\text{cov}})$ where $\Pi'_0 = (q^0_{(1,\beta_1^0)}, D_0) \parallel \cdots \parallel (q^0_{(n,\beta_n^0)}, D_0)$, $\Pi'_{\text{cov}} = (q'_{(1,\tilde{\beta}_1)}, \epsilon) \parallel \cdots \parallel (q'_{(n,\tilde{\beta}_n)}, \epsilon)$ and if $\beta_i = \epsilon$ then $\tilde{\beta}_1 = \epsilon$; otherwise if $\beta_i = A \cdot \beta'$ then $\tilde{\beta}_i = A, \epsilon$. By construction $\mathcal{Q}'$ is a simple query.

Since any set of suffixes/prefixes of a sequence β satisfies

$$|\{\beta_0 : \beta_0 \cdot \beta_1 = \beta\}| = |\{\beta_1 : \beta_0 \cdot \beta_1 = \beta\}| \leq |\beta|$$

we can clearly set that $|\mathcal{Q}'| \leq |\mathcal{Q}| + (n+m) \times \max\{|\beta_i|, |\beta_j^0| : i \in \langle n \rangle, j \in \langle m \rangle\}$ and $|\mathcal{R}'| \leq |\mathcal{R}| + 3(n+m) \times \max\{|\beta_i|, |\beta_j^0| : i \in \langle n \rangle, j \in \langle m \rangle\}$ and hence $\mathcal{P}'$ and $\mathcal{Q}'$ are clearly polynomial-time computable from $\mathcal{P}$ and $\mathcal{Q}$.

By construction $\mathcal{P}'$ has the following property: $(q'_{i,A,\beta_1}, A\,\beta) \parallel \Pi_0 \lhd \Gamma \to^*_{\mathcal{P}'} (q'_{i,A,\epsilon}, \epsilon) \parallel \Pi_0 \lhd \Gamma$ if and only if $\beta_i = A\,\beta_0\beta_1$, $\beta_1 \leq_{\text{Hig}} \beta$, i.e. checking coverability at a process level is correctly implemented by each process. Further $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}} \Pi \lhd \Gamma$ if, and only if, $\Pi'_0 \lhd \Gamma_0 \to^*_{\mathcal{P}'} \Pi \lhd \Gamma$, i.e. Π'_0 correctly sets up Π_0.

Suppose now that $\mathcal{Q}$ is a yes-instance. This means that $\Pi_0 \lhd \Gamma_0 \to^*_{\mathcal{P}} \Pi \lhd \Gamma$ such that $\Pi_{\text{cov}} \lhd \Gamma_{\text{cov}} \leq_{ACPS} \Pi \lhd \Gamma$. Clearly it is then the case that $\Pi'_0 \lhd \Gamma_0 \to^*_{\mathcal{P}'} \Pi \lhd \Gamma$. Since $\Pi_{\text{cov}} \lhd \Gamma_{\text{cov}} \leq_{ACPS} \Pi \lhd \Gamma$ we know that $\Pi = (q_1, \beta'_1) \parallel \cdots \parallel (q_n, \beta'_n) \parallel \Pi'$ and for all $i \in \langle n \rangle$ either $\beta_i = \epsilon$ or $\beta_i = A \cdot \beta''_i$, $\beta'_i = A \cdot \beta'''_i$ and $\beta''_i \leq_{\text{Hig}} \beta'''_i$.

And hence $\Pi'_0 \lhd \Gamma_0 \to^*_{\mathcal{P}'} \Pi \lhd \Gamma \to^*_{\mathcal{P}'} (q'_{(1,\tilde{\beta}_1)}, \epsilon) \parallel \cdots \parallel (q'_{(n,\tilde{\beta}_n)}, \epsilon) \parallel \Pi' \lhd \Gamma$ and hence $\mathcal{Q}'$ is a yes-instance for coverability.

For the other direction, suppose $\mathcal{Q}'$ is a yes-instance for coverability. We then know that $\Pi'_0 \lhd \Gamma_0 \to^*_{\mathcal{P}'} (q'_{(1,\tilde{\beta}_1)}, \beta'_1) \parallel \cdots \parallel (q'_{(n,\tilde{\beta}_n)}, \beta'_n) \parallel \Pi' \lhd \Gamma' =: s'$ such that

$$(q'_{(1,\tilde{\beta}_1)}, \epsilon) \parallel \cdots \parallel (q'_{(n,\tilde{\beta}_1)}, \epsilon) \lhd \Gamma \leq_{ACPS} s'.$$

Then our observation above tells us that it must be the case that (possibly reordering locally-independent transitions) $\Pi'_0 \lhd \Gamma_0 \to^*_{\mathcal{P}'} (q_1, \beta''_1\beta'_1) \parallel \cdots \parallel (q_n, \beta''_n\beta'_n) \parallel \Pi' \lhd \Gamma' \to^*_{\mathcal{P}'} (q'_{(1,\beta_1)}, \beta''_1\beta'_1) \parallel \cdots \parallel (q'_{(n,\beta_n)}, \beta''_n\beta'_n) \parallel \Pi' \lhd \Gamma'$ with either $\beta_i = \epsilon$ or $\beta_i = A\beta_i^\dagger$, $\beta''_1 = A\beta'''_i$ and $\beta_i^\dagger \leq_{\text{Hig}} \beta'''_i$ and thus $\beta_i^\dagger \leq_{\text{Hig}} \beta'''_i\beta'_i$ for all $i \in \langle n \rangle$. Further $\Pi_0 \lhd \Gamma_0 \to^*_{\mathcal{P}} (q_1, \beta''_1\beta'_1) \parallel \cdots \parallel (q_n, \beta''_n\beta'_n) \parallel \Pi' \lhd \Gamma'$ Hence $\mathcal{Q}$ is a yes-instance for coverability.

We can thus conclude that simple coverability and coverability are polynomial-time interreducible. □

Proof of Proposition 1

In this section we will give a proof of the following Proposition:

Proposition 1. Given an ACPS $\mathcal{P}$, a simple coverability query $\mathcal{Q}$ and a $\Pi^0 \lhd \Gamma^0$ there exists ACPS $\mathcal{F}_{\text{nf}}(\mathcal{P})$ in normal form, a simple coverability query $\mathcal{F}_{\text{nf}}(\mathcal{Q})$, and $\mathcal{F}_{\text{nf}}(\Pi^0 \lhd \Gamma^0)$ — all polynomial-time computable — such that: $\mathcal{Q}$ is a yes-instance if, and only if, $\mathcal{F}_{\text{nf}}(\mathcal{Q})$ is a yes-instance; and $\mathcal{P}$ is bounded (terminating) from $\Pi^0 \lhd \Gamma^0$ if, and only if, $\mathcal{F}_{\text{nf}}(\mathcal{P})$ is bounded (terminating respectively) from $\mathcal{F}_{\text{nf}}(\Pi^0 \lhd \Gamma^0)$.

We will give a proof in two steps: (i) we first transform a general ACPS $\mathcal{P}$ into an ACPS that satisfies a *pre-normal form* as defined below; (ii) secondly, we show how to transform an ACPS in pre-normal form with the desired property. We lay out our argument in the two Lemmas below, but first we define *pre-normal form*: We say an ACPS $\mathcal{P} = (\mathcal{Q}, \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R})$ is in pre-normal

if for all $(q,\beta) \xrightarrow{\lambda} (q',\beta') \in \mathcal{R}$ (i') if $\lambda \neq \epsilon$ then $\beta = \beta' = \epsilon$, otherwise either (ii') $\beta = A \in \mathcal{A}$ and $\beta' = \epsilon$ or (iii') $\beta = \epsilon$ and $\beta' = A' \in \mathcal{A}$; or (iv') $\beta = \beta' = \epsilon$; and (v') if $\lambda = \nu(q'',\beta'')$ then $\beta'' = \epsilon$.

Lemma 14. Given an ACPS $\mathcal{P}$, a simple coverability query Q and a start configuration $\Pi^0 \lhd \Gamma^0$ there exists ACPS $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ in pre-normal form, simple coverability query $\mathcal{F}^{\text{pnf}}(Q)$, and start configuration $\mathcal{F}^{\text{pnf}}(\Pi^0 \lhd \Gamma^0)$ — all polynomial-time computable — such that: (A) Q is a yes-instance if, and only if, $\mathcal{F}^{\text{pnf}}(Q)$ is a yes-instance; (B) $\mathcal{P}$ is bounded from $\Pi^0 \lhd \Gamma^0$ if, and only if, $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ is bounded from $\mathcal{F}^{\text{pnf}}(\Pi^0 \lhd \Gamma^0)$; and (C) $\mathcal{P}$ is terminating from $\Pi^0 \lhd \Gamma^0$ if, and only if, $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ is terminating from $\mathcal{F}^{\text{pnf}}(\Pi^0 \lhd \Gamma^0)$.

Proof. Let us fix an ACPS $\mathcal{P} = (\mathcal{Q}, \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R})$ and let us define the ACPS $\mathcal{P}_0 = (\mathcal{Q}', \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R}')$ and

$$\mathcal{Q}' = \mathcal{Q} \cup \left\{ q^{\text{pop}}_{\beta_0}, q'^{\text{push}}_{\beta_1} \;\middle|\; \begin{array}{l} (q,\beta) \xrightarrow{\lambda} (q',\beta') \in \mathcal{R}', \beta = \beta'_0 \cdot \beta_0, \\ \beta' = \beta_1 \cdot \beta'_1 \end{array} \right\}$$
$$\cup \left\{ q''^{\text{push}}_{\beta''_0} \;\middle|\; \begin{array}{l} (q,\beta) \xrightarrow{\nu(q'',\beta'')} (q',\beta') \in \mathcal{R}' \\ \beta'' = \beta''_0 \cdot \beta''_1 \end{array} \right\} \cup \{ q^{\text{cov}} \mid q \in \mathcal{Q} \}$$
$$\mathcal{R}' = \left\{ (q^{\text{pop}}_{\beta}, \epsilon) \xrightarrow{\lambda'} (q'^{\text{push}}_{\beta'}, \epsilon) \;\middle|\; \begin{array}{l} (q,\beta) \xrightarrow{\lambda} (q',\beta') \in \mathcal{R} \\ \text{if } \lambda = \nu(q'',\beta'') \\ \quad \text{then } \lambda' = \nu(q''^{\text{push}}_{\beta''}, \epsilon) \\ \text{otherwise } \lambda' = \lambda \end{array} \right\}$$
$$\cup \left\{ \begin{array}{l} (q,\epsilon) \xrightarrow{\epsilon} (q^{\text{pop}}_{\epsilon}, \epsilon), \\ (q'^{\text{push}}_{\epsilon}, \epsilon) \xrightarrow{\epsilon} (q',\epsilon) \end{array} \;\middle|\; (q,\beta) \xrightarrow{\lambda} (q',\beta') \in \mathcal{R}' \right\}$$
$$\cup \left\{ \begin{array}{l} (q^{\text{pop}}_{\beta}, A) \xrightarrow{\epsilon} (q^{\text{pop}}_{\beta\cdot A}, \epsilon), \\ (q'^{\text{push}}_{\beta'\cdot A'}, \epsilon) \xrightarrow{\epsilon} (q'^{\text{push}}_{\beta'}, A'), \end{array} \;\middle|\; \begin{array}{l} q^{\text{pop}}_{\beta}, q^{\text{pop}}_{\beta\cdot A} \in \mathcal{Q}' \\ q'^{\text{push}}_{\beta'\cdot A'}, q'^{\text{push}}_{\beta'} \in \mathcal{Q}' \end{array} \right\}$$

The rules of $\mathcal{P}_0$ simply implement a $(q,\beta) \xrightarrow{\lambda} (q',\beta')$ by popping β one symbol at the time and then pushing β' one symbol at the time. It is easy to see that $\mathcal{P}_0$ is in pre-normal form. Further let $\Xi = \{|\beta|, |\beta'| : (q,\beta) \xrightarrow{\lambda} (q',\beta') \in \mathcal{R}'\} \cup \{|\beta''| : (q,\beta) \xrightarrow{\nu(q'',\beta'')} (q',\beta') \in \mathcal{R}'\}$ then $|\mathcal{Q}'| \leq 2|\mathcal{Q}| + 3 \times |\mathcal{R}| \times \max(\Xi)$ and $|\mathcal{R}'| \leq 2 \times |\mathcal{Q}'| + 3 \times |\mathcal{R}|$ and so $\mathcal{P}_0$ is clearly polynomial-time computable from $\mathcal{P}$.

We will show that there is a *weak reflexive bisimulation* between $\mathcal{P}$ and $\mathcal{P}_0$.

Definition (Weak reflexive bisimulation). Suppose $(S, \xrightarrow{u}_S)$ and $(S', \xrightarrow{u}_{S'})$ are labelled transition systems we say a relation $\mathcal{B} \subseteq S \times S'$ is a *weak reflexive simulation* if for all $(s,s') \in \mathcal{B}$, if for some $t \in S$ we have $s \xrightarrow{u}_S t$ then either $(t,s') \in \mathcal{B}$ and $u = \epsilon$ or there exists $t' \in S'$ such that $s' \xrightarrow{u}{}^*_{S'} t'$ and $(t,t') \in \mathcal{B}$. We say $\mathcal{B}$ is a *weak reflexive bisimulation* relation just if both $\mathcal{B}$ and $\mathcal{B}^{-1}$ are weak reflexive simulation relations.

We temporarily label the transition systems $(\mathcal{P}, \rightarrow_{\mathcal{P}})$ and $(\mathcal{P}_0, \rightarrow_{\mathcal{P}_0})$ with rules of $\mathcal{R}$. Let us label the transition $\Pi \lhd \Gamma \xrightarrow{r}_{\mathcal{P}} \Pi' \lhd \Gamma'$ if the rule $r \in \mathcal{R}$ is used to justify the transition. We label $\mathcal{P}_0$'s transition as follows: If $\Pi \lhd \Gamma \rightarrow_{\mathcal{P}_0} \Pi' \lhd \Gamma'$ using a rule $(q^{\text{pop}}_{\beta}, \epsilon) \xrightarrow{\lambda'} (q'^{\text{push}}_{\beta'}, \epsilon) \in \mathcal{R}'$ introduced because of a rule $r = (q, \beta) \xrightarrow{\lambda} (q', \beta') \in \mathcal{R}$ we label the transition $\Pi \lhd \Gamma \xrightarrow{r}_{\mathcal{P}_0} \Pi' \lhd \Gamma'$; otherwise we label the transition with ϵ, i.e. $\Pi \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{P}_0} \Pi' \lhd \Gamma'$.

Let first define a representation function for the state of a pushdown process:

$$F(q,\beta) = \{(q,\beta)\} \cup \left\{ (q^{\text{push}}_{\beta_1}, \beta_2 \cdot \beta''') \;\middle|\; \begin{array}{l} \exists (q', \beta') \xrightarrow{\lambda} (q, \beta'') \in \mathcal{R} \\ \beta = \beta'' \cdot \beta''' \\ \beta'' = \beta_1 \cdot \beta_2 \end{array} \right\}$$
$$\cup \left\{ (q^{\text{push}}_{\beta_1}, \beta_2 \cdot \beta''') \;\middle|\; \begin{array}{l} \exists (q_0, \beta_0) \xrightarrow{\lambda} (q_1, \beta_1) \in \mathcal{R} \\ \lambda = \nu(q, \beta) \\ \beta = \beta_1 \cdot \beta_2 \end{array} \right\}$$
$$\cup \left\{ (q^{\text{pop}}_{\beta_1}, \beta_2 \cdot \beta''') \;\middle|\; \begin{array}{l} \exists (q, \beta') \xrightarrow{\lambda} (q', \beta'') \in \mathcal{R} \\ \beta = \beta' \cdot \beta''' \\ \beta' = \beta_1 \cdot \beta_2 \end{array} \right\}$$

with which we can now relate configurations of $\mathcal{P}$ and $\mathcal{P}_0$ using the relation:

$$\mathcal{B} = \{(\Pi \lhd \Gamma, \Pi' \lhd \Gamma) : \Pi = \pi_1 \parallel \cdots \parallel \pi_n, \Pi' = \pi'_1 \parallel \cdots \parallel \pi'_n, \forall i \in \langle n \rangle . \pi'_i \in F(\pi_i)\}.$$

Let us now prove that $\mathcal{B}$ is a weak reflexive simulation. Suppose $(\pi \parallel \Pi \lhd \Gamma, \pi_0 \parallel \Pi_0 \lhd \Gamma) \in \mathcal{B}$, and clearly $\pi = (q, \beta)$, and $(q, \beta) \parallel \Pi \lhd \Gamma \xrightarrow{l}_{\mathcal{P}} (q', \beta') \parallel \Pi' \lhd \Gamma'$. Clearly this must happen using rule $l = (q, \beta_0) \xrightarrow{\lambda} (q', \beta'_0)$, and, $\beta = \beta_0 \cdot \beta_1$ and $\beta' = \beta'_0 \cdot \beta_1$. We will briefly show that we can assume $\pi_0 = (q^{\text{pop}}_{\beta_0}, \beta_1)$. We observe that we can then perform the following ϵ-labelled transitions:

$$\begin{aligned} (q^{\text{push}}_{\beta''}, \beta''') \parallel \Pi_0 \lhd \Gamma &\xrightarrow{\epsilon *}_{\mathcal{P}_0} (q^{\text{push}}_{\epsilon}, \beta'' \cdot \beta''') \parallel \Pi_0 \lhd \Gamma \\ &\xrightarrow{\epsilon}_{\mathcal{P}_0} (q_\epsilon, \beta'' \cdot \beta''') \parallel \Pi_0 \lhd \Gamma = (q_\epsilon, \beta) \parallel \Pi_0 \lhd \Gamma \\ &\xrightarrow{\epsilon}_{\mathcal{P}_0} (q^{\text{pop}}_{\epsilon}, \beta) \parallel \Pi_0 \lhd \Gamma \\ = (q^{\text{pop}}_{\epsilon}, \beta_0 \cdot \beta_1) \parallel \Pi_0 \lhd \Gamma &\xrightarrow{\epsilon *}_{\mathcal{P}_0} (q^{\text{pop}}_{\beta_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma \end{aligned}$$

it should be clear that for all $\pi_0 \in F(q, \beta)$ we can ϵ-transition $\pi_0 \parallel \Pi_0 \lhd \Gamma \xrightarrow{\epsilon *}_{\mathcal{P}_0} (q^{\text{pop}}_{\beta_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma$. Hence we will assume in the following that $\pi_0 = (q^{\text{pop}}_{\beta_0}, \beta_1)$.

First we note that there is a rule $(q^{\text{pop}}_{\beta_0}, \epsilon) \xrightarrow{\lambda'} (q'^{\text{push}}_{\beta'_0}, \epsilon) \in \mathcal{R}'$. We can thus make a case analysis on λ.

- *Case:* $\lambda = \epsilon$.
  Then clearly $\Pi = \Pi'$ and $\Gamma' = \Gamma$ and $\lambda' = \epsilon$ and:

$$(q^{\text{pop}}_{\beta_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{P}_0} (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma.$$

Thus we can see that $(q'^{\text{push}}_{\beta'_0}, \beta_1) \in F(q', \beta')$ and hence $((q', \beta') \parallel \Pi' \lhd \Gamma, (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma) \in \mathcal{B}$ which is what we wanted to prove.

- *Case:* $\lambda = c\,!\,m$.
  Then clearly $\Pi = \Pi'$ and $\Gamma' = \Gamma \oplus [c \mapsto [m]]$ and $\lambda' = c\,!\,m$ and:

$$(q^{\text{pop}}_{\beta_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{P}_0} (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma'.$$

  Thus we can see that $(q'^{\text{push}}_{\beta'_0}, \beta_1) \in F(q', \beta')$ and hence $((q', \beta') \parallel \Pi' \lhd \Gamma', (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma') \in \mathcal{B}$ which is what we wanted to prove.

- *Case:* $\lambda = c\,?\,m$.
  Then clearly $\Pi = \Pi'$ and $\Gamma = \Gamma' \oplus [c \mapsto [m]]$ and $\lambda' = c\,?\,m$ and:

$$(q^{\text{pop}}_{\beta_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{P}_0} (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma'.$$

  Thus we can see that $(q'^{\text{push}}_{\beta'_0}, \beta_1) \in F(q', \beta')$ and hence $((q', \beta') \parallel \Pi' \lhd \Gamma', (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma') \in \mathcal{B}$ which is what we wanted to prove.

- *Case:* $\lambda = \nu(q'', \beta'')$.
  Then clearly $\Pi' = (q'', \beta'') \parallel \Pi$ and $\Gamma' = \Gamma$ and $\lambda' = \nu(q''^{\text{push}}_{\beta''}, \epsilon)$ and:

$$(q^{\text{pop}}_{\beta_0}, \beta_1) \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{P}_0} (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel (q''^{\text{push}}_{\beta''}, \epsilon) \parallel \Pi_0 \lhd \Gamma'.$$

  Thus we can see that $(q'^{\text{push}}_{\beta'_0}, \beta_1) \in F(q', \beta')$, $(q''^{\text{push}}_{\beta''}, \epsilon) \in F(q'', \beta'')$ and hence $((q', \beta') \parallel \Pi' \lhd \Gamma', (q'^{\text{push}}_{\beta'_0}, \beta_1) \parallel (q''^{\text{push}}_{\beta''}, \epsilon) \parallel \Pi_0 \lhd \Gamma') \in \mathcal{B}$ which is what we wanted to prove.

Hence we can conclude that $\mathcal{B}$ is a weak reflexive simulation.

Let us turn now to $\mathcal{B}^{-1}$. Suppose $(\pi \parallel \Pi \lhd \Gamma, \pi_0 \parallel \Pi_0 \lhd \Gamma) \in \mathcal{B}$ and $\pi_0 \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{P}} \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0$ using rule $r \in \mathcal{R}'$.

Let us perform a case analysis on r

- *Case:* $r = (q, \epsilon) \xrightarrow{\epsilon} (q^{\text{pop}}_{\epsilon}, \epsilon)$.
  Then clearly $\pi_0 = (q, \beta)$, $\pi = (q, \beta)$ and $\pi'_0 = (q^{\text{pop}}_{\epsilon}, \beta)$, $\Pi'_0 = \Pi_0$, and $\Gamma'_0 = \Gamma$. Further $(q^{\text{pop}}_{\epsilon}, \beta) \in F(q, \beta)$ and thus $(\pi \parallel \Pi \lhd \Gamma, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.

- *Case:* $r = (q^{\text{push}}_{\epsilon}, \epsilon) \xrightarrow{\epsilon} (q, \epsilon)$.
  Clearly $\pi_0 = (q^{\text{push}}_{\epsilon}, \beta)$, $\pi = (q, \beta)$ and $\pi'_0 = (q, \beta)$, $\Pi'_0 = \Pi_0$, and $\Gamma'_0 = \Gamma$. Further $(q, \beta) \in F(q, \beta)$ and thus $(\pi \parallel \Pi \lhd \Gamma, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.

- *Case:* $r = (q^{\text{pop}}_{\beta}, A) \xrightarrow{\epsilon} (q^{\text{pop}}_{\beta \cdot A}, \epsilon)$.
  Clearly $\pi_0 = (q^{\text{pop}}_{\beta}, A \cdot \beta')$, $\pi = (q, \beta \cdot A \cdot \beta')$ and $\pi'_0 = (q^{\text{pop}}_{\beta \cdot A}, \beta')$, $\Pi'_0 = \Pi_0$, and $\Gamma'_0 = \Gamma$. Further $(q^{\text{pop}}_{\beta \cdot A}, \beta') \in F(q, \beta \cdot A \cdot \beta')$ and thus $(\pi \parallel \Pi \lhd \Gamma, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.

- *Case:* $r = (q^{\text{push}}_{\beta\cdot A}, \epsilon) \xrightarrow{\epsilon} (q^{\text{push}}_{\beta}, A)$.
  Clearly $\pi_0 = (q^{\text{push}}_{\beta\cdot A}, \beta')$, $\pi = (q, \beta\cdot A\cdot\beta')$ and $\pi'_0 = (q^{\text{push}}_{\beta}, A\cdot\beta')$, $\Pi'_0 = \Pi_0$, and $\Gamma'_0 = \Gamma$. Further $(q^{\text{pop}}_{\beta}, \beta') \in F(q, \beta\cdot A\cdot\beta')$ and thus $(\pi \parallel \Pi \lhd \Gamma, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.

- *Case:* $r = (q^{\text{pop}}_{\beta}, \epsilon) \xrightarrow{\lambda} (q'^{\text{push}}_{\beta'}, \epsilon)$.
  Clearly $\pi_0 = (q^{\text{pop}}_{\beta}, \beta'')$, $\pi = (q, \beta\cdot\beta'')$ and $\pi'_0 = (q'^{\text{push}}_{\beta'}, \beta'')$. Further there is a rule $(q,\beta) \xrightarrow{\lambda'} (q', \beta') \in \mathcal{R}$. Let us do case analysis on λ

  - *Case:* $\lambda = \epsilon$.
    Then $\Pi'_0 = \Pi_0, \Gamma'_0 = \Gamma$ and $\lambda' = \epsilon$. And $(q, \beta\beta'') \parallel \Pi \lhd \Gamma \xrightarrow{r}_{\mathcal{P}} (q', \beta'\beta'') \parallel \Pi \lhd \Gamma$. Clearly $(q'^{\text{push}}_{\beta}, \beta'') \in F(q', \beta'\beta'')$ and thus $((q', \beta'\beta'') \parallel \Pi \lhd \Gamma, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.
  - *Case:* $\lambda = c\,!\,m$
    Then $\Pi'_0 = \Pi_0, \Gamma'_0 = \Gamma \oplus [c \mapsto [m]]$ and $\lambda' = c\,!\,m$. And $(q, \beta\beta'') \parallel \Pi \lhd \Gamma \xrightarrow{r}_{\mathcal{P}} (q', \beta'\beta'') \parallel \Pi \lhd \Gamma'_0$. Clearly $(q'^{\text{push}}_{\beta}, \beta'') \in F(q', \beta'\beta'')$ and thus $((q', \beta'\beta'') \parallel \Pi \lhd \Gamma'_0, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.
  - *Case:* $\lambda = c\,?\,m$
    Then $\Pi'_0 = \Pi_0, \Gamma = \Gamma'_0 \oplus [c \mapsto [m]]$ and $\lambda' = c\,?\,m$. And $(q, \beta\beta'') \parallel \Pi \lhd \Gamma \xrightarrow{r}_{\mathcal{P}} (q', \beta'\beta'') \parallel \Pi \lhd \Gamma'_0$. Clearly $(q'^{\text{push}}_{\beta}, \beta'') \in F(q', \beta'\beta'')$ and thus $((q', \beta'\beta'') \parallel \Pi \lhd \Gamma'_0, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.
  - *Case:* $\lambda = \nu(q''^{\text{push}}_{\beta'''}, \epsilon)$
    Then $\Pi'_0 = (q''^{\text{push}}_{\beta'''}, \epsilon) \parallel \Pi_0, \Gamma = \Gamma'_0$ and $\lambda' = \nu(q'', \beta''')$. And $(q, \beta\beta'') \parallel \Pi \lhd \Gamma \xrightarrow{r}_{\mathcal{P}} (q', \beta'\beta'') \parallel (q'', \beta''') \parallel \Pi \lhd \Gamma$. Firstly $(q'^{\text{push}}_{\beta}, \beta'') \in F(q', \beta'\beta'')$ and $(q''^{\text{push}}_{\beta'''}, \epsilon) \in F(q'', \beta''')$. Thus $((q', \beta'\beta'') \parallel (q'', \beta''') \parallel \Pi \lhd \Gamma'_0, \pi'_0 \parallel \Pi'_0 \lhd \Gamma'_0) \in \mathcal{B}$ which is what we wanted to prove.

Hence $\mathcal{B}^{-1}$ is a weak reflexive simulation and hence $\mathcal{B}$ is a weak reflexive bisimulation.

Let us now define $\mathcal{F}^{\text{pnf}}(\mathcal{P}) = (\mathcal{Q}', \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R}' \cup \mathcal{R}_{\text{cov}})$ where

$$\mathcal{R}_{\text{cov}} = \left\{ \begin{array}{l|l} (q^{\text{pop}}_{\epsilon}, \epsilon) \xrightarrow{\epsilon} (q^{\text{cov}}, \epsilon), & \\ (q^{\text{push}}_{\epsilon}, \epsilon) \xrightarrow{\epsilon} (q^{\text{cov}}, \epsilon), & q^{\text{pop}}_{\epsilon}, q^{\text{push}}_{\epsilon'} \in \mathcal{Q}', \\ (q^{\text{pop}}_{A\cdot\beta}, \epsilon) \xrightarrow{\epsilon} (q^{\text{cov}}, A), & q^{\text{pop}}_{A\cdot\beta}, q^{\text{push}}_{A'\cdot\beta'} \in \mathcal{Q}', \\ (q^{\text{push}}_{A'\cdot\beta'}, \epsilon) \xrightarrow{\epsilon} (q^{\text{cov}}, A'), & q \in \mathcal{Q} \\ (q, \epsilon) \xrightarrow{\epsilon} (q^{\text{cov}}, \epsilon), & \end{array} \right\}.$$

Adding the rules $\mathcal{R}_{\text{cov}}$ to $\mathcal{P}_0$ (which are non-reversible) only changes which configurations are reachable/coverable by a one step transition. Obviously $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ remains polynomial time computable from $\mathcal{P}$

Suppose that $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ is a simple coverbility query where $\Pi = (q_1, \beta_1) \parallel \cdots \parallel (q_k, \beta_k)$ and $\beta_i \in \mathcal{A} \cup \{\epsilon\}$ then let $\mathcal{F}^{\text{pnf}}(\mathcal{Q}) = (\mathcal{F}^{\text{pnf}}(\mathcal{P}), \Pi_0 \lhd \Gamma_0, \Pi' \lhd \Gamma)$ be a simple coverability query such that $\Pi' = (q_1^{\text{cov}}, \beta_1) \parallel \cdots \parallel (q_k^{\text{cov}}, \beta_k)$.

Suppose $\mathcal{Q}$ is a yes-instance then $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* \Pi_1 \lhd \Gamma_1$ such that $\Pi \lhd \Gamma \leq_{ACPS} \Pi_1 \lhd \Gamma_1$. Since $\mathcal{B}$ is a reflexive weak bisimulation we know $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}_0}^* \Pi_1' \lhd \Gamma_1$ such that $(\Pi_1 \lhd \Gamma_1, \Pi_1' \lhd \Gamma_1) \in \mathcal{B}$. Since $\Pi \lhd \Gamma \leq_{ACPS} \Pi_1 \lhd \Gamma_1$ we know that $\Pi_1 = (q_1, \beta_1') \parallel \cdots \parallel (q_k, \beta_k') \parallel \Pi_2$ such that for all $i \in \langle k \rangle$ either $\beta_i = \epsilon$ or $\beta_i = A_i \in \mathcal{A}$ and $\beta_1' = A_i\, \beta_1''$. Hence we can deduce that $\Pi_1' = \pi_1' \parallel \cdots \parallel \pi_k' \parallel \Pi_2'$ such that $\pi_i' \in F(q_i, \beta_i')$ for all $i \in \langle k \rangle$. Thus it is easy to see that $\Pi_1' \lhd \Gamma_1 \to_{\mathcal{F}^{\text{pnf}}(\mathcal{P})}^* (q_1^{\text{cov}}, \beta_1''') \parallel \cdots \parallel (q_k^{\text{cov}}, \beta_k''') \parallel \Pi_2' \lhd \Gamma_1$ such that for all $i \in \langle k \rangle$ it is the case that $\beta_i''' = A\beta_i''''$ and $\beta_i' = A\beta_i^\dagger$ and hence $\mathcal{F}^{\text{pnf}}(\mathcal{Q})$ is a yes-instance.

Conversely, suppose $\mathcal{F}^{\text{pnf}}(\mathcal{Q})$ is a yes instance then $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{F}^{\text{pnf}}(\mathcal{P})}^* (q_1^{\text{cov}}, \beta_1') \parallel \cdots \parallel (q_k^{\text{cov}}, \beta_k') \parallel \Pi_1' \lhd \Gamma_1$ such that for all $i \in \langle k \rangle$ either $\beta_i = \epsilon$ or $\beta_i = A_i$ and $\beta_i' = A_i\, \beta_i'$. Hence clearly (by reversing transitions from $\mathcal{R}_{\text{cov}}$) $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}_0}^* \pi_1 \parallel \cdots \parallel \pi_k \parallel \Pi_1' \lhd \Gamma_1$ where $\pi_i' \in F(q_i, \beta_i'')$ for some $\beta_1'', \ldots, \beta_k''$ such that either $\beta_i = \epsilon$ or $\beta_i'' = A_i\beta_i'''$ for all $i \in \langle k \rangle$. Since $\mathcal{B}$ is a reflexive weak bisimulation we know that $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* (q_1, \beta_1'') \parallel \cdots \parallel (q_k, \beta_k'') \parallel \Pi_1 \lhd \Gamma_1$. Hence $\mathcal{Q}$ is a yes-instance. We can thus conclude that $\mathcal{Q}$ is a yes-instance iff $\mathcal{F}^{\text{pnf}}(\mathcal{Q})$ is a yes-instance.

For boundedness, let $\Pi^0 \lhd \Gamma^0$ be a start configuration and let $\mathcal{F}^{\text{pnf}}\big(\Pi^0 \lhd \Gamma^0\big) = \Pi^0 \lhd \Gamma^0$. Suppose that $\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* \Pi \lhd \Gamma\}$ is a finite set. We notice that for all $\Pi \lhd \Gamma$ the set $\{\Pi' \lhd \Gamma : (\Pi \lhd \Gamma, \Pi' \lhd \Gamma) \in \mathcal{B}\}$ is finite. Thus using that $\mathcal{B}$ is a reflexive weak bisimulation we can infer that $\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}_0}^* \Pi \lhd \Gamma\}$ is a finite set. Since the rules in $\mathcal{R}_{\text{cov}}$ adds only a finite number of reachable configurations we can conclude $\left\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{F}^{\text{pnf}}(\mathcal{P})}^* \Pi \lhd \Gamma\right\}$ is a finite set and thus $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ is bounded from $\mathcal{F}^{\text{pnf}}\big(\Pi^0 \lhd \Gamma^0\big)$.

Conversely, suppose $\left\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{F}^{\text{pnf}}(\mathcal{P})}^* \Pi \lhd \Gamma\right\}$ is a finite set. Then since the rules in $\mathcal{R}_{\text{cov}}$ adds only a finite number of reachable configurations we can infer $\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}_0}^* \Pi \lhd \Gamma\}$ is a finite set. We further note: $\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* \Pi \lhd \Gamma\} \subseteq \left\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}_0}^* \Pi \lhd \Gamma\right\}$ and thus $\mathcal{P}$ is bounded from $\Pi_0 \lhd \Gamma_0$. Thus $\mathcal{P}$ is bounded from $\Pi_0 \lhd \Gamma_0$ if and only if $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ is bounded from $\mathcal{F}^{\text{pnf}}(\Pi_0 \lhd \Gamma_0)$.

For termination, let $\Pi_0 \lhd \Gamma_0$ be a start configuration and define again $\mathcal{F}^{\text{pnf}}\big(\Pi^0 \lhd \Gamma^0\big) = \Pi^0 \lhd \Gamma^0$. Suppose there exists an infinite path $\boldsymbol{s}$ in $\mathcal{P}$ starting from $\Pi_0 \lhd \Gamma_0$. Since $\mathcal{B}$ is a weak reflexive bisimulation and $\boldsymbol{s}$ uses an infinite sequence of labels it is clear that there is a path $\boldsymbol{s}'$ in $\mathcal{P}_0$ and $\boldsymbol{s}'$ is also an infinite path. The path $\boldsymbol{s}'$ is clearly also a path of $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ hence $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ is non-terminating from $\mathcal{F}^{\text{pnf}}\big(\Pi^0 \lhd \Gamma^0\big)$. Conversely, suppose that $\boldsymbol{s}'$ is an infinite path in $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ starting from $\Pi_0 \lhd \Gamma_0$. Since a path can only be finitely extended by rules in $\mathcal{R}_{\text{cov}}$ we can deduce that there is also an infinite path $\boldsymbol{s}''$ from $\Pi_0 \lhd \Gamma_0$ in $\mathcal{P}_0$. If $\boldsymbol{s}''$ gives rise to an infinite sequence of labels then, since $\mathcal{B}$ is a weak reflexive bisimulation, we clearly obtain a path $\boldsymbol{s}$ in $\mathcal{P}$ that is also infinite (since all transitions in $\mathcal{P}$ carry a label). Suppose for a contradiction that $\boldsymbol{s}''$ gives rise only for a finite sequence of labels. This implies there exists an infinite path $\boldsymbol{s}_0$ in $\mathcal{P}_0$ such that for all i the transition $\boldsymbol{s}(i) \xrightarrow{\epsilon}_{\mathcal{P}} \boldsymbol{s}(i+1)$ is an ϵ-transition. Inspecting the definition $\mathcal{P}_0$ we see this is implies all rules used must of the form $(q, \beta) \xrightarrow{\epsilon} (q', \beta')$. We can conclude that this is impossible since there are no cycles in $\mathcal{P}_0$'s rules with no side effects. Hence we can

conclude that $\mathcal{P}$ is non-terminating from $\Pi^0 \lhd \Gamma^0$ if, and only if, $\mathcal{F}^{\text{pnf}}(\mathcal{P})$ is non-terminating from $\mathcal{F}^{\text{pnf}}(\Pi_0 \lhd \Gamma_0)$. □

We can now use a summarisation-inspired idea to encode control-states into the stack alphabet:

Lemma 15. Given an ACPS $\mathcal{P}$ in pre-normal form, a simple coverability query $\mathcal{Q}$ and a start configuration $\Pi^0 \lhd \Gamma^0$ there exists ACPS $\mathcal{F}^{\text{nf'}}(\mathcal{P})$ satisfying: for all rules $(q,\beta) \xrightarrow{\lambda} (q',\beta')$ of $\mathcal{F}^{\text{nf'}}(\mathcal{P})$ (i) $q = q'$, (ii) $\beta = A \in \mathcal{A}$, (iii) if $\lambda \neq \epsilon$ then $\beta' = A' \in \mathcal{A}$, otherwise (iv) $\beta' \in \{\epsilon, A', B\,C : A', B, C \in \mathcal{A}\}$, and (v) $\lambda = \nu(q'',\beta)$ then $q'' = q$, and $\beta \in \mathcal{A}$; simple coverability query $\mathcal{F}^{\text{nf'}}(\mathcal{Q})$; and start configuration $\mathcal{F}^{\text{nf'}}\left(\Pi^0 \lhd \Gamma^0\right)$ — all polynomial-time computable — such that: (A) $\mathcal{Q}$ is a yes-instance if, and only if, $\mathcal{F}^{\text{nf'}}(\mathcal{Q})$ is a yes-instance; and (B) $\mathcal{P}$ is bounded (terminating) from $\Pi^0 \lhd \Gamma^0$ if, and only if, $\mathcal{F}^{\text{nf'}}(\mathcal{P})$ is bounded (terminating respectively) from $\mathcal{F}^{\text{nf'}}\left(\Pi^0 \lhd \Gamma^0\right)$.

Proof. Suppose $\mathcal{P} = (\mathcal{Q}, \mathcal{A}, \mathit{Chan}, \mathit{Msg}, \mathcal{R})$ is an ACPS in pre-normal form. We then define $\mathcal{P}' = (\mathcal{Q}', \mathcal{A}', \mathit{Chan}, \mathit{Msg}, \mathcal{R}')$ where $\mathcal{Q}' = \{q_0\}$ and q_0 is a fresh control state, $\mathcal{A}' = \{A^{(q,q')} \mid q, q' \in \mathcal{Q}, A \in \mathcal{Q} \cup \{\Theta\}\}$, where Θ is a fresh symbol, and $\mathcal{R}'$ is obtained from $\mathcal{R}$ as follows

$$\begin{aligned}\mathcal{R}' = &\left\{ A^{(q,q'')} \xrightarrow{\lambda'} A^{(q',q'')} \;\middle|\; \begin{array}{l}(q,\epsilon) \xrightarrow{\lambda} (q',\epsilon) \in \mathcal{R},\\ A \in \mathcal{A} \cup \{\Theta\}, q'', q'_0 \in \mathcal{Q},\\ \text{if } \lambda = \nu(q_0,\epsilon) \text{ then } \lambda' = \nu\Theta^{(q_0,q'_0)}\\ \text{otherwise } \lambda = \lambda'\end{array}\right\}\\ &\cup \left\{ A^{(q,q'')} \xrightarrow{\epsilon} B^{(q',q''')}A^{(q''',q'')} \;\middle|\; \begin{array}{l}(q,\epsilon) \xrightarrow{\epsilon} (q',B) \in \mathcal{R},\\ A \in \mathcal{A} \cup \{\Theta\},\\ q'', q''' \in \mathcal{Q}\end{array}\right\}\\ &\cup \left\{ A^{(q,q')} \xrightarrow{\epsilon} \epsilon \mid (q,A) \xrightarrow{\epsilon} (q',\epsilon) \in \mathcal{R} \right\}\end{aligned}$$

where rather than writing (q_0, A) we just write A. It is easy to see that $\mathcal{P}'$ is polynomial time computable from $\mathcal{P}$.

We represent the state of a pushdown process by the following function:

$$F(q, A_1A_2A_3\cdots A_n) = \left\{ A_1^{(q,q_1)} A_2^{(q_1,q_2)} A_3^{(q_2,q_3)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{q_n,q_{n+1}} \mid \Xi \right\}$$

where $\Xi = q_1, \ldots q_{n+1} \in \mathcal{Q}$. We use the former to represent a $\mathcal{P}$-configuration as a set:

$$G(\Pi \lhd \Gamma) = \left\{ \Pi' \lhd \Gamma \;\middle|\; \begin{array}{l}\Pi = \pi_1 \parallel \cdots \parallel \pi_n,\\ \Pi' = \pi'_1 \parallel \cdots \parallel \pi'_n,\\ \forall i \in \langle n \rangle .\, \pi'_i \in F(\pi_i)\end{array}\right\}.$$

Further we define a relation of configurations and sets of configurations:

$$\mathcal{B} = \{(\Pi \lhd \Gamma, G(\Pi \lhd \Gamma))\}.$$

Let us define a *co-universal powerset lifting* of the transition system induced by $\mathcal{P}'$ and $\rightarrow_{\mathcal{P}_0}$ as follows $\mathscr{P}[\mathcal{P}'] := (\mathscr{P}[\mathbb{M}[\mathcal{Q}' \times \mathcal{A}^*]], \rightarrow_{\mathscr{P}[\mathcal{P}']})$ where $S \rightarrow_{\mathscr{P}[\mathcal{P}']} S'$ just if for all $s' \in S'$ there exists $s \in S$ such that $s \rightarrow_{\mathcal{P}_0} s'$. Further we temporarily label $\mathcal{P}$ and $\mathscr{P}[\mathcal{P}']$ by $\mathcal{P}$-configurations as follows: if $s \rightarrow_{\mathcal{P}} s'$ we label by s' the transition $s \xrightarrow{s'}_{\mathcal{P}} s'$. If $S \rightarrow_{\mathscr{P}[\mathcal{P}']} S'$ such that $S' = G(s')$ for some $\mathcal{P}$-configuration s' we label by s' the transition $S \xrightarrow{s'}_{\mathscr{P}[\mathcal{P}']} S'$.

We will show that $\mathcal{B}$ is a bisimulation relation for $\mathcal{P}$ and $\mathscr{P}[\mathcal{P}']$. As a first step let us give a proof that $\mathcal{B}$ is a simulation relation: Suppose $(\pi \parallel \Pi \lhd \Gamma, G(\pi \parallel \Pi \lhd \Gamma)) \in \mathcal{B}$ and $\pi \parallel \Pi \lhd \Gamma \xrightarrow{l}_{\mathcal{P}} \pi' \parallel \Pi' \lhd \Gamma'$ using rule $r \in \mathcal{R}$. We know that $l = \pi' \parallel \Pi' \lhd \Gamma'$. So let $\pi'_1 \parallel \Pi'_1 \lhd \Gamma' \in G(\pi' \parallel \Pi' \lhd \Gamma')$ such that $\Pi'_1 \lhd \Gamma' \in G(\Pi' \lhd \Gamma')$ and $\pi'_1 \in F(\pi')$. Let us perform a case analysis on r:

- *Case:* $r = (q, \epsilon) \xrightarrow{\epsilon} (q', B)$.
  In this case $\pi = (q, A_1 \cdots A_n)$, $\pi' = (q', B\, A_1 \cdots A_n)$, $\Gamma = \Gamma'$ and $\Pi = \Pi'$. From this we can deduce: $\pi'_1 = B^{(q',q_1)}\, A_1^{(q_1,q_2)} \cdots A_n^{(q_n,q_{n+1})} \Theta^{(q_{n+1},q_{n+2})}$ for some $q_1, \ldots, q_{n+2}$. Let $\pi_1 = A_1^{(q,q_2)} \cdots A_n^{(q_n,q_{n+1})} \Theta^{(q_{n+1},q_{n+2})}$ then clearly $\pi_1 \parallel \Pi'_1 \lhd \Gamma \rightarrow_{\mathcal{P}_0} \pi'_1 \parallel \Pi'_1 \lhd \Gamma$ using rule $A_1^{(q,q_2)} \xrightarrow{\epsilon} B^{(q',q_1)}\, A_1^{(q_1,q_2)}$. Further $\pi_1 \in F(\pi)$ and hence $\pi_1 \parallel \Pi'_1 \lhd \Gamma \in G(\pi \parallel \Pi' \lhd \Gamma) = G(\pi \parallel \Pi \lhd \Gamma)$. And so since $\pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ is an arbitrary element of $G(\pi' \parallel \Pi' \lhd \Gamma')$ we can deduce $G(\pi \parallel \Pi \lhd \Gamma) \xrightarrow{l}_{\mathscr{P}[\mathcal{P}']} G(\pi' \parallel \Pi' \lhd \Gamma')$. and $(\pi' \parallel \Pi' \lhd \Gamma', G(\pi' \parallel \Pi' \lhd \Gamma')) \in \mathcal{B}$.

- *Case:* $r = (q, A) \xrightarrow{\epsilon} (q', \epsilon)$.
  In this case $\pi = (q, A\, A_1 \cdots A_n)$, $\pi' = (q', A_1 \cdots A_n)$, $\Gamma = \Gamma'$ and $\Pi = \Pi'$. From this we can deduce: $\pi'_1 = A_1^{(q',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ for some $q_1, \ldots, q_{n+1}$. Let $\pi_1 = A^{(q,q')}\, A_1^{(q',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ then clearly $\pi_1 \parallel \Pi'_1 \lhd \Gamma \rightarrow_{\mathcal{P}_0} \pi'_1 \parallel \Pi'_1 \lhd \Gamma$ using rule $A^{(q,q')} \xrightarrow{\epsilon} \epsilon$. Further $\pi_1 \in F(\pi)$ and hence $\pi_1 \parallel \Pi'_1 \lhd \Gamma \in G(\pi \parallel \Pi' \lhd \Gamma) = G(\pi \parallel \Pi \lhd \Gamma)$. And so since $\pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ is an arbitrary element of $G(\pi' \parallel \Pi' \lhd \Gamma')$ we can deduce $G(\pi \parallel \Pi \lhd \Gamma) \xrightarrow{l}_{\mathscr{P}[\mathcal{P}']} G(\pi' \parallel \Pi' \lhd \Gamma')$. and $(\pi' \parallel \Pi' \lhd \Gamma', G(\pi' \parallel \Pi' \lhd \Gamma')) \in \mathcal{B}$.

- *Case:* $r = (q, \epsilon) \xrightarrow{\lambda} (q', \epsilon)$.
  In this case $\pi = (q, A_1 \cdots A_n)$, $\pi' = (q', A_1 \cdots A_n)$. From this we can deduce: $\pi'_1 = A_1^{(q',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ for some $q_1, \ldots, q_{n+1}$. Let $\pi_1 = A_1^{(q,q_1)} \cdots A_n^{(q_{n-1},q_n)}$ $\Theta^{(q_n,q_{n+1})}$. Let us perform a case analysis on λ:

  - *Case:* $\lambda = \epsilon$.
    We have $\Pi = \Pi', \Gamma = \Gamma'$, and $\pi_1 \parallel \Pi'_1 \lhd \Gamma \rightarrow_{\mathcal{P}_0} \pi'_1 \parallel \Pi'_1 \lhd \Gamma$ using rule $A_1^{(q,q_1)} \xrightarrow{\epsilon} A_1^{(q',q_1)}$. Further $\pi_1 \in F(\pi)$ and hence $\pi_1 \parallel \Pi'_1 \lhd \Gamma \in G(\pi \parallel \Pi' \lhd \Gamma) = G(\pi \parallel \Pi \lhd \Gamma)$. Since $\pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ is an arbitrary element of $G(\pi' \parallel \Pi' \lhd \Gamma')$ we can deduce $G(\pi \parallel \Pi \lhd \Gamma) \xrightarrow{l}_{\mathscr{P}[\mathcal{P}']} G(\pi' \parallel \Pi' \lhd \Gamma')$. and $(\pi' \parallel \Pi' \lhd \Gamma', G(\pi' \parallel \Pi' \lhd \Gamma')) \in \mathcal{B}$.

- *Case:* $\lambda = c\,!\,m$.
  We have $\Pi = \Pi'$, $\Gamma' = \Gamma \oplus [c \mapsto [m]]$, and $\pi_1 \parallel \Pi'_1 \lhd \Gamma \to_{\mathcal{P}_0} \pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ using rule $A_1^{(q,q_1)} \xrightarrow{c!m} A_1^{(q',q_1)}$. Further $\pi_1 \in F(\pi)$ and hence $\pi_1 \parallel \Pi'_1 \lhd \Gamma \in G(\pi \parallel \Pi' \lhd \Gamma) = G(\pi \parallel \Pi \lhd \Gamma)$. Since $\pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ is an arbitrary element of $G(\pi' \parallel \Pi' \lhd \Gamma')$ we can deduce $G(\pi \parallel \Pi \lhd \Gamma) \xrightarrow{l}_{\mathscr{P}[\mathcal{P}']} G(\pi' \parallel \Pi' \lhd \Gamma')$. and $(\pi' \parallel \Pi' \lhd \Gamma', G(\pi' \parallel \Pi' \lhd \Gamma')) \in \mathcal{B}$.
- *Case:* $\lambda = c\,?\,m$.
  We have $\Pi = \Pi'$, $\Gamma = \Gamma' \oplus [c \mapsto [m]]$, and $\pi_1 \parallel \Pi'_1 \lhd \Gamma \to_{\mathcal{P}_0} \pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ using rule $A_1^{(q,q_1)} \xrightarrow{c?m} A_1^{(q',q_1)}$. Further $\pi_1 \in F(\pi)$ and hence $\pi_1 \parallel \Pi'_1 \lhd \Gamma \in G(\pi \parallel \Pi' \lhd \Gamma) = G(\pi \parallel \Pi \lhd \Gamma)$. Since $\pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ is an arbitrary element of $G(\pi' \parallel \Pi' \lhd \Gamma')$ we can deduce $G(\pi \parallel \Pi \lhd \Gamma) \xrightarrow{l}_{\mathscr{P}[\mathcal{P}']} G(\pi' \parallel \Pi' \lhd \Gamma')$. and $(\pi' \parallel \Pi' \lhd \Gamma', G(\pi' \parallel \Pi' \lhd \Gamma')) \in \mathcal{B}$.
- *Case:* $\lambda = \nu q'', \epsilon$.
  We have $\Pi' = (q'', \epsilon) \parallel \Pi$, $\Gamma = \Gamma'$. Hence $\Pi'_1 = \pi_0 \parallel \Pi_1$ such that $\Pi_1 \lhd \Gamma \in G(\Pi \lhd \Gamma)$ and $\pi_0 \in F(q'', \epsilon)$ which implies $\pi_0 = \Theta^{q'',q'''}$ for some q'''. Thus $\pi_1 \parallel \Pi_1 \lhd \Gamma \to_{\mathcal{P}_0} \pi'_1 \parallel \Theta^{q'',q'''} \parallel \Pi_1 \lhd \Gamma$ using rule $A_1^{(q,q_1)} \xrightarrow{\nu\Theta^{q'',q'''}} A_1^{(q',q_1)}$. Further $\pi_1 \in F(\pi)$ and hence $\pi_1 \parallel \Pi'_1 \lhd \Gamma \in G(\pi \parallel \Pi \lhd \Gamma)$. Since $\pi'_1 \parallel \Pi'_1 \lhd \Gamma'$ is an arbitrary element of $G(\pi' \parallel \Pi' \lhd \Gamma')$ we can deduce $G(\pi \parallel \Pi \lhd \Gamma) \xrightarrow{l}_{\mathscr{P}[\mathcal{P}']} G(\pi' \parallel \Pi' \lhd \Gamma')$. and $(\pi' \parallel \Pi' \lhd \Gamma', G(\pi' \parallel \Pi' \lhd \Gamma')) \in \mathcal{B}$.

We can thus conclude that $\mathcal{B}$ is a simulation relation.

For a proof that $\mathcal{B}^{-1}$ is a simulation relation we will first prove a little lemma:

Lemma. $\{(s_0, s) : s_0 \in G(s)\}$ is a simulation relation.

Proof. Suppose $\pi_0 \parallel \Pi_0 \lhd \Gamma \in G(\pi \parallel \Pi \lhd \Gamma)$ and $\pi_0 \parallel \Pi_0 \lhd \Gamma \to_{\mathcal{P}'} \pi'_0 \parallel \Pi'_0 \lhd \Gamma'$ using rule $r \in \mathcal{R}'$. Let us make a case analysis on r:

- *Case:* $r = A^{(q,q')} \xrightarrow{\epsilon} \epsilon$.
  Then $\Pi'_0 = \Pi_0, \Gamma = \Gamma', \pi_0 = A^{(q,q')} A_1^{(q',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ and $\pi'_0 = A_1^{(q',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ for some $q_1, \ldots, q_{n+1}$. We can deduce that $\pi = (q, A\, A_1 \cdots A_n)$ and we can let $\pi' = (q', A_1 \cdots A_n)$ so that $\pi'_0 \in F(\pi')$. Further we know by construction that $(q, A) \xrightarrow{\epsilon} (q', \epsilon) \in \mathcal{R}$ and thus $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi' \parallel \Pi \lhd \Gamma$ and $\pi'_0 \parallel \Pi'_0 \lhd \Gamma' \in G(\pi' \parallel \Pi' \lhd \Gamma') = G(\pi' \parallel \Pi \lhd \Gamma)$ which is what we wanted to prove.
- *Case:* $r = A^{(q,q'')} \xrightarrow{\epsilon} B^{(q',q''')} A^{(q''',q'')}$.
  Then $\Pi'_0 = \Pi_0, \Gamma = \Gamma', \pi_0 = A^{(q,q'')} A_1^{(q'',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ and $\pi'_0 = B^{(q',q''')} A^{(q''',q'')} A_1^{(q'',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ for some $q_1, \ldots, q_{n+1}$. We can deduce that $\pi = (q, A\, A_1 \cdots A_n)$ and we can let $\pi' = (q', B\, A\, A_1 \cdots A_n)$ so that $\pi'_0 \in F(\pi')$. Further we know by construction that $(q, \epsilon) \xrightarrow{\epsilon} (q', B) \in \mathcal{R}$ and thus $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi' \parallel \Pi \lhd \Gamma$ and $\pi'_0 \parallel \Pi'_0 \lhd \Gamma' \in G(\pi' \parallel \Pi' \lhd \Gamma') = G(\pi' \parallel \Pi \lhd \Gamma)$ which is what we wanted to prove.

- *Case:* $r = A^{(q,q'')} \xrightarrow{\lambda'} A^{(q',q'')}$.
  Then $\Pi'_0 = \Pi_0$, $\Gamma = \Gamma'$, $\pi_0 = A^{(q,q'')} A_1^{(q'',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ and $\pi'_0 = A^{(q',q'')}$ $A_1^{(q'',q_1)} \cdots A_n^{(q_{n-1},q_n)} \Theta^{(q_n,q_{n+1})}$ for some $q_1, \ldots, q_{n+1}$. We can deduce that $\pi = (q, A\, A_1$ $\cdots A_n)$ and we can let $\pi' = (q', A\, A_1 \cdots A_n)$ so that $\pi'_0 \in F(\pi')$. Further we know by construction that $(q, \epsilon) \xrightarrow{\lambda} (q', \epsilon) \in \mathcal{R}$. Let us perform a case analysis on λ'.

  - *Case:* $\lambda' = \epsilon$.
    Then $\lambda = \epsilon$, $\Pi = \Pi'$, $\Gamma = \Gamma'$ and thus $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi' \parallel \Pi \lhd \Gamma$ and $\pi'_0 \parallel \Pi'_0 \lhd \Gamma' \in G(\pi' \parallel \Pi' \lhd \Gamma') = G(\pi' \parallel \Pi \lhd \Gamma)$ which is what we wanted to prove.
  - *Case:* $\lambda' = c\,!\,m$.
    Then $\lambda = c\,!\,m$, $\Pi = \Pi'$, $\Gamma' = \Gamma \oplus [c \mapsto [m]]$ and thus $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi' \parallel \Pi \lhd \Gamma'$ and $\pi'_0 \parallel \Pi'_0 \lhd \Gamma' \in G(\pi' \parallel \Pi' \lhd \Gamma') = G(\pi' \parallel \Pi \lhd \Gamma')$ which is what we wanted to prove.
  - *Case:* $\lambda' = c\,?\,m$.
    Then $\lambda = c\,?\,m$, $\Pi = \Pi'$, $\Gamma = \Gamma' \oplus [c \mapsto [m]]$ and thus $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi' \parallel \Pi \lhd \Gamma'$ and $\pi'_0 \parallel \Pi'_0 \lhd \Gamma' \in G(\pi' \parallel \Pi' \lhd \Gamma') = G(\pi' \parallel \Pi \lhd \Gamma')$ which is what we wanted to prove.
  - *Case:* $\lambda' = \nu\Theta^{(q''',q'''')}$.
    Then $\lambda = \nu(q''', \epsilon)$, $\Pi' = (q''', \epsilon) \parallel \Pi$, $\Gamma' = \Gamma$ and thus $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi'(q''', \epsilon) \parallel\parallel$ $\Pi \lhd \Gamma'$ and $\pi'_0 \parallel \Pi'_0 \lhd \Gamma' \in G(\pi' \parallel \Pi' \lhd \Gamma') = G(\pi' \parallel \Pi \lhd \Gamma')$ which is what we wanted to prove.

which concludes the proof. □

Now we can use the above Lemma to show that $\mathcal{B}^{-1}$ is a simulation relation. Hence suppose $(\pi \parallel \Pi \lhd \Gamma, G(\pi \parallel \Pi \lhd \Gamma)) \in \mathcal{B}$ and $G(\pi \parallel \Pi \lhd \Gamma) \xrightarrow{l}_{\mathscr{P}[\mathcal{P}']} G(\pi' \parallel \Pi' \lhd \Gamma')$ then clearly $G(\pi' \parallel \Pi' \lhd \Gamma') \neq \emptyset$ and hence we have $s \in G(\pi \parallel \Pi \lhd \Gamma)$ and $s' \in G(\pi' \parallel \Pi' \lhd \Gamma')$ such that $s \to_{\mathcal{P}'} s'$ from which the above Lemma let's us conclude that $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi' \parallel \Pi' \lhd \Gamma'$. Hence we can conclude that $\mathcal{B}$ is a bisimulation.

Let us define $\mathcal{F}^{\text{nf}'}(\mathcal{P}) = (\mathcal{Q}', \mathcal{A}' \cup \mathcal{A}_{\text{cov}}, \mathit{Chan}, \mathit{Msg}, \mathcal{R}' \cup \mathcal{R}_{\text{cov}})$ where $\mathcal{A}_{\text{cov}} = \{\Theta_{\text{cov}}^{(q,A)}, \Theta_{\text{cov}}^{(q,\epsilon)} : q \in \mathcal{Q}, A \in \mathcal{A}\}$ and

$$\mathcal{R}_{\text{cov}} = \left\{ A^{q,q'} \xrightarrow{\epsilon} \Theta_{\text{cov}}^{(q,A)}, A^{q,q'} \xrightarrow{\epsilon} \Theta_{\text{cov}}^{(q,\epsilon)} : q, q' \in \mathcal{Q}, A \in \mathcal{A} \right\}$$
$$\cup \left\{ \Theta^{q,q'} \xrightarrow{\epsilon} \Theta_{\text{cov}}^{(q,\epsilon)} : q, q' \in \mathcal{Q} \right\}.$$

It is easy to see that $\mathcal{F}^{\text{nf}'}(\mathcal{P})$ is polynomial-time computable from $\mathcal{P}$.

Now suppose $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi_{\text{cov}} \lhd \Gamma_{\text{cov}})$ is a simple coverability query. Since $\mathcal{Q}$ is simple we may assume that $\Pi_0 = (q_1^0, \epsilon) \parallel \cdots \parallel (q_k^0, \epsilon)$ and $\Pi_{\text{cov}} = (q_1, \beta_1) \parallel \cdots (q_n, \beta_n)$ such that $\beta_i \in \mathcal{A} \cup \{\epsilon\}$ since a trivial polynomial-time transform $\mathcal{P}$ may set up a general $\Pi_0 \lhd \Gamma_0$ from a simple query $\mathcal{Q}$.

Fix a $q^\dagger \in \mathcal{Q}$. We may define the query $\mathcal{F}^{\text{nf}'}(\mathcal{Q}) = (\mathcal{F}^{\text{nf}'}(\mathcal{P}), \mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0), \mathcal{F}^{\text{nf}'}(\Pi_{\text{cov}} \lhd \Gamma_{\text{cov}}))$ where

$$\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) = \Theta^{(q_1^0, q^\dagger)} \parallel \cdots \parallel \Theta^{q_k^0, q^\dagger} \lhd \Gamma_0 \text{ and}$$
$$\mathcal{F}^{\text{nf}'}(\Pi_{\text{cov}} \lhd \Gamma_{\text{cov}}) = \Theta_{\text{cov}}^{(q_1, \beta_1)} \parallel \cdots \parallel \Theta_{\text{cov}}^{(q_n, \beta_n)} \lhd \Gamma.$$

$\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \in G(\Pi_0 \lhd \Gamma_0)$ and $\mathcal{F}^{\text{nf}'}(\Pi_{\text{cov}} \lhd \Gamma_{\text{cov}}) \in G(\Pi_{\text{cov}} \lhd \Gamma_{\text{cov}})$ and both the former are polynomial-time computable.

Now suppose $\mathcal{Q}$ is a yes-instance, then $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* \Pi \lhd \Gamma$ such that $\Pi_{\text{cov}} \lhd \Gamma_{\text{cov}} \leq_{ACPS} \Pi \lhd \Gamma$ and since $\mathcal{B}$ is a bisimulation we also know that $G(\Pi_0 \lhd \Gamma_0) \to_{\mathscr{P}[\mathcal{P}']}^* G(\Pi \lhd \Gamma)$. Further we must have $\Pi = (q_1, \beta_1') \parallel \cdots (q_n, \beta_n') \parallel \Pi_1$ where either $\beta_i = \epsilon$ or $\beta_i = A_i$ and $\beta_i' = A_i \cdot \beta_i''$. By definition for all $\Pi' \lhd \Gamma \in G(\Pi \lhd \Gamma)$ there exists a $\Pi_0' \lhd \Gamma_0 \in G(\Pi_0 \lhd \Gamma_0)$ such that $\Pi_0' \lhd \Gamma_0 \to_{\mathcal{P}'}^* \Pi' \lhd \Gamma$. We can deduce that $\Pi' = A_1^{q_1, q_1'} \bar{\beta}_1 \parallel \cdots A_n^{q_n, q_n'} \bar{\beta}_n \parallel \Pi_1'$ where either $A_i = \beta_i$ or $\beta_i = \epsilon$ and $A_i = \Theta$. We can then see that $\Pi' \lhd \Gamma \to_{\mathcal{F}^{\text{nf}'}(\mathcal{P})}^* \Theta_{\text{cov}}^{(q_1, \beta_1)} \bar{\beta}_1 \parallel \cdots \Theta_{\text{cov}}^{(q_n, \beta_n)} \bar{\beta}_n \parallel \Pi_1'$. Since $\Pi_0' = \Theta^{(q_1^0, q_1^1)} \parallel \cdots \parallel \Theta^{q_k^0, q_k^1}$ may differ from $\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0)$ only by the choice of the q_j^1 which cannot play a rôle in any reduction; this allows us to deduce: $\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \to_{\mathcal{F}^{\text{nf}'}(\mathcal{P})}^* \Theta_{\text{cov}}^{(q_1, \beta_1)} \bar{\beta}_1 \parallel \cdots \Theta_{\text{cov}}^{(q_n, \beta_n)} \bar{\beta}_n \parallel \Pi_1''$. Thus $\mathcal{F}^{\text{nf}'}(\mathcal{Q})$ is a yes-instance.

Conversely, suppose $\mathcal{F}^{\text{nf}'}(\mathcal{Q})$ is a yes-instance then $\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \to_{\mathcal{F}^{\text{nf}'}(\mathcal{P})}^* \Theta_{\text{cov}}^{(q_1, \beta_1)} \bar{\beta}_1 \parallel \cdots \Theta_{\text{cov}}^{(q_n, \beta_n)} \bar{\beta}_n \parallel \Pi_1'$. Reversing transitions using rules from $\mathcal{R}_{\text{cov}}$ we can see that $\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \to_{\mathcal{P}'}^* A_1^{q_1, q_1'} \bar{\beta}_1 \parallel \cdots \parallel A_n^{q_n, q_n'} \bar{\beta}_n \parallel \Pi_1''$ where either $A_i = \beta_i$ or $\beta_i = \epsilon$ and $A_i = \Theta$. Since $\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \in G(\Pi_0 \lhd \Gamma_0)$ the Lemma above implies that $A_1^{q_1, q_1'} \bar{\beta}_1 \parallel \cdots \parallel A_n^{q_n, q_n'} \bar{\beta}_n \parallel \Pi_1'' \in G((q_1, \beta_1\, \beta_1') \parallel \cdots \parallel (q_n, \beta_n\, \beta_n') \parallel \Pi_1)$ for some β_i' and Π_1 and $\Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* (q_1, \beta_1\, \beta_1') \parallel \cdots \parallel (q_n, \beta_n\, \beta_n') \parallel \Pi_1$. Thus $(\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi_{\text{cov}} \lhd \Gamma_{\text{cov}})$ is a yes-instance.

For boundedness, suppose given $\Pi_0 \lhd \Gamma_0$ suppose $\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* \Pi \lhd \Gamma\}$ is a finite set. Then since $\mathcal{B}$ is a bisimulation this implies $\{G(\Pi \lhd \Gamma) : G(\Pi_0 \lhd \Gamma_0) \to_{\mathscr{P}[\mathcal{P}']}^* G(\Pi \lhd \Gamma)\}$ is a finite set which implies that $\{\Pi \lhd \Gamma : \mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \to_{\mathcal{P}'}^* \Pi \lhd \Gamma\}$ is a finite set since $G(\Pi \lhd \Gamma)$ is a finite set for all $\Pi \lhd \Gamma$. Since rules from $\mathcal{R}_{\text{cov}}$ only add a finite of finite number of configurations we can deduce that $\{\Pi \lhd \Gamma : \mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \to_{\mathcal{F}^{\text{nf}'}(\mathcal{P})}^* \Pi \lhd \Gamma\}$ is a finite set. Conversely, suppose $\{\Pi \lhd \Gamma : \mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \to_{\mathcal{F}^{\text{nf}'}(\mathcal{P})}^* \Pi \lhd \Gamma\}$ is a finite set then since rules from $\mathcal{R}_{\text{cov}}$ only add a finite of finite number of configurations we can infer that $\{\Pi \lhd \Gamma : \mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \to_{\mathcal{P}'}^* \Pi \lhd \Gamma\}$ is finite set. Clearly this implies that $\{G(\Pi \lhd \Gamma) : G(\Pi_0 \lhd \Gamma_0) \to_{\mathscr{P}[\mathcal{P}']}^* G(\Pi \lhd \Gamma)\}$ is a finite set and thus $\{\Pi \lhd \Gamma : \Pi_0 \lhd \Gamma_0 \to_{\mathcal{P}}^* \Pi \lhd \Gamma\}$ is a finite set since $\mathcal{B}$ is a bisimulation.

For termination, since $\mathcal{B}$ is a bisimulation clearly there is an infinite path from $\Pi_0 \lhd \Gamma_0$ in $\mathcal{P}$ iff there is an infinite path from $G(\Pi_0 \lhd \Gamma_0)$ in $\mathscr{P}[\mathcal{P}']$. And the latter is clearly possible if, and only if, there is an infinite path from a $\mathcal{F}^{\text{nf}'}(\Pi_0 \lhd \Gamma_0) \in G(\Pi_0 \lhd \Gamma_0)$ which is implied by the Lemma above and by $\mathcal{B}$ being a bisimulation. This concludes the proof. □

We can now give a proof of Proposition 1:

Proof of Proposition 1. Let $\mathcal{F}'(\mathcal{P}) = \mathcal{F}^{\text{nf}'}\big(\mathcal{F}^{\text{pnf}}(\mathcal{P})\big), \mathcal{F}'(\mathcal{Q}) = \mathcal{F}^{\text{nf}'}\big(\mathcal{F}^{\text{pnf}}(\mathcal{Q})\big)$, and $\mathcal{F}'(\Pi_0 \lhd \Gamma_0) = \mathcal{F}^{\text{nf}'}\big(\mathcal{F}^{\text{pnf}}(\Pi_0 \lhd \Gamma_0)\big)$. Lemma 14 and Lemma 15 then implies that $\mathcal{F}'(\mathcal{Q})$ is a yes-instance if, and

only if, Q is a yes-instance; $\mathcal{F}'(\mathcal{P})$ is bounded from $\mathcal{F}'(\Pi_0 \lhd \Gamma_0)$ if, and only if, $\mathcal{P}$ is bounded from $\Pi_0 \lhd \Gamma_0$; and $\mathcal{F}'(\mathcal{P})$ is terminating from $\mathcal{F}'(\Pi_0 \lhd \Gamma_0)$ if, and only if, $\mathcal{P}$ is terminating from $\Pi_0 \lhd \Gamma_0$. $\mathcal{F}'(\mathcal{P})$ is not quite in normal form yet, since it contains rules of the form $A \xrightarrow{\lambda} B$ where normal form requires the RHS to be ϵ, clearly we can by introducing a polynomial number of non-terminals to remedy this; replacing such rules by pairs::

$$A \xrightarrow{\lambda} B \mapsto \left\{ A \xrightarrow{\epsilon} C^{\lambda}\, B, C^{\lambda} \xrightarrow{\lambda} \epsilon \right\}.$$

We call the the resulting ACPS $\mathcal{F}_{\text{nf}}(\mathcal{P})$, and take the query $\mathcal{F}_{\text{nf}}(Q) = \mathcal{F}'(Q)$, and $\mathcal{F}_{\text{nf}}(\Pi_0 \lhd \Gamma_0) = \mathcal{F}'(\Pi_0 \lhd \Gamma_0)$ and it is trivial to see that the result holds. □

B.2 Proofs for Section 4.2

In this section we give a proof of Proposition 2. For this purpose, we fix a $\mathcal{P}$ in normal form and hence have also a fixed $\mathcal{G}(\mathcal{P})$. The proof of Proposition 2 exploits the fact, that we may accelerate any transitions with commutative side-effects (send, spawn, ϵ) and that we may delay blocking/non-commutative transitions until the last possible point. This means we can synchronise reductions of $\mathcal{P}$ and $\mathcal{G}(\mathcal{P})$ at configurations where all processes have non-commutative non-terminals as a head symbol. We will relabel the transition relations several times in order to chose different synchronisation points.

Let us first extend $\simeq_{I(\mathcal{G}(\mathcal{P}))}$ to parallel compositions. We write $\pi_1 \parallel \cdots \parallel \pi_n \simeq_{I(\mathcal{G}(\mathcal{P}))} \bar{\pi}_1 \parallel \cdots \parallel \bar{\pi}_n$ just if $\pi_i \simeq_{I(\mathcal{G}(\mathcal{P}))} \bar{\pi}_i$ for all $i \in \langle n \rangle$.

For this section let us write $\to_{\mathcal{G}(\mathcal{P})}$ for $\to_{\mathcal{G}_{\text{std}}}$ to make clear that the transition relation is induced by $\mathcal{G}(\mathcal{P})$. We temporarily label $\to_{\mathcal{P}}$ and $\to_{\mathcal{G}(\mathcal{P})}$ by $\mathcal{P}$-configurations as follows: If $\Pi \lhd \Gamma \to_{\mathcal{P}} \Pi' \lhd \Gamma'$ then we label the transition $\Pi \lhd \Gamma \xrightarrow{\Pi'_0 \lhd \Gamma'}_{\mathcal{P}} \Pi' \lhd \Gamma'$ for $\Pi'_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi'$. If $\Pi \lhd \Gamma \to_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$ and $\Pi' \simeq_{I(\mathcal{G}(\mathcal{P}))} \bar{\Pi}', \bar{\Pi}' \in \mathbb{M}[\mathcal{N}^*]$ then we label the transition $\Pi \lhd \Gamma \xrightarrow{\bar{\Pi}' \lhd \Gamma'}_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$; otherwise we label it by ϵ, i.e. $\Pi \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$.

Lemma 16. The relation $\mathcal{R} = \{(\Pi \lhd \Gamma, \bar{\Pi} \lhd \Gamma) : \bar{\Pi} \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi, \bar{\Pi} \in \mathbb{M}[\mathcal{N}^*]\}$ is a weak simulation relation for $\mathcal{P}$ and $\mathcal{G}(\mathcal{P})$.

Proof. Suppose $(\pi \parallel \Pi \lhd \Gamma, \pi_0 \parallel \Pi_0 \lhd \Gamma) \in \mathcal{R}$ and $\pi \parallel \Pi \lhd \Gamma \to_{\mathcal{P}} \pi' \parallel \Pi' \lhd \Gamma'$. We may assume that $\pi = A\,\beta$ and a rule $A \xrightarrow{\lambda} \beta'$ is used in the transition. Since $\pi_0 \parallel \Pi_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} A\,\beta \parallel \Pi$ we can w.l.o.g. assume that $\pi_0 = A\,\bar{\beta}$ and $\bar{\beta} \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta$ (by using transitivity of $\simeq_{I(\mathcal{G}(\mathcal{P}))}$ otherwise). Let us perform a case analysis on λ:

- *Case:* $\lambda = \epsilon$
  Since $\mathcal{P}$ is in normal form we know that $\beta' \in \{\epsilon, B\,C : B, C \in \mathcal{A}\}$, $\pi' = \beta'\,\beta$, $\Pi' = \Pi$, $\Gamma' = \Gamma$ and $A \to \beta' \in \mathcal{G}(\mathcal{P})$. Then clearly $A\,\bar{\beta} \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{G}(\mathcal{P})} \beta'\bar{\beta} \parallel \Pi_0 \lhd \Gamma$ and $\beta'\,\bar{\beta} \parallel \Pi_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta'\,\beta \parallel \Pi = \pi' \parallel \Pi'$. Further clearly $\beta'\,\bar{\beta} \parallel \Pi_0 \in \mathbb{M}[\mathcal{N}^*]$ and thus we may take $l = \pi' \parallel \Pi'$.

- *Case:* $\lambda = c\,!\,m$
  Since $\mathcal{P}$ is in normal form we know that $\beta' = \epsilon$, $\pi' = \beta$, $\Pi' = \Pi$, $\Gamma' = \Gamma \oplus [c \mapsto [m]]$ and $A \to c\,!\,m \in \mathcal{G}(\mathcal{P})$. Then clearly $A\,\bar{\beta} \parallel \Pi_0 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} c\,!\,m\bar{\beta} \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{G}(\mathcal{P})} \bar{\beta} \parallel \Pi_0 \lhd \Gamma'$ and $\bar{\beta} \parallel \Pi_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta \parallel \Pi = \pi' \parallel \Pi'$. Further clearly $\bar{\beta} \parallel \Pi_0 \in \mathbb{M}[\mathcal{N}^*]$ and thus we may take $l = \pi' \parallel \Pi'$.

- *Case:* $\lambda = c\,?\,m$
  Since $\mathcal{P}$ is in normal form we know that $\beta' = \epsilon$, $\pi' = \beta$, $\Pi' = \Pi$, $\Gamma = \Gamma' \oplus [c \mapsto [m]]$ and $A \to c\,?\,m \in \mathcal{G}(\mathcal{P})$. Then clearly $A\,\bar{\beta} \parallel \Pi_0 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} c\,?\,m\bar{\beta} \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{G}(\mathcal{P})} \bar{\beta} \parallel \Pi_0 \lhd \Gamma'$ and $\bar{\beta} \parallel \Pi_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta \parallel \Pi = \pi' \parallel \Pi'$. Further clearly $\bar{\beta} \parallel \Pi_0 \in \mathbb{M}[\mathcal{N}^*]$ and thus we may take $l = \pi' \parallel \Pi'$.

- *Case:* $\lambda = \nu A'$
  Since $\mathcal{P}$ is in normal form we know that $\beta' = \epsilon$, $\pi' = \beta$, $\Pi' = A' \parallel \Pi$, $\Gamma = \Gamma'$ and $A \to \nu A' \in \mathcal{G}(\mathcal{P})$. Then clearly $A\,\bar{\beta} \parallel \Pi_0 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} \nu A'\bar{\beta} \parallel \Pi_0 \lhd \Gamma \xrightarrow{l}_{\mathcal{G}(\mathcal{P})} \bar{\beta} \parallel A' \parallel \Pi_0 \lhd \Gamma$. Further $\bar{\beta} \parallel A' \parallel \Pi_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta \parallel A' \parallel \Pi = \pi' \parallel \Pi'$, and Further clearly $\bar{\beta} \parallel A' \parallel \Pi_0 \in \mathbb{M}[\mathcal{N}^*]$ and thus we may take $l = \pi' \parallel \Pi'$.

Hence $\mathcal{R}$ is a weak simulation which is what we wanted to prove. □

We temporarily relabel $\to_{\mathcal{P}}$ and $\to_{\mathcal{G}(\mathcal{P})}$ by side effects in Λ as follows: If $\Pi \lhd \Gamma \to_{\mathcal{P}} \Pi' \lhd \Gamma'$ using rule $\beta \xrightarrow{\lambda} \beta'$ then we label the transition $\Pi \lhd \Gamma \xrightarrow{\lambda}_{\mathcal{P}} \Pi' \lhd \Gamma'$. If $\lambda\,\beta \parallel \Pi \lhd \Gamma \to_{\mathcal{G}(\mathcal{P})} \beta \parallel \Pi' \lhd \Gamma'$ using rules (R-4), (R-5) and $\lambda \in \Sigma(\mathcal{P})$ then we label the transition $\lambda\,\beta \parallel \Pi \lhd \Gamma \xrightarrow{\lambda}_{\mathcal{G}(\mathcal{P})} \beta \parallel \Pi' \lhd \Gamma'$; otherwise we label it by ϵ, i.e. $\Pi \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$. Further we label the transitive closures as usual: If $\Pi_0 \lhd \Gamma_0 \xrightarrow{l_1}_{\mathcal{G}(\mathcal{P})} \Pi_1 \lhd \Gamma_1 \xrightarrow{l_2}_{\mathcal{G}(\mathcal{P})} \cdots \xrightarrow{l_n}_{\mathcal{G}(\mathcal{P})} \Pi_n \lhd \Gamma_n$ then we label $\Pi_0 \lhd \Gamma_0 \xrightarrow{l_1 \cdots l_n}{}^*_{\mathcal{G}(\mathcal{P})} \Pi_n \lhd \Gamma_n$ and similarly for $\xrightarrow{\boldsymbol{l}}{}^*_{\mathcal{P}}$.

We capture the language of side effects of a $\beta \in \mathcal{N}^{\text{com}}$ in $\mathcal{P}$ and $\mathcal{G}(\mathcal{P})$ as follows: $\mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta) = \{\boldsymbol{\lambda} \mid \beta \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}}{}^*_{\mathcal{G}(\mathcal{P})} \epsilon \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma(\boldsymbol{\lambda})\}$ and $\mathcal{L}_{\mathcal{P}}(\beta) = \{\boldsymbol{\lambda} \mid \beta \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}}{}^*_{\mathcal{P}} \epsilon \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma(\boldsymbol{\lambda})\}$ where a $\boldsymbol{\lambda}$ determines a "delta" to the configuration in terms of sent messages and spawned processes:

$$\Pi(\boldsymbol{\lambda}) := \left\{ \Big|\Big|_{A \in \mathcal{N}} \left(\|_1^{\mathbb{M}(\boldsymbol{\lambda})(\nu A)} A \right) \right\},$$

$$\Gamma(\boldsymbol{\lambda}) := \left\{ \bigoplus_{c \in Chan} \bigoplus_{m \in Msg} \left[c \mapsto \left[m^{\mathbb{M}(\boldsymbol{\lambda})(c!m)} \right] \right] \right\}.$$

Further we note that $\mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta)$ is effectively the *Parikh* language defined by the CFG $\mathcal{G}(\mathcal{P})$ of β in the standard derivation sense. We can now make precise how we can accelerate the execution of commutative non-terminals:

Lemma 17. Suppose $\beta, \bar{\beta} \in \mathcal{N}^{\text{com}*}$ then:

(i) $\mathcal{L}_{\mathcal{P}}(\beta) \neq \emptyset$ and for all $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{P}}(\beta)$ we have the transitions: $\beta\,\beta' \parallel \Pi \lhd \Gamma \to^*_{\mathcal{P}} \beta' \parallel \Pi \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\boldsymbol{\lambda})$; and

(ii) $\mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta}) \neq \emptyset$ and for all $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta})$ we have the transitions: $\bar{\beta}\,\bar{\beta}' \parallel \bar{\Pi} \lhd \bar{\Gamma} \to^*_{\mathcal{G}(\mathcal{P})} \bar{\beta}' \parallel \bar{\Pi} \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma(\boldsymbol{\lambda})$.

And if $\beta \simeq_{I(\mathcal{G}(\mathcal{P}))} \bar{\beta}$ then we have the set equality:

$$\{(\Pi(\boldsymbol{\lambda}), \Gamma(\boldsymbol{\lambda})) \mid \boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{P}}(\beta)\} = \left\{(\Pi(\boldsymbol{\lambda}), \Gamma(\boldsymbol{\lambda})) \mid \boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta})\right\}.$$

Proof. Let us first prove that $\mathcal{L}_{\mathcal{P}}(\beta) \neq \emptyset$. First since $\beta \in \mathcal{N}^{\mathrm{com}*}$ we may see them as non-terminals of $\mathcal{G}(\mathcal{P})$ and rewrite them. Suppose $\beta = A_1 \cdots A_n$ then we know that A_i is commutative and thus $\mathcal{L}(A_i) \neq \emptyset$ for all $i \in \langle n \rangle$. So let us pick $\boldsymbol{\lambda}_i \in \mathcal{L}(A_i)$ let us show in a brief induction that if $\boldsymbol{\lambda} \in \mathcal{L}(A)$ then $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{P}}(A)$. Let us suppose $A \to^n \boldsymbol{\lambda}$. If $n = 1$ then we use a rule $A \to \boldsymbol{\lambda}$ in $\mathcal{G}(\mathcal{P})$; hence there exists a rule $A \xrightarrow{\boldsymbol{\lambda}} \epsilon$ in $\mathcal{P}$ and thus $A \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}}{}^*_{\mathcal{G}(\mathcal{P})} \epsilon \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma(\boldsymbol{\lambda})$. If $n = k + 1$ and the claim holds for all A' and $k' \leq k$ then we can see that $A \to \beta \to^k \boldsymbol{\lambda}$. Now since $\mathcal{G}(\mathcal{P})$ is in Chomsky normal form the first step must be a rule of the form $A \to B\,C$ and $\beta = B\,C$. This implies $\boldsymbol{\lambda} = \boldsymbol{\lambda}'\boldsymbol{\lambda}''$, $B \to^{k'} \boldsymbol{\lambda}'$, $C \to^{k''} \boldsymbol{\lambda}''$, and $k = k' + k''$, which yields using the IH that $\boldsymbol{\lambda}' \in \mathcal{L}_{\mathcal{P}}(B)$ and $\boldsymbol{\lambda}'' \in \mathcal{L}_{\mathcal{P}}(C)$. Further we know there is a rule $A \xrightarrow{\epsilon} B\,C$ in $\mathcal{P}$ and thus $A \lhd \emptyset \xrightarrow{\epsilon}{}^*_{\mathcal{G}(\mathcal{P})} B\,C \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}'}{}^*_{\mathcal{G}(\mathcal{P})} C \parallel \Pi(\boldsymbol{\lambda}') \lhd \Gamma(\boldsymbol{\lambda}') \xrightarrow{\boldsymbol{\lambda}''}{}^*_{\mathcal{G}(\mathcal{P})} \epsilon \parallel \Pi(\boldsymbol{\lambda}'\boldsymbol{\lambda}') \lhd \Gamma(\boldsymbol{\lambda}'')$ which concludes the induction. Hence we can infer that in fact $\boldsymbol{\lambda}_i \in \mathcal{L}(A_i)$ and so we can see that $A_1 \cdots A_n \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}_1}{}^*_{\mathcal{G}(\mathcal{P})} A_2 \cdots A_n \parallel \Pi(\boldsymbol{\lambda}_1) \lhd \Gamma(\boldsymbol{\lambda}_1) \xrightarrow{\boldsymbol{\lambda}_2}{}^*_{\mathcal{G}(\mathcal{P})} \cdots \xrightarrow{\boldsymbol{\lambda}_n}{}^*_{\mathcal{G}(\mathcal{P})} \cdots \epsilon \parallel \Pi(\boldsymbol{\lambda}_1 \cdots \boldsymbol{\lambda}_n) \lhd \Gamma(\boldsymbol{\lambda}_1 \cdots \boldsymbol{\lambda}_n)$ and thus $\boldsymbol{\lambda}_1 \cdots \boldsymbol{\lambda}_n \in \mathcal{L}_{\mathcal{P}}(\beta)$ which is what we wanted to prove. For the second claim of (i) suppose $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{P}}(\beta)$ then by definition $\beta \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}}{}^*_{\mathcal{G}(\mathcal{P})} \epsilon \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma(\boldsymbol{\lambda})$. It is then trivial to see that $\beta\,\beta' \parallel \Pi \lhd \Gamma \to^*_{\mathcal{P}} \beta' \parallel \Pi \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\boldsymbol{\lambda})$.

For (ii), let us first prove that $\mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta}) \neq \emptyset$. First we note again that $\bar{\beta} \in \mathcal{N}^{\mathrm{com}*}$ and thus $\mathcal{L}(\bar{\beta}) \neq \emptyset$. Hence take $\boldsymbol{\lambda} \in \mathcal{L}(\bar{\beta})$. By continuously commuting non-terminals to the leftmost-position we can see that $\bar{\beta} \lhd \emptyset \xrightarrow{\epsilon}{}^*_{\mathcal{G}(\mathcal{P})} \boldsymbol{\lambda} \lhd \emptyset$ and then clearly $\boldsymbol{\lambda} \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}}{}^*_{\mathcal{G}(\mathcal{P})} \epsilon \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma(\boldsymbol{\lambda})$ thus $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta})$. For the second claim of (ii) let $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta})$ then by definition $\bar{\beta} \lhd \emptyset \xrightarrow{\boldsymbol{\lambda}}{}^*_{\mathcal{G}(\mathcal{P})} \epsilon \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma(\boldsymbol{\lambda})$; it is hence trivial to see that $\bar{\beta}\,\bar{\beta}' \parallel \bar{\Pi} \lhd \bar{\Gamma} \to^*_{\mathcal{P}} \bar{\beta}' \parallel \bar{\Pi} \parallel \Pi(\boldsymbol{\lambda}) \lhd \bar{\Gamma} \oplus \Gamma(\boldsymbol{\lambda})$.

Since $\mathcal{G}(\mathcal{P})$, restricted to non-terminals of $\mathcal{N}^{\mathrm{com}}$, is effectively a completely commutative context-free grammar derived from $\mathcal{P}$ we can see that $\{\mathbb{M}(\lambda) \mid \lambda \in \mathcal{L}_{\mathcal{P}}(\beta)\} = \{\mathbb{M}(\bar{\lambda}) \mid \bar{\lambda} \simeq_{I(\mathcal{G}(\mathcal{P}))} \lambda \in \mathcal{L}_{\mathcal{P}}(\beta)\} = \{\mathbb{M}(\bar{\lambda}) \mid \bar{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta})\}$ which implies immediately that

$$\{(\Pi(\boldsymbol{\lambda}), \Gamma(\boldsymbol{\lambda})) \mid \boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{P}}(\beta)\} = \left\{(\Pi(\boldsymbol{\lambda}), \Gamma(\boldsymbol{\lambda})) \mid \boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\bar{\beta})\right\}.$$

□

As a final step in our argument let us relabel the transition relation again. We temporarily relabel $\to_{\mathcal{P}}$ and $\to_{\mathcal{G}(\mathcal{P})}$ by $\mathcal{P}$-configurations as follows: If $\Pi \lhd \Gamma \to_{\mathcal{P}} \Pi' \lhd \Gamma'$ then we label the transition $\Pi \lhd \Gamma \xrightarrow{\Pi'_0 \lhd \Gamma'}_{\mathcal{P}} \Pi' \lhd \Gamma'$, if $\Pi' \in \mathbb{M}[\mathcal{N}^{\neg\mathrm{com}} \cdot \mathcal{A}^*]$ and $\Pi'_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi'$; otherwise we label it with ϵ: $\Pi \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{P}} \Pi' \lhd \Gamma'$. If $\Pi \lhd \Gamma \to_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$ and $\Pi' \simeq_{I(\mathcal{G}(\mathcal{P}))} \bar{\Pi}'$,

$\bar{\Pi}' \in \mathbb{M}[\mathcal{N}^{\neg com} \cdot \mathcal{N}^*]$ then we label the transition $\Pi \lhd \Gamma \xrightarrow{\bar{\Pi}' \lhd \Gamma'}_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$; otherwise we label it by ϵ, i.e. $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$.

Lemma 18. The relation $\mathcal{R}' = \{(\Pi \lhd \Gamma, \bar{\Pi} \lhd \Gamma) : \bar{\Pi} \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi, \bar{\Pi} \in \mathbb{M}[\mathcal{N}^{\neg com} \cdot \mathcal{N}^*]\}$ is a weak bisimulation relation for $\mathcal{P}$ and $\mathcal{G}(\mathcal{P})$.

Proof. Let us first prove that $\mathcal{R}'$ is a weak simulation. Suppose $(\Pi \lhd \Gamma, \Pi_0 \lhd \Gamma) \in \mathcal{R}'$ and $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}^*_{\mathcal{P}} \Pi' \lhd \Gamma'$ such that $\Pi' \in \mathbb{M}[\mathcal{N}^{\neg com} \cdot \mathcal{A}^*]$. Lemma 16 then tells us that $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}^*_{\mathcal{P}} \Pi'_0 \lhd \Gamma'$ and $\Pi'_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi'$. The latter implies that $\Pi'_0 \in \mathbb{M}[\mathcal{N}^{\neg com} \cdot \mathcal{N}^*]$ since $\Pi' \in \mathbb{M}[\mathcal{N}^{\neg com} \cdot \mathcal{A}^*]$, i.e. each process is headed by a non-commutative non-terminal which obviously are invariant under commutation. Hence we can deduce that $\mathcal{R}'$ is a weak simulation.

Let us first prove that $\mathcal{R}'^{-1}$ is a weak simulation. Suppose $(\Pi \lhd \Gamma, \Pi_0 \lhd \Gamma) \in \mathcal{R}'$ and $\Pi_0 \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}^*_{\mathcal{G}(\mathcal{P})} \Pi'_0 \lhd \Gamma'$ where $\Pi'_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi'$ for some $\Pi' \in \mathbb{M}[\mathcal{N}^{\neg com} \cdot \mathcal{A}^*]$. W.l.o.g. we may assume this path may not be split to expose two labels l, l' (otherwise we may consider a maximal splitting of the path to obtain several labels and apply the below to each in turn one-by-one). We can infer that $\Pi_0 = A\,\beta_0 \parallel \Pi_1$ and the transition may be split $A\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{l}_{\mathcal{G}(\mathcal{P})} \beta'\,\beta_0 \parallel \Pi'_1 \lhd \Gamma_1 \xrightarrow{l'}^*_{\mathcal{G}(\mathcal{P})} \Pi'_0 \lhd \Gamma'$, $l \cdot l' = \Pi' \lhd \Gamma'$ and using a rule $A \to \beta' \in \mathcal{G}(\mathcal{P})$. Further we may infer that $\Pi = A\,\beta \parallel \Pi_2$. Let us make a case analysis on β'.

- *Case:* $\beta' = B\,C$, B non-commutative.
  Firstly, we notice $\Pi'_1 = \Pi_1, \Gamma = \Gamma_1$ and $B\,C\,\beta_0 \parallel \Pi_1 \in \mathbb{M}[\mathcal{N}^{\neg com} \cdot \mathcal{N}^*]$ which implies that $l = \Pi' \lhd \Gamma'$ and also $\Pi'_0 \lhd \Gamma' = \beta'\,\beta_0 \parallel \Pi_1$. Further there is a rule $A \xrightarrow{\epsilon} B\,C$ in $\mathcal{P}$ and thus $A\,\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{l}_{\mathcal{P}} B\,C\,\beta \parallel \Pi_2 \lhd \Gamma$ and $B\,C\,\beta \parallel \Pi_2 \simeq_{I(\mathcal{G}(\mathcal{P}))} B\,C\,\beta_0 \parallel \Pi_1$ which is what we wanted to prove.

- *Case:* $\beta' = B\,C$, B commutative.
  Firstly, we notice $\Pi'_1 = \Pi_1$ and $\Gamma = \Gamma_1$. Since A was non-commutative it must be the case that C is non-commutative. Further since $\Pi_0 \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}^*_{\mathcal{G}(\mathcal{P})} \Pi'_0 \lhd \Gamma'$ may not be further split we may assume that for some $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(B)$ we may rewrite the transition as: $A\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} B\,C\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{l'}^*_{\mathcal{G}(\mathcal{P})} C\,\beta_0 \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\boldsymbol{\lambda})$, i.e. $\Pi'_0 \lhd \Gamma' = C\,\beta_0 \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\boldsymbol{\lambda})$. Now we know that there is a rule $A \xrightarrow{\epsilon} B\,C$ in $\mathcal{P}$ and thus $A\,\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{P}} B\,C\,\beta \parallel \Pi_2 \lhd \Gamma$ and Lemma 17 then gives us that $B\,C\,\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{l'}^*_{\mathcal{P}} C\,\beta \parallel \Pi_2 \parallel \Pi(\bar{\boldsymbol{\lambda}}) \lhd \Gamma \oplus \Gamma(\bar{\boldsymbol{\lambda}})$ for some $\bar{\boldsymbol{\lambda}} \in \mathcal{L}_{\mathcal{P}}(B)$ such that $\Pi(\bar{\boldsymbol{\lambda}}) = \Pi(\boldsymbol{\lambda}), \Gamma(\boldsymbol{\lambda}) = \Gamma(\bar{\boldsymbol{\lambda}})$. Further $C\,\beta \parallel \Pi_2 \parallel \Pi(\boldsymbol{\lambda}) \simeq_{I(\mathcal{G}(\mathcal{P}))} C\,\beta_0 \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda})$ which is what we wanted to prove.

- *Case:* $\beta' = \epsilon$.
  Firstly, we notice $\Pi'_1 = \Pi_1$ and $\Gamma = \Gamma_1$ and we may write $\beta_0 = \beta'_0\beta''_0$ such that $\beta'_0 \in \mathcal{N}^{com*}$ and $\beta''_0 \in (\mathcal{N}^{\neg com}\mathcal{N}^{com*})^*$. Further since $\Pi_0 \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}^*_{\mathcal{G}(\mathcal{P})} \Pi'_0 \lhd \Gamma'$ may not be further split we may assume that for some $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta'_0)$ we may rewrite the

transition as: $A\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} \beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{l'}{}^{*}_{\mathcal{G}(\mathcal{P})} \beta_0'' \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\boldsymbol{\lambda})$, i.e. $\Pi_0' \lhd \Gamma' = \beta_0'' \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\boldsymbol{\lambda})$. Now we know that there is a rule $A \xrightarrow{\epsilon} \epsilon$ in $\mathcal{P}$ and thus $A\,\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{P}} \beta \parallel \Pi_2 \lhd \Gamma$ and since $\beta \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0$ we may rewrite $\beta = \beta'\beta''$ such that $\beta' \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0'$ and $\beta'' \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0''$, hence clearly $\beta' \in \mathcal{N}^{\mathrm{com}*}$ and $\beta'' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}*})^*$, and Lemma 17 then gives us that $\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{l'}{}^{*}_{\mathcal{P}} \beta'' \parallel \Pi_2 \parallel \Pi(\bar{\boldsymbol{\lambda}}) \lhd \Gamma \oplus \Gamma(\bar{\boldsymbol{\lambda}})$ for some $\bar{\boldsymbol{\lambda}} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta')$ such that $\Pi(\bar{\boldsymbol{\lambda}}) = \Pi(\boldsymbol{\lambda}), \Gamma(\boldsymbol{\lambda}) = \Gamma(\bar{\boldsymbol{\lambda}})$. Further $\beta'' \parallel \Pi_2 \parallel \Pi(\boldsymbol{\lambda}) \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0'' \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda})$ which is what we wanted to prove.

- *Case:* $\beta' = c\,!\,m$.
  Firstly, we notice $\Pi_1' = \Pi_1$ and $\Gamma = \Gamma_1$ and we may write $\beta_0 = \beta_0'\beta_0''$ such that $\beta_0' \in \mathcal{N}^{\mathrm{com}*}$ and $\beta_0'' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}*})^*$. Further since $\Pi_0 \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}{}^{*}_{\mathcal{G}(\mathcal{P})} \Pi_0' \lhd \Gamma'$ may not be further split we may assume that for some $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta_0')$ we may rewrite the transition as: $A\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} c\,!\,m\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{l'}{}^{*}_{\mathcal{G}(\mathcal{P})} \beta_0'' \parallel \Pi_1 \parallel \Pi(c\,!\,m \cdot \boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(c\,!\,m \cdot \boldsymbol{\lambda})$, i.e. $\Pi_0' \lhd \Gamma' = \beta_0'' \parallel \Pi_1 \parallel \Pi(c\,!\,m \cdot \boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(c\,!\,m \cdot \boldsymbol{\lambda})$. Now we know that there is a rule $A \xrightarrow{c!m} \epsilon$ in $\mathcal{P}$ and thus $A\,\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{P}} \beta \parallel \Pi_2 \parallel \Pi(c\,!\,m) \lhd \Gamma \oplus \Gamma(c\,!\,m)$ and since $\beta \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0$ we may use analogous reasoning to the $\beta' = \epsilon$ case to obtain that $\beta \parallel \Pi_2 \parallel \Pi(c\,!\,m) \lhd \Gamma \oplus \Gamma(c\,!\,m) \xrightarrow{l'}{}^{*}_{\mathcal{P}} \beta'' \parallel \Pi_2 \parallel \Pi(c\,!\,m \cdot \boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(c\,!\,m \cdot \boldsymbol{\lambda})$ such that $\beta'' \parallel \Pi_2 \parallel \Pi(c\,!\,m \cdot \boldsymbol{\lambda}) \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0'' \parallel \Pi_1 \parallel \Pi(c\,!\,m \cdot \boldsymbol{\lambda})$ which is what we wanted to prove.

- *Case:* $\beta' = \nu A'$.
  Firstly, we notice $\Pi_1' = \Pi_1$ and $\Gamma = \Gamma_1$ and we may write $\beta_0 = \beta_0'\beta_0''$ such that $\beta_0' \in \mathcal{N}^{\mathrm{com}*}$ and $\beta_0'' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}*})^*$. Further since $\Pi_0 \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}{}^{*}_{\mathcal{G}(\mathcal{P})} \Pi_0' \lhd \Gamma'$ may not be further split we may assume that for some $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta_0')$ we may rewrite the transition as: $A\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} \nu A'\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{l'}{}^{*}_{\mathcal{G}(\mathcal{P})} \beta_0'' \parallel \Pi_1 \parallel \Pi(\nu A' \cdot \boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\nu A' \cdot \boldsymbol{\lambda})$, i.e. $\Pi_0' \lhd \Gamma' = \beta_0'' \parallel \Pi_1 \parallel \Pi(\nu A' \cdot \boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\nu A' \cdot \boldsymbol{\lambda})$. Now we know that there is a rule $A \xrightarrow{\nu A'} \epsilon$ in $\mathcal{P}$ and thus $A\,\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{P}} \beta \parallel \Pi_2 \parallel \Pi(\nu A') \lhd \Gamma \oplus \Gamma(\nu A')$ and since $\beta \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0$ we may use analogous reasoning to the $\beta' = \epsilon$ case to obtain that $\beta \parallel \Pi_2 \parallel \Pi(\nu A') \lhd \Gamma \oplus \Gamma(\nu A') \xrightarrow{l'}{}^{*}_{\mathcal{P}} \beta'' \parallel \Pi_2 \parallel \Pi(\nu A' \cdot \boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\nu A' \cdot \boldsymbol{\lambda})$ such that $\beta'' \parallel \Pi_2 \parallel \Pi(c\,!\,m \cdot \boldsymbol{\lambda}) \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0'' \parallel \Pi_1 \parallel \Pi(c\,!\,m \cdot \boldsymbol{\lambda})$ which is what we wanted to prove.

- *Case:* $\beta' = c\,?\,m$.
  Firstly, we notice $\Pi_1' = \Pi_1$ and $\Gamma = \Gamma_1$ and we may write $\beta_0 = \beta_0'\beta_0''$ such that $\beta_0' \in \mathcal{N}^{\mathrm{com}*}$ and $\beta_0'' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}*})^*$. Further since $\Pi_0 \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}{}^{*}_{\mathcal{G}(\mathcal{P})} \Pi_0' \lhd \Gamma'$ may not be further split we may assume that for some $\boldsymbol{\lambda} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta_0')$ we may rewrite the transition as: $A\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} c\,?\,m\,\beta_0 \parallel \Pi_1 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{G}(\mathcal{P})} \beta_0 \parallel \Pi_1 \lhd \Gamma \ominus [c \mapsto m] \xrightarrow{l'}{}^{*}_{\mathcal{G}(\mathcal{P})} \beta_0'' \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \oplus \Gamma(\boldsymbol{\lambda})$, i.e. $\Pi_0' \lhd \Gamma' = \beta_0'' \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \ominus [c \mapsto m] \oplus \Gamma(\boldsymbol{\lambda})$. Now we know that there is a rule $A \xrightarrow{c?m} \epsilon$ in $\mathcal{P}$ and thus $A\,\beta \parallel \Pi_2 \lhd \Gamma \xrightarrow{\epsilon}_{\mathcal{P}} \beta \parallel \Pi_2 \parallel$

$\lhd \Gamma \ominus [c \mapsto m]$ and since $\beta \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_0$ we may use analogous reasoning to the $\beta' = \epsilon$ case to obtain that $\beta \parallel \Pi_2 \lhd \Gamma \ominus [c \mapsto m] \xrightarrow{l'}{}^*_{\mathcal{P}} \beta'' \parallel \Pi_2 \parallel \Pi(\boldsymbol{\lambda}) \lhd \Gamma \ominus [c \mapsto m] \oplus \Gamma(\boldsymbol{\lambda})$ such that $\beta'' \parallel \Pi_2 \parallel \Pi(\boldsymbol{\lambda}) \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta''_0 \parallel \Pi_1 \parallel \Pi(\boldsymbol{\lambda})$ which is what we wanted to prove.

This concludes the proof.

□

Proposition 2. Suppose $\mathcal{P}$ is an ACPS in normal form and $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ a simple coverability query. Then, $\mathcal{Q}$ is a yes-instance, if and only if, $\mathcal{Q}' = (\mathcal{G}(\mathcal{P}), \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ is a yes-instance. Hence simple coverability, boundedness and termination for ACPS and APCPS polynomial-time interreduce. A simple APCPS query $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ satisfies: $\pi \simeq_{I(\mathcal{G})} A \in \mathcal{N}$ for all π in Π and Π_0. Further $\mathcal{P}$ is K-shaped from $\Pi_0 \lhd \Gamma_0$ if, and only if, $\mathcal{G}(\mathcal{P})$ is K-shaped from $\Pi_0 \lhd \Gamma_0$.

Proof. First suppose $\mathcal{Q} = (\mathcal{P}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ is a simple coverability query for ACPS in normal form, $\Pi = A_1 \parallel \cdots \parallel A_n$. W.l.o.g. we may assume that all A_i are non-commutative wrt to $I(\mathcal{G}(\mathcal{P}))$ (otherwise we may introduce a fresh channel c_{cov}, message m_{cov}, non-terminals A'_i and local states $0, \ldots, K, K+1, \infty$, counting the number of non-commutative non-terminals on stack — widening at $\geq K+2$, rules $(k, A_i) \xrightarrow{\epsilon} A'_i$ for $k \leq K+1$, $A'_i \xrightarrow{c_{\text{cov}}?m_{\text{cov}}} \epsilon$ and change Π to $\Pi = A'_1 \parallel \cdots \parallel A'_n$; applying polynomial-time normal form reduction afterwards; this transformation may increase the shaped constraint from K-shaped to $K+1$-shaped, since (∞, A_i) will clearly be commutative, and is also polynomial).

Defining then $\mathcal{Q}' = (\mathcal{G}(\mathcal{P}), \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ is then clearly a simple coverability query and can be polynomial-time computed.

Since $\mathcal{G}(-)$ is clearly a bijection and both $\mathcal{G}(-)$ and $\mathcal{G}(-)^{-1}$ are polynomial time computable we may simply show that $\mathcal{Q}$ is a yes-instance if, and only if, $\mathcal{Q}'$ is a yes-instance.

Suppose $\mathcal{Q}$' is a yes-instance then $\Pi_0 \lhd \Gamma_0 \to^*_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$ such that $\Pi \lhd \Gamma \leq_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$. Let us maximally split this path so that

$$\Pi_0 \lhd \Gamma_0 \xrightarrow{\Pi_1 \lhd \Gamma_1}{}^*_{\mathcal{G}(\mathcal{P})} \Pi_1 \lhd \Gamma_1 \xrightarrow{\Pi_2 \lhd \Gamma_2}{}^*_{\mathcal{G}(\mathcal{P})} \cdots \xrightarrow{\Pi_n \lhd \Gamma_n}{}^*_{\mathcal{G}(\mathcal{P})} \Pi_n \lhd \Gamma_n \xrightarrow{\epsilon}{}^*_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma'$$

using the third labelling of this section. Since $\mathcal{Q}'$ is simple we know that $\Pi = A_1 \parallel \cdots \parallel A_n$ and thus $\Pi' \simeq_{I(\mathcal{G}(\mathcal{P}))} A_1\beta_1 \parallel \cdots \parallel A_n\beta_n \parallel \Pi''$. Further since the path above is a maximal split we may assume that $\Pi_n \simeq_{I(\mathcal{G}(\mathcal{P}))} A_1\beta_1 \parallel \cdots \parallel A_n\beta_n \parallel \Pi'''$ and $\Pi''' \lhd \Gamma_n \xrightarrow{\epsilon}{}^*_{\mathcal{G}(\mathcal{P})} \Pi'' \lhd \Gamma'$ where the latter may not be split into two paths exposing a label. Hence we may conclude that $\Pi''' = B_1\,\beta_1^\dagger\,\beta_1^{\dagger'} \parallel \cdots \parallel B_k\,\beta_k^\dagger\beta_k^{\dagger'} \parallel \Pi''''$ where $B_1, \ldots, B_k \in \mathcal{N}^{\neg\text{com}}$, $\beta_1^\dagger, \ldots, \beta_k^\dagger \in \mathcal{N}^{\text{com}*}$ and $\beta_1^{\dagger'}, \ldots, \beta_k^{\dagger'} \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$. that after a one rule step for each process 1 up to k we can go to $\Pi''' \lhd \Gamma_n \xrightarrow{\epsilon}{}^*_{\mathcal{G}(\mathcal{P})} \beta_1^{\dagger''}\,\beta_1^{\dagger'} \parallel \cdots \parallel \beta_k^{\dagger''}\beta_k^{\dagger'} \parallel \Pi'''' \lhd \Gamma^\dagger \xrightarrow{\epsilon}{}^*_{\mathcal{G}(\mathcal{P})} \Pi'' \lhd \Gamma'$ where $\beta_1^{\dagger''}, \ldots, \beta_k^{\dagger''} \in \mathcal{N}^{\text{com}*}$, $\Pi'' = \Pi'''' \parallel \Pi(\boldsymbol{\lambda}_1^\dagger \cdots \boldsymbol{\lambda}_k^\dagger)$, $\Gamma' = \Gamma^\dagger \oplus \Gamma(\boldsymbol{\lambda}_1^\dagger \cdots \boldsymbol{\lambda}_k^\dagger)$ for some $\boldsymbol{\lambda}_i^\dagger\boldsymbol{\lambda}_i^{\dagger'} \in \mathcal{L}_{\mathcal{G}(\mathcal{P})}(\beta_i^{\dagger''})$ for each i.

We will now show that we can follow these transitions with $\mathcal{P}$. First Lemma 18 tells us that we find the following path $\Pi_0 \lhd \Gamma_0 \xrightarrow{\Pi_1 \lhd \Gamma_1}{}^*_{\mathcal{P}} \bar{\Pi}_1 \lhd \Gamma_1 \xrightarrow{\Pi_2 \lhd \Gamma_2}{}^*_{\mathcal{P}} \cdots \xrightarrow{\Pi_n \lhd \Gamma_n}{}^*_{\mathcal{P}} \bar{\Pi}_n \lhd \Gamma_n$ such that $\Pi_i \simeq_{I(\mathcal{G}(\mathcal{P}))} \bar{\Pi}_i$ for all $i \in \langle n \rangle$. Now $\bar{\Pi}_n = A_1\bar{\beta}_1 \parallel \cdots \parallel A_n\bar{\beta}_n \parallel \bar{\Pi}'''$ and $\Pi''' = B_1\,\bar{\beta}_1^\dagger\,\bar{\beta}_1^{\dagger'} \parallel$

$\cdots \parallel B_k\,\bar{\beta}_k^{\dagger}\bar{\beta}_k^{\dagger'} \parallel \bar{\Pi}''''$ since $A_1,\ldots,A_n,B_1,\ldots,B_k \in \mathcal{N}^{\neg\mathrm{com}}$. Now since $B_1,\ldots,B_k \in \mathcal{N}^{\neg\mathrm{com}}$ it is easy to see that we can follow the $\mathcal{G}(\mathcal{P})$-path one rule step per process in: $\bar{\Pi}''' \lhd \Gamma_n \xrightarrow{\epsilon}{}^{*}_{\mathcal{G}(\mathcal{P})}$ $\bar{\beta}_1^{\dagger''}\,\bar{\beta}_1^{\dagger'} \parallel \cdots \parallel \bar{\beta}_k^{\dagger''}\bar{\beta}_k^{\dagger'} \parallel \bar{\Pi}''' \lhd \Gamma^{\dagger}$ where $\bar{\beta}_1^{\dagger''},\ldots,\bar{\beta}_k^{\dagger''} \in \mathcal{N}^{\mathrm{com}*}$ and $\bar{\beta}_i^{\dagger''} \simeq_{I(\mathcal{G}(\mathcal{P}))} \beta_i^{\dagger''}$ for all $i \in \langle k\rangle$. We can then pick $\bar{\boldsymbol{\lambda}}_i^{\dagger} \in \mathcal{L}_{\mathcal{P}}(\bar{\beta}_i^{\dagger''})$ such that $\bar{\boldsymbol{\lambda}}_i^{\dagger} \simeq_{I(\mathcal{G}(\mathcal{P}))} \boldsymbol{\lambda}_i^{\dagger}\boldsymbol{\lambda}_i^{\dagger'}$. Lemma 17 then tells us we can reach $\bar{\Pi}''' \lhd \Gamma_n \xrightarrow{\epsilon}{}^{*}_{\mathcal{G}(\mathcal{P})} \bar{\Pi}^{\dagger} \parallel \bar{\Pi}'''' \parallel \Pi(\bar{\boldsymbol{\lambda}}_1^{\dagger}\cdots\bar{\boldsymbol{\lambda}}_k^{\dagger}) \lhd \Gamma^{\dagger} \oplus \Gamma^{\dagger} \oplus \Gamma(\bar{\boldsymbol{\lambda}}_1^{\dagger}\cdots\bar{\boldsymbol{\lambda}}_k^{\dagger})$ where we write $\bar{\Pi}^{\dagger} = \bar{\beta}_1^{\dagger'} \parallel \cdots \parallel \bar{\beta}_k^{\dagger'}$. We note that $\Gamma \leq \Gamma(\boldsymbol{\lambda}_1^{\dagger}\cdots\boldsymbol{\lambda}_k^{\dagger}) \leq \Gamma(\bar{\boldsymbol{\lambda}}_1^{\dagger}\cdots\bar{\boldsymbol{\lambda}}_k^{\dagger})$. We can now use these paths to extend our $\mathcal{P}$ path to a covering configuration: $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{P}} \bar{\Pi}_n \lhd \Gamma_n \to^{*}_{\mathcal{P}}$ $A_1\bar{\beta}_1 \parallel \cdots \parallel A_n\bar{\beta}_n \parallel \bar{\Pi}^{\dagger} \parallel \bar{\Pi}''' \parallel \Pi(\bar{\boldsymbol{\lambda}}_1^{\dagger}\cdots\bar{\boldsymbol{\lambda}}_k^{\dagger}) \lhd \Gamma^{\dagger}\oplus\Gamma^{\dagger}\oplus\Gamma(\bar{\boldsymbol{\lambda}}_1^{\dagger}\cdots\bar{\boldsymbol{\lambda}}_k^{\dagger})$ which clearly implies that $\Pi \lhd \Gamma$ is coverable in $\mathcal{P}$ from $\Pi_0 \lhd \Gamma_0$ Hence $\mathcal{Q}$ is a yes-instance.

Conversely, suppose $\mathcal{Q}$ is a yes-instance. Then $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{P}} \Pi' \lhd \Gamma'$ such that $\Pi \lhd \Gamma \leq_{ACPS}$ $\Pi' \lhd \Gamma'$. Since $\mathcal{Q}$ is a simple query we know that $\Pi = A_1 \parallel \cdots \parallel A_n$ and hence we may conclude that $\Pi' = A_1\,\beta_1 \parallel \cdots \parallel A_n\,\beta_n \parallel \Pi''$. Lemma 16 then allows us to conclude that $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{G}(\mathcal{P})} \Pi'_0 \lhd \Gamma'$ where $\Pi'_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi'$ i.e. $\Pi'_0 \simeq_{I(\mathcal{G}(\mathcal{P}))} A_1\,\beta_1 \parallel \cdots \parallel A_n\,\beta_n \parallel \Pi''$ and thus clearly $\Pi \lhd \Gamma \leq_{\mathcal{G}_{\mathrm{std}}} \Pi'_0 \lhd \Gamma'$ which implies $\mathcal{Q}$' is a yes-instance.

For the second claim, we will prove that there is a reachable K-shaped configuration from $\Pi_0 \lhd \Gamma_0$ in $\mathcal{P}$ if, and only if, there is a reachable K-shaped configuration from $\Pi_0 \lhd \Gamma_0$ in $\mathcal{G}(\mathcal{P})$.

Suppose then that $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{P}} \Pi \lhd \Gamma$ and that $\Pi \lhd \Gamma$ is K-shaped. An application of Lemma 16 then yields that $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{G}(\mathcal{P})} \Pi' \lhd \Gamma$ where $\Pi' \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi$ which clearly implies that Π' is K-shaped as processes of Π' are only (commutative) permutations of processes in Π. Conversely, suppose $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{G}(\mathcal{P})} \Pi \lhd \Gamma$ and $\Pi \lhd \Gamma$ is K-shaped. As before let us maximally split this path so that $\Pi_0 \lhd \Gamma_0 \xrightarrow{\Pi_1 \lhd \Gamma_1}{}^{*}_{\mathcal{G}(\mathcal{P})} \Pi_1 \lhd \Gamma_1 \xrightarrow{\Pi_2 \lhd \Gamma_2}{}^{*}_{\mathcal{G}(\mathcal{P})} \cdots \xrightarrow{\Pi_n \lhd \Gamma_n}{}^{*}_{\mathcal{G}(\mathcal{P})}$ $\Pi_n \lhd \Gamma_n \xrightarrow{\epsilon}{}^{*}_{\mathcal{G}(\mathcal{P})} \Pi \lhd \Gamma$ using the third labelling of this section. Now $\Pi = \beta_1\beta'_1 \parallel \cdots \parallel \beta_n\beta'_n$ with $\beta_i \in \mathcal{N}^{\mathrm{com}*}$ and $\beta'_i \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}*})^*$. Using Lemma 17 we know that we can extend the path above to $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{G}(\mathcal{P})} \Pi \lhd \Gamma \to^{*}_{\mathcal{G}(\mathcal{P})} \beta'_1 \parallel \cdots \parallel \beta'_n \lhd \Gamma' =: \Pi' \lhd \Gamma'$. Clearly $beta'_1 \parallel \cdots \parallel \beta'_n \in \mathbb{M}[\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^*]$ and since $\Pi \lhd \Gamma$, and we have only "executed off" all commutative β_i's, it is easy to see that $\Pi' \lhd \Gamma'$ is also K-shaped. Since $beta'_1 \parallel \cdots \parallel \beta'_n \in \mathbb{M}[\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^*]$ we can invoke Lemma 18 to see $\Pi_0 \lhd \Gamma_0 \to^{*}_{\mathcal{P}} \bar{\Pi}' \lhd \Gamma'$ such that $\bar{\Pi}' \simeq_{I(\mathcal{G}(\mathcal{P}))} \Pi'$. Hence as above we may conclude that $\bar{\Pi}' \lhd \Gamma'$ is K-shaped.

Hence we can conclude that there is a reachable K-shaped configuration from $\Pi_0 \lhd \Gamma_0$ in $\mathcal{P}$ if, and only if, there is a reachable K-shaped configuration from $\Pi_0 \lhd \Gamma_0$ in $\mathcal{G}(\mathcal{P})$. From which we can deduce that $\mathcal{P}$ is K-shaped if, and only if, $\mathcal{G}(\mathcal{P})$ is K-shaped. □

Standard semantics	***Alternative semantics***
(R-1): $A\,\gamma \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \beta\,\gamma \parallel \Pi \lhd \Gamma$	(R′-1): $A\,M\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \beta\,M\,\gamma' \parallel \Pi' \lhd \Gamma$
(R-2): $A\,\gamma \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} B\,C\,\gamma \parallel \Pi \lhd \Gamma$	(R′-2): $A\,M\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} B\,(\mathbb{M}(w) \oplus M)\,\gamma' \parallel \Pi' \lhd \Gamma$
(R-3): $(c\,?\,m)\,\gamma \parallel \Pi \lhd ([m] \oplus q)^c, \Gamma \to_{\mathcal{G}_{\text{std}}} \gamma \parallel \Pi \lhd q^c, \Gamma$	(R′-3): $(c\,?\,m)\,\gamma' \parallel \Pi' \lhd ([m] \oplus q)^c, \Gamma \to_{\mathcal{G}_{\text{alt}}} \gamma' \parallel \Pi' \lhd q^c, \Gamma$
(R-4): $(c\,!\,m)\,\gamma \parallel \Pi \lhd q^c, \Gamma \to_{\mathcal{G}_{\text{std}}} \gamma \parallel \Pi \lhd ([m] \oplus q)^c, \Gamma$	(R′-4): $(c\,!\,m)\,\gamma' \parallel \Pi' \lhd q^c, \Gamma \to_{\mathcal{G}_{\text{alt}}} \gamma' \parallel \Pi' \lhd ([m] \oplus q)^c, \Gamma$
(R-5): $(\nu X)\,\gamma \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \gamma \parallel X \parallel \Pi \lhd \Gamma$	(R′-5): $(\nu X)\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \gamma' \parallel X \parallel \Pi' \lhd \Gamma$
	(R′-6): $M\,X\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} X\,\gamma' \parallel \Pi' \parallel \Pi(M) \lhd \Gamma \oplus \Gamma(M)$
	(R′-7): $M'\,\gamma' \parallel \Pi' \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} M''\,\gamma' \parallel \Pi' \parallel \Pi(M') \lhd \Gamma \oplus \Gamma(M')$

Let (†) be a condition on a $\beta \in \mathcal{N}^*$: $\beta = B\,C$ and C is commutative. Rules (R-2) and (R′-2) have a side condition: $A \to B\,C \in \mathcal{G}$, $B\,C$ satisfies (†), and $C \to^* w$. Rules (R-1) and (R′-1) have a side condition: $A \to \beta \in \mathcal{G}$ and β does *not* satisfy (†). In rule (R′-6) we require the multiset $M \in \mathbb{M}[\Sigma^{\text{com}}]$ whereas in rule (R′-7) M' may be an element of $\mathbb{M}[\Sigma^{\text{com}} \cup \mathcal{N}]$ and $M'' \in \mathbb{M}[\mathcal{N}]$. We use the abbreviations: $\Pi(M) := \|_{A \in \mathcal{N}} \|_1^{M(\nu A)}\ A$, and $\Gamma(M) := \bigoplus_{c \in \mathit{Chan}} \bigoplus_{m \in \mathit{Msg}} [c \mapsto [m^{M(c!m)}]]$.

Table B.1: Transition semantics for APCPS

B.3 Proofs for Section 4.3

Lemma 2. Suppose $\bar{\Pi} \lhd \Gamma \in \mathcal{F}_{Sum}(\Pi \lhd \Gamma)$ such that $\bar{\Pi} \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \bar{\Pi}' \lhd \Gamma'$ for some $\bar{\Pi}' \lhd \Gamma'$. Then, there exists a configuration $\Pi' \lhd \Gamma'$ such that $\Pi \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$ and $\bar{\Pi}' \lhd \Gamma' \in \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$.

Proof. Let $\Pi_1 \in \mathbb{M}[Procs_{\text{alt}}]$, $\Pi_2 \in \mathbb{M}[Procs]$ and $\Gamma, \Gamma' \in Chan \to \mathbb{M}[Msg]$ such that $\Pi_1 \lhd \Gamma \in \mathcal{F}_{Sum}(\Pi_2 \lhd \Gamma)$ and suppose that $\Pi_1 \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma'$.

We will prove that there exists a Π_2' such that $\Pi_1' \lhd \Gamma' \in \mathcal{F}_{Sum}(\Pi_2' \lhd \Gamma')$ and $\Pi_2 \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} \Pi_2' \lhd \Gamma'$ by case analysis on the rule used for $\Pi_1 \lhd \Gamma \to_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma'$. First, though, we notice that $\Pi_1 = \gamma \parallel \Pi_1^0$, $\Pi_1 = \gamma' \parallel \Pi_1'^0$, and $\Pi_2 = \alpha \parallel \Pi_2^0$ such that $\gamma \in \mathcal{F}_{Sum}(\alpha)$ and $\Pi_1^0 \in \mathcal{F}_{Sum}(\Pi_2^0)$.

- Rule (R′-1)
  In this case, $\Pi_1^0 = \Pi_1'^0$ and $\Gamma = \Gamma'$. We know that $\gamma = X M \gamma_0$ and $\gamma' = \gamma_1 M \gamma_0$. Thus $\alpha = X \beta \alpha_0$ such that $M \in \mathcal{F}_{Sum}(\beta)$ and $\gamma_0 \in \mathcal{F}_{Sum}(\alpha_0)$.

  Further, $X \to \gamma_1 \in G$ and γ_1 does not satisfy (†).

  Clearly $X \beta \alpha_0 \parallel \Pi_2^0 \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \gamma_1 \beta \alpha_0 \parallel \Pi_2^0 \lhd \Gamma =: \Pi_2' \lhd \Gamma'$. Therefore, writing $\alpha' := \gamma_1 \beta \alpha_0$ gives us $\gamma' = \gamma_1 M \gamma_0 \in \mathcal{F}_{Sum}(\alpha')$ as desired.

- Rule (R′-2)
  In this case, $\Pi_1^0 = \Pi_1'^0$ and $\Gamma = \Gamma'$. $X \to AB \in G$ and AB satisfies (†), i.e. B commutative and $B \to^* w$. Further, we know that $\gamma = X M \gamma_0$ and $\gamma' = A(\mathbb{M}(w) \oplus M)\gamma_0$. Thus, $\alpha = X \beta \alpha_0$ such that $M \in \mathcal{F}_{Sum}(\beta)$ and $\gamma_0 \in \mathcal{F}_{Sum}(\alpha_0)$.

  Then, $X \beta \alpha_0 \parallel \Pi_2^0 \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} AB \beta \alpha_0 \parallel \Pi_2^0 \lhd \Gamma =: \Pi_2' \lhd \Gamma'$.

  Clearly, $\mathbb{M}(w) \oplus M \in \mathcal{F}_{Sum}(B\,\beta)$. Hence, writing $\alpha' := A\,B\,\beta\alpha_0$ yields $\gamma' = A(\mathbb{M}(w) \oplus M)\gamma_0 \in \mathcal{F}_{Sum}(\alpha')$ as desired. This concludes the proof of this case.

- Rule (R′-3)
  Then $\Pi_1 = (c\,?\,m)\,\gamma \parallel \Pi_1^0$ and $\Gamma = \Gamma' \oplus [c \mapsto [m]]$ and $\Pi_1' = \gamma \parallel \Pi_1^0$. Hence $\Pi_2 = (c\,?\,m)\alpha \parallel \Pi_2^0$ with $\gamma \in \mathcal{F}_{Sum}(\alpha)$ and so $\Pi_1' \in \mathcal{F}_{Sum}(\alpha \parallel \Pi_2^0)$. And using rule (R-3) $(c\,?\,m)\,\alpha \parallel \Pi_2^0 \lhd \Gamma' \oplus [c \mapsto [m]] \to_{\mathcal{G}_{\text{std}}} \alpha \parallel \Pi_2^0 \lhd \Gamma'$.

- Rule (R′-4)
  Then $\Pi_1 = (c_j\,!\,m)\,\gamma \parallel \Pi_1^0$, $\Pi_1' = \gamma \parallel \Pi_1^0$ and $\Gamma' = \Gamma \oplus [c \mapsto [m]]$. Then $\Pi_2 = (c_j\,!\,m)\,\alpha \parallel \Pi_2^0$ with $\gamma \in \mathcal{F}_{Sum}(\alpha)$ and so $\Pi_1' \in \mathcal{F}_{Sum}(\alpha \parallel \Pi_2^0)$. By rule (R-4) we can see $(c_j\,!\,m)\,\alpha \parallel \Pi_2^0 \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \alpha \parallel \Pi_2^0 \lhd \Gamma \oplus [c \mapsto [m]]$.

- Rule (R′-5)
  Then $\Pi_1 = (\nu X)\,\gamma \parallel \Pi_1^0$, $\Pi_1' = \gamma \parallel X \parallel \Pi_1^0$ and $\Gamma' = \Gamma$. Hence $\Pi_2 = (\nu X)\,\alpha \parallel \Pi_2^0$ with $\gamma \in \mathcal{F}_{Sum}(\alpha)$ and so $\Pi_1' \in \mathcal{F}_{Sum}(\alpha \parallel X \parallel \Pi_2^0)$. By rule (R-5) $(\nu X)\,\alpha \parallel \Pi_2^0 \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \alpha \parallel X \parallel \Pi_2^0 \lhd \Gamma$.

- Rule (R′-6)
  Then $\Pi_1 = M\,X\,\gamma \parallel \Pi_1^0$ such that $M \in CompleteSummary$, $\Gamma' = \Gamma \oplus \Gamma(M)$, $X \in \mathcal{N}^{\neg\text{com}}$, and $\Pi_1' = X\,\gamma \parallel \Pi_1^0 \parallel \Pi(M)$. Also $\Pi_2 = \beta\,X\,\alpha \parallel \Pi_2^0$ and $M\,X\,\gamma \in \mathcal{F}_{Sum}(\beta\,X\,\alpha)$.

Hence $X\,\gamma \in \mathcal{F}_{Sum}(X\,\alpha)$ and $M \in \mathcal{F}_{Sum}(\beta)$. Thus, $\beta \to^* w$, $w \in \Sigma^{\text{com}*}$ such that $\mathbb{M}(w) = M$. Hence using rules (R-4), (R-5), (R-1), and (R-2) repeatedly we can see that $\beta\,X\,\alpha \parallel \Pi_2^0 \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} X\,\alpha \parallel \Pi_2^0 \parallel \Pi(w) \lhd \Gamma \oplus \Gamma(\mathbb{M}(w)) = X\,\alpha \parallel \Pi_2^0 \parallel \Pi(M) \lhd \Gamma \oplus \Gamma(M) =: \Pi_2' \lhd \Gamma'$. and $\Pi_1' \in \mathcal{F}_{Sum}(\Pi_2')$.

- Rule (R′-7)
  Then $\Pi_1 = M\,X\,\gamma \parallel \Pi_1^0$ and $\Pi_1' = M'\,X\,\gamma \parallel \Pi_1^0 \parallel \Pi(M)$ such that $M \in$ *PartialSummary*, $M' \in$ *DegenSummary*, $\Gamma' = \Gamma \oplus \Gamma(M)$ and $X \in \mathcal{N}^{\neg\text{com}}$.

  Also $\Pi_2 = \beta\,X\,\alpha \parallel \Pi_2^0$ and $M\,X\,\gamma \in [\![\beta\,X\,\alpha]\!]$. Thus $M \in \mathcal{F}_{Sum}(\beta)$ and hence $\beta \to^* w$, $w \in (\Sigma^{\text{com}} \cup \mathcal{N}^{\text{com}})^* \setminus \Sigma^{\text{com}*}$ such that $\mathbb{M}(w) = M$. Then $w \simeq_I w_0 w_1$ such that $w_0 \in \Sigma^{\text{com}*}$ and $w_1 \in \mathcal{N}^{\text{com}}$ and $M' = \mathbb{M}(w_1)$.

  Hence $M'\,X\,\gamma \in [\![w_1\,X\,\alpha]\!]$ and thus $w_1 X\,\alpha \parallel \Pi_2^0 \oplus \Pi(M) \lhd \Gamma \oplus \Gamma(M) =: \Pi_2' \lhd \Gamma'$ and $\Pi_1' \lhd \Gamma' \mathcal{F}_{Sum}(\Pi_2' \lhd \Gamma')$.

  Using rules rules (R-4), (R-5), (R-1), and (R-2) repeatedly we can see that $\beta\,X\,\alpha \parallel \Pi_2^0 \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} w_1 X\,\alpha \parallel \Pi_2^0 \parallel \Pi(w_0) \lhd \Gamma \oplus \Gamma(\mathbb{M}(w_0)) = w_1 X\,\alpha \parallel \Pi_2^0 \parallel \Pi(w) \lhd \Gamma \oplus \Gamma(M(w)) = w_1 X\,\alpha \parallel \Pi_2^0 \parallel \Pi(M) \lhd \Gamma \oplus \Gamma(M) = \Pi_2' \lhd \Gamma'$.

Hence the claim holds in all cases. □

Corollary 10. Given an APCPS P if $\bar{\Pi}_0 \lhd \Gamma_0 \to^*_{\mathcal{G}_{\text{alt}}} \bar{\Pi} \lhd \Gamma$ and $\bar{\Pi}_0 \lhd \Gamma_0 \in \mathcal{F}_{Sum}(\Pi_0 \lhd \Gamma_0)$ then $\Pi_0 \lhd \Gamma_0 \to^*_{\mathcal{G}_{\text{std}}} \Pi \lhd \Gamma$ such that $\bar{\Pi} \lhd \Gamma \in \mathcal{F}_{Sum}(\Pi \lhd \Gamma)$.

Proof. Follows trivially by induction from Lemma 2. □

A Few Technical Lemmas for the Co-Universal Powerset Lifting

The next few lemmas establish cases in which the Co-universal powerset lifting of the alternative semantics can simulate a step of the standard semantics. They are technical in nature but we build upon them in the following.

Lemma 19. If $\alpha \lhd \emptyset \to_{\mathcal{G}_{\text{std}}} \beta \lhd \emptyset$ such that $\alpha \in \mathcal{N}^*$ then $\mathcal{F}_{Sum}(\alpha \lhd \emptyset) \to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\beta \lhd \emptyset)$.

Proof. Since $\alpha \lhd \emptyset \to_{\mathcal{G}_{\text{std}}} \beta \lhd \emptyset$ we have $\alpha = X\alpha_0$ and $\beta = \alpha_1\alpha_0$ such that $X \to \alpha_1$. And so $\mathcal{F}_{Sum}(\alpha) = X\mathcal{F}_{Sum}(\alpha_0)$. We will proceed by case analysis on $X \to \alpha_1$.

- $X \to \alpha_1$, α_1 does not satisfy (†).
  Take $\alpha_1\delta \in \alpha_1 \cdot \mathcal{F}_{Sum}(\alpha_0) = \mathcal{F}_{Sum}(\alpha_1\alpha_0) = \mathcal{F}_{Sum}(\beta)$. Then $X\delta \in \mathcal{F}_{Sum}(\alpha)$ and $X\delta \lhd \emptyset \to_{\mathcal{G}_{\text{alt}}} \alpha_1\delta \lhd \emptyset$. Hence $\mathcal{F}_{Sum}(\alpha \lhd \emptyset) \to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\beta \lhd \emptyset)$.

- $X \to A\,B$, $B \in \mathcal{N}^{\text{com}}$.
  Suppose $AM'\delta \in \mathcal{F}_{Sum}(AB\alpha_0) = \mathcal{F}_{Sum}(\alpha_1\alpha_0) = \mathcal{F}_{Sum}(\beta)$. Then clearly $M' = \mathbb{M}(w) \oplus M$ such that $B \to^* w$ and $M\delta \in \mathcal{F}_{Sum}(\alpha_0)$. Now then $XM\delta \in \mathcal{F}_{Sum}(\alpha)$ and $XM\delta \lhd \emptyset \to_{\mathcal{G}_{\text{alt}}} A(\mathbb{M}(w) \oplus M)\delta \lhd \emptyset$. Thus $\mathcal{F}_{Sum}(\alpha \lhd \emptyset) \to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\beta \lhd \emptyset)$.

□

Lemma 20. (i) If $a \in \Sigma^{\mathrm{com}}$ and $a\alpha\alpha' \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\mathrm{std}}} \alpha\alpha' \parallel \Pi \oplus \Pi(\mathbb{M}(a)) \lhd \Gamma \oplus \Gamma(\mathbb{M}(a))$ where $\alpha \in (\mathcal{N} \cup \{\epsilon\})\mathcal{N}^{\mathrm{com}*}$, $\alpha' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}})^*$, then

$$\mathcal{F}_{Sum}(a\alpha\alpha' \parallel \Pi) \lhd \Gamma \to_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi \oplus \Pi(\mathbb{M}(a))) \lhd \Gamma \oplus \Gamma(\mathbb{M}(a)).$$

(ii) If $(c\,?\,m)\alpha\alpha' \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]]) \to_{\mathcal{G}_{\mathrm{std}}} \alpha\alpha' \parallel \Pi \lhd \Gamma$ where $\alpha \in (\mathcal{N} \cup \{\epsilon\})\mathcal{N}^{\mathrm{com}*}$, $\alpha' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}})^*$, then

$$\mathcal{F}_{Sum}((c\,?\,m)\alpha\alpha' \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]]) \to_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi) \lhd \Gamma.$$

Proof of Claim 1. We show that $\mathcal{F}_{Sum}(a\alpha\alpha' \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi' \lhd \Gamma')$ by case analysis on a:

- $a = c\,!\,m$
  Then $\Pi \oplus \Pi(\mathbb{M}(a)) = \Pi$,
  Take $\gamma \parallel \pi \in \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi)$ then $(c\,!\,m)\gamma \parallel \pi \in \mathcal{F}_{Sum}((c\,!\,m)\alpha\alpha' \parallel \Pi)$. Using rule (R′-4) we see that
  $$(c\,!\,m)\gamma \parallel \pi \lhd \Gamma \to_{\mathcal{G}_{\mathrm{alt}}} \gamma \parallel \pi \lhd \Gamma \oplus [c \mapsto [m]]).$$
  Hence we conclude $\mathcal{F}_{Sum}((c\,!\,m)\alpha\alpha' \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi \lhd \oplus[c \mapsto [m]])$.
- $a = \nu X$
  Then $\Pi \oplus \Pi(\mathbb{M}(\nu X)) = \Pi \parallel X, \Gamma = \Gamma$
  Take $\gamma \parallel X \parallel \pi \in \mathcal{F}_{Sum}(\alpha\alpha' \parallel X \parallel \Pi) = \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi \parallel X)$ then $(\nu X)\gamma \parallel \pi \in \mathcal{F}_{Sum}((\nu X)\alpha\alpha' \parallel \Pi)$. Using rule (R′-5) we see that
  $$(\nu X)\gamma \parallel \pi \lhd \Gamma \to_{\mathcal{G}_{\mathrm{alt}}} \gamma \parallel X \parallel \pi \lhd \Gamma.$$
  Hence we conclude $\mathcal{F}_{Sum}((\nu X)\alpha\alpha' \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi \parallel X \lhd \Gamma)$.

□

Proof of Claim 2. Take $\gamma \parallel \pi \in \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi) = \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi)$ then $(c\,?\,m)\gamma \parallel \pi \in \mathcal{F}_{Sum}((c\,?\,m)\alpha\alpha' \parallel \Pi)$. Then using rule (R′-3) we see that

$$(c\,?\,m)\gamma \parallel \pi \lhd \Gamma \oplus [c \mapsto [m]]) \to_{\mathcal{G}_{\mathrm{alt}}} \gamma \parallel \pi \lhd \Gamma.$$

Hence we conclude $\mathcal{F}_{Sum}((c\,?\,m)\alpha\alpha' \parallel \Pi \lhd \Gamma \oplus [c \mapsto [m]]) \to_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi \lhd \Gamma)$.

□

Lemma 21. If $a_1 \cdots a_n \in \Sigma^{\mathrm{com}*}$, $\alpha_i \in \mathcal{N}^{\mathrm{com}*}$ and $\alpha' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}})^*$,

$$\begin{aligned}
\alpha_1\alpha' \lhd \emptyset &\to^*_{\mathcal{G}_{\mathrm{std}}} a_1\alpha_2\alpha' \lhd \emptyset \\
&\to_{\mathcal{G}_{\mathrm{std}}} \alpha_2\alpha' \parallel \Pi(\mathbb{M}(a_1)) \lhd \Gamma(\mathbb{M}(a_1)) \\
&\to^*_{\mathcal{G}_{\mathrm{std}}} \cdots \to^*_{\mathcal{G}_{\mathrm{std}}} \\
&a_n\alpha_{n+1}\alpha' \parallel \Pi(\mathbb{M}(a_1 \cdots a_{n-1})) \lhd \Gamma(\mathbb{M}(a_1 \cdots a_{n-1})) \\
&\to^*_{\mathcal{G}_{\mathrm{std}}} \alpha_{n+1}\alpha' \parallel \Pi(\mathbb{M}(a_1 \cdots a_n)) \lhd \Gamma(\mathbb{M}(a_1 \cdots a_n))
\end{aligned}$$

then

$$\mathcal{F}_{Sum}(\alpha_1\alpha' \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha_{n+1}\alpha' \parallel \Pi(\mathbb{M}(a_1 \cdots a_n)) \lhd \Gamma(\mathbb{M}(a_1 \cdots a_n))).$$

Proof. We prove the claim by induction on n. For $n = 0$ the claim is vacuously true.

For $n = k + 1$, assuming the claim holds for k it is enough to show that if

$$\begin{aligned}\alpha_{k+1}\alpha' \parallel \Pi(a_1 \cdots a_k) \lhd \Gamma(a_1 \cdots a_k) &\to^*_{\mathcal{G}_{\text{std}}} a_{k+1}\alpha_{k+2}\alpha' \parallel \Pi(a_1 \cdots a_k) \lhd \Gamma(a_1 \cdots a_k) \\ &\to^*_{\mathcal{G}_{\text{std}}} \alpha_{k+2}\alpha' \parallel \Pi(a_1 \cdots a_{k+1}) \lhd \Gamma(a_1 \cdots a_{k+1})\end{aligned}$$

then

$$\begin{aligned}\mathcal{F}_{Sum}(\alpha_{k+1}\alpha' \parallel \Pi(a_1 \cdots a_k) \lhd \Gamma(a_1 \cdots a_k)) &\to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \\ &\mathcal{F}_{Sum}(\alpha_{k+2}\alpha' \parallel \Pi(a_1 \cdots a_{k+1}) \lhd \Gamma(a_1 \cdots a_{k+1}))\end{aligned}$$

which we obtain by repeatedly applying Lemma 19 and then Lemma 20. □

Lemma 22. If $a\alpha\alpha' \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \alpha\alpha' \parallel \Pi' \lhd \Gamma' \to^*_{\mathcal{G}_{\text{std}}} \alpha' \parallel \Pi'' \lhd \Gamma''$ where $a \in \Sigma$, $\alpha \in \mathcal{N}^{\text{com}*}$, $\alpha' \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}})^*$, $\alpha \to^* w \in {\Sigma^*}/{\simeq_I}$, $\Pi'' = \Pi' \oplus \Pi(\mathbb{M}(w))$, $\Gamma'' = \Gamma' \oplus \Gamma(\mathbb{M}(w))$ then

$$\mathcal{F}_{Sum}(a\alpha\alpha' \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi' \lhd \Gamma') \to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha' \parallel \Pi'' \lhd \Gamma'').$$

Proof. For $\mathcal{F}_{Sum}(a\alpha\alpha' \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi' \lhd \Gamma')$ we appeal to Lemma 20. Now since $\alpha \to^* w$ we have $\alpha\alpha' \parallel \Pi' \lhd \Gamma' \to^*_{\mathcal{G}_{\text{std}}} \alpha' \parallel \Pi' \oplus \Pi(\mathbb{M}(w)) \lhd \Gamma' \oplus \Gamma(\mathbb{M}(w))$.

Let $M = \mathbb{M}(w)$ and take

$$\gamma \parallel \pi' \oplus \Pi(M) \lhd \Gamma' \oplus \Gamma(M) \in \mathcal{F}_{Sum}(\alpha' \parallel \Pi' \oplus \Pi(\mathbb{M}(w))) \lhd \Gamma' \oplus \Gamma(\mathbb{M}(w)).$$

Then note $M\gamma \parallel \pi' \in \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi')$ and using rule (R′-6)

$$M\gamma \parallel \pi' \lhd \Gamma' \to_{\mathcal{G}_{\text{alt}}} \gamma \parallel \pi' \oplus \Pi(M) \lhd \Gamma' \oplus \Gamma(M).$$

□

Lemma 23. If $a\alpha\alpha' \parallel \Pi \lhd \Gamma \to_{\mathcal{G}_{\text{std}}} \alpha\alpha' \parallel \Pi' \lhd \Gamma' \to^*_{\mathcal{G}_{\text{std}}} w\alpha' \parallel \Pi' \lhd \Gamma' \to^*_{\mathcal{G}_{\text{std}}} w'\alpha' \parallel \Pi'' \lhd \Gamma''$ where $a \in \Sigma$, $\alpha \in \mathcal{N}^{\text{com}*}$, $\alpha' \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}})^*$, $\alpha'' \in \mathcal{N}^{\neg\text{com}*}$ and $\alpha \to^* w$ where $w \in (\Sigma \cup \mathcal{N})^*/{\simeq_I}$, $w' \in \mathcal{N}^*/{\simeq_I}$, $\Pi'' = \Pi' \oplus \Pi(\mathbb{M}(w))$, $\Gamma'' = \Gamma' \oplus \Gamma(\mathbb{M}(w))$ then

$$\begin{aligned}\mathcal{F}_{Sum}(a\alpha\alpha' \parallel \Pi \lhd \Gamma) &\to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi' \lhd \Gamma') \\ &\to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathbb{M}(w') \cdot \mathcal{F}_{Sum}(\alpha') \parallel \mathcal{F}_{Sum}(\Pi'') \lhd \Gamma''.\end{aligned}$$

Proof. For $\mathcal{F}_{Sum}(a\alpha\alpha' \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi' \lhd \Gamma')$ we appeal to Lemma 20. Now since $\alpha \to^* w$ we have $\alpha\alpha' \parallel \Pi' \lhd \Gamma' \to^*_{\mathcal{G}_{\text{std}}} w'\alpha' \parallel \Pi' \oplus \Pi(\mathbb{M}(w)) \lhd \Gamma' \oplus \Gamma(\mathbb{M}(w))$.

Let $M' = \mathbb{M}(w')$ then

$M'\gamma \parallel \pi' \oplus \Pi(M) \lhd \Gamma' \oplus \Gamma(M) \in \mathbb{M}(w') \cdot \mathcal{F}_{Sum}(\alpha') \parallel \mathcal{F}_{Sum}(\Pi'') \lhd \Gamma''$ and $M\gamma \parallel \pi' \in \mathcal{F}_{Sum}(\alpha\alpha' \parallel \Pi')$ such that $M = \mathbb{M}(w)$. Using rule (R′-7)

$$M\gamma \parallel \pi' \lhd \Gamma' \to_{\mathcal{G}_{\text{alt}}} M'\gamma \parallel \pi' \oplus \Pi(M) \lhd \Gamma' \oplus \Gamma(M).$$

□

Lemma 24. Let $X \in \mathcal{N}$, $\alpha \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}*})^*$, $w \in \Sigma^{\mathrm{com}}$ and $\beta, \alpha' \in \mathcal{N}^{\mathrm{com}*}$

(i) If $X\beta\alpha \lhd \emptyset \to^*_{\mathcal{G}_{\mathrm{std}}} \alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w))$ then

$$\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w)))$$

(ii) If $X\beta\alpha \lhd \emptyset \to^*_{\mathcal{G}_{\mathrm{std}}} \alpha'\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w))$, then

$$\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathbb{M}(\alpha') \cdot \mathcal{F}_{Sum}(\alpha) \parallel \mathcal{F}_{Sum}(\Pi(\mathbb{M}(w))) \lhd \Gamma(\mathbb{M}(w)).$$

Proof of Claim 1. Then $X\beta\alpha \to^* a\alpha_0\alpha$ where $a \in \Sigma^{\mathrm{com}} \cup \{\epsilon\}$ such that $a\alpha_0 \to^* w$ so by Lemma 19 $\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(a\alpha_0\alpha \lhd \emptyset)$. By Lemma 22

$$\mathcal{F}_{Sum}(a\alpha_0\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w)))$$

and clearly also

$$\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(a\alpha_0\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w))).$$

□

Proof of Claim 2. Then $X\beta\alpha \to^* a\alpha_0\alpha$ where $a \in \Sigma^{\mathrm{com}} \cup \{\epsilon\}$ such that $a\alpha_0 \to^* w\alpha'$ so by Lemma 19 $\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(a\alpha_0\alpha \lhd \emptyset)$. By Lemma 23 $\mathcal{F}_{Sum}(a\alpha_0\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha'\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w)))$ and clearly also

$$\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(a\alpha_0\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w))).$$

□

Lemma 25. Let $X \in \mathcal{N}^{\neg\mathrm{com}}$, $\beta, \beta' \in \mathcal{N}^{\mathrm{com}*}$ and $\alpha, \alpha' \in (\mathcal{N}^{\neg\mathrm{com}}\mathcal{N}^{\mathrm{com}*})^*$.

(i) If $X\beta\alpha \lhd \emptyset \to^*_{\mathcal{G}_{\mathrm{std}}} \alpha'\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w))$ then

$$\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(\alpha'\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w)))$$

(ii) If $X\beta\alpha \lhd \emptyset \to^*_{\mathcal{G}_{\mathrm{std}}} (c\,?\,m)\beta'\alpha'\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w))$, then

$$\mathcal{F}_{Sum}(X\beta\alpha) \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}((c\,?\,m)\beta'\alpha'\alpha \lhd \emptyset)$$

Proof of Claim 1. Then

$$X\beta\alpha \lhd \emptyset \to^*_{\mathcal{G}_{\mathrm{std}}} X'\alpha'\alpha \parallel \Pi(w_0) \lhd \Gamma(w_0)$$

such that $X' \to^* a\alpha_0$ where $a \in \Sigma^{\mathrm{com}} \cup \{\epsilon\}$, $\alpha_0 \in \mathcal{N}^{\mathrm{com}*}$ such that $a\alpha_0 \to^* w_1$ and $w = w_0w_1$. By Lemma 21

$$\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\mathrm{alt}}]} \mathcal{F}_{Sum}(X'\alpha'\alpha \parallel \Pi(w_0)) \lhd \Gamma(w_0)$$

Then the proof of Lemma 24 Claim 1 applies to give the result. □

Proof of Claim 2. Then

$$X\beta\alpha \lhd \emptyset \to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} X'X''\beta'\alpha'\alpha \parallel \Pi(w_0) \lhd \Gamma(\mathbb{M}(w_0))$$

such that $X'' \to^* c\,?\,m$, and $X' \to^* w_1$ where $w = w_0w_1$. Hence

$$X'X''\beta'\alpha'\alpha \parallel \Pi(w_0) \lhd \Gamma(w_0) \to^*_{\mathcal{G}_{\text{std}}} (c\,?\,m)\beta'\alpha'\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w)).$$

so by Lemma 21 and Lemma 19.

$$\mathcal{F}_{Sum}(X\beta\alpha \lhd \emptyset) \to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}((c\,?\,m)\beta'\alpha'\alpha \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma(\mathbb{M}(w))).$$

□

Establishing Simulations

We now establish that the co-universal powerset lifting of the alternative semantics simulates the standard semantics.

Proposition 9. $\mathcal{R}$ is a weak simulation for $\mathcal{G}_{\text{std}}$ and $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$.

Proof. To prove the simulation suppose we have a $\Pi \lhd \Gamma \in \mathit{Sync}$ such that $\Pi \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$ and $\Pi' \lhd \Gamma' \in \mathit{Sync}$. We assume without loss of generality that during $\Pi \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$ only one process makes progress (note this can be achieved by delaying receptions and performing sends and spawns as early as possible) and none of the intermediate steps are configurations of *Sync*.

Hence, $\Pi = \alpha \parallel \Pi^0$, $\Pi' = \alpha' \parallel \Pi^0 \oplus \Pi(\mathbb{M}(w))$ and $\Gamma' \oplus \Gamma(w') = \Gamma \oplus \Gamma(\mathbb{M}(w))$ for some $w \in \Sigma^{\text{com}*}$ and $w' \in \{\epsilon\} \cup \Sigma^{\text{com}}$. Further, $\alpha = X\alpha_0\alpha_1$, $X \in \mathcal{N}^{\neg\text{com}}$, $\alpha_0 \in \mathcal{N}^{\text{com}*}$, $\alpha_1 \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ and $\alpha' = \alpha'_0\alpha_1$ where $\alpha'_0 \in \epsilon \cup (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^+$, i.e. either we increase the call-stack or we pop one non-commutative non-terminal off the call-stack. Otherwise we would get a contradiction to $\Pi' \lhd \Gamma' \in \mathit{Sync}$.

- Case $\alpha'_0 = \epsilon$.
  Then $X\alpha_0\alpha_1 \parallel \Pi^0 \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} X'\alpha_2\alpha_0\alpha_1 \parallel \Pi^0 \lhd \Gamma$ such that $X' \in \mathcal{N}^{\text{com}}$, $\alpha_2 \in \mathcal{N}^{\text{com}*}$ and $X'\alpha_2\alpha_0\alpha_1 \parallel \Pi^0 \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} w\alpha_1 \parallel \Pi^0 \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} \alpha_1 \parallel \Pi^0 \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma \oplus \Gamma(\mathbb{M}(w))$, where $w \in \Sigma^{\text{com}*}$ such that $\Pi' = \alpha' \parallel \Pi^0 \parallel \Pi(\mathbb{M}(w))$ and $\Gamma' = \Gamma \oplus \Gamma(\mathbb{M}(w))$. Lemma 20 then allows us to conclude that

  $$\mathcal{F}_{Sum}(X\alpha_0\alpha_1 \parallel \Pi^0 \lhd \Gamma) \to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(X'\alpha_2\alpha_0\alpha_1 \parallel \Pi^0 \lhd \Gamma)$$

  and Lemma 24.1 gives us

  $$\mathcal{F}_{Sum}(X'\alpha_2\alpha_0\alpha_1 \parallel \Pi^0 \lhd \Gamma) \to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha_1 \parallel \Pi^0 \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma \oplus \Gamma(\mathbb{M}(w))).$$

  Hence we can conclude $\mathcal{F}_{Sum}(\Pi \lhd \Gamma) \to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$ as desired.

- Case $\alpha_0' \neq \epsilon$.
  Follows directly from Lemma 25.1. Hence we can conclude

$$\mathcal{F}_{Sum}(\Pi \lhd \Gamma) \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$$

  as desired.

Thus $\mathcal{R}$ is a weak simulation. □

Proposition 3. $\mathcal{R}$ is a weak bisimulation for $\mathcal{G}_{\text{std}}$ and $\mathscr{P}^{\mathsf{A}}[\mathcal{G}_{\text{alt}}]$.

Proof. It remains to prove that $\mathcal{R}^{-1}$ is a weak simulation. Suppose we have a $\Pi \lhd \Gamma \in \mathit{Sync}$ and $\Pi' \lhd \Gamma' \in \mathit{Sync}$ such that $\mathcal{F}_{Sum}(\Pi \lhd \Gamma) \xrightarrow{\Pi' \lhd \Gamma'}{}^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$. Since clearly $\mathcal{F}_{Sum}(\Pi' \lhd \Gamma') \neq \emptyset$ there is a $\bar{\Pi}' \lhd \Gamma' \in \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$. Hence by definition there is a $\bar{\Pi} \lhd \Gamma \in \mathcal{F}_{Sum}(\Pi \lhd \Gamma)$ such that $\bar{\Pi} \lhd \Gamma \rightarrow^*_{\mathcal{G}_{\text{alt}}} \Pi' \lhd \Gamma'$. Lemma 2 then implies that $\Pi \lhd \Gamma \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$. Hence $\mathcal{R}^{-1}$ is a weak simulation. □

Connection between Reachability and Coverability Decision Problems

We now strengthen the connection between coverability and rechability.

Proposition 10. If $\Pi_0 \lhd \Gamma_0 \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$ then $\mathcal{F}_{Sum}(\Pi_0 \lhd \Gamma_0) \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$

Proof. Define the set $P_{\Pi^f} = \{\alpha \mid \exists \Pi. \alpha \parallel \Pi = \Pi^f\}$ for some given $\Pi^f \in \mathbb{M}[\mathit{Procs}]$. Further define the set of configurations

$$P := \{\Pi \lhd \Gamma \mid \forall \alpha \in \Pi, \alpha \in \mathcal{N}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^* \cup \Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^* \cup \mathcal{N} \cup P_{\Pi'}\}.$$

Now suppose that for some Π, Π' and Γ, Γ'

$$\Pi \lhd \Gamma := \Pi_0 \lhd \Gamma_0 \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi_1 \lhd \Gamma_1 \rightarrow^*_{\mathcal{G}_{\text{std}}} \cdots \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi_n \lhd \Gamma_n =: \Pi' \lhd \Gamma'$$

such that $\Pi_i \lhd \Gamma_i \in P$ for $i = 0, \ldots n$. Without loss of generality we can assume that for all $i = 0, ..., n$, $\Pi_i = \Pi_i^a \parallel \Pi_i^f$ such that for all $\alpha \in \Pi_i^f$ we have $\alpha \in P_{\Pi^f}$ and α is not involved in any transitions in $\Pi_i \lhd \Gamma_i \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi_n \lhd \Gamma_n$. Note that we are not loosing generality, since a reduction $\alpha \parallel \Pi \lhd \Gamma \rightarrow^*_{\mathcal{G}_{\text{std}}} \alpha \parallel \Pi' \lhd \Gamma'$ can either be pre-empted or goes through a process state in $\Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$. Note this also means that $\Pi_{i+1}^f = \Pi_i^f \parallel \Pi'^f_i$. We further assume w.l.o.g that for each i it is the case that $\Pi_i^a = \alpha \parallel \Pi_i'$ and $\Pi_{i+1}^a = \alpha' \parallel \Pi_i' \oplus \Pi(\mathbb{M}(w))$ and $\Gamma_{k+1} \oplus \Gamma(\mathbb{M}(w')) = \Gamma_k \oplus \Gamma(\mathbb{M}(w))$ for some $w \in \Sigma^{\text{com}*}$ and $w' \in \{\epsilon\} \cup \Sigma^{\text{com}}$, i.e. during each $\Pi_i^a \lhd \Gamma_i \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi_{i+1}^a \lhd \Gamma_{i+1}$ only one process makes progress (note this can be achieved by delaying receptions and performing sends and spawns as early as possible) and none of the intermediate steps are configurations of P.

We will prove by induction on n:

$$\mathcal{F}_{Sum}(\Pi_0^a \parallel \tilde{\Pi}_0^f \lhd \Gamma_0) \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \cdots \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\Pi_n \parallel \tilde{\Pi}_n^f \lhd \Gamma_n)$$

where for all $i = 0, \ldots n$ and $\alpha \in \Pi_i^f$ we have either $\Pi_i^f(\alpha) = \tilde{\Pi}_i^f(\mathcal{F}_{Sum}(\alpha))$ or $\Pi_i^f(\alpha) = \tilde{\Pi}_i^f(\mathbb{M}(\alpha_0) \cdot \mathcal{F}_{Sum}(\alpha_1))$, $\alpha = \alpha_0\alpha_1$.

- $n = 0$.
  The claim holds trivially.

- $n = k + 1$, assuming the claim holds for k.
  To prove the inductive claim we need to show that from $\Pi_k = \alpha \parallel \Pi'_k$, $\Pi_{k+1} = \alpha' \parallel \Pi'_k \parallel \Pi(\mathbb{M}(w))$ and $\Gamma_{k+1} \oplus \Gamma(w') = \Gamma_k \oplus \Gamma(\mathbb{M}(w))$ where $\Pi_k \lhd \Gamma_k \rightarrow^*_{\mathcal{G}_{\text{std}}} \Pi_{k+1} \lhd \Gamma_{k+1}$. We want to prove $\mathcal{F}_{Sum}(\Pi_k \lhd \Gamma_k) \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\Pi_{k+1} \lhd \Gamma_{k+1})$. We will do so by a case analysis on the shape of α and α'.

  - $\alpha, \alpha' \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$
    From Proposition 3 we infer that

    $$\mathcal{F}_{Sum}(\alpha \lhd \Gamma_k) \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha' \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma_{k+1})$$

    Hence, clearly $\mathcal{F}_{Sum}(\alpha \parallel \Pi'_k \lhd \Gamma_k) \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\alpha' \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma_{k+1})$.
  - $\alpha \in \Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ and $\alpha' \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$
    Follows from Lemma 22.
  - $\alpha \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ and $\alpha' \in \Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$
    Follows from Lemma 25.2
  - $\alpha \in \mathcal{N}$ and $\alpha' \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$
    We can assume that $\alpha \in \mathcal{N}^{\text{com}}$ since otherwise a case above already applies. By the definition of $\mathcal{N}^{\text{com}}$ we can thus infer that $\alpha' = \epsilon$ since otherwise α would not be commutative. Thus Lemma 24.1 applies.
  - $\alpha \in \mathcal{N}$ and $\alpha' \in \Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$
    There is nothing to prove for this case as, similarly to the case above, either $\alpha \in \mathcal{N}^{\neg\text{com}}$ and so a case above applies or $\alpha \in \mathcal{N}^{\text{com}}$ but then

    $$\alpha' \notin \Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$$

    which is impossible; so the former must be the case.
  - $\alpha \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ and $\alpha' \in P_{\Pi_f}$
    If $\alpha' \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^* \cup \Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ the above cases apply. Otherwise it must be the case that

    $$\alpha' \in \mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^* \cup \Sigma^{\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*.$$

    - $\alpha' \in \mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$
      So it must be the case that $\alpha = X\alpha_0\alpha_1$, $X \in \mathcal{N}^{\neg\text{com}}$, $\alpha_0 \in \mathcal{N}^{\text{com}*}$ $\alpha_1 \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ and $\alpha' = \alpha'_0\alpha'_1\alpha_1$ where $\alpha'_1 \in (\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$, $\alpha'_0 \in \mathcal{N}^{\text{com}*}$
      Lemma 24.2 applies to give

      $$\mathcal{F}_{Sum}(X\alpha_0\alpha_1 \parallel \Pi_k \lhd \Gamma_k) \rightarrow^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathbb{M}(\alpha'_0)\cdot\mathcal{F}_{Sum}(\alpha'_1\alpha_1) \parallel \mathcal{F}_{Sum}(\Pi_k \parallel \Pi(\mathbb{M}(w)) \lhd \Gamma_k \oplus \Gamma(\mathbb{M}(w)))$$

- $\alpha' \in \Sigma^{\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$
  Follows from Lemma 21

- $\alpha \in \Sigma^{\neg\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ and $\alpha' \in P_{\Pi_f}$
  Unless $\alpha' \in \mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^* \cup \Sigma^{\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ this case is covered by a case above. The remaining follows from Lemma 23.

- $\alpha \in \mathcal{N}$ and $\alpha' \in P_{\Pi_f}$
  Unless $\alpha' \in \mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^* \cup \Sigma^{\text{com}}\mathcal{N}^{\text{com}*}(\mathcal{N}^{\neg\text{com}}\mathcal{N}^{\text{com}*})^*$ this case is covered by a case above. The remaining follows from Lemma 19 and Lemma 21

This concludes the proof of the inductive step.

Now we apply the above for the case that $\Pi_n \lhd \Gamma_n = \Pi' \lhd \Gamma'$. We can then conclude that

$$\mathcal{F}_{Sum}(\Pi_0 \lhd \Gamma_0) \to^*_{\mathscr{P}[\mathcal{G}_{\text{alt}}]} \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$$

which concludes the proof. □

Corollary 11. If $\Pi_0 \lhd \Gamma_0 \to^*_{\mathcal{G}_{\text{std}}} \Pi' \lhd \Gamma'$ then for all $\bar{\Pi}' \lhd \Gamma' \in \mathcal{F}_{Sum}(\Pi' \lhd \Gamma')$ there exists $\bar{\Pi}_0 \lhd \Gamma_0 \in \mathcal{F}_{Sum}(\Pi_0 \lhd \Gamma_0)$ such that $\Pi_0 \lhd \Gamma_0 \to^*_{\mathcal{G}_{\text{alt}}} \Pi' \lhd \Gamma'$.

Theorem 7 (Equivalence of Simple Coverability). *Let $Q_{std} = (\mathcal{G}_{std}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$ and $Q_{alt} = (\mathcal{G}_{alt}, \Pi_0 \lhd \Gamma_0, \Pi \lhd \Gamma)$. Then, query Q_{std} is a yes-instance for simple coverabililty for the standard semantics if, and only if, Q_{alt} is a yes-instance for simple coverability for the alternative semantics.*

Proof. We will first prove the $\Longrightarrow$ -direction. Let Q_{std} be a yes-instance of the simple coverabililty problem and $\Pi = A_1 \parallel \cdots \parallel A_n$. Then, a configuration $A_1\alpha_1 \parallel \cdots \parallel A_n\alpha_n \parallel \Pi' \lhd \Gamma$ for some $\alpha_1, \ldots, \alpha_n \in (\Sigma \cup \mathcal{N})^*/_{\simeq_I}$ is $\to_{\mathcal{G}_{\text{std}}}$ reachable. By Corollary 11 $A_1\bar{\alpha}_1 \parallel \cdots \parallel A_n\bar{\alpha}_n \parallel \bar{\Pi}' \lhd \Gamma$ is reachable for $\to_{\mathcal{G}_{\text{alt}}}$ and thus Q_{alt} is a yes-instance.

For the $\Longleftarrow$-direction let Q_{alt} be a yes-instance of simple coverabililty. Then a configuration $\gamma_1 \parallel \cdots \parallel \gamma_n \parallel \Pi \lhd \Gamma$ is $\to_{\mathcal{G}_{\text{alt}}}$ reachable and for $i = 1, \ldots, n$ either $\gamma_i = A_i\gamma_i'$ or $\gamma_i = M_i\gamma_i'$ such that $A_i \in M_i$. By Proposition we can conclude that $\alpha_1 \parallel \cdots \parallel \alpha_n \parallel \Pi' \lhd \Gamma$ is $\to_{\mathcal{G}_{\text{std}}}$ reachable such that $\gamma_1 \parallel \cdots \parallel \gamma_n \parallel \Pi \lhd \Gamma \in \mathcal{F}_{Sum}(\alpha_1 \parallel \cdots \parallel \alpha_n \parallel \Pi' \lhd \Gamma)$. That means for $i = 1, \ldots, n$ either $\alpha_i = A_i\alpha_i'$ or $\alpha_i = \beta_i\alpha_i'$ such that $\beta_i \in \mathcal{N}^{\text{com}*}$ and $\beta_i \to^* w_i^0 w_i^1$ such that $\mathbb{M}(w_i^1) = M_i$ and $w_i^0 \in \Sigma^{\text{com}*}$. It follows, by using $\simeq_I$ where necessary and choosing rewrite rules to expose A_i, that $\beta_i \to^* w_i'^0 A_i \beta_i'$ where $w_i'^0 \in \Sigma^{\text{com}*}$ and $\beta_i \in \mathcal{N}^{\text{com}*}$. Hence $\alpha_1 \parallel \cdots \parallel \alpha_n \parallel \Pi' \lhd \Gamma \to^*_{\mathcal{G}_{\text{std}}} \alpha_1'' \parallel \cdots \parallel \alpha_n'' \parallel \Pi' \lhd \Gamma \oplus \Gamma(w_0'^0 \cdots w_n'^0)$ where either $\alpha_i'' = \alpha_i = A_i\alpha_i$ and $w_i'^0 = \epsilon$ or $\alpha_i'' = A_i\beta_i'$. Thus we can conclude that Q_{std} is a yes-instance for simple coverability. □

B.4 Proofs for Section 4.4

Lemma 3. The transition relation $\rightarrow_{\mathcal{G}_{\text{alt}}}$ is strictly monotone with respect to the well-order $\leq_{\mathcal{G}_{\text{alt}}}$.

Proof. Suppose $\Pi_1 \lhd \Gamma_1, \Pi_1' \lhd \Gamma_1', \Pi_2 \lhd \Gamma_2 \in \mathit{State}_{\text{alt}}$ such that $\Pi_1 \lhd \Gamma_1 <_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$ and $\Pi_1 \lhd \Gamma_1 \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$. We will show that there $\exists \Pi_2'$ such that $\Pi_2 \lhd \Gamma_2 \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ and $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$. Since $\Pi_1 \lhd \Gamma_1 \leq_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$ and $\Pi_1 \lhd \Gamma \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma'$, we can infer the components of the configuration involved in the latter transition. That means $\Pi_1 = \gamma \parallel \Pi_1^0$ and $\Pi_2 = \delta \parallel \Pi_2^0$ such that $\Pi_1^0 \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$, $\Gamma_1 \leq_{\mathit{Chans}} \Gamma_2$, $\gamma, \delta \in \mathit{Procs}_{\text{alt}}$ and $\gamma \leq_{\mathit{Procs}_{\text{alt}}} \delta$. Further, $\Pi_1' = \gamma' \parallel \Pi_1'^0$. Since $\Pi_1 \lhd \Gamma_1 <_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$, we further know that one of the following three cases holds: either $\Pi_1^0 <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$, $\Gamma_1 <_{\mathit{Chans}} \Gamma_2$, or $\gamma <_{\mathit{Procs}_{\text{alt}}} \delta$. Our proof will be by case analysis on $\Pi_1 \lhd \Gamma \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$.

- $\Pi_1 \lhd \Gamma_1 \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$ using Rules (R′-1) or (R′-2).
  We can immediately infer that $\Gamma_1 = \Gamma_1'$, $\Pi_1'^0 = \Pi_1^0$, $\gamma = A\, M_0\, X_1\, M_1\, X_2\, M_2 \cdots X_j\, M_j$ and $\delta = A\, M_0\, X_1 \mathbb{M}_1'\, X_2\, M_2' \cdots X_j\, M_j'$ with $M_i \leq_{\mathit{Summary}} M_i'$ for $i \leq j \leq k$. Further we know that there is a rule $A \rightarrow \beta \in \mathcal{G}$. We perform a case analysis on whether β satisfies (†).

  - β satisfies (†).
    Thus, $\beta = B\,C$, C commutative, $C \rightarrow_{\text{seq}}^* w$ and $\gamma' = B(\mathbb{M}(w) \oplus M_0) X_1 M_1 \cdots X_j M_j$. Hence, if we define $\delta' = B(\mathbb{M}(w) \oplus M_0') X_1 M_1' X_2 M_2' \cdots X_j M_j'$ then

    $$\delta \parallel \Pi_2^0 \lhd \Gamma_2 \rightarrow_{\mathcal{G}_{\text{alt}}} \delta' \parallel \Pi_2^0 \lhd \Gamma_2 =: \Pi_2' \lhd \Gamma_2'.$$

    Clearly $(\mathbb{M}(w) \oplus M_0) \leq_{\mathit{Summary}} (\mathbb{M}(w) \oplus M_0')$ and thus $\gamma' \leq_{\mathit{Procs}_{\text{alt}}} \delta'$. If $\Pi_1^0 <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ or $\Gamma_1 <_{\mathit{Chans}} \Gamma_2$, then $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ is immediate. Otherwise, $\gamma <_{\mathit{Procs}_{\text{alt}}} \delta$ thus two cases arise: $M_0 <_{\mathit{Summary}} M_0'$ or $X_1 M_1 \cdots X_j M_j <_{\mathit{Procs}_{\text{alt}}} X_1' M_1 \cdots X_j' M_j$. From both cases $\gamma' <_{\mathit{Procs}_{\text{alt}}} \delta'$ follows immediately. Hence we can conclude $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ as desired.
  - β does not satisfy (†).
    Thus $\gamma' = \beta\, M_0 X_1 M_1 \cdots X_j M_j$. Further since γ' in $\mathit{Procs}_{\text{alt}}$ it is the case that $j < k$. If we define $\delta' := \beta\, M_0' X_1 M_1' \cdots X_j M_j'$, we can see that

    $$\delta \parallel \Pi_2^0 \lhd \Gamma_2 \rightarrow_{\mathcal{G}_{\text{alt}}} \delta' \parallel \Pi_2^0 \lhd \Gamma_2.$$

    Thus, clearly $\gamma' \leq_{\mathit{Procs}_{\text{alt}}} \delta'$. If $\Pi_1^0 <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ or $\Gamma_1 <_{\mathit{Chans}} \Gamma_2$, then $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ is immediate. Otherwise, $\gamma <_{\mathit{Procs}_{\text{alt}}} \delta$ thus $M_0 X_1 M_1 \cdots X_j M_j <_{\mathit{Procs}_{\text{alt}}} M_0 X_1' M_1 \cdots X_j' M_j$. This implies $\gamma' <_{\mathit{Procs}_{\text{alt}}} \delta'$ follows immediately. Hence we can conclude $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ as desired.

- $\Pi_1 \lhd \Gamma_1 \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$ using (R′-3).
  Thus we can conclude (i) $\gamma = c\,?\,m\gamma'$, (ii) $\Pi_1' = \gamma' \parallel \Pi_1^0$, (iii) $\Gamma_1 = \Gamma_1' \oplus [c \mapsto [m]]$. Further since $\Pi_1 \lhd \Gamma_1 \leq_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$ we infer (iv) $\delta = c\,?\,m\delta'$ with $\gamma' \leq_{\mathit{Procs}_{\text{alt}}} \delta'$ and (v) $\Gamma_2 = \Gamma_2' \oplus [c \mapsto [m]]$.

Then we have $\Pi_2 \lhd \Gamma_2 \to_{\mathcal{G}_{\text{alt}}} \delta' \parallel \Pi_2^0 \lhd \Gamma_2'$.

Writing $\Pi_2' := \delta' \parallel \Pi_2^0$ it remains to show $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$.

Now (a) $\gamma' \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \delta'$, (b) $\Pi_1^0 \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ by assumption and (c) since $\Gamma_1 \leq_{\mathit{Chans}} \Gamma_2$ and clearly $\Gamma_1' \leq_{\mathit{Chans}} \Gamma_2'$.

Hence we conclude $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$.

- $\Pi_1 \lhd \Gamma_1 \to_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$ using (R′-4).
Thus we can conclude (i) $\gamma = c\,!\,m\,\gamma'$, (ii) $\Pi_1' = \gamma' \parallel \Pi_1^0$, (iii) $\Gamma_1' = \Gamma_1 \oplus [c \mapsto [m]]$. Further since $\Pi_1 \lhd \Gamma_1 \leq_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$ we infer (iv) $\delta = c\,!\,m\,\delta'$ with $\gamma' \leq_{\mathit{Procs}_{\text{alt}}} \delta'$.

Then we have $\Pi_2 \lhd \Gamma_2 \to_{\mathcal{G}_{\text{alt}}} \delta' \parallel \Pi_2^0 \lhd \Gamma_2 \oplus [c \mapsto [m]]$.

Writing $\Pi_2' := \delta' \parallel \Pi_2^0$ and $\Gamma_2' := \Gamma_2 \oplus [c \mapsto [m]]$ it remains to show $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$.

Now (a) $\gamma' \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \delta'$, (b) $\Pi_1^0 \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ by assumption and (c) since $\Gamma_1 \leq_{\mathit{Chans}} \Gamma_2$, $\oplus$ and $\Gamma(\cdot)$ monotonic we have $\Gamma_1 \oplus [c \mapsto [m]] \leq_{\mathit{Chans}} \Gamma_2 \oplus [c \mapsto [m]]$.

Hence we conclude $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$. Further, if $\Pi_1^0 <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ or $\Gamma_1 <_{\mathit{Chans}} \Gamma_2$, then $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ is immediate. Otherwise, $\gamma <_{\mathit{Procs}_{\text{alt}}} \delta$ and thus $\gamma' <_{\mathit{Procs}_{\text{alt}}} \delta'$ follows immediately. Hence we can conclude $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ as desired.

- $\Pi_1 \lhd \Gamma_1 \to_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$ using (R′-5).
Thus we can conclude (i) $\gamma = (\nu X)\,\gamma'$, (ii) $\Pi_1' = \gamma' \parallel \Pi_1^0 \parallel X$, (iii) $\Gamma_1' = \Gamma_1$. Further since $\Pi_1 \lhd \Gamma_1 \leq_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$ we infer (iv) $\delta = (\nu X)\,\delta'$ with $\gamma' \leq_{\mathit{Procs}_{\text{alt}}} \delta'$.

Then we have $\Pi_2 \lhd \Gamma_2 \to_{\mathcal{G}_{\text{alt}}} \delta' \parallel \Pi_2^0 \parallel X \lhd \Gamma_2$.

Writing $\Pi_2' := \delta' \parallel \Pi_2^0 \parallel X$ it remains to show $\Pi_1' \lhd \Gamma_1 \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2$. By assumption (a) $\gamma' \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \delta'$, (b) $\Pi_1^0 \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ and (c) clearly $X \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} X$.

Hence we conclude $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$. Further, if $\Pi_1^0 <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ or $\Gamma_1 <_{\mathit{Chans}} \Gamma_2$, then $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ is immediate. Otherwise, $\gamma <_{\mathit{Procs}_{\text{alt}}} \delta$ and thus $\gamma' <_{\mathit{Procs}_{\text{alt}}} \delta'$ follows immediately. Hence we can conclude $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ as desired.

- $\Pi_1 \lhd \Gamma_1 \to_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$ using (R′-6).
Thus we can conclude (i) $\gamma = M_1 X \gamma_1$, (ii) $\Pi_1' = \gamma' \parallel \Pi_1^0 \parallel \Pi(M_1)$, (iii) $\gamma' = X\gamma_1$ and (iv) $\Gamma_1' = \Gamma_1 \oplus \Gamma(M_1)$. Further since $\Pi_1 \lhd \Gamma_1 \leq_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$ we infer (v) $\delta = M_2 X \delta_1$ with $M_1 \leq_{\mathit{Summary}} M_2$ and $X\gamma_1 \leq_{\mathit{Procs}_{\text{alt}}} X\delta_1$.

Let $\delta' := X\delta_1$ then clearly $\gamma' \leq_{\mathit{Procs}_{\text{alt}}} \delta'$ and we have

$$\Pi_2 \lhd \Gamma_2 \to_{\mathcal{G}_{\text{alt}}} \delta' \parallel \Pi_2^0 \parallel \Pi(M_2) \lhd \Gamma_2 \oplus \Gamma(M_2).$$

Writing $\Pi_2' := \delta' \parallel \Pi_2^0 \parallel \Pi(M_2)$ and $\Gamma_2' := \Gamma_2 \oplus \Gamma(M_2)$ we will now show $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$.

Now (a) $\gamma' \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \delta'$, (b) $\Pi_1^0 \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ by assumption, (c) since $M_1 \leq_{\mathit{Summary}} M_2$ and since $\Pi(-)$ is clearly monotonic $\Pi(M_1) \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi(M_2)$ and (d) lastly since $\Gamma(-)$ is monotonic we can conclude $\Gamma(M_1) \leq_{\mathit{Chans}} \Gamma(M_2)$.

Hence we conclude $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$. Further, if $\Pi_1^0 <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ or $\Gamma_1 <_{\mathit{Chans}} \Gamma_2$, then $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ is immediate. Otherwise, $\gamma <_{\mathit{Procs}_{\text{alt}}} \delta$ and thus there are two possible cases: either $M_1 <_{\mathit{Summary}} M_2$ or $X\gamma_1 <_{\mathit{Procs}_{\text{alt}}} X\delta_1$. The latter case immediately implies $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$. While the former, implies one of $\Pi(M_1) <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi(M_2)$ or $\Gamma(M_1) <_{\mathit{Chans}} \Gamma(M_2)$ which immediately implies $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ as desired.

- $\Pi_1 \lhd \Gamma_1 \rightarrow_{\mathcal{G}_{\text{alt}}} \Pi_1' \lhd \Gamma_1'$ using (R′-7).
  Thus we can conclude (i) $\gamma = M_1 X\gamma_1$ with $M_1 \in \mathit{PartialSummary}$, (ii) $\Pi_1' = \gamma' \parallel \Pi_1^0 \parallel \Pi(M_1)$, (iii) $\gamma' = M_1' X\gamma_1$ with $M_1' \in \mathit{DegenSummary}$ and $M_1' = M_1 \upharpoonright (\mathcal{N} \cup \mathcal{L})$, (iv) $\Gamma_1' = \Gamma_1 \oplus \Gamma(M_1)$. Further since $\Pi_1 \lhd \Gamma_1 \leq_{\mathcal{G}_{\text{alt}}} \Pi_2 \lhd \Gamma_2$ we infer (v) $\delta = M_2 X\delta_1$ with $M_1 \leq_{\mathit{Summary}} M_2$ and $X\gamma_1 \leq_{\mathit{Procs}_{\text{alt}}} X\delta_1$.

  Let $M_2' := M_2 \upharpoonright (\mathcal{N} \cup \mathcal{L})$ and $\delta' := M_2' X\delta_1$ then we have

  $$\Pi_2 \lhd \Gamma_2 \rightarrow_{\mathcal{G}_{\text{alt}}} \delta' \parallel \Pi_2^0 \parallel \Pi(M_2) \lhd \Gamma_2 \oplus \Gamma(M_2).$$

  Writing $\Pi_2' := \delta' \parallel \Pi_2^0 \parallel \Pi(M_2)$ and $\Gamma_2' := \Gamma_2 \oplus \Gamma(M_2)$ we will now show $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$.

  Now since $\cdot \upharpoonright \cdot$ is monotonic in the first argument and $M_1 \leq_{\mathit{Summary}} M_2$ we conclude $M_1' \leq_{\mathit{Summary}} M_2'$ and thus (a) $\gamma' \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \delta'$, (b) $\Pi_1^0 \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ by assumption, (c) and since $\Pi(-)$ is monotonic $\Pi(M_1) \leq_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi(M_2)$ and (d) lastly since $\Gamma(-)$ is monotonic we can conclude $\Gamma(M_1) \leq_{\mathit{Chans}} \Gamma(M_2)$.

  Hence we conclude $\Pi_1' \lhd \Gamma_1' \leq_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$. Further, if $\Pi_1^0 <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi_2^0$ or $\Gamma_1 <_{\mathit{Chans}} \Gamma_2$, then $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ is immediate. Otherwise, $\gamma <_{\mathit{Procs}_{\text{alt}}} \delta$ and thus there are two possible cases: either $M_1 <_{\mathit{Summary}} M_2$ or $X\gamma_1 <_{\mathit{Procs}_{\text{alt}}} X\delta_1$. The latter case immediately implies $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$. While the former, implies one of $M_1' <_{\mathit{Summary}} M_2'$, $\Pi(M_1) <_{\mathbb{M}[\mathit{Procs}_{\text{alt}}]} \Pi(M_2)$, $\Gamma(M_1) <_{\mathit{Chans}} \Gamma(M_2)$ which immediately implies $\Pi_1' \lhd \Gamma_1' <_{\mathcal{G}_{\text{alt}}} \Pi_2' \lhd \Gamma_2'$ as desired.

□

Appendix C

Appendix for Chapter 5

C.1 Proofs for Section 5.1

Lemma 4. $(\mathcal{M}^{colour}, \leq_{\mathcal{M}^{colour}})$ and $(\mathit{Config}, \leq_{\mathit{Config}})$ are WQO.

Proof. Let for all $P \subseteq \mathcal{P}^{col}$ define $M(P)$ to be set of complex tokens that exactly contain coloured tokens only of colours not in P, i.e.

$$M(P) = \left\{ m \in \mathcal{M}^{colour} : \forall p \in P.m(p) = \emptyset, \forall p' \notin p'.m(p') \neq \emptyset \right\}.$$

It should be clear that for each $m \in \mathcal{M}^{colour}$ there is a unique $P \subseteq \mathcal{P}^{col}$ such that $m \in M(P)$. Hence we can see that $\mathcal{M}^{colour}$ is isomorphic to the *finite* disjoint union $\sum_{P \subseteq \mathcal{P}^{col}} M(P)$. Let $P \subseteq \mathcal{P}^S$ we can see that $M(P)$ is isomorphic to $\Pi_{p \in P} \{\emptyset\} \times \Pi_{p \notin P} \mathbb{M}\{\bullet\}$ and so clearly $\leq_{M(P)} := \Pi_{p \in P} = \times \Pi_{p \notin P} \subseteq_{\mathbb{M}\{\bullet\}}$ is a WQO (eliding the isomorphism) for $M(P)$. Hence $\sum_{P \subseteq \mathcal{P}^{col}} \leq_{M(P)}$ is a WQO for $\sum_{P \subseteq \mathcal{P}^{col}} M(P)$ and thus for $\mathcal{M}^{colour}$ (eliding the isomorphism).

Suppose $m, m' \in \mathcal{M}^{colour}$ such that $m \leq_{\mathcal{M}^{colour}} m'$. Let $P = \{p : m(p) = \emptyset\}$ since $m \leq_{\mathcal{M}^{colour}} m'$ it is easy to see that $P = \{p : m'(p) = \emptyset\}$. Hence $m, m' \in M(P)$ and we also know that for all $p \notin P$ we have $0 < |m(p)| \leq |m(p')|$, i.e. $m(p) \subseteq_{\mathbb{M}[\{\bullet\}]} m'(p)$, hence $m \leq_{M(P)} m'$ and thus $m \leq_{\sum_{P \subseteq \mathcal{P}^{col}} M(P)} m'$. In the opposite direction suppose m, m' are such $m \leq_{\sum_{P \subseteq \mathcal{P}^{col}} M(P)} m'$ then both $m, m' \in M(P)$ for some $P \subseteq \mathcal{P}^{col}$ and $m \leq_{M(P)} m'$ which means that for all $p \in P$ it is the case that $0 = |m(p)| = |m'(p)| = |\emptyset|$ and for all $p \notin P$ we have both $m(p), m'(p) \neq \emptyset$ and $m(p) \subseteq_{\mathbb{M}[\{\bullet\}]} m'(p)$, i.e. $0 < |m(p)| \leq |m'(p)|$ and hence $m \leq_{\mathcal{M}^{colour}} m'$. $(\mathcal{M}^{colour}, \leq_{\mathcal{M}^{colour}})$ is then isomorphic to a WQO and hence is a WQO.

For $(\mathit{Config}, \leq_{\mathit{Config}})$. It is clear that $\left(\mathbb{M}[\bullet], \subseteq_{\mathbb{M}[\bullet]}\right)$ is a WQO; since $\mathcal{P}^S$ is finite, Dickson's lemma tells us that $(\mathcal{M}^{simple}, \leq_{\mathcal{M}^{simple}})$ is a WQO where $\leq_{\mathcal{M}^{simple}}$ is the extension from $\subseteq_{\mathbb{M}[\bullet]}$ to $\subseteq_{\mathcal{P}^S \to \mathbb{M}[\bullet]}$. Further since multi sets over a WQO are again a WQO, $(\mathbb{M}[\mathcal{M}^{colour}], \subseteq_{\mathbb{M}[\mathcal{M}^{colour}]})$ is a WQO and so, we can infer that $(\mathit{Config}, \leq_{\mathit{Config}})$ is a WQO which concludes the proof. □

Lemma 26. $\oplus$ is monotone in both arguments with respect to $\leq_{\mathit{Config}}$ and $\leq_{\mathcal{M}^{colour}}$.

Proof. Suppose we have $m, m', m_0, m_0' \in \mathcal{M}^{colour}$ such that $m \leq_{\mathcal{M}^{colour}} m_0$ and $m' \leq_{\mathcal{M}^{colour}} m_0'$. Let $p \in \mathcal{P}^{col}$ If $|(m \oplus m')(p)| = 0$ we can conclude that $|m(p)| = |m'(p)| = 0$ then $\leq_{\mathcal{M}^{colour}}$ guarantees that $|m(p)| = |m_0(p)| = 0$ and $|m'(p)| = |m_0'(p)| = 0$ and hence $|(m_0 \oplus m_0')(p)| = 0$.

Otherwise either $|m(p)|$ or $|m'(p)| > 0$. W.l.o.g. assume $|m(p)| > 0$ then we know that $|m(p)| \leq_{\mathcal{M}^{colour}} |m_0(p)|$ and $|m_0(p)| > \mathbf{0}$. Further we can easily see that $|m'(p)| \leq |m'_0(p)|$. Hence we can see that $0 < |(m \oplus m')(p)| \leq |(m_0 \oplus m'_0)(p)|$. Hence we can deduce that $m \oplus m' \leq_{\mathcal{M}^{colour}} m_0 \oplus m'_0$ which is what we want to prove.

Suppose we have $s, s', s_0, s'_0 \in \mathit{Config}$ and $s \leq_{\mathit{Config}} s_0$ and $s' \leq_{\mathit{Config}} s'_0$. Let $p \in \mathcal{P}^S$ then clearly

$$|(s \oplus s')(p)| = |s(p)| + |s'(p)| \leq |s_0(p)| + |s'_0(p)| = |(s_0 \oplus s'_0)(p)|.$$

Further let $p' \in \mathcal{P}^C$ then

$$\begin{aligned}(s \oplus s')(p') &= \left[m_1, \ldots, m_n, m'_1, \ldots, m'_{n'}\right] \text{ and}\\ (s_0 \oplus s'_0)(p') &= \left[M_1, \ldots, M_N, M'_1, \ldots, M'_{N'}\right]\end{aligned}$$

where we may assume:

$$\begin{aligned} s(p') &= [m_1, \ldots, m_n], & s'(p') &= \left[m'_1, \ldots, m'_{n'}\right],\\ s_0(p') &= [M_1, \ldots, M_N] \text{ and} & s'_0(p') &= \left[M'_1, \ldots, M'_{N'}\right].\end{aligned}$$

Further since $s \leq_{\mathit{Config}} s_0$ and $s' \leq_{\mathit{Config}} s'_0$ we know that $n \leq N$ and $n' \leq N'$ and for all $i \in \langle n \rangle$ we have $m_i \leq_{\mathcal{M}^{colour}} M_i$ and $m'_j \leq_{\mathcal{M}^{colour}} M'_j$ for all $j \in \langle n' \rangle$. This gives us an injection that pairs up m_i with M_i and m'_j with M'_j for $i \in \langle n \rangle$ and $j \in \langle n' \rangle$. We can thus conclude that $(s \oplus s')(p') \leq_{\mathbb{M}[\mathcal{M}^{colour}]} (s_0 \oplus s'_0)(p')$ which of course implies $s \oplus s' \leq_{\mathit{Config}} s_0 \oplus s'_0$. □

Lemma 27. Suppose we have $s, s_0, s' \in \mathit{Config}$, $s \leq_{\mathit{Config}} s_0$ and $s'(p)(m) = 0$ if both $p \in \mathcal{P}^C$ and $m \neq \mathbf{0}$ then $s \ominus s' \leq s_0 \ominus s'$.

Proof. Let $p \in \mathcal{P}^S$ then clearly

$$|(s \ominus s')(p)| = |s(p)| - |s'(p)| \leq |s_0(p)| - |s'(p)| = |(s_0 \ominus s')(p)|.$$

Further let $p' \in \mathcal{P}^C$ then

$$\begin{aligned}(s \ominus s')(p')(\mathbf{0}) = s(p')(\mathbf{0}) - s'(p')(\mathbf{0}) \leq s_0(p')(\mathbf{0}) - s'(p')(\mathbf{0}) = (s_0 \ominus s')(p')(\mathbf{0})\\ (s \ominus s')(p') \restriction \{\mathbf{0}\}^{\mathrm{c}} = s(p') \restriction \{\mathbf{0}\}^{\mathrm{c}} \leq s_0(p') \restriction \{\mathbf{0}\}^{\mathrm{c}} = (s_0 \ominus s')(p') \restriction \{\mathbf{0}\}^{\mathrm{c}}\end{aligned}$$

Hence $s \oplus s' \leq_{\mathit{Config}} s_0 \oplus s'_0$. □

Lemma 28. Let $p \in \mathcal{P}^C$ and $s, s' \in \mathit{Config}$. Suppose $m, m' \in \mathcal{M}^{colour}$ such that

$$m' = \min\left\{m_0 \in s'(p) : m \leq_{\mathcal{M}^{colour}} m_0\right\}.$$

If $s \oplus [p \mapsto m] \leq_{\mathit{Config}} s_0 \oplus [p \mapsto m'] = s'$ then $s \leq_{\mathit{Config}} s_0$.

Proof. Let $p \in \mathcal{P}^S$ then clearly

$$|s(p)| = |(s \oplus [p \mapsto m])(p)| \leq |(s_0 \oplus \left[p \mapsto m'\right])(p)| = |s_0(p)|.$$

Further let $p' \in \mathcal{P}^C$ such that $p \neq p'$ then

$$s(p') = (s \oplus [p \mapsto m])(p') \leq_{\mathbb{M}[\mathcal{M}^{colour}]} (s_0 \oplus [p \mapsto m])(p') = s_0(p').$$

Focussing on p we see:

$$s(p) = [m_1, \ldots, m_n], \qquad s_0(p) = [m'_1, \ldots, m'_{n'}],$$

where we know that

$$(s \oplus [p \mapsto m])(p) = [m_1, \ldots, m_n, m] \text{ and } (s_0 \oplus [p \mapsto m'])(p) = [m'_1, \ldots, m_{n'}, m']$$

where $n \leq n'$ and there is an bijection h from $M := [m_1, \ldots, m_n, m]$ to $M' := [m'_1, \ldots, m_n, m']$. such that $m^0 \leq_{\mathcal{M}^{colour}} h(m)^0$ for all $m^0 \in M$. Suppose h pairs up m with m' then clearly h is an injection justifying $s(p) \leq_{\mathbb{M}[\mathcal{M}^{colour}]} s_0(p)$. Otherwise say $h(m_i) = m'$ then since $m \leq_{\mathcal{M}^{colour}} h(m)$ we know that $m' \leq_{\mathcal{M}^{colour}} h(m)$ since m' is a minimum. Thus $h\,[m_i \mapsto h(m)]$ is an injection justifying $s(p) \leq_{\mathbb{M}[\mathcal{M}^{colour}]} s_0(p)$. We can thus conclude $s \leq_{Config} s_0$. □

Lemma 29. $\restriction$ is monotone in the first argument with respect to $\leq_{\mathcal{M}^{colour}}$.

Proof. Let $m, m' \in \mathcal{M}^{colour}$ such that $m \leq_{\mathcal{M}^{colour}} m'$ and $P \subseteq \mathcal{P}^{col}$. Suppose $p \in P$ and $0 < m(p)$ then $0 < |m(p)| = |(m \restriction P)(p)| = |m(p)| \leq |m'(p)| = |(m' \restriction P)(p)|$. If $p \in P$ and $|m(p)| = 0$ then $0 = |m(p)| = |(m \restriction P)(p)| = |m(p)| = |m'(p)| = |(m' \restriction P)(p)|$. Otherwise $p' \notin P$ then $|(m \restriction P)(p')| = 0 = |(m' \restriction P)(p')|$. Hence clearly $m \restriction P \leq_{\mathcal{M}^{colour}} m' \restriction P$. □

Proposition 5. Given an NNCT $\mathcal{N}$, $(Config, \to_{\mathcal{N}}, \leq_{Config})$ is a strict WSTS.

Proof. Let $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$ and suppose $s, s' \in Config$ such that $s <_{Config} s'$. Further suppose we can make the transition $s \xrightarrow{r}_{\mathcal{N}} t$ using rule $r \in \mathcal{R}$. Let us perform a case analysis on r

- *Case:* $r = (I, O) \in SimpleRules$.
  Since r is enabled at s we know that $s \ominus I \in Config$. Lemma 27 then yields that $s \ominus I \leq_{Config} s' \ominus I$ and hence clearly r is enabled at s'. Thus $s \xrightarrow{r}_{\mathcal{N}} s' \ominus I \oplus O =: t'$. Since $t = s \ominus I \oplus O$ Lemma 26 gives us $t \leq_{Config} t'$. Since $s \neq s'$ it is clear that $t \neq t'$ and hence we obtain $t <_{Config} t'$ which is what we want to prove.

- *Case:* $r = ((p, I), (p', c, O)) \in ComplexRules$.
  Since r is enabled at s we know that for some $m \in s(p)$, $s \ominus I\,[p \mapsto m] \in Config$ and

  $$t = s \ominus I\,[p \mapsto m] \oplus O \oplus [p' \mapsto m \oplus c].$$

  First Lemma 27 then yields that $s \ominus I \leq_{Config} s' \ominus I$. Since $s <_{Config} s'$ there exists $m' \in s'(p)$ such that $m \leq_{\mathcal{M}^{colour}} m'$; w.l.o.g. we can assume that $m' = \min\{m_0 \in s'(p) : m \leq_{\mathcal{M}^{colour}} m_0\}$. Since $I \in Config_S$ it is also the case that $m' = \min\{m_0 \in s'(p) \ominus I : m \leq_{\mathcal{M}^{colour}} m_0\}$. Lemma 28 then yields that $s \ominus I \ominus [p \mapsto m] \leq_{Config} s' \ominus I \ominus [p \mapsto m']$ hence it is easy to see that r is enabled at s'. Further

  $$s' \xrightarrow{r}_{\mathcal{N}} s' \ominus I\,[p \mapsto m'] \oplus O \oplus [p' \mapsto m' \oplus c] =: t'.$$

Since $m \leq_{\mathcal{M}^{colour}} m'$ Lemma 26 yields $m \oplus c \leq_{\mathcal{M}^{colour}} m' \oplus c$. Hence it is easy to see $[p' \mapsto m \oplus c] \leq_{Config} [p \mapsto m' \oplus c]$. Lemma 26 then gives us that $t \leq_{Config} t'$. Since $s \neq s'$ either $s' \ominus I\,[p \mapsto m'] \neq s \ominus I\,[p \mapsto m]$ or $m \neq m'$. Noting this we can see that $t \neq t'$ and hence $t <_{Config} t'$ which is what we want to prove.

- *Case:* $r = ((p, I), (p', P, O)) \in$ *TransferRules*.
Since r is enabled at s we know that for some $m \in s(p)$ $s \ominus I\,[p \mapsto m] \in$ *Config* and

$$t = s \ominus I\,[p \mapsto m] \oplus O \oplus \left[p' \mapsto m_{\overline{P}}\right] \oplus (m_P \circ \zeta^{-1})$$

where $m_P = m \restriction P$ and $m_{\overline{P}} = m \restriction (\mathcal{P}^{col} \setminus P)$. First Lemma 27 then yields that $s \ominus I \leq_{Config} s' \ominus I$. Since $s <_{Config} s'$ there exists $m' \in s'(p)$ such that $m \leq_{\mathcal{M}^{colour}} m'$; w.l.o.g. we can assume that $m' = \min\{m_0 \in s'(p) : m \leq_{\mathcal{M}^{colour}} m_0\}$. Since $I \in Config_S$ it is also the case that $m' = \min\{m_0 \in s'(p) \ominus I : m \leq_{\mathcal{M}^{colour}} m_0\}$. Lemma 28 then yields that $s \ominus I\,[p \mapsto m] \leq_{Config} s' \ominus I\,[p \mapsto m']$ hence it is easy to see that r is enabled at s'. Further

$$s' \xrightarrow{r}_{\mathcal{N}} s' \ominus I\left[p \mapsto m'\right] \oplus O \oplus \left[p' \mapsto m'_{\overline{P}}\right] \oplus (m'_P \circ \zeta^{-1}) =: t'.$$

where $m_P = m \restriction P$ and $m_{\overline{P}} = m \restriction (\mathcal{P}^{col} \setminus P)$. Since $m \leq_{\mathcal{M}^{colour}} m'$ Lemma 29 yields both $m_P \leq_{\mathcal{M}^{colour}} m'_P$ and $m_{\overline{P}} \leq_{\mathcal{M}^{colour}} m'_{\overline{P}}$. Hence it is easy to see $\left[p' \mapsto m_{\overline{P}}\right] \leq_{Config} \left[p \mapsto m'_{\overline{P}}\right]$. Further clearly $|(m_P \circ \zeta^{-1})(p)| \leq |(m_P \circ \zeta^{-1})(p)|$ for all $p \in \mathcal{P}^S$ and thus $(m_P \circ \zeta^{-1}) \leq_{Config} (m_P \circ \zeta^{-1})$. Lemma 26 then gives us that $t \leq_{Config} t'$. Since $s \neq s'$ either $s' \ominus I\,[p \mapsto m'] \neq s \ominus I\,[p \mapsto m]$ or $m \neq m'$. The later implies that either $m_P \neq m'_P$ or $m_{\overline{P}} \neq m'_{\overline{P}}$. Noting this we can see that $t \neq t'$ and hence $t <_{Config} t'$ which is what we want to prove.

□

C.2 Proofs for Section 5.2

Theorem 10. *Simple coverability for K-shaped APCPS in the alternative semantics EXPTIME-time reduces to simple coverability for NNCT.*

Proof. Fix a K-shaped APCPS $\mathcal{G} = (\Sigma, I, \mathcal{N}, \mathcal{R}, S)$ from $\Pi_0 \lhd \Gamma_0$ where $K \geq 1$ and a simple coverability query $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, A_1^{\text{cov}} \parallel \cdots \parallel A_n^{\text{cov}} \lhd \Gamma^{\text{cov}})$. We first define a simulating NNCT $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$.

- For each *msg* $\in$ *Msg* and *c* $\in$ *Chan*, we introduce a simple place $p^S_{c,msg}$.
- For each $X \in \mathcal{N}$, we introduce a simple place $p^S_{\nu X}$.
- For each $0 \leq k \leq K$, $X_k \cdots X_1 \in (\mathcal{N}^{\neg\text{com}})^*$, $\ell_{k+1}\,\ell_k \cdots \ell_1 \in (\{A_i^{\text{cov}} : i \in \langle n \rangle\} \cup \{+,-\})^*$ and $\$ \in \Sigma \cup \{\epsilon\} \cup \mathcal{N}$, we introduce a complex place $p^C_{\$',\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1}$; we also introduce a complex place $p^C_{\$',(+)\,N\,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1}$ for each $\$' \in \Sigma \cup \{\epsilon\}$ and $N \in \mathcal{N}$.
- We introduce an auxiliary simple place p^{CCFG}_{budget} and for each $X \in \mathcal{N}$, we introduce a simple place p^{CCFG}_X; for each $0 \leq k \leq K$, $X_k \cdots X_1 \in (\mathcal{N}^{\neg\text{com}})^*$, $\ell_{k+1}\,\ell_k \cdots \ell_1 \in (\mathscr{P}[\{A_i^{\text{cov}} : i \in \langle n \rangle\}] \cup \{+\})^*$ and $\$ \in \Sigma \cup \{\epsilon\} \cup \mathcal{N}$, we introduce a complex place $p^{CCFG}_{\$,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1}$, which will be used by $\mathcal{N}$ to implement a CCFG widget.
- For each non-terminal A_i^{cov} in the coverability query where $i \in \langle n \rangle$ $\mathcal{N}$ has a simple place $p^{cov}_{A_i}$.
- The NNCT $\mathcal{N}$ will further have four special simple places: p^{sim}, p^{CCFG}, p^{CCFG+} and p^{Query}.
- For each $1 \leq k \leq K+1$ and $e \in \Sigma^{\text{com}}$, we introduce a colour $p^I_{k,e}$.
- We define a map $\zeta : \mathcal{P}^{col} \to \mathcal{P}^S$ by $p^I_{k,c!msg} \mapsto p^S_{c,m}$ and $p^I_{k,\nu X} \mapsto p^S_{\nu X}$.

Let us further define three special simple markings $s_{sim} = [p^{sim} \mapsto [\bullet]]$, $s_{CCFG} = [p^{CCFG} \mapsto [\bullet]]$, $s_{CCFG+} = [p^{CCFG+} \mapsto [\bullet]]$ and $s_{Query} = [p^{Query} \mapsto [\bullet]]$.

An APCPS configuration $\Pi \lhd \Gamma \in State_{\text{alt}}$ is represented as an NNCT configuration as follows:

- For each *c* $\in$ *Chan* and *msg* $\in$ *Msg* place $p^S_{c,msg}$ contains precisely one $\bullet$-token for each occurrence of *msg* in *c* — we can formalise this as a function $\mathcal{F}^\Gamma(\Gamma) = [p^S_{c,msg} \mapsto [\bullet^{\Gamma(c)(msg)}] \mid c \in Chan, msg \in Msg]$.
- Let $0 \leq k \leq K+1$. The representation of the state of a process $\pi = \$\,M_{k+1}\,\gamma \in \Pi$ with $\gamma = X_k\,M_k \cdots X_1\,M_1$ is defined by a case analysis in three cases – a general case and two edge cases:
  - If either $0 < k \leq K$ or $k = 0$ and it is not the case that both $\$ \in \mathcal{N}$ and $M_1 = \emptyset$ then we represent π as follows: (i) for each $e \in M_i$ there is one $p^I_{i,e}$-coloured token in m where $1 \leq i \leq k+1$ and $e \in \Sigma^{\text{com}}$ — or equivalently $m = \mathcal{F}^I(M_1, \ldots, M_{k+1})$ where we define the function $\mathcal{F}^I(M_1, \ldots, M_{k+1}) = [p^I_{i,e} \mapsto [\bullet^{M_i(e)}] \mid 1 \leq i \leq k+1, e \in \Sigma^{\text{com}}]$; (ii) for each $1 \leq i \leq k+1$, if the cache $M_i \in$ *CompleteSummary* then let $\ell_i = +$; otherwise let $\ell_i \in \{-\} \cup \{A_j^{\text{cov}} \mid j \in \langle n \rangle, M_i(A_j^{\text{cov}}) > 0\}$. We refer to a possible value of ℓ_i as a *character* of M_i. The sequence of \$, the non-commutative non-terminals $X_k \cdots X_1$ and a possible choice of characters $\ell_{k+1}, \cdots, \ell_1$ of $M_{k+1}, \cdots M_1$ is represented as the complex place in which m located, i.e. m is placed in $p^C_{\$,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1}$. We formalise the representation of one process as the set of markings

$\mathcal{F}^\pi(\$\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1) = \{[p^C_{\$,\ell_{k+1}\, X_k\, \ell_k \cdots X_1\, \ell_1} \mapsto [\mathcal{F}^I(M_1, \ldots, M_{k+1})]] \mid \ell_i \text{ character of } M_i, i \in \langle k+1 \rangle\}$, where the possible characters of $M_{k+1}, \cdots M_1$ can be thought of non-deterministically chosen.

- If $k = 0$, $\$ \in \mathcal{N}$, and $M_1 = \emptyset$ then we may represent π in addition to the representation above also as a $\bullet$ token in $p^S_{\nu\$}$ hence the representation of π is defined as $\mathcal{F}^\pi(\pi) = \{[p^S_{\nu\$} \mapsto [\bullet]]\} \cup \mathcal{F}^\pi(\$\, \emptyset)$ where $\mathcal{F}^\pi(\$\, \emptyset)$ is as defined in the general case above.
- If $k = K+1$ and $\pi = \$\, M_{K+2}\, \gamma \in \Pi$ with $\gamma = X_{K+1}\, M_{K+1}\, X_K\, M_K \cdots X_1\, M_1$ then since $\mathcal{G}$ is a K-shaped APCPS it will be the case that $M_{K+2} = \emptyset$ and $X_{K+1} \in \mathcal{N}$ and $\$ \in \Sigma \cup \epsilon$. We notice that any character for M_{k+2} must be $+$ and so $\$\, M_{K+2}\, \gamma$ is represented by a complex token $m = \mathcal{F}^I(M_1, \ldots, M_{K+2}) := \mathcal{F}^I(M_1, \ldots, M_{K+1})$ and with the set of markings $\mathcal{F}^\pi(\$\, M_{K+2}\, \gamma) = \{[p^C_{\$,(+)\, X_{K+1}\, \ell_{K+1}\, X_k\, \ell_k \cdots X_1\, \ell_1} \mapsto [\mathcal{F}^I(M_1, \ldots, M_{K+2})]] \mid \ell_i \text{ character of } M_i, i \in \langle k+1 \rangle\}$. For uniformity we will not treat this special case explicitly in the following but we will note that this special representation applies when $k = K+1$.

Representing all processes in $\Pi = \pi_1 \parallel \cdots \parallel \pi_n$ can then be formalised as

$$\mathcal{F}^\Pi(\Pi) = \left\{ \bigoplus_{i=1}^{n} s_i \mid s_i \in \mathcal{F}^\pi(\pi_i) \right\}.$$

The representation of $\Pi \lhd \Gamma$ is a set of configurations where Γ is represented in $\mathcal{N}$'s simple places by $\mathcal{F}^\Gamma(\Gamma)$ and Π is represented in $\mathcal{N}$'s complex and simple places by a configuration from $\mathcal{F}^\Pi(\Pi)$ together with a $\bullet$-token in p_{sim} or formally as:

$$\mathcal{F}(\Pi \lhd \Gamma) = \{s \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim}) \mid s \in \mathcal{F}^\Pi(\Pi)\}.$$

We can define the relation $R \subseteq State_{\text{alt}} \times \mathscr{P}[Config]$ by

$$R = \{(\Pi \lhd \Gamma, \mathcal{F}(\Pi \lhd \Gamma)) \mid \Pi \lhd \Gamma \in State_{\text{alt}}\};$$

we will prove in the following that R is a weak bisimulation between a labelled version of $(State_{\text{alt}}, \rightarrow_{\mathcal{G}_{\text{alt}}})$ and a labelled version of a "co-universal powerset lifting" of $(Config, \rightarrow_{\mathcal{N}})$.

Let us turn to the implementation of the alternative semantics as defined in the right column of Table B.1 in $\mathcal{N}$'s rules. In the following we write $\Xi = X_k\, \ell_k \cdots X_1\, \ell_1$ to save space.

The alternative semantics defines transitions of the form $\gamma \parallel \Pi \lhd \Gamma \rightarrow_{\mathcal{G}_{\text{alt}}} \gamma' \parallel \Pi' \lhd \Gamma'$. We will describe $\mathcal{N}$'s rules by a case analysis on the rule that justifies the transition and relate the forms of γ, γ', Π, Π', Γ and Γ' to guide the reader's intuition.

- **Rule** (R′-1): $\Pi = \Pi'$, $\Gamma = \Gamma'$ and $A \rightarrow \beta$ is a $\mathcal{G}$ rule.

We perform case analysis on β:

- $\beta = B\, C$ and C non-commutative.
  For each sequence of non-commutative non-terminals and cache characters $A\, \ell_{k+1}\, X_k\, \ell_k \cdots X_1\, \ell_1$ where $k < K$ we introduce a complex rule
  $$((p^C_{A,\ell_{k+1}\, \Xi}, s_{sim}), (p^C_{B,(+)\, C\, \ell_{k+1}\, \Xi}, \emptyset, s_{sim}))$$
  that moves a complex token from a place encoding $A\, \ell_{k+1}\, \Xi$ to a place representing $B\, (+)\, C\, \ell_{k+1}\, \Xi$.

- $\beta = a$ where $a \in \Sigma \cup \{\epsilon\}$
  For each sequence of non-commutative non-terminals and cache characters $A\,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1$ where $k \leq K$ we introduce a complex rule $((p^C_{A,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{a,\ell_{k+1}\,\Xi}, \emptyset, s_{sim}))$ that moves a complex token from a place encoding $A\,\ell_{k+1}\,\Xi$ to a place representing $a\,\ell_{k+1}\,\Xi$.

• **Rule** (R′-2): $A \to B\,C$ is a $\mathcal{G}$-rule and C commutative.
Note that a reduction $C \to^*_{\text{seq}} w \in (\mathcal{N}^{\text{com}} \cup \Sigma^{\text{com}})^*$ is that of a commutative context-free grammar (CCFG) for which a Petri-net encoding exists [GM12] by Ganty and Majumdar which builds on an earlier encoding [Esp97] by Esparza. Ganty and Majumdar leverage a recent result [Esp+11]: every word of a CCFG has a *bounded-index derivation*. The CCFG widget can thus be augmented with a budget counter that ensures that the Petri-net encoding respects boundedness of index. Termination of such a CCFG computation is signaled by a transition which is only enabled when the budget counter reaches the set budget.

We make use of a trivially modified CCFG widget *à la* Ganty and Majumdar to implement transitions justified by Rule (R′-2). We will first define the rules of the CCFG widget and how it is activated.

Let us define a few abbreviations. Let $s_b^{budget} = [p^{CCFG}_{budget} \mapsto [\bullet^b]]$ and for each $N \in \mathcal{N}^{\text{com}}$ let $s_N = [p^{CCFG}_N \mapsto [\bullet]]$.

For each sequence of non-commutative non-terminals and cache characters $X_k\,\ell_k \cdots X_1\,\ell_1$ where $k \leq K$ we introduce the following complex rules:

* $((p^C_{A,(+)\,\Xi}, s_{sim}), (p^{CCFG}_{B,(+)\,\Xi}, \emptyset, s_{CCFG+} \oplus O))$,
* $((p^C_{A,(+)\,\Xi}, s_{sim}), (p^{CCFG}_{B,(-)\,\Xi}, \emptyset, s_{CCFG} \oplus O))$,
* $((p^C_{A,(-)\,\Xi}, s_{sim}), (p^{CCFG}_{B,(-)\,\Xi}, \emptyset, s_{CCFG} \oplus O))$ and
* $((p^C_{A,\ell\,\Xi}, s_{sim}), (p^{CCFG}_{B,\ell\,\Xi}, \emptyset, s_{CCFG} \oplus O))$ for each $\ell \in \{A^{\text{cov}}_i : i \in \langle n \rangle\}$

where $O = s^{budget}_{|\mathcal{N}|} \oplus s_C$. Each of rules activates the simulation of the CCFG widget. The topmost cache character is set according to whether the cache is a *CompleteSummary* (+), a potential *PartialSummary* (-) or exposing non-terminal A^{cov}_i in a *PartialSummary*. Further the s_{CCFG+}-"mode" will enforce that the CCFG widget computes a *CompleteSummary* while the s_{CCFG}-"mode" allows a *PartialSummary* and the exposal of a non-terminal. Further a token is placed in place p^{CCFG}_C and the budget place is initialised with $|\mathcal{N}|$ •-tokens. The CCFG widget maintains the invariant that the number of tokens in the set of places $\bigcup_{N \in \mathcal{N}} p^{CCFG}_N$ plus the contents of p^{budget} equals $|\mathcal{N}| + 1$.

To exit the simulation of the CCFG widget we add the rules:

* $((p^{CCFG}_{B,(+)\,\Xi}, I), (p^C_{B,(+)\,\Xi}, \emptyset, s_{sim}))$,
* $((p^{CCFG}_{B,(-)\,\Xi}, I), (p^C_{B,(+)\,\Xi}, \emptyset, s_{sim}))$ and
* $((p^{CCFG}_{B,\ell\,\Xi}, I), (p^C_{B,\ell\,\Xi}, \emptyset, s_{sim}))$ for each $\ell \in \{A^{\text{cov}}_i : i \in \langle n \rangle\}$

where $I = s_{CCFG} \oplus s^{budget}_{|\mathcal{N}|+1}$. Since we require that p^{budget} contains $|\mathcal{N}| + 1$ $\bullet$-tokens the invariant tells us that the set of places $\bigcup_{N\in\mathcal{N}} p^{CCFG}_N$ must be empty when one of the above rules is enabled.

We turn to how the CCFG widget implements $\mathcal{G}$'s commutative rules. For each $\mathcal{G}$-rule r which involves only commutative non-terminals we do a case analysis on r:

(I) $r = X \to Y\ Z$ and X, Y, Z commutative
We add the complex rules
- $((p^{CCFG}_{B,(+)\,\Xi}, s_{CCFG+} \oplus I), (p^{CCFG}_{B,(+)\,\Xi}, \emptyset, s_{CCFG+} \oplus O))$,
- $((p^{CCFG}_{B,(-)\,\Xi}, s_{CCFG} \oplus I), (p^{CCFG}_{B,(-)\,\Xi}, \emptyset, s_{CCFG} \oplus O))$, and
- $((p^{CCFG}_{B,\ell\,\Xi}, s_{CCFG} \oplus I), (p^{CCFG}_{B,\ell\,\Xi}, \emptyset, s_{CCFG} \oplus O))$ for each $\ell \in \{A^{\text{cov}}_i : i \in \langle n\rangle\}$ where $I = s_X \oplus s^{budget}_2$ and $O = s_Y \oplus s_Z \oplus s^{budget}_1$. Further if $W \in \{Y, Z\}$ and $W = A^{\text{cov}}_i$ then we add the complex rule $((p^{CCFG}_{B,(-)\,\Xi}, s_{CCFG} \oplus I), (p^{CCFG}_{B,W\,\Xi}, \emptyset, s_{CCFG} \oplus O))$. We notice that a non-terminal can only be non-deterministically exposed in s_{CCFG}-"mode" and not changed after it has been set.

(II) $r = X \to e, e \in \Sigma^{\text{com}} \cup \{\epsilon\}$
We add the complex rules
(i) $((p^{CCFG}_{B,(+)\,\Xi}, s_{CCFG+} \oplus I), (p^{CCFG}_{B,(+)\,\Xi}, c, s_{CCFG+} \oplus O))$,
(ii) $((p^{CCFG}_{B,(-)\,\Xi}, s_{CCFG} \oplus I), (p^{CCFG}_{B,(-)\,\Xi}, c, s_{CCFG} \oplus O))$ and
(iii) $((p^{CCFG}_{B,\ell\,\Xi}, s_{CCFG} \oplus I), (p^{CCFG}_{B,\ell\,\Xi}, c, s_{CCFG} \oplus O))$ for each $\ell \in \{A^{\text{cov}}_i : i \in \langle n\rangle\}$ where $I = s_X$, $O = s^{budget}_1$ and $c = [p^I_{k+1,e} \mapsto [\bullet]]$.

(III) For each $X \in \mathcal{N}^{\text{com}}$
We add the simple rule: $(s_{CCFG} \oplus I, s_{CCFG} \oplus O)$ and where $I = s_X$ and $O = s^{budget}_1$.

The CCFG widget defined above is essentially the same as Ganty and Majumdar's in [GM12] with one difference: our CCFG widget injects for each terminal symbol $e \in \Sigma^{\text{com}}$ a token of colour $p^I_{k,e}$ into the unique complex token located in some $p^{CCFG}_{B,\ell\,\Xi}$ instead of placing a token into a designated place p_e.

Further our CCFG widget can be thought of as implementing two CCFGs derived from $\mathcal{G}$'s rules. One, indicated by the "mode" s_{CCFG+}, implements the CCFG induced by $\mathcal{G}$'s commutative rules; the other, indicated by the "mode" s_{CCFG}, allows: (a) $\mathcal{G}$ to produce partial words using rules introduced by (III), which can be thought of allowing a non-terminal X to rewrite to a terminal $\bar{X}$ that is ignored; and (b) a non-terminal A^{cov}_i may change the location of the unique complex token m from some place $p^{CCFG}_{B,(-)\,\Xi}$ to $p^{CCFG}_{B,A^{\text{cov}}_i\,\Xi}$ which reflects the representation of the topmost cache's character for the process represented by m. This concludes the description of the implementation of Rule (R′-2).

- **Rule** (R′-3): $\Pi = \Pi', \Gamma = ([msg] \oplus q)^c, \Gamma'$ and $\gamma = (c\ ?\ msg)\,\gamma'$.

Let $\$ = c\ ?\ msg$. For each sequence of non-terminals and cache characters $\$\,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1$ where $k \leq K + 1$ we introduce a complex rule $((p^C_{\$,\ell_{k+1}\,\Xi}, I), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, \emptyset, O))$ where $I = \left[p^S_{c,msg} \mapsto [\bullet]\right] \oplus s_{sim}$ and $O = s_{sim}$ that moves a complex token from a place encoding $\$\,\ell_{k+1}\,\Xi$

to a place representing $\epsilon\,\ell_{k+1}\,\Xi$ while removing a $\bullet$-token from the simple place representing messages *msg* in channel *c*.

- **Rule** (R′-4): $\Pi = \Pi', \Gamma, ([msg] \oplus q)^c = \Gamma'$ and $\gamma = (c\,!\,msg)\,\gamma'$.
Let $\$ = c\,!\,msg$. For each sequence of non-terminals and cache characters $\$\,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1$ where $k \leq K+1$ we introduce a complex rule $((p^C_{\$,\ell_{k+1}\,\Xi}, I), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, \emptyset, O))$ where $I = s_{sim}$ and $O = \left[p^S_{c,msg} \mapsto [\bullet]\right] \oplus s_{sim}$ that moves a complex token from a place encoding $\$\,\ell_{k+1}\,\Xi$ to a place representing $\epsilon\,\ell_{k+1}\,\Xi$ while adding a $\bullet$-token from the simple place representing messages *msg* in channel *c*.

- **Rule** (R′-5). $\Pi \parallel X = \Pi', \Gamma = \Gamma'$ and $\gamma = \nu X\,\gamma'$

Let $\$ = \nu X$. For each sequence of non-terminals and cache characters $\$\,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1$ where $k \leq K+1$ we introduce a complex rule $((p^C_{\$,\ell_{k+1}\,\Xi}, I), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, \emptyset, O))$ where $I = s_{sim}$ and $O = \left[p^S_{\nu X} \mapsto [\bullet]\right] \oplus s_{sim}$ that moves a complex token from a place encoding $\$\,\ell_{k+1}\,\Xi$ to a place representing $\epsilon\,\ell_{k+1}\,\Xi$ while adding a $\bullet$-token to the simple "spawning" place $p^S_{\nu X}$. Additionally, for every $N \in \mathcal{N}$ we introduce a simple rule $r_N = (\left[p^S_{\nu N} \mapsto [\bullet]\right] \oplus s_{sim}, \left[p^C_{N,+} \mapsto [\emptyset]\right] \oplus s_{sim})$ that removes a $\bullet$-token from $p^S_{\nu N}$ and adds the empty complex token $\emptyset$ to a complex place representing $N+$. For reference we will call the rule r_N a weak spawn rule for the non-terminal N.

- **Rule** (R′-6): $\Pi' = \Pi \oplus \Pi(M), \Gamma' = \Gamma \oplus \Gamma(M), \gamma = M\,\gamma'$ and $M \in$ *CompleteSummary*.
For each sequence of non-terminals and cache characters $\epsilon\,(+)\,X_k\,\ell_k \cdots X_1\,\ell_1$ where $k \leq K+1$ we introduce a transfer rule

$$((p^C_{\epsilon,+\,X_k\,\ell_k \cdots X_1\,\ell_1}, s_{sim}), (p^C_{X_k,\ell_k \cdots X_1\,\ell_1}, P, s_{sim}))$$

where $P = \left\{p^I_{k+1,e} \mid e \in \Sigma^{\text{com}}\right\}$ if $k \leq K$ and $P = \emptyset$ otherwise. This rule moves a complex token from a place encoding $\epsilon, +, \Xi$ to a place representing Ξ while it simulates the immediate despatch of the commutative concurrency actions, send $c\,!\,msg$ and spawn νX, that are present in the top-level cache M which is in *CompleteSummary* since its character is $+$.

- **Rule** (R′-7): $\Pi' = \Pi \oplus \Pi(M)$, $\Gamma' = \Gamma \oplus \Gamma(M)$, $\gamma = M\,\gamma_0$, $\gamma' = M'\,\gamma_0$, $M' = M \restriction (\mathcal{N}^{\text{com}} \cup \{A^{\text{cov}}_i : i \in \langle n \rangle\})$ and $M \in$ *PartialSummary*
For each sequence of non-terminals and cache characters $\epsilon\,\ell_{k+1}\,X_k\,\ell_k \cdots X_1\,\ell_1$ where $k \leq K+1$, $\ell_{k+1} \in \{-\} \cup \{A^{\text{cov}}_i : i \in \langle n \rangle\}$ where $k \leq K$ we introduce a transfer rule $((p^C_{\epsilon,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, P, s_{sim})$ where $P = \left\{p^I_{k+1,e} \mid e \in \Sigma^{\text{com}}\right\}$. This rule moves a takes a complex token from a place encoding $\epsilon, +, \Xi$ and places it back while it simulates the immediate despatch of the commutative concurrency actions, send $c\,!\,msg$ and spawn νX, that are present in the top-level cache M which is in *PartialSummary* since its character is $\ell_{k+1} \in \{-\} \cup \{A^{\text{cov}}_i : i \in \langle n \rangle\}$.

The coverability query $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, A^{\text{cov}}_1 \parallel \cdots \parallel A^{\text{cov}}_n \lhd \Gamma^{\text{cov}})$ is implemented by a family of widgets $W_1, \cdots, W_n$ where for each i, W_i is a disjunction of the non-emptiness test of complex places of the shapes $p^C_{\epsilon,A^{\text{cov}}_i\,\Xi}$ and $p^C_{A^{\text{cov}}_i,\ell\,\Xi'}$, as ℓ ranges over $\widehat{\mathcal{L}} := \{A^{\text{cov}}_i : i \in \langle n \rangle\} \cup \{+, -\}$,

and Ξ and Ξ' range over $(\mathcal{N}^{\neg\text{com}} \cdot \widehat{\mathcal{L}})^*$ of the appropriate lengths. A widget signals that the non-emptiness test is satisfied by placing a $\bullet$-token into the simple place $p^{cov}_{A_i}$. The intention is that we can use a coverability query for $\mathcal{N}$ asking for at least one token in each of $p^{cov}_{A_1}, \ldots, p^{cov}_{A_n}$ to implement the coverability query $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, A_1^{\text{cov}} \parallel \cdots \parallel A_n^{\text{cov}} \lhd \Gamma^{\text{cov}})$.

Formally, there is a simple rule (s_{sim}, s_{Query}) that non-deterministically terminates the simulation of $\mathcal{G}$ and activates the processing of the coverability query. Then for each A_i^{cov} where $i \in \langle n \rangle$, we implement the widget W_i by the following complex rules: for all non-commutative non-terminals and cache characters $X_k\, \ell_k \cdots X_1\, \ell_1$ where $k \leq K$, and $\$ \in \Sigma \cup \{\epsilon\} \cup \mathcal{N}$ we introduce complex rules $((p^C_{A_i^{\text{cov}}, \ell_{k+1}\, \Xi}, s_{Query}), (p^C_{\epsilon,(-)\,\Xi}, \emptyset, s_{Query} \oplus O))$ and $((p^C_{\epsilon, A_i^{\text{cov}}\, \Xi}, s_{Query}), (p^C_{\epsilon,(-)\,\Xi}, \emptyset, s_{Query} \oplus O))$ where $O = [p^{cov}_{A_i} \mapsto [\bullet]]$.

Let us briefly inspect the size of $\mathcal{N}$ and let us assume that all $n, K, |\mathcal{R}|, |\mathcal{N}|, |\Sigma| > 0$. It is clear that there exists a constants c, c' and c'' such that $\mathcal{N}$ has no more than $c \cdot n \cdot |\mathcal{R}| \cdot |\mathcal{P}^C|^2$ complex rules, $\mathcal{N}$ has no more than $c' \cdot |\mathcal{R}| \cdot |\mathcal{N}|$ simple rules and $\mathcal{N}$ has no more than $c'' \cdot |\mathcal{P}^C|^2$ transfer rules. Hence there exists a constant c''' such that $\mathcal{N}$ has no more than $c''' \cdot n \cdot |\mathcal{R}| \cdot |\mathcal{N}| \cdot |\mathcal{P}^C|^2$ rules. Further there exists constants d, d', d'', d''' such that $\mathcal{N}$ has no more than $d \cdot n \cdot |\Sigma| \cdot |\mathcal{N}|$ simple places, $\mathcal{N}$ has no more than $d' \cdot K \cdot |\Sigma|$ colours, and $\mathcal{N}$ has no more than $d'' \cdot |\mathcal{N}|^{d''' \cdot K} \cdot |\Sigma|^{d''' \cdot K}$ complex places. It is thus easy to see that $\mathcal{N}$ can be computed from $\mathcal{G}$ in Exptime.

We will now prove that it checking whether $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, A_1^{\text{cov}} \parallel \cdots \parallel A_n^{\text{cov}} \lhd \Gamma^{\text{cov}})$ is a yes-instance of simple coverability reduces to a coverability check on $\mathcal{N}$.

In order to clarify which model induces a transition we will write $\to_{\mathcal{G}}$ for $\to_{\mathcal{G}_{\text{alt}}}$ in the following. We label the transition system $(\mathit{State}_{\text{alt}}, \to_{\mathcal{G}})$ in the following way: if $\Pi \lhd \Gamma, \Pi' \lhd \Gamma' \in \mathit{State}_{\text{alt}}$ such that $\Pi \lhd \Gamma \to_{\mathcal{G}} \Pi' \lhd \Gamma'$ then the labelled version has the transition $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_{\mathcal{G}} \Pi' \lhd \Gamma'$. Next, let us define the transition system $(\mathscr{P}[\mathit{Config}], \to_{\mathscr{P}[\mathcal{N}]})$ which is a *weak* couniversal power lifting of the transition system $(\mathit{Config}, \to_{\mathcal{N}})$. Suppose $S, S' \subseteq \mathit{Config}$ then $S \to_{\mathscr{P}[\mathcal{N}]} S'$ just if for all $s' \in S'$ there exists an $s \in S$ such that there are number of weak spawn rules (see $\bullet$) $r_1, \ldots, r_m$ where $m \geq 0$ such that $s \xrightarrow{r_1}_{\mathcal{N}} s_1 \xrightarrow{r_2}_{\mathcal{N}} \cdots \xrightarrow{r_m}_{\mathcal{N}} s_m$ and $s_m \to_{\mathcal{N}} s'$. We label $(\mathscr{P}[\mathit{Config}], \to_{\mathscr{P}[\mathcal{N}]})$ in the following way: suppose $S, S' \subseteq \mathit{Config}$ such that $S \to_{\mathscr{P}[\mathcal{N}]} S'$; if there exists $\Pi \lhd \Gamma \in \mathit{State}_{\text{alt}}$ such that $S' = \mathcal{F}(\Pi \lhd \Gamma)$ then we label the transition by $S \xrightarrow{\Pi \lhd \Gamma}_{\mathscr{P}[\mathcal{N}]} S'$; otherwise we label the transition by $S \xrightarrow{\epsilon}_{\mathscr{P}[\mathcal{N}]} S'$.

We will now prove that R is a weak bisimulation between the labelled versions of $(\mathit{State}_{\text{alt}}, \to_{\mathcal{G}})$ and $\left(\mathscr{P}[\mathit{Config}], \to_{\mathscr{P}[\mathcal{N}]}\right)$.

Let us first prove R is a weak simulation. Suppose $(\gamma \parallel \Pi \lhd \Gamma, \mathcal{F}(\gamma \parallel \Pi \lhd \Gamma)) \in R$ and $\gamma \parallel \Pi \lhd \Gamma \xrightarrow{\Pi_0 \lhd \Gamma_0}{}^*_{\mathcal{G}} \gamma' \parallel \Pi' \lhd \Gamma'$ then clearly $\gamma \parallel \Pi \lhd \Gamma \xrightarrow{\gamma' \parallel \Pi' \lhd \Gamma'}_{\mathcal{G}} \gamma' \parallel \Pi' \lhd \Gamma'$.

We want to show that $\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \xrightarrow{\gamma' \parallel \Pi' \lhd \Gamma'}{}^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$. So let $s' \in \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ then by definition $s' = s'_0 \oplus (\mathcal{F}^{\Gamma}(\Gamma') \oplus s_{sim})$ for some $s'_0 \in \mathcal{F}^{\Pi}(\gamma' \parallel \Pi')$. We can further decompose s'_0 and obtain $s'_0 = s'_1 \oplus s'_2$ where $s'_1 \in \mathcal{F}^{\pi}(\gamma')$ and $s'_2 \in \mathcal{F}^{\Pi}(\Pi')$.

Let us perform a case analysis on the rule that justifies the transition $\gamma \parallel \Pi \lhd \Gamma \xrightarrow{\gamma' \parallel \Pi' \lhd \Gamma'}_{\mathcal{G}} \gamma' \parallel \Pi' \lhd \Gamma'$:

- **Rule** (R′-1): $\Pi = \Pi'$, $\Gamma = \Gamma'$, $\gamma = A\, M\, X_k\, M_k \cdots X_1\, M_1$, and $\gamma' = \delta\, M\, X_k\, M_k \cdots X_1\, M_1$ for some $\mathcal{G}$ rule $A \to \delta$.

We can see that $s'_1 = [p^C_{\overline{\delta},\ell_{k+1}\,\Xi} \mapsto [\mathcal{F}^I(M_1,\ldots,M_{k+1})]]$ where ℓ_i is a character of M_i for each $i \in \langle k+1\rangle$, and $\overline{\delta}$ depends on the form of δ: $\overline{\delta} = B\,(+)\,C$ if $\delta = B\,C$, $k < K$ and C non-commutative; and $\overline{\delta} = a$ if $\delta = a$ and $a \in \Sigma \cup \{\epsilon\}$. Note that it is now obvious that it is impossible that γ' is represented in a simple place.

Let $s = (s_1 \oplus s'_2) \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim})$ we write $s_1 := [p^C_{A,\ell_{k+1}\,\Xi} \mapsto [\mathcal{F}^I(M_1,\ldots,M_{k+1})]]$. Clearly $s_1 \in \mathcal{F}^\pi(AMX_kM_k\cdots X_1M_1) = \mathcal{F}^\pi(\gamma)$ and thus $s \in \mathcal{F}(\gamma \parallel \Pi \lhd \Gamma)$.

We know that $\mathcal{N}$ has a complex rule $((p^C_{A,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{\overline{\delta}\,\ell_{k+1}\,\Xi}, \emptyset, s_{sim}))$ Thus we know we can make the transition $s \to_{\mathcal{N}} (s'_1 \oplus s_2) \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim})$ and clearly $s' = (s'_1 \oplus s_2) \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim})$.

Since $s \in \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ was arbitrary we can conclude that in fact $\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ and thus clearly

$$\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \xrightarrow{\gamma' \parallel \Pi' \lhd \Gamma'}{}^*_{\mathscr{P}[\mathcal{N}]}\ \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$$

which is what we wanted to prove.

- **Rule** (R′-2): $\Pi = \Pi'$, $\Gamma = \Gamma'$, $\gamma = A\,M\,X_k\,M_k\cdots X_1\,M_1$, $\gamma' = \delta\,M'\,X_k\,M_k\cdots X_1\,M_1$ for some $\mathcal{G}$ rule $A \to B\,C$, C commutative, $C \to^* w$, and $M' = M \oplus \mathbb{M}(w)$.
In this case we can assume that $s'_1 = [p^C_{B,\ell'\,\Xi} \mapsto [\mathcal{F}^I(M_1,\ldots,M_k,M')]]$ where ℓ' is a character for $M' = M \oplus \mathbb{M}(w)$.

Let us define cache characters $\ell, \overline{\ell}$ and $\overline{\ell}'$ by a case analysis on ℓ' and $\mathbb{M}(w)$:

(C1) *Case:* $\ell' = +$: Define $\ell = \overline{\ell} = \overline{\ell}' = +$.
(C2) *Case:* $\ell' = -$ and $M \in$ *PartialSummary*: Define $\ell = \overline{\ell} = \overline{\ell}' = -$.
(C3) *Case:* $\ell' = -$ and $M \in$ *CompleteSummary*: Define $\ell = +$ and $\overline{\ell} = \overline{\ell}' = -$.
(C4) *Case:* $\ell' \in \{A^{\text{cov}}_i : i \in \langle n\rangle\}$, $M \in$ *PartialSummary* and $\mathbb{M}(w)(\ell') > 0$: Define $\ell = \overline{\ell} = -$ and $\overline{\ell}' = \ell'$.
(C5) *Case:* $\ell' \in \{A^{\text{cov}}_i : i \in \langle n\rangle\}$, $M \in$ *CompleteSummary* and $\mathbb{M}(w)(\ell') > 0$: Define $\ell = +$ and $\overline{\ell} = -$ and $\overline{\ell}' = \ell'$.
(C6) *Case:* $\ell' \in \{A^{\text{cov}}_i : i \in \langle n\rangle\}$ and $\mathbb{M}(w)(\ell') = 0$: Define $\ell = \overline{\ell} = \overline{\ell}' = \ell'$.

Let us show that the above choices ensure that ℓ is a character for M. If $M \in$ *CompleteSummary* then either $M' = M \oplus \mathbb{M}(w) \in$ *CompleteSummary* and so $\ell' = +$ and hence $\ell = +$ and thus ℓ is a character for M. Otherwise $\ell' \in \{-\} \cup \{A^{\text{cov}}_i : i \in \langle n\rangle\}$ and so since $M \in$ *CompleteSummary* by definition we have $\ell = +$ and hence ℓ is a character for M. If $M \in$ *PartialSummary* then clearly $M \oplus \mathbb{M}(w) \in$ *PartialSummary* and so either $\ell' = -$ or $\ell' \in \{A^{\text{cov}}_i : i \in \langle n\rangle\}$. So if $\ell' = -$ or $\mathbb{M}(w)(\ell') > 0$ then $\ell = -$ which is a character M. Otherwise, $\ell' \in \{A^{\text{cov}}_i : i \in \langle n\rangle\}$ and $\mathbb{M}(w)(\ell') = 0$ then $\ell = \ell'$ and since ℓ' is not a character for $\mathbb{M}(w)$ but for M' it must be that case that $\ell = \ell'$ is character for M. Hence we can conclude that in all cases ℓ is a character for M.

Define $s^{s'}$ by $s^{s'} = s_1^{s'} \oplus s'_2 \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim})$ where $s_1^{s'} = [p^C_{A,\ell\,\Xi} \mapsto [\mathcal{F}^I(M_1,\ldots,M_k,M)]]$. It is clear that $s_1^{s'} \in \mathcal{F}^\pi(\gamma)$ and hence $s^{s'} \in \mathcal{F}(\gamma \parallel \Pi \lhd \Gamma)$.

Our definitions of ℓ and $\overline{\ell}$ ensure that there is a complex rule of the form $\boldsymbol{r}(1) = ((p^{C}_{A,\ell\,\Xi}, s_{sim}), (p^{CCFG}_{B,\overline{\ell}\,\Xi}, \emptyset, s_{CCFG+?} \oplus O))$ which is enabled at $\boldsymbol{s}^{s'}(1)$ to active a CCFG widget computation where $CCFG+? = CCFG+$ if $\overline{\ell} = +$; and $CCFG+? = CCFG$ otherwise; and $O = s^{budget}_{|\mathcal{N}|} \oplus [p^{CCFG}_{C} \mapsto [\bullet]]$.

Hence we can make a transition $\boldsymbol{s}^{s'}(1) \xrightarrow{\boldsymbol{r}(1)}_{\mathcal{N}} \boldsymbol{s}^{s'}(2)$ where $\boldsymbol{s}^{s'}(2) = (\boldsymbol{s}_1{}^{s'}(2) \oplus s'_2) \oplus (\mathcal{F}^{\Gamma}(\Gamma) \oplus s_{CCFG+?} \oplus s^{budget}_{|\mathcal{N}|} \oplus [p^{CCFG}_{C} \mapsto [\bullet]])$ and $\boldsymbol{s'}^{s'}(2) = [p^{CCFG}_{B,\overline{\ell}\,\Xi} \mapsto [\mathcal{F}^{I}(M_1, \ldots, M_{k+1})]]$.

We appeal to [GM12] that the CCFG widget allows us to simulate the computation $C \to^* w$ in L steps, where L only depends on the derivation $C \to^* w$ and not on $\boldsymbol{s'}^{s'}(2)$, so that we get $\boldsymbol{s}^{s'}(1+1) \xrightarrow{\boldsymbol{r}(2)}_{\mathcal{N}} \boldsymbol{s}^{s'}(1+2) \cdots \xrightarrow{\boldsymbol{r}(1+L)}_{\mathcal{N}} \boldsymbol{s}^{s'}(1+L)$ where for all $2 \leq i \leq L+1$ the rule $\boldsymbol{r}(i)$ is introduced by cases (I)–(III) above; and for each terminal $e \in \Sigma^{\mathrm{com}}$ occurring in w we inject a $p^{I}_{k+1,e}$-coloured token into $\mathcal{F}^{I}(M_1, \ldots, M_{k+1})$, i.e. if $M_0 \leq_{\mathbb{M}} \mathbb{M}(w)$ then injecting a $p^{I}_{k+1,e}$-coloured token $\mathcal{F}^{I}(M_1, \ldots, M_{k+1} \oplus M_0)$ yields $\mathcal{F}^{I}(M_1, \ldots, M_{k+1} \oplus (M_0 \oplus [e]))$, and for each $2 \leq i \leq L+1$ the configuration $\boldsymbol{s}^{s'}(i)$ has $s_{CCFG+?}$ as a submarking. Further, we can see that in the computation of $C \to^* w$ it is possible to expose a non-terminal ℓ_0 if $\mathbb{M}(w)(\ell_0) > 0$ and $\overline{\ell} = -$. We can thus see that if cases (C1)–(C5) applied for the definition of $\overline{\ell}$ and $\overline{\ell}'$ then we can assume that the above computation ignores all non-terminal; if case (C6) applied then $\mathbb{M}(w)(\ell') > 0$ and $\overline{\ell} = -$, so we can expose $\ell' = \overline{\ell}'$ along the above computation.

Hence we can conclude that $\boldsymbol{s}^{s'}(1+L) = (\boldsymbol{s}_1{}^{s'}(1+L) \oplus s_2) \oplus (\mathcal{F}^{\Gamma}(\Gamma) \oplus s_{CCFG+?} \oplus s^{budget}_{|\mathcal{N}|+1})$ where $\boldsymbol{s'}^{s}(1+L) = [p^{CCFG}_{B,\overline{\ell}'\,\Xi} \mapsto [\mathcal{F}^{I}(M_1, \ldots, M_{k+1} \oplus (\mathbb{M}(w) \upharpoonright \Sigma^{\mathrm{com}}))]]$.

We note that $\overline{\ell}' = \ell'$ in all cases of (C1)–(C6).

It is then the case that a complex rule of the form $((p^{CCFG}_{B,\ell'\,\Xi}, s_{CCFG+?} \oplus s^{budget}_{|\mathcal{N}|+1}), (p^{C}_{B,\ell'\,\Xi}, \emptyset, s_{sim}))$ is enabled and we can make the transition $\boldsymbol{s}^{s'}(L+1) \to_{\mathcal{N}} \boldsymbol{s}^{s'}(L+2)$ where $\boldsymbol{s}^{s'}(L+2) = (\boldsymbol{s}_1{}^{s'}(L+2) \oplus s_2) \oplus (\mathcal{F}^{\Gamma}(\Gamma) \oplus s_{sim})$ and

$$\boldsymbol{s'}^{s'}(L+2) = [p^{C}_{B,\ell'\,\Xi} \mapsto [\mathcal{F}^{I}(M_1, \ldots, M_{k+1} \oplus (\mathbb{M}(w) \upharpoonright \Sigma^{\mathrm{com}}))]].$$

Now $\mathcal{F}^{I}(M_1, \ldots, M_{k+1} \oplus (\mathbb{M}(w) \upharpoonright \Sigma^{\mathrm{com}})) = \mathcal{F}^{I}(M_1, \ldots, M \oplus \mathbb{M}(w))$. Thus $\boldsymbol{s'}^{s'}(L+2) = s'_1$ and hence $\boldsymbol{s}^{s'}(L+2) = s'$.

We now need to lift these paths to $\mathscr{P}[\mathcal{N}]$. The recipe above gives us for each $s' \in \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ a path $\boldsymbol{s}^{s'}(1) \cdots \boldsymbol{s}^{s'}(L+2)$. We note that L is in fact independent of s' since it is only dependent on the derivation $C \to^* w$. Further $\boldsymbol{s}^{s'}(1) \in \mathcal{F}(\gamma \parallel \Pi \lhd \Gamma)$, $\boldsymbol{s}^{s'}(L+2) = s'$ and each for all $2 \leq i < L+1$ the configuration $\boldsymbol{s}^{s'}(i)$ contains either submarking s_{CCFG+} or s_{CCFG} and hence $\nexists \Pi_0 \lhd \Gamma_0$ such that $\boldsymbol{s}^{s'}(i) \in \mathcal{F}(\Pi_0 \lhd \Gamma_0)$. Let us define the following subsets of *Config*: for $1 \leq i \leq L+2$ let $\boldsymbol{S}(i) = \left\{ \boldsymbol{s}^{s'}(i) \mid s' \in \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma') \right\}$. By definition we have for all $\boldsymbol{s}^{s'}(i+1) \in \boldsymbol{S}(i+1)$ that $\boldsymbol{s}^{s'}(i) \to_{\mathcal{N}} \boldsymbol{s}^{s'}(i+1) \in \boldsymbol{S}(i+1)$ if $1 \leq i < L+2$ and hence $\boldsymbol{S}(1) \to_{\mathscr{P}[\mathcal{N}]} \boldsymbol{S}(2) \to_{\mathscr{P}[\mathcal{N}]} \cdots \to_{\mathscr{P}[\mathcal{N}]} \boldsymbol{S}(L+1) \to_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ since clearly $\boldsymbol{S}(L+2) = \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$. Looking at the labelled version of $\mathscr{P}[\mathcal{N}]$ our reasoning

above implies that For all $1 < i < L+2$ $\nexists \Pi_0 \lhd \Gamma_0$ such that $\boldsymbol{S}(i) = \mathcal{F}(\Pi_0 \lhd \Gamma_0)$ and hence $\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \supseteq \boldsymbol{S}(1) \xrightarrow{\gamma' \parallel \Pi' \lhd \Gamma'}{}^{*}_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ which is what we wanted to prove.

- **Rule** (R′-3),(R′-4): $\gamma = \$\,\gamma'$ and $\$ \in \Sigma$.
We will prove the case where the transition is justified by rule (R′-3); the case using rule (R′-4) is analogous.

In this case $\Pi = \Pi', \Gamma = ([msg] \oplus q)^c, \Gamma_0, \Gamma' = q^c, \Gamma_0$ and $\$ = c\,?\,msg$.

If $\gamma' = M_{k+1}\, X_k\, M_k \cdots X_1\, M_1$ is easy to see that $s_1' = [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [\mathcal{F}^I(M_1, \ldots, M_{k+1})]]$ where ℓ_i is a character of M_i for each $i \in \langle k+1 \rangle$; and $\mathcal{F}^\Gamma(\Gamma) = \mathcal{F}^\Gamma(\Gamma') \oplus [p^S_{c,msg} \mapsto [\bullet]]$. Hence let $s = (s_1 \oplus s_2') + (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim})$ where we write $s_1 := [p^C_{c?msg,\ell_{k+1}\,\Xi} \mapsto [\mathcal{F}^I(M_1, \ldots, M_{k+1})]]$.

The complex rule $((p^C_{c?msg,\ell_{k+1}\,\Xi}, \left[p^S_{c,msg} \mapsto [\bullet]\right] \oplus s_{sim}), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, \emptyset, s_{sim}))$ is then enabled at s and we can make the transition $s \to_{\mathcal{N}} (s_1' \oplus s_2') + (\mathcal{F}^\Gamma(\Gamma') \oplus s_{sim}) = s'$

Since $\gamma = (c\,?\,msg)\,\gamma'$ we know that $s_1 \in \mathcal{F}^\pi((c\,?\,msg)\,\gamma')$ from which we can conclude that $s_1 \oplus s_2' \in \mathcal{F}^\Pi(\gamma \parallel \Pi') = \mathcal{F}^\Pi(\gamma \parallel \Pi)$ and hence $(s_1 \oplus s_2') \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim}) \in \mathcal{F}(\gamma \parallel \Pi \lhd \Gamma)$. Since $s' \in \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ was arbitrary we can conclude that in fact $\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ and thus clearly

$$\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \xrightarrow{\gamma' \parallel \Pi' \lhd \Gamma'}{}^{*}_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$$

which concludes this case.

- **Rule** (R′-5): $\Pi' = \Pi \parallel X, \Gamma = \Gamma'$, and $\gamma = \nu X\, \gamma'$.
If $\gamma' = M_{k+1}\, X_k\, M_k \cdots X_1\, M_1$ is easy to see that $s_1' = [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [\mathcal{F}^I(M_1, \ldots, M_{k+1})]]$ where ℓ_i is a character of M_i for each $i \in \langle k+1 \rangle$. Further since $s_2' \in \mathcal{F}^\Pi(\Pi') = \mathcal{F}^\Pi(\Pi \parallel X)$ we know that $s_2' = s_2'' \oplus s_X'$ where $s_2'' \in \mathcal{F}^\Pi(\Pi)$ and $s_X' \in \mathcal{F}^\pi(X)$. Thus we can define $s = (s_1 \oplus s_2'') + (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim})$ where we write $s_1 := [p^C_{\nu X,\ell_{k+1}\,\Xi} \mapsto [\mathcal{F}^I(M_1, \ldots, M_{k+1})]]$.

Hence the complex rule

$$((p^C_{\nu X,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, \emptyset, [p^S_{\nu X} \mapsto [\bullet]] \oplus s_{sim}))$$

is enabled at s and we can make the transition $s \to_{\mathcal{N}} (s_1' \oplus s_2'' \oplus s_X) \oplus (\mathcal{F}^\Gamma(\Gamma') \oplus s_{sim})$ where we write $s_X = [p^S_{\nu X} \mapsto [\bullet]]$.

Since $s_X' \in \mathcal{F}^\pi(X)$ either $s_X' = s_X$ or $s_X' = \left[p^C_{X,(+)} \mapsto [\emptyset]\right]$. In the latter case we can use a weak spawn rule to make the transition

$$(s_1' \oplus s_2'' \oplus s_X) \oplus (\mathcal{F}^\Gamma(\Gamma') \oplus s_{sim}) \to_{\mathcal{N}} s'.$$

Since $\gamma = (\nu X)\, \gamma'$ we know that $s_1 \in \mathcal{F}^\pi(\gamma)$ from which we can conclude that $s_1 \oplus s_2'' \in \mathcal{F}^\Pi(\gamma \parallel \Pi)$. Therefore $(s_1 \oplus s_2'') \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim}) \in \mathcal{F}(\gamma \parallel \Pi \lhd \Gamma)$.

Since $s' \in \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ was arbitrary we can conclude that $\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$ and thus clearly

$$\mathcal{F}(\gamma \parallel \Pi \lhd \Gamma) \xrightarrow{\gamma' \parallel \Pi' \lhd \Gamma'}{}^{*}_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \parallel \Pi' \lhd \Gamma')$$

which is what we wanted to prove.

- **Rule** (R′-6)–(R′-7): $\Pi' = \gamma' \parallel \Pi \parallel \Pi(M)$, $\Gamma' = \Gamma \oplus \Gamma(M)$, $\gamma = \epsilon\, M\, \gamma_0$ and $\gamma' = \epsilon\, M'\, \gamma_0$. We will prove the case where the transition is justified by rule (R′-6); the case using rule (R′-7) is proved similarly.

In this case, in fact $M' = \emptyset$ and thus $\gamma' = \gamma_0$ which implies $\gamma = \epsilon\, M\, \gamma'$. If $\gamma' = X_k\, M_k \cdots X_1\, M_1$ is easy to see that $s_1' = [p^C_{X_k\, \ell_k \cdots X_1\, \ell_1} \mapsto [\mathcal{F}^I(M_1, \ldots, M_k)]]$ where ℓ_i is a character of M_i for each $i \in \langle k \rangle$. Further since $s_2' \in \mathcal{F}^\Pi(\Pi') = \mathcal{F}^\Pi(\Pi \parallel \Pi(M))$ we decompose s_2' into $s_2' = s_2'' \oplus s'_{\Pi(M)}$ where $s_2'' \in \mathcal{F}^\Pi(\Pi)$ and $s'_{\Pi(M)} \in \mathcal{F}^\Pi(\Pi(M))$.

We also know that $M \in \mathit{CompleteSummary}$, so let $\ell = +$, $s = (s_1 \oplus s_2'') \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim})$ and $s_1 = [p^C_{\epsilon,+\,\Xi} \mapsto [\mathcal{F}^I(M_1, \ldots, M_k, M)]]$. Thus $s_1 \in \mathcal{F}^\pi(M\, \gamma') = \mathcal{F}^\pi(\gamma)$ and hence $s \in \mathcal{F}(\gamma \parallel \Pi \lhd \Gamma)$. The transfer rule $r = ((p^C_{\epsilon,+\,\Xi}, s_{sim}), (p^C_\Xi, P, s_{sim}))$ is then enabled at s where $P = \left\{ p^I_{k+1,e} \mid e \in \Sigma^{\text{com}} \right\}$ if $k \leq K$ and $P = \emptyset$ otherwise (in which case $M = \emptyset$). Let $m = \mathcal{F}^I(M_1, \ldots, M_k, M)$, $m_P = m \upharpoonright P$ and $m_{\overline{P}} = m \upharpoonright (\mathcal{P}^{col} \setminus P)$. Then we know that using r we can make the transition

$$s \to_{\mathcal{N}} (s_{\overline{P}} \oplus s_2'') \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus s_{sim}) \oplus \left(m_P \circ \zeta^{-1}\right) =: s_0'$$

where $s_{\overline{P}} = \left[p^C_\Xi \mapsto \left[m_{\overline{P}}\right]\right]$.

It is not hard to see that $m_{\overline{P}} = \mathcal{F}^I(M_1, \ldots, M_k, \emptyset) = \mathcal{F}^I(M_1, \ldots, M_k)$ and hence $s_{\overline{P}} = s_1'$.

Let $\mathcal{G}^\Gamma = \left\{ p^S_{c,msg} \mid c \in \mathit{Chan}, msg \in \mathit{Msg} \right\}$ and $\mathcal{G}^\nu = \{ p^S_{\nu X} \mid X \in \mathcal{N} \}$. We notice that for all $c \in \mathit{Chan}$, $msg \in \mathit{Msg}$

$$\begin{aligned}\mathcal{F}^\Gamma\left(\Gamma(M \upharpoonright \{c\,!\,msg\})\right) =& \mathcal{F}^\Gamma\left(\left[msg^{M(c!msg)}\right]^c\right) = \left[p^S_{c,msg} \mapsto \left[\bullet^{M(c!msg)}\right]\right] \\ =& \left[p^S_{c,msg} \mapsto m_P(c\,!\,msg)\right] = \left(m_P \circ \zeta^{-1}\right) \upharpoonright \left\{p^S_{c,msg}\right\}\end{aligned}$$

from which we conclude that

$$\left(m_P \circ \zeta^{-1}\right) \upharpoonright \mathcal{G}^\Gamma = \bigoplus_{\substack{c \in \mathit{Chan}, \\ msg \in \mathit{Msg}}} \mathcal{F}^\Gamma\left(\Gamma(M \upharpoonright \{c\,!\,msg\})\right) = \mathcal{F}^\Gamma\left(\Gamma(M)\right)$$

and $\mathcal{F}^\Gamma(\Gamma') = \mathcal{F}^\Gamma(\Gamma) \oplus \left(m_P \circ \zeta^{-1}\right) \upharpoonright \mathcal{G}^\Gamma$.

Further for each $X \in \mathcal{N}$ let $s_X = \left[p^S_{\nu X} \mapsto [\bullet]\right]$; then it is the case that $\bigoplus_{i=1}^{M(\nu(X))} s_X \in \mathcal{F}^\Pi(\Pi(M \upharpoonright \{\nu X\}))$ and thus $\bigoplus_{X \in \mathcal{N}} \bigoplus_{i=1}^{M(\nu(X))} s_X \in \mathcal{F}^\Pi(\Pi(M))$. Now $\bigoplus_{i=1}^{M(\nu(X))} s_X = \left(m_P \circ \zeta^{-1}\right) \upharpoonright \left\{p^S_{\nu X}\right\}$ and thus $\bigoplus_{X \in \mathcal{N}} \bigoplus_{i=1}^{M(\nu(X))} s_X = (m_P \circ \zeta^{-1}) \upharpoonright \mathcal{G}^\nu$. Since $(m_P \circ \zeta^{-1}) = (m_P \circ \zeta^{-1}) \upharpoonright \mathcal{G}^\Gamma \oplus (m_P \circ \zeta^{-1}) \upharpoonright \mathcal{G}^\nu$ we have

$$s_0' = (s_1' \oplus s_2 \oplus \left(m_P \circ \zeta^{-1} \upharpoonright \mathcal{G}^\nu\right)) \oplus (\mathcal{F}^\Gamma(\Gamma') \oplus s_{sim}).$$

Let us inspect $s'_{\Pi(M)}$. Since $s'_{\Pi(M)} \in \mathcal{F}^\Pi(\Pi(M))$ either $s'_{\Pi(M)} = m_P \circ \zeta^{-1} \upharpoonright \mathcal{G}^\nu$ or using a number of weak spawn rules we can transition in s_0' from $m_P \circ \zeta^{-1} \upharpoonright \mathcal{G}^\nu$ to $s'_{\Pi(M)}$. This

implies (using a number of weak spawn rules after the first step) we can make the transition $s \to_{\mathcal{N}} s'_0 \to^*_{\mathcal{N}} s'$.

Since $s' \in \mathcal{F}(\gamma' \,\|\, \Pi' \lhd \Gamma')$ was arbitrary we can conclude that in fact $\mathcal{F}(\gamma \,\|\, \Pi \lhd \Gamma) \to_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \,\|\, \Pi' \lhd \Gamma')$ and thus clearly

$$\mathcal{F}(\gamma \,\|\, \Pi \lhd \Gamma) \xrightarrow{\gamma' \| \Pi' \lhd \Gamma'}{}^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\gamma' \,\|\, \Pi' \lhd \Gamma')$$

which is what we wanted to prove.

We can thus deduce that R is a weak simulation.

Let us first prove a lemma that will be the basis of our proof that R^{-1} is a weak simulation.

Lemma. Suppose $\boldsymbol{s}(1), \cdots \boldsymbol{s}(m)$ is a sequence of configurations such that (A) for each $i \in \langle m-1 \rangle$ $\boldsymbol{s}(i) = s^i_0 \xrightarrow{\boldsymbol{r}^i(1)}_{\mathcal{N}} s^i_1 \xrightarrow{\boldsymbol{r}^i(2)}_{\mathcal{N}} \cdots \xrightarrow{\boldsymbol{r}^i(m^i-1)}_{\mathcal{N}} s^i_{m^i-1} \xrightarrow{\boldsymbol{r}^i(m^i)}_{\mathcal{N}} s^i_{m^i} = \boldsymbol{s}(i+1)$ where only one rule, $\boldsymbol{r}^i(j^i)$ say, is not a weak spawn rule; (B) $\boldsymbol{s}(1) \in \mathcal{F}(\Pi \lhd \Gamma)$ and $\boldsymbol{s}(m) \in \mathcal{F}(\Pi' \lhd \Gamma')$; and (C) for all $2 \leq i < m$ there *does not* exist $\Pi_0 \lhd \Gamma_0$ such that $\boldsymbol{s}(i) \in \mathcal{F}(\Pi_0 \lhd \Gamma_0)$. Then $\Pi \lhd \Gamma \to_{\mathcal{G}} \Pi' \lhd \Gamma'$.

Proof. It is easy to see that if a configuration $s \in \mathcal{F}(\Pi_0 \lhd \Gamma_0)$ for some $\Pi_0 \lhd \Gamma_0$ and $s \xrightarrow{r}_{\mathcal{N}} s'$ where r is a weak spawn rule then $s' \in \mathcal{F}(\Pi_0 \lhd \Gamma_0)$. Hence if we look at the transitions $\boldsymbol{s}(1) \xrightarrow{\boldsymbol{r}^1(1)}_{\mathcal{N}} s^1_1 \xrightarrow{\boldsymbol{r}^1(2)}_{\mathcal{N}} \cdots \xrightarrow{\boldsymbol{r}^1(j^1)}_{\mathcal{N}} s^1_{j^1}$ we can conclude that for all $0 \leq l < j^1$ we have $s^1_l \in \mathcal{F}(\Pi \lhd \Gamma)$. Thus we can decompose $s^1_{j^1-1}$ as $s^1_{j^1-1} = s^\Pi \oplus \mathcal{F}^\Gamma(\Gamma) \oplus s_{sim}$ where $s^\Pi \in \mathcal{F}^\Pi(\Pi)$.

Let us do a case analysis on which rule of the alternative semantics justifies the introduction of $\boldsymbol{r}^1(j^1)$ into $\mathcal{N}$'s rules.

- **Rule** (R′-2).

There are many candidate rules $\boldsymbol{r}^1(j^1)$ at first sight. However, we know that $s^1_{j^1-1}$ does not have s_{CCFG} or s_{CCFG+} as a submarking. Hence we must have $\boldsymbol{r}^1(j^1) = ((p^C_{A,\ell_{k+1}\,\Xi}, s_{sim}), (p^{CCFG}_{B,\overline{\ell}\,\Xi}, \emptyset, s_{CCFG+?} \oplus O))$ for some $\mathcal{G}$ rule $A \to B\,C$ where C is commutative and $\overline{\ell} \in \{+,-\}$ and $CCFG{+}? \in \{CCFG{+}, CCFG\}$ if $\ell_{k+1} = +$; $\overline{\ell} = \ell_{k+1}$, and $CCFG{+}? = CCFG$ otherwise; and $O = s^{budget}_{|\mathcal{N}|} \oplus [p^{CCFG}_C \mapsto [\bullet]]$.

Thus with $s^1_{j^1}$ a computation of the CCFG starts and until for some i, j the rule $\boldsymbol{r}^i(j)$ exists the CCFG computation, i.e. $\boldsymbol{r}^i(j) = ((p^{CCFG}_{B,\overline{\ell}'\,\Xi}, s_{CCFG+?} \oplus s^{budget}_{|\mathcal{N}|+1}), (p^C_{B,\overline{\ell}'\,\Xi}, \emptyset, s_{sim}))$ where $\overline{\ell}' \in \{-\} \cup \{A^{\text{cov}}_i \mid i \in \langle n \rangle\, \mathbb{M}(w)(A^{\text{cov}}_i) > 0\}$ if $\overline{\ell} = -$ and $\overline{\ell}' = \overline{\ell}$ otherwise. Along this path between the configurations $s^1_{j^1}$ and s^i_j the submarking s_{sim} cannot appear and hence no weak spawning rule can apply. Thus we know that $m^1 = j^1$, $j = 1$ and for $1 < i' < i$ it is the case that $m^{i'} = 1$. We know that $s^\Pi \in \mathcal{F}^\Pi(\Pi)$ and hence $\Pi = A\,M_{k+1}\,X_k\,M_k \cdots X_1\,M_1 \parallel \Pi_0$ and $[p^C_{A,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(A\,M_{k+1}\,X_k\,M_k \cdots X_1\,M_1)$ where $m = \mathcal{F}^I(M_1, \ldots, M_{k+1})$. Further it is clear that $m \in p^C_{A,\ell_{k+1}\,\Xi}$ in $s^1_{j^1-1}$. Hence the CCFG widget guarantees [GM12] that $s^i_j = (s^\Pi \ominus [p^C_{A,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{B,\overline{\ell}'\,\Xi} \mapsto [m']]) \oplus \mathcal{F}^\Gamma(\Gamma) \oplus s_{sim}$ where $m' = \mathcal{F}^I(M_1, \ldots, M_{k+1} \oplus \mathbb{M}(w))$ for some w such that $C \to^* w$.

Further the CCFG widget guarantees that if $\overline{\ell}' = +$ then $M_{k+1} \oplus \mathbb{M}(w) \in \mathit{CompleteSummary}$ and $M_{k+1} \oplus \mathbb{M}(w) \in \mathit{PartialSummary}$ otherwise. Hence $[p^C_{B,\overline{\ell}'\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(B, (M_{k+1} \oplus \mathbb{M}(w))\, X_k\, M_k \cdots X_1\, M_1)$ and we can deduce that $s^\Pi \ominus [p^C_{A,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{B,\overline{\ell}'\,\Xi} \mapsto [m']] \in \mathcal{F}^\Pi(B\,(M_{k+1} \oplus \mathbb{M}(w))\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0)$. It is thus the case that $s^i_1 \in \mathcal{F}^\Pi(B\,(M_{k+1} \oplus \mathbb{M}(w))\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \lhd \Gamma)$ which implies (a) $m^i = 1$, (b) $m = i$ and thus (c) $\Pi' \lhd \Gamma' = B\,(M_{k+1} \oplus \mathbb{M}(w))\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \lhd \Gamma$.

We can easily check that $\Pi \lhd \Gamma \rightarrow_\mathcal{G} \Pi' \lhd \Gamma'$ and hence $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_\mathcal{G} \Pi' \lhd \Gamma'$ which is what we wanted to prove.

- *Rule* (R′-1).

We can assume that $\boldsymbol{r}^1(j^1) = ((p^C_{A,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{\overline{\delta}\,\ell_{k+1}\,\Xi}, \emptyset, s_{sim}))$ for some non-terminals and cache characters $A\ell_{k+1}\, X_k\, \ell_k \cdots X_1\, \ell_1$ and $\mathcal{G}$ rule $A \rightarrow \delta$ where $k \leq K+1$ and $\overline{\delta}$ depends on δ and k: $\overline{\delta} = B$, $(+)\, C$ if $\delta = B\, C$, $k < K$; and $\overline{\delta} = a$ if $\delta = a$ with $a \in \Sigma \cup \{\epsilon\}$.

Hence we can deduce there exists a complex token m located at place $p^C_{A,\ell_{k+1}\,\Xi}$ in $s^1_{j^1-1}$ and $s^1_{j^1} = (s^\Pi \ominus [p^C_{A,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\overline{\delta},\ell_{k+1}\,\Xi} \mapsto [m]]) \oplus \mathcal{F}^\Gamma(\Gamma) \oplus s_{sim}$.

We know $s^\Pi \in \mathcal{F}^\Pi(\Pi)$ and hence $\Pi = A\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0$ and $[p^C_{A,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(A\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1)$. Then clearly $[p^C_{\overline{\delta},\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(\delta\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1)$ and we can deduce that $s^\Pi \ominus [p^C_{A,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\overline{\delta}\,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\Pi(\delta\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0)$.

It is thus easy to see that $s^1_{j^1} \in \mathcal{F}^\Pi(\delta\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \lhd \Gamma)$. From our assumptions above this implies (a) $m^1 = j^1$, (b) $m = 1$ and thus (c) $\Pi' \lhd \Gamma' = \delta\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \lhd \Gamma$. It is also easy to see that $\Pi \lhd \Gamma \rightarrow_\mathcal{G} \Pi' \lhd \Gamma'$ and hence $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_\mathcal{G} \Pi' \lhd \Gamma'$ which is what we wanted to prove.

- **Rules** (R′-3),(R′-4).

We will prove the case that the introduction of $\boldsymbol{r}^1(j^1)$ is justified by rule (R′-4). The case of (R′-3) are proved similarly.

We can thus assume that $\boldsymbol{r}^1(j^1) = ((p^C_{c!msg,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, \emptyset, [p^S_{c,msg} \mapsto [\bullet]] \oplus s_{sim}))$ for some non-terminals and cache characters $\ell_{k+1}\, X_k\, \ell_k \cdots X_1\, \ell_1$ where $k \leq K+1$. Hence there exists a complex token m located at place $p^C_{c!msg,\ell_{k+1}\,\Xi}$ and $s^1_{j^1} = (s^\Pi \ominus [p^C_{c!msg,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]]) \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus [p^S_{c,msg} \mapsto [\bullet]]) \oplus s_{sim}$.

Since $s^\Pi \in \mathcal{F}^\Pi(\Pi)$ we can deduce that $\Pi = c\,!\,msg\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0$ and $[p^C_{c!msg,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(c\,!\,msg\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1)$. Then clearly $[p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1)$ and we can deduce that $s^\Pi \ominus [p^C_{c!msg,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\Pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0)$. Further it easy to see that $\mathcal{F}^\Gamma(\Gamma) \oplus [p^S_{c,msg} \mapsto [\bullet]] = \mathcal{F}^\Gamma(\Gamma \oplus [msg]^c)$. We can then conclude that $s^1_{j^1} \in \mathcal{F}^\Pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \lhd \Gamma \oplus [msg]^c)$. From our assumptions above this implies (a) $m^1 = j^1$, (b) $m = 1$ and thus (c) $\Pi' \lhd \Gamma' = \epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \lhd \Gamma \oplus [msg]^c$. It is also easy to see that $\Pi \lhd \Gamma \rightarrow_\mathcal{G} \Pi' \lhd \Gamma'$ and hence $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_\mathcal{G} \Pi' \lhd \Gamma'$ which is what we wanted to prove.

- **Rule** (R′-5).
Since by definition $\boldsymbol{r}^1(j^1)$ is not a weak spawn rule, we can assume that $\boldsymbol{r}^1(j^1) = ((p^C_{\nu X,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, \emptyset, s_X \oplus s_{sim}))$ for $X \in \mathcal{N}$ and some non-terminals and cache characters $\ell_{k+1}\, X_k\, \ell_k \cdots X_1\, \ell_1$ where $k \leq K+1$ and $s_X = [p^S_{\nu X} \mapsto [\bullet]]$. Hence there exists a complex token m located at place $p^C_{\nu X,\ell_{k+1}\,\Xi}$ and $s^1_{j^1} = (s^\Pi \ominus [p^C_{\nu X,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]]) \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus [p^S_{\nu X} \mapsto [\bullet]]) \oplus s_{sim}$.

Since $s^\Pi \in \mathcal{F}^\Pi(\Pi)$ we can deduce that $\Pi = \nu X\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0$ and
$[p^C_{\nu X,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(c\,!\,msg\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1)$.
Then clearly $[p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1)$ and we can deduce that $s' := s^\Pi \ominus [p^C_{\nu X,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\Pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0)$. Clearly $s_X \in \mathcal{F}^\pi(X)$ and thus we can deduce that $s' \oplus s_X \in \mathcal{F}^\Pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \parallel X)$

We can then conclude that $s^1_{j^1} \in \mathcal{F}^\Pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \parallel X \lhd \Gamma)$. From our assumptions above this implies (a) $m^1 = j^1$, (b) $m = 1$ and thus (c) $\Pi' \lhd \Gamma' = \epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \parallel X \lhd \Gamma$. It is also easy to see that $\Pi \lhd \Gamma \to_{\mathcal{G}} \Pi' \lhd \Gamma'$ and hence $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_{\mathcal{G}} \Pi' \lhd \Gamma'$ which is what we wanted to prove.

- **Rule** (R′-6)–(R′-7).
We will give a proof of the case that the introduction of $\boldsymbol{r}^1(j^1)$ is justified by rule (R′-7). The proof in the case that $\boldsymbol{r}^1(j^1)$ introduced to implement rule (R′-6) is analogous.

We can thus assume that $\boldsymbol{r}^1(j^1) = ((p^C_{\epsilon,\ell_{k+1}\,\Xi}, s_{sim}), (p^C_{\epsilon,\ell_{k+1}\,\Xi}, P, s_{sim})$ for some non-terminals and cache characters $\ell_{k+1}\, X_k\, \ell_k \cdots X_1\, \ell_1$ where $k \leq K$, $P = \left\{p^I_{k+1,e} \mid e \in \Sigma^{\text{com}}\right\}$ and $\ell_{k+1} \in \{-\} \cup \{A^{\text{cov}}_i : i \in \langle n \rangle\}$.

Hence there exists a complex token m located at place $p^C_{\epsilon,\ell_{k+1}\,\Xi}$ and $s^1_{j^1} = (s^\Pi \ominus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m_{\overline{P}}]]) \oplus (\mathcal{F}^\Gamma(\Gamma) \oplus (m_P \circ \zeta^{-1})) \oplus s_{sim}$ where $m_P = m \restriction P$ and $m_{\overline{P}} = m \restriction (\mathcal{P}^{col} \setminus P)$.

Since $s^\Pi \in \mathcal{F}^\Pi(\Pi)$ we can deduce that $\Pi = \epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0$ and $[p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]] \in \mathcal{F}^\pi(c\,!\,msg\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1)$. We further know that $m = \mathcal{F}^I(M_1, \ldots, M_k, M_{k+1})$ and $M_{k+1} \in \mathit{PartialSummary}$. It is easy to see that $m_{\overline{P}} = \mathcal{F}^I(M_1, \ldots, M_k, M')$ where we write $M' = M_{k+1} \restriction (\mathcal{N}^{\text{com}} \cup \{A^{\text{cov}}_i : i \in \langle n \rangle\})$.

Then clearly $[p^C_{\epsilon\,\ell_{k+1}\,\Xi} \mapsto [m_{\overline{P}}]] \in \mathcal{F}^\pi(\epsilon, M'\, X_k\, M_k \cdots X_1\, M_1)$ and we can deduce that $s^\Pi \ominus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m]] \oplus [p^C_{\epsilon,\ell_{k+1}\,\Xi} \mapsto [m_{\overline{P}}]] \in \mathcal{F}^\Pi(\epsilon\, M'\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0)$.

Inspecting case **Rules** (R′-6),(R′-7) in the proof of R is a simulation we can see that $(m_P \circ \zeta^{-1}) \restriction \mathcal{G}^\Gamma = \mathcal{F}^\Gamma(\Gamma(M_{k+1}))$ and $(m_P \circ \zeta^{-1}) \restriction \mathcal{G}^\nu \in \mathcal{F}^\Pi(\Pi(M_{k+1}))$ where $\mathcal{G}^\Gamma = \{p^S_{c,msg} \mid c \in \mathit{Chan}, msg \in \mathit{Msg}\}$ and $\mathcal{G}^\nu = \{p^S_{\nu X} \mid X \in \mathcal{N}\}$. Further $(m_P \circ \zeta^{-1}) \restriction \mathcal{G}^\Gamma \oplus (m_P \circ \zeta^{-1}) \restriction \mathcal{G}^\nu = (m_P \circ \zeta^{-1})$.

Hence we obtain $\mathcal{F}^\Gamma(\Gamma) \oplus (m_P \circ \zeta^{-1}) \restriction \mathcal{G}^\Gamma = \mathcal{F}^\Gamma(\Gamma \oplus \Gamma(M_{k+1}))$. We can then conclude that $s^1_{j^1} \in \mathcal{F}^\Pi(\epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \parallel \Pi(M_{k+1}) \lhd \Gamma \oplus \Gamma(M_{k+1}))$. From our

assumptions above this implies (a) $m^1 = j^1$, (b) $m = 1$ and thus (c) $\Pi' \lhd \Gamma' = \epsilon\, M_{k+1}\, X_k\, M_k \cdots X_1\, M_1 \parallel \Pi_0 \parallel \Pi(M_{k+1}) \lhd \Gamma \oplus \Gamma(M_{k+1})$. It is also easy to see that $\Pi \lhd \Gamma \rightarrow_{\mathcal{G}} \Pi' \lhd \Gamma'$ and hence $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_{\mathcal{G}} \Pi' \lhd \Gamma'$ which is what we wanted to prove.

□

Let us now turn to R^{-1}. Suppose $(\Pi \lhd \Gamma, \mathcal{F}(\Pi \lhd \Gamma)) \in R$ and $\mathcal{F}(\Pi \lhd \Gamma) \xrightarrow{\Pi' \lhd \Gamma'}^*_{\mathscr{P}[\mathcal{N}]} S$. Hence $\mathcal{F}(\Pi \lhd \Gamma) \xrightarrow{\Pi' \lhd \Gamma'}^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi' \lhd \Gamma') \xrightarrow{\epsilon}^*_{\mathscr{P}[\mathcal{N}]} S$. By definition this implies that there exists non-empty subsets of *Config*, $\boldsymbol{S}(1), \ldots, \boldsymbol{S}(m)$ say, such that $m \geq 1$,

$$\mathcal{F}(\Pi \lhd \Gamma) = \boldsymbol{S}(1) \xrightarrow{\epsilon}_{\mathscr{P}[\mathcal{N}]} \boldsymbol{S}(2) \xrightarrow{\epsilon}_{\mathscr{P}[\mathcal{N}]} \cdots \xrightarrow{\epsilon}_{\mathscr{P}[\mathcal{N}]} \boldsymbol{S}(m-1) \xrightarrow{\Pi' \lhd \Gamma'}_{\mathscr{P}[\mathcal{N}]} \boldsymbol{S}(m) = \mathcal{F}(\Pi' \lhd \Gamma')$$

Thus for $i \in \langle m \rangle$ we may pick configuration $\boldsymbol{s}(i) \in \boldsymbol{S}(i)$ such that for each $i \in \langle m-1 \rangle$ $\boldsymbol{s}(i) = s^i_0 \xrightarrow{\boldsymbol{r}^i(1)}_{\mathcal{N}} s^i_1 \xrightarrow{\boldsymbol{r}^i(2)}_{\mathcal{N}} \cdots \xrightarrow{\boldsymbol{r}^i(m^i-1)}_{\mathcal{N}} s^i_{m^i-1} \xrightarrow{\boldsymbol{r}^i(m^i)}_{\mathcal{N}} s^i_{m^i} = \boldsymbol{s}(i+1)$ where only one rule, $\boldsymbol{r}^i(j^i)$ say, is not a weak spawn rule. We further know that for all $2 \leq i < m$ there *does not* exist $\Pi_0 \lhd \Gamma_0$ such that $\boldsymbol{s}(i) \in \mathcal{F}(\Pi_0 \lhd \Gamma_0)$. Since clearly $\boldsymbol{s}(1) \in \mathcal{F}(\Pi \lhd \Gamma)$ and $\boldsymbol{s}(m) \in \mathcal{F}(\Pi' \lhd \Gamma')$ the Lemma above applies to give us $\Pi \lhd \Gamma \xrightarrow{\Pi' \lhd \Gamma'}_{\mathcal{G}} \Pi' \lhd \Gamma'$. We can thus conclude that R^{-1} is a weak simulation and hence R is a weak bisimulation.

Further suppose that we have $s \in \mathcal{F}(\Pi \lhd \Gamma)$ for some $\Pi \lhd \Gamma$ and $s' \in \mathcal{F}(\Pi' \lhd \Gamma')$ and $s \rightarrow^*_{\mathcal{N}} s'$ then we can decompose this path into a sequence of configurations $\boldsymbol{s}(1) = s \rightarrow_{\mathcal{N}} \boldsymbol{s}(2) \rightarrow_{\mathcal{N}} \cdots \rightarrow_{\mathcal{N}} \boldsymbol{s}(l) = s'$. Let $i_1, \ldots, i_{l'}$ be the indices, in order, such that $\boldsymbol{s}(i_j) \in \mathcal{F}(\Pi_j \lhd \Gamma_j)$ for some $\Pi_1 \lhd \Gamma_1, \ldots, \Pi_{l'} \lhd \Gamma_{l'}$. Clearly for all $1 \leq j \leq l'$ we can apply the Lemma above to the path $\boldsymbol{s}(i_j) \rightarrow^*_{\mathcal{N}} \boldsymbol{s}(i_{j+1})$ to obtain $\Pi_1 \lhd \Gamma_1 \xrightarrow{\Pi_2 \lhd \Gamma_2}_{\mathcal{G}} \Pi_2 \lhd \Gamma_2 \xrightarrow{\Pi_3 \lhd \Gamma_3}_{\mathcal{G}} \cdots \xrightarrow{\Pi_{l'} \lhd \Gamma_{l'}}_{\mathcal{G}} \Pi_{l'} \lhd \Gamma_{l'}$ which means that since R is a weak bisimulation that $\mathcal{F}(\Pi \lhd \Gamma) \rightarrow^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi' \lhd \Gamma')$. Hence we can conclude that if $s \in \mathcal{F}(\Pi \lhd \Gamma)$ for some $\Pi \lhd \Gamma$ and $s' \in \mathcal{F}(\Pi' \lhd \Gamma')$ and $s \rightarrow^*_{\mathcal{N}} s'$ then $\mathcal{F}(\Pi \lhd \Gamma) \rightarrow^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi' \lhd \Gamma')$.

We know that $\Pi_0 = A^0_1 \parallel \cdots \parallel A^0_{n'}$ so let $s_{cov} = s_{\mathit{Query}} \oplus [p^{cov}_{A_1} \mapsto [\bullet], \ldots, p^{cov}_{A_n} \mapsto [\bullet]] \oplus [p^S_{c,msg} \mapsto [\bullet^{\Gamma^{\mathrm{cov}}(c)(m)}] \mid c \in \mathit{Chan}, msg \in \mathit{Msg}]$ and $s_0 = s_{sim} \oplus [p^S_{\nu A^0_i} \mapsto [\bullet] \mid i \in \langle n' \rangle] \oplus [p^S_{c,msg} \mapsto [\bullet^{\Gamma_0(c)(m)}] \mid c \in \mathit{Chan}, msg \in \mathit{Msg}]$. We note that $s_0 \in \mathcal{F}(\Pi_0 \lhd \Gamma_0)$.

We will now show that $(\mathcal{N}, s_0, s_{cov})$ is a yes-instance of the coverability problem if and only if $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, A^{\mathrm{cov}}_1 \parallel \cdots \parallel A^{\mathrm{cov}}_n \lhd \Gamma^{\mathrm{cov}})$ is a yes-instance of simple coverability.

Suppose $(\mathcal{N}, s_0, s_{cov})$ is a yes-instance of the coverability problem. Then there exists a $s' \in \mathit{Config}$ such that $s_0 \rightarrow^*_{\mathcal{N}} s'$ and $s_{cov} \leq_{\mathit{Config}} s'$. That means that there exists a configuration s'' such that $s_0 \rightarrow^*_{\mathcal{N}} s''$, $s'' \rightarrow_{\mathcal{N}} \boldsymbol{s}(1) \rightarrow_{\mathcal{N}} \cdots \rightarrow_{\mathcal{N}} \boldsymbol{s}(l)$, $\boldsymbol{s}(l) = s'$, $s'' \in \mathcal{F}(\Pi \lhd \Gamma)$ for some $\Pi \lhd \Gamma$, and for each $1 \leq i \leq l$ there does not exists a $\bar{\Pi} \lhd \bar{\Gamma}$ such that $\boldsymbol{s}(i) \in \mathcal{F}(\bar{\Pi} \lhd \bar{\Gamma})$. We can thus appeal to our reasoning above to obtain that $\mathcal{F}(\Pi_0 \lhd \Gamma_0) \rightarrow^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi \lhd \Gamma)$. We can decompose this path to get $\mathcal{F}(\Pi_0 \lhd \Gamma_0) \xrightarrow{\Pi_1 \lhd \Gamma_1}^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi_1 \lhd \Gamma_1) \xrightarrow{\Pi_2 \lhd \Gamma_2}^*_{\mathscr{P}[\mathcal{N}]} \cdots \xrightarrow{\Pi_{l'} \lhd \Gamma_{l'}}^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi_{l'} \lhd \Gamma_{l'}) = \mathcal{F}(\Pi \lhd \Gamma)$. Since R is a weak bisimulation we thus know that $\Pi_0 \lhd \Gamma_0 \xrightarrow{\Pi_1 \lhd \Gamma_1}^*_{\mathcal{G}} \Pi_1 \lhd \Gamma_1 \xrightarrow{\Pi_2 \lhd \Gamma_2}^*_{\mathcal{G}} \cdots \xrightarrow{\Pi_{l'} \lhd \Gamma_{l'}}^*_{\mathcal{G}} \Pi_{l'} \lhd \Gamma_{l'} = \Pi \lhd \Gamma$ and thus $\Pi_0 \lhd \Gamma_0 \rightarrow^*_{\mathcal{G}} \Pi \lhd \Gamma$.

Let us turn to an inspection of s''. We know that $s'' \in \mathcal{F}(\Pi \lhd \Gamma)$, for all $1 \leq i \leq l$ there *does not* exist $\bar{\Pi} \lhd \bar{\Gamma}$ such that $\boldsymbol{s}(i) \in \mathcal{F}(\bar{\Pi} \lhd \bar{\Gamma})$, and $s_{cov} \leq_{\mathit{Config}} s' = \boldsymbol{s}(l)$. Since $s_{\mathit{Query}} \leq_{\mathit{Config}} s_{cov}$, we can conclude that $s_{\mathit{Query}} \leq_{\mathit{Config}} s'$. This implies that for some $0 \leq i < l$ we have the transition $\boldsymbol{s}(j) \xrightarrow{r}_{\mathcal{N}} \boldsymbol{s}(j+1)$ via the rule $r = (s_{\mathit{sim}}, s_{\mathit{Query}})$ which switches from simulation mode to query mode and where we write $\boldsymbol{s}(0) = s''$. Inspecting the rules of $\mathcal{N}$ we note that this mode switch cannot be reversed. We can deduce that $j = 0$ since using any other rule of $\mathcal{N}$ for the transition $s'' \xrightarrow{r}_{\mathcal{N}} \boldsymbol{s}(1)$ would either place $\boldsymbol{s}(1) \in \mathcal{F}(\bar{\Pi} \lhd \bar{\Gamma})$ for some $\bar{\Pi} \lhd \bar{\Gamma}$ immediately, or start a CCFG computation disabling r until completion which would produce another $\boldsymbol{s}(j) \in \mathcal{F}(\Pi_0 \lhd \Gamma_0)$ for some $\Pi_0 \lhd \Gamma_0$ before r could be enabled. Hence we can conclude that $s''(p^{\mathit{cov}}_{A_i}) = \emptyset$ for all $1 \leq i \leq n$ since $s'' \in \mathcal{F}(\Pi \lhd \Gamma)$ and for all $1 \leq j \leq l$ such that $\boldsymbol{s}(j) \xrightarrow{\boldsymbol{r}(j)}_{\mathcal{N}} \boldsymbol{s}(j+1)$ the rule $\boldsymbol{r}(j)$ is a rule of the form $((p^C_{A_i^{\mathrm{cov}},\ell_{k+1}}\Xi, s_{\mathit{Query}}), (p^C_{\epsilon,(-)}\Xi, \emptyset, s_{\mathit{Query}} \oplus [p^{\mathit{cov}}_{A_i} \mapsto [\bullet]]))$ or $((p^C_{\epsilon,A_i^{\mathrm{cov}}}\Xi, s_{\mathit{Query}}), (p^C_{\epsilon,(-)}\Xi, \emptyset, s_{\mathit{Query}} \oplus [p^{\mathit{cov}}_{A_i} \mapsto [\bullet]]))$. Since $s_{cov} \leq_{\mathit{Config}} s'$ this means that at least there are indices $i_1, \ldots, i_n$ (not necessarily in order) where for each $1 \leq j \leq n$ rule $\boldsymbol{r}(i_j)$ places a $\bullet$-token into place $p^{\mathit{cov}}_{A_j}$. Since the rules $\boldsymbol{r}(1), \ldots, \boldsymbol{r}(l)$ only remove complex tokens from places of the form $p^C_{A,\ell_{k+1}}\Xi$ or $p^C_{\epsilon,A}\Xi$ we can conclude that there exists complex tokens $m_1, ..., m_n$ located in places $p_1, ..., p_n$ in configuration s'' such that for all $1 \leq j \leq n$ the place $p_j = p^C_{A_j,\ell_{k+1}}\Xi$ or $p^C_{\epsilon,A_j}\Xi$. Since $s'' \in \mathcal{F}(\Pi \lhd \Gamma)$ this implies there exists processes $\pi_1, \ldots, \pi_n$ such that $\Pi = \pi_1 \parallel \cdots \parallel \pi_n \parallel \Pi_0$ and each $\pi_j = A_j^{\mathrm{cov}} \cdot \gamma_j$ or $\epsilon\, M_j \cdot \gamma_j$ with $M_j(A_j^{\mathrm{cov}}) > 0$ for $1 \leq j \leq n$. Further it is also easy to see that Γ^{cov} is covered by Γ. Hence we can deduce that $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, A_1^{\mathrm{cov}} \parallel \cdots \parallel A_n^{\mathrm{cov}} \lhd \Gamma^{\mathrm{cov}})$ is a yes-instances of simple coverability for the alternative semantics.

For the other direction of the reduction suppose $(\mathcal{G}, \Pi_0 \lhd \Gamma_0, A_1^{\mathrm{cov}} \parallel \cdots \parallel A_n^{\mathrm{cov}} \lhd \Gamma^{\mathrm{cov}})$ is a yes-instances of simple coverability in the alternative semantics. This means that

$$\Pi_0 \lhd \Gamma_0 \xrightarrow{\Pi_1 \lhd \Gamma_1}_{\mathcal{G}} \Pi_1 \lhd \Gamma_1 \xrightarrow{\Pi_2 \lhd \Gamma_2}_{\mathcal{G}} \cdots \xrightarrow{\Pi_l \lhd \Gamma_l}_{\mathcal{G}} \Pi_l \lhd \Gamma_l =: \Pi \lhd \Gamma$$

where $\Pi = \pi_1 \parallel \cdots \parallel \pi_n \parallel \Pi_0$ and each $\pi_j = A_j^{\mathrm{cov}} \cdot \gamma_j$ or $\epsilon\, M_j \cdot \gamma_j$ with $M_j(A_j^{\mathrm{cov}}) > 0$ for $1 \leq j \leq n$.

Since R is a weak bisimulation we obtain

$$\mathcal{F}(\Pi_0 \lhd \Gamma_0) \xrightarrow{\Pi_1 \lhd \Gamma_1}{}^*_{\mathscr{P}[\mathcal{N}]} \cdots \xrightarrow{\Pi_{l-1} \lhd \Gamma_{l-1}}{}^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi_{l-1} \lhd \Gamma_{l-1}) \xrightarrow{\Pi \lhd \Gamma}{}^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi \lhd \Gamma).$$

Thus we may pick $\boldsymbol{s}(1), \ldots, \boldsymbol{s}(l)$ such that for all $1 \leq j \leq l$ we have $\boldsymbol{s}(j) \in \mathcal{F}(\Pi_j \lhd \Gamma_j)$ and for $j < l$ we have $\boldsymbol{s}(j) \to_{\mathcal{N}} \boldsymbol{s}(j+1)$. Hence $\boldsymbol{s}(1) \to^*_{\mathcal{N}} \boldsymbol{s}(l)$. Further since $\mathcal{F}(\Pi_0 \lhd \Gamma_0) \to^*_{\mathscr{P}[\mathcal{N}]} \mathcal{F}(\Pi_1 \lhd \Gamma_1)$ and $s_0 \to^*_{\mathcal{N}} s$ for all $s \in \mathcal{F}(S \lhd \emptyset)$ we can conclude that $s_0 \to^*_{\mathcal{N}} \boldsymbol{s}(l) =: s'$.

Let us now inspect s'. Since $s' \in \mathcal{F}(\Pi \lhd \Gamma)$ we can deduce that there exists complex tokens $m_1, \ldots, m_n$ located in places $p_1, ..., p_n$ in configuration s' such that for all $1 \leq j \leq n$ the place $p_j = p^C_{A_j^{\mathrm{cov}},\ell_{k+1}}\Xi$ or $p^C_{\epsilon,A_j^{\mathrm{cov}}}\Xi$. Hence we may extend the derivation $s_0 \to^*_{\mathcal{N}} s' \xrightarrow{r}_{\mathcal{N}} s'_0 \xrightarrow{\boldsymbol{r}(1)}_{\mathcal{N}} s'_1 \xrightarrow{\boldsymbol{r}(2)}_{\mathcal{N}} \cdots \xrightarrow{\boldsymbol{r}(n)}_{\mathcal{N}} s'_n$ where $r = (s_{\mathit{sim}}, s_{\mathit{Query}})$ switches from simulation mode to query mode and the rule $\boldsymbol{r}(j) = ((p^C_{A_j^{\mathrm{cov}},\ell_{k+1}}\Xi, s_{\mathit{Query}}), (p^C_{\epsilon,(-)}\Xi, \emptyset, s_{\mathit{Query}} \oplus [p^{\mathit{cov}}_{A_j} \mapsto [\bullet]]))$ or $((p^C_{\epsilon,A_j^{\mathrm{cov}}}\Xi, s_{\mathit{Query}}), (p^C_{\epsilon,(-)}\Xi, \emptyset, s_{\mathit{Query}} \oplus [p^{\mathit{cov}}_{A_j} \mapsto [\bullet]]))$. and thus we can deduce that s'_n has one

•-token located in each place $p^{cov}_{A_1}, \ldots, p^{cov}_{A_n}$ and s_{Query} as a submarking, i.e. $s_{\text{cov}} \leq_{Config} s'_n$. Hence we can deduce that $(\mathcal{N}, s_0, s_{cov})$ is a yes-instance of the coverability problem.

Hence we can conclude that the simple coverability problem for K-shaped APCPS in the alternative semantics EXPTIME reduces to the coverability problem of NNCT. □

Theorem 11. *Simple coverability, boundedness and termination for a total-transfer NNCT EXPTIME reduces to simple coverability, boundedness and termination respectively for a 4-shaped APCPS in the alternative semantics.*

Proof. Fix a total-transfer NNCT $\mathcal{N} = (\mathcal{P}^S, \mathcal{P}^C, \mathcal{P}^{col}, \mathcal{R}, \zeta)$ and a coverability query $(\mathcal{N}, s_0, s_{\text{cov}})$ with a simple query, i.e. $s_0(p) = s_{\text{cov}}(p) = \emptyset$ for all $p \in \mathcal{P}^C$. Let us also fix an enumeration of $\mathcal{P}^S = \left\{p_{(1)}, \ldots, p_{(n_S)}\right\}$, $\mathcal{P}^C = \left\{p'_{(1)}, \ldots, p'_{(n_C)}\right\}$ and $\mathcal{P}^{col} = \left\{p^{col}_{(1)}, \ldots, p^{col}_{(n_{col})}\right\}$. Since $\mathcal{N}$ is a total-transfer NNCT we know that for all $r \in$ *TransferRules* it is the case that $r = ((p, I), (p', \mathcal{P}^{col}, O))$.

Let us define an APCPS $\mathcal{G} = (\Sigma, I, \mathcal{N}, \mathcal{R}, S)$ that will simulate $\mathcal{N}$. The APCPS $\mathcal{G}$ has a channel c_p for each simple or complex place p plus a special channel $c_?$ and let the set of channels be $Chan = \{c_p \mid p \in \mathcal{P}^S \cup \mathcal{P}^C\} \cup \{c_?\}$. For $\mathcal{G}$'s messages let us first inspect $\mathcal{N}$'s complex rules: let Ξ be the set of complex tokens representing "injected" coloured tokens in $\mathcal{N}$'s complex rules plus the set of complex tokens that are created by simple rules, i.e. $\Xi = \{c \mid ((p, I), (p', c, O)) \in ComplexRules\} \cup \{c \mid p \in \mathcal{P}^C, (I, O) \in SimpleRules, O(p)(c) \geq 1\}$ then $\mathcal{G}$ will have a message $m_{p,d}$ for each $p \in \mathcal{P}^C$ and $d \in \Xi$ and two special messages $m_{p,\emptyset}, m_{p,\downarrow}$ for each $p \in \mathcal{P}^C$. Further $\mathcal{G}$ will have a message •; and hence we can let the set of messages be $Msg = \left\{msg_{p,d} \mid p \in \mathcal{P}^C, d \in \Xi \cup \{\emptyset, \downarrow\}\right\} \cup \{\bullet\}$.

The APCPS $\mathcal{G}$ will have non-terminals N_r, N^I_r for each rule $r \in \mathcal{R}$, non-terminals $N^{S,O}_r, N^{C,O}_r$ for each $r \in SimpleRules$, and non-terminals N^O_r, N^c_r for each $r \in ComplexRules \cup TransferRules$. Additionally $\mathcal{G}$ will have non-terminals $N^{\nu p}, N^p, N^p_\emptyset$ for every $p \in \mathcal{P}^C$, a non-terminal N^c for each $c \in \Xi$ and two special non-terminals $N^{\nu?}$ and N^{sim}. Hence $\mathcal{G}$'s set of non-terminals are

$$\begin{aligned}\mathcal{N} = &\left\{N^{\nu p}, N^p, N^p_\emptyset \mid p \in \mathcal{P}^C\right\} \cup \{N^c \mid c \in \Xi\} \cup \{N^{\nu?}, N^{sim}\} \\ &\cup \{N_r, N^I_r \mid r \in \mathcal{R}\} \cup \{N^{S,O}_r, N^{C,O}_r \mid r \in SimpleRules\} \\ &\cup \{N^O_r, N^c_r \mid r \in ComplexRules \cup TransferRules\}\end{aligned}$$

Let us also define a designated non-terminal A_{cov} to label the coverability query. We can then set $\mathcal{G}$'s alphabet $\Sigma = \{c\,!\,msg, c\,?\,msg, \nu X \mid c \in Chan, msg \in Msg, X \in \mathcal{N}\}$. Further let us use the standard independence relation for APCPS I over $\Sigma \cup \mathcal{N}$. A configuration $s \in Config$ will be represented by an APCPS configuration in the following way:

- for each simple place $p \in \mathcal{P}^S$, the channel c_p will contain $|s(p)|$ •-messages — we can formalise this by a function $\mathcal{F}^\Gamma$ that we define as $\mathcal{F}^\Gamma(s) = \left[\bullet^{|s(p_{(1)})|}\right]^{c_{p_{(1)}}}, \ldots, \left[\bullet^{|s(p_{(n_S)})|}\right]^{c_{p_{(n_S)}}}$;
- for each complex place $p \in \mathcal{P}^C$, suppose $s(p) = [m_1, \ldots, m_k]$ then for each $m_i \neq \emptyset$ there will be one process with a process-state $N^p \cdot \widetilde{\Gamma}(m_i \circ \zeta^{-1}) \cdot N^{\nu?}$ where $\widetilde{\Gamma}$ is a function that takes a mapping $m_0 : \mathcal{P}^S \to \mathbb{N}$ and transforms it into a multiset $\mathbb{M}[c_{\mathcal{P}^S}\,!\,\bullet]$ by $\widetilde{\Gamma}(m_0)(c_p\,!\,\bullet) = |m_0(p)|$. If $m_i = \emptyset$ there will be one process with a process-state $N^p_\emptyset \cdot \emptyset \cdot N^{\nu?} = N^p_\emptyset \cdot \widetilde{\Gamma}(m_i \circ \zeta^{-1}) \cdot N^{\nu?}$. We can formalise this with two functions

$\mathcal{F}^{\Pi}(s,p) = \left|\right|_{i=1}^{k} N^p_{m_i} \cdot \widetilde{\Gamma}(m_i \circ \zeta^{-1}) \cdot N^{\nu?}$ and $\mathcal{F}^{\Pi}(s) = \left|\right|_{i=1}^{n_C} \mathcal{F}^{\Pi}(s, p'_{(i)})$ where $N^p_{m_i} = N^p$ if $m_i \neq \emptyset$ and $N^p_\emptyset$ otherwise;

- in addition we have one administrative process, that implements the execution of $\mathcal{N}$'s rules and is in the process-state N^{sim} when representing the configuration s.

Formally, we define a representation function $\mathcal{F}$ by $\mathcal{F}(s) = N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s)$ and define the relation $R \subseteq \mathit{Config} \times \mathit{State}_{\text{alt}}$ by $R = \{(s, \mathcal{F}(s)) \mid s \in \mathit{Config}\}$. We will prove that R is a weak bisimulation for $(\mathit{Config}, \to_{\mathcal{N}})$ and $(\mathit{State}_{\text{alt}}, \to_{\mathcal{G}})$ where we write $\to_{\mathcal{G}}$ for $\to_{\mathcal{G}_{\text{alt}}}$ here and in the following.

We will now define and explain how the administrative process is implementing $\mathcal{N}$'s rules. For each rule $r \in \mathcal{R}$ the APCPS $\mathcal{G}$ has the rule $N^{sim} \to N_r N^{sim}$ which guesses one of $\mathcal{N}$'s rules to execute next. Depending on whether r is a simple, complex or transfer rule the implementation is different:

- $r = (I, O) \in \mathit{SimpleRules}$

  If r is a simple rule, then N_r rewrites to three non-terminals: N^I_r which removes $\bullet$-messages as described by I, $N^{O,S}_r$ which sends $\bullet$-messages as described by the effect of O on $\mathcal{N}$'s simple places and $N^{O,C}_r$ which spawns new processes which will represent the newly created complex tokens as described by O on $\mathcal{N}$'s complex places. Let us enumerate the set $\Xi = \left\{c_{(1)}, \ldots, c_{(n_\Xi)}\right\}$ then we can formally implement the above by the following rules:

$$
\begin{aligned}
N_r &\to N^I_r \cdot N^{S,O}_r \cdot N^{C,O}_r \\
N^I_r &\to \left(c_{p_{(1)}}\ ?\ \bullet\right)^{|I(p_{(1)})|} \cdots \left(c_{p_{(n_S)}}\ ?\ \bullet\right)^{|I(p_{(n_S)})|} \\
N^{S,O}_r &\to \left(c_{p_{(1)}}\ !\ \bullet\right)^{|O(p_{(1)})|} \cdots \left(c_{p_{(n_S)}}\ !\ \bullet\right)^{|O(p_{(n_S)})|} \\
N^{C,O}_r &\to \left(\nu N^{\nu p'_{(1)}} \left(c_?\ !\ \mathit{msg}_{p'_{(1)}, c_{(1)}}\right)(c_?\ ?\ \bullet)\right)^{O(p'_{(1)})(c_{(1)})} \cdots \\
&\qquad \left(\nu N^{\nu p'_{(i)}} \left(c_?\ !\ \mathit{msg}_{p'_{(i)}, c_{(j)}}\right)(c_?\ ?\ \bullet)\right)^{O(p'_{(i)})(c_{(j)})} \cdots \\
&\qquad \left(\nu N^{\nu p'_{(n_C)}} \left(c_?\ !\ \mathit{msg}_{p'_{(n_C)}, c_{(n_\Xi)}}\right)(c_?\ ?\ \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}
\end{aligned}
$$

  and the rules for complex token representing processes we have the following rule for each $c \in \Xi$ such that $c \neq \emptyset$ and $p \in \mathcal{P}^C$:

$$
\begin{aligned}
N^{\nu p} &\to \left(c_?\ ?\ \mathit{msg}_{p,c}\right)(c_?\ !\ \bullet)\, N^p \cdot N^c \cdot N^{\nu?} \\
N^{\nu p} &\to \left(c_?\ ?\ \mathit{msg}_{p,\emptyset}\right)(c_?\ !\ \bullet)\, N^p_\emptyset \cdot N^\emptyset \cdot N^{\nu?}
\end{aligned}
$$

  and for each $c \in \Xi$:

$$
N^c \to \left(c_{\zeta\left(p^{col}_{(1)}\right)}\ !\ \bullet\right)^{|c(p^{col}_{(1)})|} \cdots \left(c_{\zeta\left(p^{col}_{(n_{col})}\right)}\ !\ \bullet\right)^{|c(p^{col}_{(n_{col})})|}
$$

- $r = ((p, I), (p', c, O)) \in \mathit{ComplexRules}$

  If r is a complex rule, then N_r rewrites to three non-terminals: N^I_r which removes $\bullet$-messages as described by I, N^O_r which sends $\bullet$-messages as described by O and N^C_r which

forces one process representing one complex token m in complex place p to change state so that the process afterwards represents $m \oplus c$ in complex place p'. Formally we can implement this with the following rules:

$$\begin{aligned}
N_r &\to N_r^I \cdot N_r^O \cdot N_r^C \\
N_r^I &\to \left(c_{p_{(1)}} \,?\, \bullet\right)^{|I(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,?\, \bullet\right)^{|I(p_{(n_S)})|} \\
N_r^O &\to \left(c_{p_{(1)}} \,!\, \bullet\right)^{|O(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,!\, \bullet\right)^{|O(p_{(n_S)})|} \\
N_r^C &\to (c_p \,!\, msg_{p',c}) \cdot (c_p \,?\, \bullet)
\end{aligned}$$

and for each $c \in \Xi \setminus \{\emptyset\}$ and $p \in \mathcal{P}^C$ we have the following rule for complex token representing processes:

$$\begin{aligned}
N^p &\to (c_p \,?\, msg_{p',c}) \cdot (c_p \,!\, \bullet) \cdot N^{p'} N^c \\
N^p_\emptyset &\to (c_p \,?\, msg_{p',c}) \cdot (c_p \,!\, \bullet) \cdot N^{p'} N^c \\
N^p_\emptyset &\to (c_p \,?\, msg_{p',\emptyset}) \cdot (c_p \,!\, \bullet) \cdot N^{p'}_\emptyset N^\emptyset
\end{aligned}$$

- $r = ((p, I), (p', \mathcal{P}^{col}, O)) \in \mathit{TransferRules}$
  If r is a transfer rule, then N_r rewrites to three non-terminals: N_r^I which removes $\bullet$-messages as described by I, N_r^O which sends $\bullet$-messages as described by O and N_r^C which forces one process representing one complex token m in complex place p to change state and make its summary effective, which transfers $(m \circ \zeta^{-1})$ to the channels, so that the process afterwards represents the empty complex token $\emptyset$ in complex place p'. Formally we can implement this with the following rules:

$$\begin{aligned}
N_r &\to N_r^I \cdot N_r^O \cdot N_r^C \\
N_r^I &\to \left(c_{p_{(1)}} \,?\, \bullet\right)^{|I(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,?\, \bullet\right)^{|I(p_{(n_S)})|} \\
N_r^O &\to \left(c_{p_{(1)}} \,!\, \bullet\right)^{|O(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,!\, \bullet\right)^{|O(p_{(n_S)})|} \\
N_r^C &\to (c_p \,!\, msg_{p,\downarrow}) \cdot (c_p \,?\, \bullet) \cdot (c_? \,!\, msg_{p',\emptyset}) \cdot (c_? \,?\, \bullet)
\end{aligned}$$

and for each $p \in \mathcal{P}^C$ we have the following rule for complex token representing processes:

$$\begin{aligned}
N^p &\to (c_p \,?\, msg_{p,\downarrow}) \cdot (c_p \,!\, \bullet) \\
N^{\nu^?} &\to (c_? \,?\, msg_{p,\emptyset}) \cdot (c_? \,!\, \bullet) \cdot N^p_\emptyset \cdot N^{\nu^?}
\end{aligned}$$

Further we add two more rules:

$$\begin{aligned}
S &\to \left(c_{p_{(1)}} \,!\, \bullet\right)^{|s_0(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,!\, \bullet\right)^{|s_0(p_{(n_S)})|} N^{sim} \\
N^{sim} &\to \left(c_{p_{(1)}} \,?\, \bullet\right)^{|s_{\text{cov}}(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,?\, \bullet\right)^{|s_{\text{cov}}(p_{(n_S)})|} A_{\text{cov}}.
\end{aligned}$$

The first rule simply sets up the simulation from s_0. The second rule has the purpose to encode $\mathcal{N}$'s coverability query with the intention that a configuration $A_{\text{cov}} \parallel \Pi \lhd \Gamma$ is only reachable if and only if s_{cov} is coverable.

First, let us note that clearly $\mathcal{G}$ can be constructed from $\mathcal{N}$ in Exptime. Also all non-terminals except for the N_r^O, $N_r^{O,S}$ and N^ϵ are non-commutative. It is easy to see that $\mathcal{G}$ has shaped stacks since the only non-terminal loop increasing the "call-stack" is in the rule

$$N^p \to (c_p \,?\, msg_{p',c}) \cdot (c_p \,!\, \bullet) \cdot N^{p'} N^c$$

and N^c is a commutative non-terminal. It is easy to see that picking $K = 4$ is adequate.

Before we go on to prove R is a weak bisimulation let us first analyse $\mathcal{F}^\Pi$ and $\mathcal{F}^\Gamma$.

It is easy to see that for all $s, s' \in \mathit{Config}$ we have

$$\mathcal{F}^\Pi(s) \parallel \mathcal{F}^\Pi(s') = \mathcal{F}^\Pi(s \oplus s'), \qquad \mathcal{F}^\Gamma(s) \oplus \mathcal{F}^\Gamma(s') = \mathcal{F}^\Gamma(s \oplus s'),$$

if for all $p \in \mathcal{P}^S$ we have $s'(p) = \emptyset$ then

$$\mathcal{F}^\Gamma(s) = \mathcal{F}^\Gamma(s \oplus s') = \mathcal{F}^\Gamma(s \ominus s')$$

and if for all $p \in \mathcal{P}^C$ we have $s'(p) = \emptyset$ then

$$\mathcal{F}^\Pi(s) = \mathcal{F}^\Pi(s \oplus s') = \mathcal{F}^\Pi(s \ominus s')$$

Let us label APCPS and NNCT transitions in the following way: for $s, s' \in \mathit{Config}$ such that $s \to_\mathcal{N} s'$ let us label $\to_\mathcal{N}$ such that we write $s \xrightarrow{s'}_\mathcal{N} s'$; for $\Pi \lhd \Gamma, \Pi' \lhd \Gamma' \in \mathit{State}_{\text{alt}}$ such that $\Pi \lhd \Gamma \to_\mathcal{G} \Pi' \lhd \Gamma'$ and $\exists s_0 \in \mathit{Config}$ such that $\mathcal{F}(s_0) = \Pi' \lhd \Gamma'$ let us label $\to_\mathcal{G}$ such that we write $\Pi \lhd \Gamma \xrightarrow{s_0}_\mathcal{G} \Pi' \lhd \Gamma'$; if however $\nexists s_0 \in \mathit{Config}$ such that $\mathcal{F}(s_0) = s'$ then let us label $\to_\mathcal{G}$ such that we write $\Pi \lhd \Gamma \xrightarrow{\epsilon}_\mathcal{G} \Pi' \lhd \Gamma'$.

Let us clarify some notation: if $\Gamma \in (\mathit{Chan} \to \mathbb{M}[\mathit{Msg}])$ such that for $\Gamma', \Gamma'' \in (\mathit{Chan} \to \mathbb{M}[\mathit{Msg}])$ we can decompose Γ such that $\Gamma = \Gamma' \oplus \Gamma''$ then $\Gamma' = \Gamma \ominus \Gamma''$.

Let us now prove that R above is a weak simulation. So suppose we have $(s, \mathcal{F}(s)) \in R$ and $s \xrightarrow{s_0}{}^*_\mathcal{N} s'$ then from the labelling we clearly have $s \xrightarrow{s'}_\mathcal{N} s'$. Hence there exists a $r \in \mathcal{R}$ such that $s \xrightarrow{r}_\mathcal{N} s'$. Note that we focus on the general case where a complex token is non-empty. Inspecting the rules of $\mathcal{G}$ we can see that the arguments below can easily be adapted to cover the edge case where it is empty. We will perform a case analysis on r.

- *Case:* $r \in \mathit{SimpleRules}$.

Then $r = (I, O)$ and $s' = (s \ominus I) \oplus O$ and also $s \ominus I \in \mathit{Config}$. We know that $\mathcal{F}(s) = N^{sim} \parallel \mathcal{F}^\Pi(s) \lhd \mathcal{F}^\Gamma(s)$ and since $s \ominus I \in \mathit{Config}$ we have that $|s(p_0)| \geq |I(p_0)|$ for all $p_0 \in \mathcal{P}^S$. Hence by definition

$$\mathcal{F}^\Gamma(s) = \Gamma \oplus \left[\bullet^{|I(p_{(1)})|}\right]^{c_{p_{(1)}}} \oplus \cdots \oplus \left[\bullet^{|I(p_{(n_S)})|}\right]^{c_{p_{(n_S)}}}$$

for some $\Gamma \in (\mathit{Chan} \to \mathbb{M}[\mathit{Msg}])$. Hence we can see that we can perform the following transitions:

$$\begin{aligned}
\mathcal{F}(s) &\xrightarrow{\epsilon}_\mathcal{G} N_r N^{sim} \parallel \mathcal{F}^\Pi(s) \lhd \mathcal{F}^\Gamma(s) \xrightarrow{\epsilon}_\mathcal{G} N_r^I N_r^{O,S} N_r^{O,C} N^{sim} \parallel \mathcal{F}^\Pi(s) \lhd \mathcal{F}^\Gamma(s) \\
&\xrightarrow{\epsilon}_\mathcal{G} \left(c_{p_{(1)}} \,?\, \bullet\right)^{|I(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,?\, \bullet\right)^{|I(p_{(n_S)})|} N_r^{O,S} N_r^{O,C} N^{sim} \parallel \mathcal{F}^\Pi(s) \lhd \mathcal{F}^\Gamma(s) \\
&\xrightarrow{\epsilon}_\mathcal{G} \left(c_{p_{(1)}} \,?\, \bullet\right)^{|I(p_{(1)})|-1} \cdots \left(c_{p_{(n_S)}} \,?\, \bullet\right)^{|I(p_{(n_S)})|} N_r^{O,S} N_r^{O,C} N^{sim} \parallel \mathcal{F}^\Pi(s) \lhd \mathcal{F}^\Gamma(s) \ominus [\bullet]^{c_{p_{(1)}}} \\
&\xrightarrow{\epsilon}_\mathcal{G} \cdots
\end{aligned}$$

$$\xrightarrow{\epsilon}_{\mathcal{G}} \left(c_{p_{(2)}} ? \bullet\right)^{|I(p_{(2)})|} \cdots \left(c_{p_{(n_S)}} ? \bullet\right)^{|I(p_{(n_S)})|} N_r^{O,S} N_r^{O,C} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s) \ominus \left[\bullet^{|I(p_{(1)})|}\right]^{c_{p_{(1)}}}$$
$$\xrightarrow{\epsilon}_{\mathcal{G}} \cdots$$
$$\xrightarrow{\epsilon}_{\mathcal{G}} N_r^{O,S} N_r^{O,C} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s) \ominus \left[\bullet^{|I(p_{(1)})|}\right]^{c_{p_{(1)}}} \ominus \left[\bullet^{|I(p_{(2)})|}\right]^{c_{p_{(2)}}} \ominus \cdots \ominus \left[\bullet^{|I(p_{(n_S)})|}\right]^{c_{p_{(n_S)}}}$$
$$= N_r^{O,S} N_r^{O,C} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I)$$

We can expand $N_r^{O,S}$ and perform the send actions similarly to the way we expanded N_r^I and its receive actions.

$$\mathcal{F}(s) \xrightarrow{\epsilon}_{\mathcal{G}}^{*} \left(c_{p_{(1)}} ! \bullet\right)^{|O(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} ! \bullet\right)^{|O(p_{(n_S)})|} N_r^{O,C} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I)$$
$$\xrightarrow{\epsilon}_{\mathcal{G}}^{*} N_r^{O,C} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I) \oplus \left[\bullet^{|O(p_{(1)})|}\right]^{c_{p_{(1)}}} \oplus \cdots \oplus \left[\bullet^{|O(p_{(n_S)})|}\right]^{c_{p_{(n_S)}}}$$
$$= N_r^{O,C} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \restriction \mathcal{P}^S))$$

We continue with expanding $N_r^{O,C}$. Similarly to above we can then perform all the spawns and synchronisations.

$$\mathcal{F}(s) \xrightarrow{\epsilon}_{\mathcal{G}}^{*} \left(\nu N^{\nu p'_{(1)}} \left(c_? ! \, msg_{p'_{(1)}, c_{(1)}}\right) (c_? ? \bullet)\right)^{O(p'_{(1)})(c_{(1)})} \cdots$$
$$\left(\nu N^{\nu p'_{(n_C)}} \left(c_? ! \, msg_{p'_{(n_C)}, c_{(n_\Xi)}}\right) (c_? ? \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}$$
$$N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \restriction \mathcal{P}^S))$$
$$\xrightarrow{\epsilon}_{\mathcal{G}} \left(c_? ! \, msg_{p'_{(1)}, c_{(1)}}\right) (c_? ? \bullet)$$
$$\left(\nu N^{\nu p'_{(1)}} \left(c_? ! \, msg_{p'_{(1)}, c_{(1)}}\right) (c_? ? \bullet)\right)^{O(p'_{(1)})(c_{(1)})-1} \cdots$$
$$\left(\nu N^{\nu p'_{(n_C)}} \left(c_? ! \, msg_{p'_{(n_C)}, c_{(n_\Xi)}}\right) (c_? ? \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}$$
$$N^{sim} \parallel N^{\nu p'_{(1)}} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \restriction \mathcal{P}^S))$$

Using the rule $N^{\nu p'_{(1)}} \to \left(c_? ? \, msg_{p, c_{(1)}}\right) (c_? ! \bullet) \, N^{p'_{(1)}} \cdot N^{c_{(1)}} \cdot N^{\nu ?}$ we can derive

$$N^{c_{(1)}} \to^{*} \left(c_{\zeta\left(p^{col}_{(1)}\right)} ! \bullet\right)^{|c_{(1)}(p^{col}_{(1)})|} \cdots \left(c_{\zeta\left(p^{col}_{(n_{col})}\right)} ! \bullet\right)^{|c_{(1)}(p^{col}_{(n_{col})})|} =: w$$

and the equality $\mathbb{M}(w) = \widetilde{\Gamma}(c_{(1)} \circ \zeta^{-1})$ holds. Thus

$$\mathcal{F}(s) \xrightarrow{\epsilon}_{\mathcal{G}}^{*} \left(c_? ! \, msg_{p'_{(1)}, c_{(1)}}\right) (c_? ? \bullet) \left(\nu N^{\nu p'_{(1)}} \left(c_? ! \, msg_{p'_{(1)}, c_{(1)}}\right) (c_? ? \bullet)\right)^{O(p'_{(1)})(c_{(1)})-1} \cdots$$
$$\left(\nu N^{\nu p'_{(i)}} \left(c_? ! \, msg_{p'_{(i)}, c_{(j)}}\right) (c_? ? \bullet)\right)^{O(p'_{(i)})(c_{(j)})} \cdots$$
$$\left(\nu N^{\nu p'_{(n_C)}} \left(c_? ! \, msg_{p'_{(n_C)}, c_{(n_\Xi)}}\right) (c_? ? \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}$$
$$N^{sim} \parallel \left(c_? ? \, msg_{p, c}\right) (c_? ! \bullet) \, N^{p'_{(1)}} \cdot \widetilde{\Gamma}(c_{(1)} \circ \zeta^{-1}) \cdot N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \restriction \mathcal{P}^S))$$

$$\xrightarrow{\epsilon}_{\mathcal{G}}^{*} \left(\nu N^{\nu p'_{(1)}} \left(c_? \,!\, msg_{p'_{(1)},c_{(1)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(1)})(c_{(1)})-1} \cdots$$
$$\left(\nu N^{\nu p'_{(i)}} \left(c_? \,!\, msg_{p'_{(i)},c_{(j)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(i)})(c_{(j)})} \cdots$$
$$\left(\nu N^{\nu p'_{(n_C)}} \left(c_? \,!\, msg_{p'_{(n_C)},c_{(n_\Xi)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}$$
$$N^{sim} \parallel N^{p'_{(1)}} \cdot \widetilde{\Gamma}(c_{(1)} \circ \zeta^{-1}) \cdot N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \upharpoonright \mathcal{P}^S))$$

we can continue like this:

$$\mathcal{F}(s) \xrightarrow{\epsilon}_{\mathcal{G}}^{*} \left(\nu N^{\nu p'_{(1)}} \left(c_? \,!\, msg_{p'_{(1)},c_{(2)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(1)})(c_{(2)})} \cdots$$
$$\left(\nu N^{\nu p'_{(i)}} \left(c_? \,!\, msg_{p'_{(i)},c_{(j)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(i)})(c_{(j)})} \cdots$$
$$\left(\nu N^{\nu p'_{(n_C)}} \left(c_? \,!\, msg_{p'_{(n_C)},c_{(n_\Xi)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}$$
$$N^{sim} \parallel \Big\|_{i=1}^{O(p'_{(1)}c_{(1)})} N^{p'_{(1)}} \cdot \widetilde{\Gamma}(c_{(1)} \circ \zeta^{-1}) \cdot N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \upharpoonright \mathcal{P}^S))$$

$$\mathcal{F}(s) \xrightarrow{\epsilon}_{\mathcal{G}}^{*} \left(\nu N^{\nu p'_{(2)}} \left(c_? \,!\, msg_{p'_{(2)},c_{(1)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(2)})(c_{(1)})} \cdots$$
$$\left(\nu N^{\nu p'_{(i)}} \left(c_? \,!\, msg_{p'_{(i)},c_{(j)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(i)})(c_{(j)})} \cdots$$
$$\left(\nu N^{\nu p'_{(n_C)}} \left(c_? \,!\, msg_{p'_{(n_C)},c_{(n_\Xi)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}$$
$$N^{sim} \parallel \Big\|_{i=1}^{O(p'_{(1)}c_{(1)})} N^{p'_{(1)}} \cdot \widetilde{\Gamma}(c_{(1)} \circ \zeta^{-1}) \cdot N^{\nu ?} \parallel \cdots$$
$$\parallel \Big\|_{i=1}^{O(p'_{(1)}c_{(n_\Xi)})} N^{p'_{(1)}} \cdot \widetilde{\Gamma}(c_{(n_\Xi)} \circ \zeta^{-1}) \cdot N^{\nu ?}$$
$$\parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \upharpoonright \mathcal{P}^S))$$
$$= \left(\nu N^{\nu p'_{(2)}} \left(c_? \,!\, msg_{p'_{(2)},c_{(1)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(2)})(c_{(1)})} \cdots$$
$$\left(\nu N^{\nu p'_{(i)}} \left(c_? \,!\, msg_{p'_{(i)},c_{(j)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(i)})(c_{(j)})} \cdots$$
$$\left(\nu N^{\nu p'_{(n_C)}} \left(c_? \,!\, msg_{p'_{(n_C)},c_{(n_\Xi)}}\right)(c_? \,?\, \bullet)\right)^{O(p'_{(n_C)})(c_{(n_\Xi)})}$$
$$N^{sim} \parallel \mathcal{F}^{\Pi}(O, p'_{(1)}) \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \upharpoonright \mathcal{P}^S))$$

$$\xrightarrow{\alpha}_{\mathcal{G}}^{*} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \parallel \left(\Big\|_{i=1}^{n_C} \mathcal{F}^{\Pi}(O, p'_{(i)})\right) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \upharpoonright \mathcal{P}^S))$$
$$= N^{sim} \parallel \mathcal{F}^{\Pi}(s) \parallel \mathcal{F}^{\Pi}(O \upharpoonright \mathcal{P}^C) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus (O \upharpoonright \mathcal{P}^S))$$

It just remains to analyse the last configuration:

$$\mathcal{F}^{\Pi}(s) \parallel \mathcal{F}^{\Pi}(O \restriction \mathcal{P}^{C}) = \mathcal{F}^{\Pi}(s) \parallel \mathcal{F}^{\Pi}(O) = \mathcal{F}^{\Pi}(s \oplus O) = \mathcal{F}^{\Pi}(s \ominus I \oplus O) = \mathcal{F}^{\Pi}(s')$$

and

$$\mathcal{F}^{\Gamma}(s \ominus I \oplus (O \restriction \mathcal{P}^{S})) = \mathcal{F}^{\Gamma}(s \ominus I \oplus O) = \mathcal{F}^{\Gamma}(s')$$

from which we can deduce $\mathcal{F}(s) \xrightarrow{\alpha}{}^{*}_{\mathcal{G}} \mathcal{F}(s')$, $(s', \mathcal{F}(s')) \in R$ and clearly $\alpha = s'$ which is what we wanted to prove.

- *Case:* $r \in \mathit{ComplexRules}$.

Then $r = ((p, I), (p', \oplus, O))$ and $s' = (s \ominus I \ominus [p \mapsto [m]]) \oplus O \oplus [p' \mapsto [m \oplus c]]$ for some $m \in s(p)$. Again $\mathcal{F}(s) = N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s)$ and similarly to the case above

$$\begin{aligned}
\mathcal{F}(s) &\xrightarrow{\epsilon}{}^{*}_{\mathcal{G}} N^{C}_{r} N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&\xrightarrow{\epsilon}_{\mathcal{G}} (c_p \,!\, msg_{p',c}) \cdot (c_p \,?\, \bullet) N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&\xrightarrow{\epsilon}_{\mathcal{G}} (c_p \,?\, \bullet) N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \oplus \left[msg_{p',c} \right]^{c_p} \\
&= (c_p ? \bullet) N^{sim} \parallel \mathcal{F}^{\Pi}([p \mapsto [m]]) \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \oplus \left[msg_{p',c} \right]^{c_p} \\
&= (c_p ? \bullet) N^{sim} \parallel N^{p} \widetilde{\Gamma}\left(m \circ \zeta^{-1}\right) N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \oplus \left[msg_{p',c} \right]^{c_p}
\end{aligned}$$

It is immediate that we can derive

$$N^{c} \to^{*} \left(c_{\zeta\left(p^{col}_{(1)}\right)} \,!\, \bullet \right)^{|c(p^{col}_{(1)})|} \cdots \left(c_{\zeta\left(p^{col}_{(n_{col})}\right)} \,!\, \bullet \right)^{|c(p^{col}_{(n_{col})})|} =: w$$

and that the equality $\mathbb{M}(w) = \widetilde{\Gamma}(c \circ \zeta^{-1})$ holds. Thus

$$\begin{aligned}
\mathcal{F}(s) &\xrightarrow{\epsilon}{}^{*}_{\mathcal{G}} (c_p \,?\, \bullet) N^{sim} \parallel (c_p \,?\, msg_{p',c}) \cdot (c_p \,!\, \bullet) \cdot N^{p'} \widetilde{\Gamma}\left(m \circ \zeta^{-1}\right) \oplus \widetilde{\Gamma}\left(c \circ \zeta^{-1}\right) N^{\nu ?} \\
&\qquad \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \oplus \left[msg_{p',c} \right]^{c_p} \\
&= (c_p \,?\, \bullet) N^{sim} \parallel (c_p \,?\, msg_{p',c}) \cdot (c_p \,!\, \bullet) \cdot N^{p'} \widetilde{\Gamma}\left((m \oplus c) \circ \zeta^{-1}\right) N^{\nu ?} \\
&\qquad \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \oplus \left[msg_{p',c} \right]^{c_p} \\
&\xrightarrow{\alpha}{}^{*}_{\mathcal{G}} N^{sim} \parallel N^{p'} \left((m \oplus c) \circ \zeta^{-1} \circ \widetilde{\Gamma} \right) N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \\
&\qquad \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&= N^{sim} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]] \oplus \left[p' \mapsto [m \oplus c]\right]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O)
\end{aligned}$$

Lastly, we clearly have

$$\begin{aligned}
\mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]] \oplus \left[p' \mapsto [m \oplus c]\right]) &= \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]] \ominus I \oplus \left[p' \mapsto [m \oplus c]\right] \oplus O) \\
&= \mathcal{F}^{\Pi}(s') \\
\mathcal{F}^{\Gamma}(s \ominus I \oplus O) = \mathcal{F}^{\Gamma}(s \ominus [p \mapsto [m]] \ominus I \oplus \left[p' \mapsto [m \oplus c]\right] \oplus O) &= \mathcal{F}^{\Gamma}(s')
\end{aligned}$$

from which we can deduce $\mathcal{F}(s) \xrightarrow{\alpha}{}^{*}_{\mathcal{G}} \mathcal{F}(s')$, $(s', \mathcal{F}(s')) \in R$ and clearly $\alpha = s'$ which is what we wanted to prove.

- $r \in \mathit{TransferRules}$.
Then $r = ((p, I), (p', \mathcal{P}^{col}, O))$ since r is total and $s' = (s \ominus I \ominus [p \mapsto [m]]) \oplus O \oplus [p' \mapsto [\emptyset]] \oplus (m \circ \zeta^{-1})$ for some $m \in s(p)$. Analogously to the cases above:

$$\begin{aligned}
\mathcal{F}(s) &\xrightarrow{\epsilon}{}^{*}_{\mathcal{G}} N_r^C N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&\xrightarrow{\epsilon}_{\mathcal{G}} (c_p \,!\, msg_{p,\downarrow}) \cdot (c_p \,?\, \bullet) \cdot (c_? \,!\, msg_{p',\emptyset}) \cdot (c_? \,?\, \bullet) N^{sim} \parallel \mathcal{F}^{\Pi}(s) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&\xrightarrow{\epsilon}{}^{*}_{\mathcal{G}} (c_p \,!\, msg_{p,\downarrow}) \cdot (c_p \,?\, \bullet) \cdot (c_? \,!\, msg_{p',\emptyset}) \cdot (c_? \,?\, \bullet) N^{sim} \parallel N^{p} \widetilde{\Gamma}\left(m \circ \zeta^{-1}\right) N^{\nu ?} \\
&\qquad \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&\xrightarrow{\epsilon}_{\mathcal{G}} (c_p ! msg_{p,\downarrow}) \cdot (c_p ? \bullet) \cdot (c_? ! msg_{p',\emptyset}) \cdot (c_? ? \bullet) N^{sim} \parallel (c_p ? msg_{p',\downarrow}) \cdot (c_p ! \bullet) \widetilde{\Gamma}\left(m \circ \zeta^{-1}\right) N^{\nu ?} \\
&\qquad \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&\xrightarrow{\epsilon}{}^{*}_{\mathcal{G}} (c_? \,!\, msg_{p',\emptyset}) \cdot (c_? \,?\, \bullet) N^{sim} \parallel \widetilde{\Gamma}\left(m \circ \zeta^{-1}\right) N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \\
&\xrightarrow{\epsilon}_{\mathcal{G}} (c_? \,!\, msg_{p',\emptyset}) \cdot (c_? \,?\, \bullet) N^{sim} \parallel N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \\
&\qquad \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O) \oplus \Gamma\left(\widetilde{\Gamma}(m \circ \zeta^{-1})\right)
\end{aligned}$$

We can see that

$$\mathcal{F}^{\Gamma}(s \ominus I \oplus O) \oplus \Gamma\left(\widetilde{\Gamma}(m \circ \zeta^{-1})\right) = \mathcal{F}^{\Gamma}(s \ominus I \oplus O \oplus m \circ \zeta^{-1})$$

and so

$$\begin{aligned}
\mathcal{F}(s) &\xrightarrow{\epsilon}{}^{*}_{\mathcal{G}} (c_? \,!\, msg_{p',\emptyset}) \cdot (c_? \,?\, \bullet) N^{sim} \parallel (c_? \,?\, msg_{p,\emptyset}) \cdot (c_? \,!\, \bullet) \cdot N_{\emptyset}^{p'} \cdot N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \\
&\qquad \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O \oplus (m \circ \zeta^{-1})) \\
&\xrightarrow{\alpha}{}^{*}_{\mathcal{G}} N^{sim} \parallel N_{\emptyset}^{p'} \cdot N^{\nu ?} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O \oplus (m \circ \zeta^{-1})) \\
&= N^{sim} \parallel \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]] \oplus \left[p' \mapsto [\emptyset]\right]) \lhd \mathcal{F}^{\Gamma}(s \ominus I \oplus O \oplus (m \circ \zeta^{-1}))
\end{aligned}$$

Analysing the last configuration:

$$\begin{aligned}
\mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]] \oplus \left[p' \mapsto [\emptyset]\right]) &= \mathcal{F}^{\Pi}(s \ominus [p \mapsto [m]] \ominus I \oplus \left[p' \mapsto [\emptyset]\right] \oplus O \oplus (m \circ \zeta^{-1})) \\
&= \mathcal{F}^{\Pi}(s') \\
\mathcal{F}^{\Gamma}(s \ominus I \oplus O \oplus (m \circ \zeta^{-1})) &= \mathcal{F}^{\Gamma}(s \ominus [p \mapsto [m]] \ominus I \oplus \left[p' \mapsto [\emptyset]\right] \oplus O \oplus (m \circ \zeta^{-1})) \\
&= \mathcal{F}^{\Gamma}(s')
\end{aligned}$$

from which we can deduce $\mathcal{F}(s) \xrightarrow{\alpha}{}^{*}_{\mathcal{G}} \mathcal{F}(s')$, $(s', \mathcal{F}(s')) \in R$ and clearly $\alpha = s'$ which is what we wanted to prove.
Hence we can conclude that R is a weak simulation.

Let us now investigate R^{-1}. Suppose we have $(s, \mathcal{F}(s)) \in R$ and $F(s) \xrightarrow{s'}{}^{*}_{\mathcal{G}} t$ then from the labelling we know we have $\mathcal{F}(s) \xrightarrow{s'}{}^{*}_{\mathcal{G}} \mathcal{F}(s') \xrightarrow{\epsilon}{}^{*}_{\mathcal{G}} t$. Inspecting the rules of $\mathcal{G}$ we can see that the only paths (modulo different interleavings) to make the transitions $\mathcal{F}(s) \xrightarrow{s'}{}^{*}_{\mathcal{G}} \mathcal{F}(s')$ are the three described in the proof that R is a simulation and depends on which rule r of $\mathcal{N}$ is simulated.

Inspecting the transition paths we can see that if $r \in$ *SimpleRules* then $s' = s \ominus I \oplus O$ and since N_r^I can be fully executed we can deduce that $|s(p)| \geq |I(p)|$ for all $p \in \mathcal{P}^S$ and hence $s \ominus I \in$ *Config*. Thus we can conclude that $s \xrightarrow{r}_{\mathcal{N}} s'$ and hence $s \xrightarrow{s'}_{\mathcal{N}} s'$.

If $r \in$ *ComplexRules* then we can see that $(s \ominus I \ominus [p \mapsto [m]]) \oplus O \oplus [p' \mapsto [m \oplus c]]$ for some $m \in s(p)$ and as above since N_r^I can be fully executed we can deduce that $s \ominus I \in$ *Config* which implies $s \xrightarrow{r}_{\mathcal{N}} s'$ and hence $s \xrightarrow{s'}_{\mathcal{N}} s'$.

For the third case if $r \in$ *ComplexRules* inspecting the transition path above we can see that $s' = s \ominus [p \mapsto [m]] \ominus I \oplus [p' \mapsto [\emptyset]] \oplus O \oplus (m \circ \zeta^{-1})$ for some $m \in s(p)$ as above since N_r^I can be fully executed we can deduce that $s \ominus I \in$ *Config*. Hence $s \xrightarrow{r}_{\mathcal{N}} s'$ and hence $s \xrightarrow{s'}_{\mathcal{N}} s'$.

Thus we can conclude that R^{-1} is a weak simulation and thus R is a weak bisimulation.

In order to finish the proof we will now prove that a check of coverability of s_{cov} for $\mathcal{N}$ is equivalent to a program-point coverability check of l_{cov} for $\mathcal{G}$.

First suppose $(\mathcal{N}, s_0, s_{\text{cov}})$ is a yes-instance of NNCT coverability with a simple query. Hence there exists a path $s_0 \to_{\mathcal{N}} s_1 \to_{\mathcal{N}} \cdots \to_{\mathcal{N}} s_n$ and $s_{\text{cov}} \leq_{\textit{Config}} s_n$. Attaching the labels to the transition system as we have described above this path turns into $s_0 \xrightarrow{s_1}_{\mathcal{N}} s_1 \xrightarrow{s_2}_{\mathcal{N}} \cdots \xrightarrow{s_n}_{\mathcal{N}} s_n$. Since R is a weak bisimulation we know that we can find a path $S \xrightarrow{s_0}{}^*_{\mathcal{G}} \mathcal{F}(s_0) \xrightarrow{s_1}{}^*_{\mathcal{G}} \mathcal{F}(s_1) \xrightarrow{s_2}{}^*_{\mathcal{G}} \cdots \xrightarrow{s_n}{}^*_{\mathcal{G}} \mathcal{F}(s_n)$. We know that for all $p \in \mathcal{P}^S$ it is the case that $|s_{\text{cov}}(p)| \leq |s_n(p)|$ hence $s_n \ominus (s_{\text{cov}} \restriction \mathcal{P}^S) \in \textit{State}_{\text{alt}}$. Thus

$$
\begin{aligned}
\mathcal{F}(s_n) &= N^{\textit{sim}} \parallel \mathcal{F}^{\Pi}(s_n) \lhd \mathcal{F}^{\Gamma}(s_n) \\
&\xrightarrow{\epsilon}_{\mathcal{G}} \left(c_{p_{(1)}} \,?\, \bullet\right)^{|s_{\text{cov}}(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,?\, \bullet\right)^{|s_{\text{cov}}(p_{(n_S)})|} l_{\text{cov}} \parallel \mathcal{F}^{\Pi}(s_n) \lhd \mathcal{F}^{\Gamma}(s_n) \\
&\xrightarrow{\epsilon}{}^*_{\mathcal{G}} l_{\text{cov}} \parallel \mathcal{F}^{\Pi}(s_n) \lhd \mathcal{F}^{\Gamma}(s_n) \ominus \left[\bullet^{|s_{\text{cov}}(p_{(1)})|}\right]^{c_{p_{(1)}}} \ominus \cdots \ominus \left[\bullet^{|s_{\text{cov}}(p_{(n_S)})|}\right]^{c_{p_{(n_S)}}} \\
&= A_{\text{cov}} \parallel \mathcal{F}^{\Pi}(s_n) \lhd \mathcal{F}^{\Gamma}(s_n \ominus (s_{\text{cov}} \restriction \mathcal{P}^S))
\end{aligned}
$$

Hence $(\mathcal{G}, S \lhd \emptyset, A_{\text{cov}} \lhd \emptyset)$ is a yes-instance of alternative simple coverability and thus the query $(\mathcal{G}, S \lhd \emptyset, A_{\text{cov}} \lhd \emptyset)$ is a yes-instance of standard simple coverability.

Suppose $(\mathcal{G}, S \lhd \emptyset, A_{\text{cov}} \lhd \emptyset)$ is a yes-instance of standard simple coverability. Hence we know that $(\mathcal{G}, S \lhd \emptyset, A_{\text{cov}} \lhd \emptyset)$ is a yes-instance of alternative simple coverability. Hence $S \to^*_{\mathcal{G}} A_{\text{cov}}\alpha \parallel \Pi \lhd \Gamma$ for some α, Π and Γ. Let us view this path in the labelled transition system we can then obtain: $S \xrightarrow{s_0}{}^*_{\mathcal{G}} \mathcal{F}(s_0) \xrightarrow{s_1}{}^*_{\mathcal{G}} \mathcal{F}(s_1) \xrightarrow{s_2}{}^*_{\mathcal{G}} \cdots \xrightarrow{s_n}{}^*_{\mathcal{G}} \mathcal{F}(s_n) \xrightarrow{\epsilon}{}^*_{\mathcal{G}} A_{\text{cov}}\alpha \parallel \Pi \lhd \Gamma$ where it is clear that the first simulation state set up by S can only by s_0.

We know that $\mathcal{F}(s_n) = N^{\textit{sim}} \parallel \mathcal{F}^{\Pi}(s_n) \lhd \mathcal{F}^{\Gamma}(s_n)$. Since on the sequential level it must be the case that $N^{\textit{sim}} \to^* \beta A_{\text{cov}}\alpha$, but inspecting the rules of $\mathcal{G}$ we can see that only $N^{\textit{sim}} \to^* \left(c_{p_{(1)}} \,?\, \bullet\right)^{|s_{\text{cov}}(p_{(1)})|} \cdots \left(c_{p_{(n_S)}} \,?\, \bullet\right)^{|s_{\text{cov}}(p_{(n_S)})|} A_{\text{cov}}$ we can conclude that $\alpha = \epsilon$. Further, along these transition no administrative messages to channels $c_?$, c_p for $p \in \mathcal{P}^C$ are sent and thus any process in $\mathcal{F}^{\Pi}(s_n)$ is blocked on its initial receive action and thus cannot receive or send any messages to a c_p where $p \in \mathcal{P}^S$. Hence it must be the case that

$$
\Gamma = \mathcal{F}^{\Gamma}(s_n) \ominus \left[\bullet^{|s_{\text{cov}}(p_{(1)})|}\right]^{c_{p_{(1)}}} \ominus \cdots \ominus \left[\bullet^{|s_{\text{cov}}(p_{(n_S)})|}\right]^{c_{p_{(n_S)}}} = \mathcal{F}^{\Gamma}(s_n \ominus s_{\text{cov}})
$$

since $s_{\text{cov}}(p) = \emptyset$ for all $p \in \mathcal{P}^C$. We can thus conclude that $s_{\text{cov}} \leq_{\mathit{Config}} s_n$. Since R is a weak bisimulation we know that there is a path $s_0 \xrightarrow{s_1}{}^*_{\mathcal{N}} s_1 \xrightarrow{s_2}_{\mathcal{N}} \cdots \xrightarrow{s_n}{}^*_{\mathcal{N}} s_n$ and hence $s_0 \to^*_{\mathcal{N}} s_n$ and $s_{\text{cov}} \leq_{\mathit{Config}} s_n$ thus $(\mathcal{N}, s_0, s_{\text{cov}})$ is a yes-instance of coverability with a simple query.

For boundedness, suppose the set $\{\Pi \lhd \Gamma : S \lhd \emptyset \to^*_{\mathcal{G}} \Pi \lhd \Gamma\}$ is finite. This implies that the set $\{\mathcal{F}(s) : \mathcal{F}(s_0) \to^*_{\mathcal{G}} \mathcal{F}(s)\}$ is finite and since R is a weak bisimulation this means that $\{s : s_0 \lhd \lhd \to^*_{\mathcal{N}} s\}$ is finite. Conversely, suppose $\{s : s_0 \lhd \lhd \to^*_{\mathcal{N}} s\}$ is finite then clearly $\{\mathcal{F}(s) : \mathcal{F}(s_0) \to^*_{\mathcal{G}} \mathcal{F}(s)\}$ is finite. Since in $\mathcal{G}$ there are only finitely many more steps along weak bisimulation steps we can also conclude that $\{\Pi \lhd \Gamma : S \lhd \emptyset \to^*_{\mathcal{G}} \Pi \lhd \Gamma\}$ is finite. Hence $\mathcal{G}$ is bounded from from $S \lhd \emptyset$ if, and only if, $\mathcal{N}$ is bounded from s_0.

For termination, suppose there is an infinite path from $S \lhd \emptyset$ in $\mathcal{G}$, then since there are only finitely many ϵ steps along weak bisimulation steps and R is a weak bisimulation we can conclude that $\mathcal{N}$ has an infinite path from s_0. Conversely, if $\mathcal{N}$ has an infinite path from s_0 then there is an infinite path from $S \lhd \emptyset$ in $\mathcal{G}$ since R is a weak bisimulation and every step . $\mathcal{N}$ is bounded from s_0. Hence $\mathcal{G}$ is terminating from from $S \lhd \emptyset$ if, and only if, $\mathcal{N}$ is terminating from s_0. □

C.3 Proofs for Section 5.3

Lemma 7. Suppose $\boldsymbol{s}$ is a covering path for s_{cov} in $\mathcal{N}_{i+1}$ and $|\boldsymbol{s}(1)(p_{(i+1)})| \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$. Then there exists $\boldsymbol{s}'$ a covering path for s_{cov} in $\mathcal{N}_{i+1}$ such that $\boldsymbol{s}'(1) = \boldsymbol{s}(1)$ and $|\boldsymbol{s}'| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Proof. Let $m_\infty = [\bullet \mapsto \infty]$, let $s = \boldsymbol{s}(1)$ and define $\boldsymbol{s}_\infty(i) = \boldsymbol{s}(j)\left[p_{(i+1)} \mapsto m_\infty\right]$ for all $1 \leq j \leq |\boldsymbol{s}|$. Since $\boldsymbol{s}$ is covering path for s_{cov} in $\mathcal{N}_{i+1}$, we can easily see that $\boldsymbol{s}_\infty$ is a covering path for s_{cov} in $\mathcal{N}_i$. Hence there must exists a covering path $\boldsymbol{s}'_\infty$ for s_{cov} in $\mathcal{N}_i$ such that $|\boldsymbol{s}'_\infty| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$. We can "replay" the path $\boldsymbol{s}'_\infty$ from s instead of $s\left[p_{(i+1)} \mapsto m_\infty\right]$ to obtain a configuration sequence $\boldsymbol{s}_0$ from s so that $\boldsymbol{s}_0(j)(k) = \boldsymbol{s}'_\infty(j)(k)$ for all $1 \leq j \leq |\boldsymbol{s}'_\infty|$ and $k \neq i+1$. An easy induction shows that for all $1 \leq j \leq |\boldsymbol{s}'_\infty|$ we have $|\boldsymbol{s}_0(j)(i+1)| \geq R' \times \rho_{\mathcal{N}_i}(s_{\text{cov}}) - j \times R$ from which we can conclude

$$|\boldsymbol{s}_0(j)(i+1)| \geq R' - R + (R' - R) \times (\rho_{\mathcal{N}_i}(s_{\text{cov}}) - 1) \geq R' - R \geq s_{\text{cov}}(p_{(i+1)}).$$

since $\rho_{\mathcal{N}_i}(s_{\text{cov}}) \geq 1$ for all i. Hence we can conclude that $\boldsymbol{s}_0$ is a path in $\mathcal{N}_{i+1}$, further $\boldsymbol{s}_0$ is a path from s covering s_{cov} in $\mathcal{N}_{i+1}$ and by construction $|\boldsymbol{s}_0| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$. which is what we wanted to prove. □

Lemma 8. For all covering paths $\boldsymbol{s}$ for s_{cov} in $\mathcal{N}_{i+1}$ one of two cases applies:
(C$_1$) $\|\boldsymbol{s}\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$; or
(C$_2$) $\boldsymbol{s} = \boldsymbol{s}_1 \cdot s_p \cdot \boldsymbol{s}_2$ such that $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$ and $\|s_p\|_{\mathcal{N}_{i+1}} \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Proof. Let $\boldsymbol{s}$ be a covering path $\boldsymbol{s}$ for s_{cov}. Let us do a case analysis:

(i) *Case 1:* $\|\boldsymbol{s}\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{cov})$.
Case (C$_1$) holds.

(ii) *Case 2:* $\|\boldsymbol{s}\|^*_{\mathcal{N}_{i+1}} \not< R' \cdot \rho_{\mathcal{N}_i}(s_{cov})$.
Hence we can split $\boldsymbol{s}$ in two paths $\boldsymbol{s} = \boldsymbol{s}_1\boldsymbol{s}_0$ such that $\boldsymbol{s}_0$ is the longest prefix of $\boldsymbol{s}$ such that $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$ and $\|\boldsymbol{s}_0(1)\|_{\mathcal{N}_{i+1}} \not< R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$, i.e. $\|\boldsymbol{s}_0(1)\|_{\mathcal{N}_{i+1}} \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$. Letting $s_p = \boldsymbol{s}_0(1)$ and $\boldsymbol{s}_2 = \boldsymbol{s}_0(2) \cdots \boldsymbol{s}_0(|\boldsymbol{s}_0|)$ means (C$_2$) applies.

□

Lemma 9. Suppose $\boldsymbol{s}$ is a covering path for s_{cov} in $\mathcal{N}_{i+1}$. If $L \in \mathbb{N}$ such that $|\boldsymbol{s}| \leq L$ and $\|\boldsymbol{s}(1)\|_{\mathcal{N}_{i+1}} < L \cdot R'$ then there exists a covering path $\boldsymbol{s}''$ for s_{cov} in $\mathcal{N}_{i+1}$ such that $\boldsymbol{s}''(1) \leq_{\mathit{Config}} \boldsymbol{s}(1)$, $\|\boldsymbol{s}''(1)\|_{\mathcal{N}_{i+1};C} < L \cdot R'$, and $|\boldsymbol{s}''| \leq L$.

Proof. We will construct a path removing all superfluous complex tokens first. Suppose we label every complex token in $\boldsymbol{s}(1)$ as "not-moved" and along $\boldsymbol{s}$ if a complex token is moved we change the label to "moved". Since $|\boldsymbol{s}| \leq L$ and each transition can move at most one complex token, and remove R complex empty token it must be the case that for each $p \in \mathcal{P}^C$ the size of the multiset $M_{\text{remove}} = |\,[m \text{ "not-moved"} \mid m \in \boldsymbol{s}(|\boldsymbol{s}|)(p)]\,| \geq |[m \text{ "not-moved"} \mid m \in \boldsymbol{s}(1)(p)]| - R \times L$. Hence the complex tokens in M_{remove} play no rôle in $\boldsymbol{s}$, appear in $\boldsymbol{s}(j)(p)$ for all $1 \leq j \leq |\boldsymbol{s}|$ and could thus safely be remove without affecting the validity of the path. However, in $\boldsymbol{s}(|\boldsymbol{s}|)(p)$

there are complex tokens, at most $\|s_{\text{cov}}\|_{\mathcal{N}_{ns};C}$ many, that are required to justify $s_{\text{cov}} \leq_{\mathit{Config}} \boldsymbol{s}(|\boldsymbol{s}|)$ and hence we cannot remove these tokens without losing the cover of s_{cov}. We can be on the safe side an assume the former tokens are all "not-moved" tokens and hence we may only remove $|\boldsymbol{s}(1)(p)| - (R \times L + \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C})$ tokens. We can do this for all $p \in \mathcal{P}^C$ and hence we get a new path $\boldsymbol{s}''_0$ which is a covering path for s_{cov} in $\mathcal{N}_{i+1}$, $|\boldsymbol{s}''_0| \leq L$ and $|\boldsymbol{s}''_0(1)(p')| < R \times L + \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C}$ for all $p' \in \mathcal{P}^C$ and since $\boldsymbol{s}''_0(1)$ is obtained from $\boldsymbol{s}(1)$ by removing tokens we can see that $\boldsymbol{s}''_0(1) \leq_{\mathit{Config}} \boldsymbol{s}(1)$ and hence also $\|\boldsymbol{s}''_0(1)\|_{\mathcal{N}_{i+1}} < (L+1) \cdot R'$.

Let us now show that we can also reduce the number of coloured tokens. We will label coloured tokens in $\boldsymbol{s}''_0(1)$ with three statuses *untouched*, *ejected* and *consumed*. Initially this record of $\boldsymbol{s}''_0(1)$'s coloured tokens marks all of them as *untouched*. Along each transition of $\boldsymbol{s}''_0$ we update this record as follows. If the transition $\boldsymbol{s}''_0(j) \rightarrow_{\mathcal{N}_{i+1}} \boldsymbol{s}''_0(j+1)$ is a transfer transition then we look at all the coloured tokens that are ejected in this transition to simple places and label the coloured tokens whose origin lies in $\boldsymbol{s}''_0(1)$ as *ejected*. If in the transition $\boldsymbol{s}''_0(j) \rightarrow_{\mathcal{N}_{i+1}} \boldsymbol{s}''_0(j+1)$ a simple token is removed and its origin is as a coloured token in $\boldsymbol{s}''_0(1)$ then we mark that coloured token in $\boldsymbol{s}''_0(1)$ as *consumed*. Since $|\boldsymbol{s}| \leq L$ and each transition can remove at most R simple tokens it must be the case that at most $R \times L$ coloured tokens were marked as *consumed* by the above process.

However, for each $p' \in \mathcal{P}^C$ and $p^{col} \in \mathcal{P}^{col}$ there are coloured tokens in $\boldsymbol{s}''_0(|\boldsymbol{s}''_0|)(p')(p^{col})$, at most $\|s_{\text{cov}}\|_{\mathcal{N}_{ns};C}$ many, that are required to justify $s_{\text{cov}} \leq_{\mathit{Config}} \boldsymbol{s}''_0(|\boldsymbol{s}''_0|)$ and hence we cannot remove these tokens without losing the cover of s_{cov}. As above it is safe to label these coloured tokens *consumed* in $\boldsymbol{s}''_0(1)$. Hence for each $p' \in \mathcal{P}^C$ and $p^{col} \in \mathcal{P}^{col}$ there are at most $R \times L + \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C}$ that are labelled *consumed* in $\boldsymbol{s}''_0(1)$.

It should be clear that we can remove all coloured tokens labelled *untouched* along $\boldsymbol{s}''_0$ without affecting the validity of the path. We can also remove coloured tokens that are labelled *ejected* along $\boldsymbol{s}''_0$ since the transfer transition may still fire, however, fewer tokens will be transferred. Since these coloured tokens are not labelled as consumed it is clear that their absence as simple tokens cannot disable a rule that is fired along $\boldsymbol{s}''_0$. For a rule r that removes an empty complex token m we can see that if m is empty along $\boldsymbol{s}''_0$ then removing coloured tokens from $\boldsymbol{s}''_0(1)$ could only possibly lead to less tokens inside m, hence it is impossible to disable r. Thus we can obtain a new covering path $\boldsymbol{s}''$ for s_{cov} in $\mathcal{N}_{i+1}$, such that $|\boldsymbol{s}''| \leq L$ and $|\boldsymbol{s}''(1)(p')| < R \times L + \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C}$ for all $p' \in \mathcal{P}^C$ and $|\boldsymbol{s}''(1)(p')(p^{col})| < R \times L + \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C}$ for all $p' \in \mathcal{P}^C$, for all $p^{col} \in \mathcal{P}^{col}$ since $\boldsymbol{s}''(1)$ is obtained from $\boldsymbol{s}(1)$ by removing tokens we can see that $\boldsymbol{s}''(1) \leq_{\mathit{Config}} \boldsymbol{s}(1)$ and hence also $\|\boldsymbol{s}''_0(1)\|_{\mathcal{N}_{i+1}} < L \cdot R'$.

Since

$$\begin{aligned} R \times L + \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C} &\leq R + R \times (L-1) + \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C} + 1 \\ &= R \times (L-1) + R' \\ &\leq R' \times (L-1) + R' \\ &= R' \times L \end{aligned}$$

we can see that the above implies that $\|\boldsymbol{s}''_0(1)\|_{\mathcal{N}_{i+1};C} < L \cdot R'$ which is what we wanted to prove. □

Corollary 5. For all covering paths $\boldsymbol{s}$ for s_{cov} in $\mathcal{N}_{i+1}$ either

(C_1') $\|\boldsymbol{s}\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$; or
(C_2') there exist paths $\boldsymbol{s}_1$ and $s_{p'} \cdot \boldsymbol{s}_2$ such that $\boldsymbol{s}_1$ is a covering path for $s_{p'}$, $\boldsymbol{s}_1(1) = \boldsymbol{s}(1)$ and $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$; $s_{p'} \cdot \boldsymbol{s}_2$ is a covering path for s_{cov} and $|\boldsymbol{s}_2'| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$; and $\left\|s_{p'}\right\|_{\mathcal{N}_{i+1};C} \leq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Proof. Let $\boldsymbol{s}$ be a covering path $\boldsymbol{s}$ for s_{cov} in $\mathcal{N}_{i+1}$. According to Lemma 8 we can consider the following two cases:

(i) $\|\boldsymbol{s}\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.
Clearly case (C_1') applies.

(ii) $\boldsymbol{s} = \boldsymbol{s}_1 \cdot s_p \cdot \boldsymbol{s}_2$ such that $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$ and $\|s_p\|_{\mathcal{N}_{i+1}} \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.
Since $\|s_p\|_{\mathcal{N}_{i+1}} \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$ we can assume w.l.o.g. that $|s_p(p_{(i+1)})| \geq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$ (we can always change the enumeration $\mathcal{P}^S$'s elements). Lemma 7 applies to $s_p \cdot \boldsymbol{s}_2$ and gives us a covering path $\boldsymbol{s}_2'$ for s_{cov} from s_p in $\mathcal{N}_{i+1}$ such that $|\boldsymbol{s}_2'| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$.
Let $s_0 = \boldsymbol{s}_1(|\boldsymbol{s}_1|)$. Clearly $s_0 \cdot \boldsymbol{s}_2'$ is a covering path for s_{cov} in $\mathcal{N}_{i+1}$, with length $|s_0 \cdot \boldsymbol{s}_2'| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}}) + 1$, and $\|s_0\|_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$ since $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.
We are thus able to apply Lemma 9 to $s_0 \cdot \boldsymbol{s}_2'$ which yields a new covering path $s_{p'} \cdot \boldsymbol{s}_2''$ for s_{cov} in $\mathcal{N}_{i+1}$ such that $s_{p'} \leq_{\mathit{Config}} s_0$, $\left\|s_{p'}\right\|_{\mathcal{N}_{i+1};C} < R' \cdot (\rho_{\mathcal{N}_i}(s_{\text{cov}}) + 1)$, and $|s_{p'} \cdot \boldsymbol{s}_2''| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}}) + 1$.
We can then conclude that $\boldsymbol{s}_1$ is a covering path in $\mathcal{N}_{i+1}$ for $s_{p'}$, $\boldsymbol{s}_1(1) = \boldsymbol{s}(1)$, and $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$. Further $s_{p'} \cdot \boldsymbol{s}_2''$ is a covering path for s_{cov} in $\mathcal{N}_{i+1}$, $|\boldsymbol{s}_2''| \leq \rho_{\mathcal{N}_i}(s_{\text{cov}})$, and $\left\|s_{p'}\right\|_{\mathcal{N}_{i+1};C} \leq R' \cdot \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

□

Proposition 6. (i) $\rho_{\mathcal{N}_0}(s_{\text{cov}}) \leq d_{\mathcal{N}_0}\left(S_{(0,R')}\right)$ and (ii) $\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) \leq d_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right) + \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Proof. In order to prove (i) let $\boldsymbol{s}$ be a covering path for s_{cov} in $\mathcal{N}_0$. Since all simple places are ignored it is trivial to see that $\|\boldsymbol{s}\|^*_{\mathcal{N}_0} < 1 \leq R'$. Further $\|s_{\text{cov}}\|_{\mathcal{N}_0;C} \leq \|s_{\text{cov}}\|_{\mathcal{N}_{ns};C} \leq R'$. Hence we have $\boldsymbol{s}(1), s_{\text{cov}} \in S_{(0,R')}$ which implies we can find a path $\boldsymbol{s}'$ from $\boldsymbol{s}(1)$ covering s_{cov} in $\mathcal{N}_0$ with $|\boldsymbol{s}'| \leq d_{\mathcal{N}_0}\left(S_{(0,R')}\right)$. Hence $\mathit{dist}_{\mathcal{N}_0}(\boldsymbol{s}(1), s_{\text{cov}}) \leq d_{\mathcal{N}_0}\left(S_{(0,R')}\right)$. Since $\boldsymbol{s}$ was arbitrary we can conclude that $\rho_{\mathcal{N}_0}(s_{\text{cov}}) \leq d_{\mathcal{N}_0}\left(S_{(0,R')}\right)$.

For claim (ii) let s be an element of Config^∞ such that $\|s\|_{\mathcal{N}_{i+1}} < \infty$. We want to show that

$$\mathit{dist}_{\mathcal{N}_{i+1}}(s, s_{\text{cov}}) \leq d_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right) + \rho_{\mathcal{N}_i}(s_{\text{cov}}).$$

Let us do a case analysis:

(i) *Case 1: There is no covering path for s_{cov} from s in $\mathcal{N}_{i+1}$.*
Then $\mathit{dist}_{\mathcal{N}_{i+1}}(s, s_{\text{cov}}) = 0$ by definition and the inequality holds trivially.

(ii) *Case 2: There is a covering path $\boldsymbol{s}$ for s_{cov} from s in $\mathcal{N}_{i+1}$.*
Corollary 5 allows us to consider two cases:

Case (A): $\|\boldsymbol{s}'\|^*_{\mathcal{N}_{i+1}} < B_i$.
In this case it is easy to see that $\|s_{\text{cov}}\|_{\mathcal{N}_i+1;C} \leq \|s_{\text{cov}}\|_{\mathcal{N}_{n_S};C} \leq R' \leq B_i$. Hence we can see $s, s_{\text{cov}} \in S_{(i+1,B_i)}$ which implies we can find a path $\boldsymbol{s}'$ from s covering s_{cov} in $\mathcal{N}_{i+1}$ with $|\boldsymbol{s}'| \leq \mathit{d}_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right)$. Hence $\mathit{dist}_{\mathcal{N}_{i+1}}(s, s_{\text{cov}}) \leq \mathit{d}_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right)$.

Case (B): There exist paths $\boldsymbol{s}_1$ and $s_{p'} \cdot \boldsymbol{s}_2$ such that $\boldsymbol{s}_1$ is a covering path for $s_{p'}$, $\boldsymbol{s}_1(1) = s$ and $\|\boldsymbol{s}_1\|^*_{\mathcal{N}_{i+1}} < B_i$; $s_{p'} \cdot \boldsymbol{s}_2$ is a covering path for s_{cov} and $|\boldsymbol{s}'_2| \leq \rho_{\mathcal{N}_i}(s_{cov})$; and $\left\|s_{p'}\right\|_{\mathcal{N}_{i+1};C} \leq B_i$.
We can see $s, s_{p'} \in S_{(i+1,B_i)}$ which implies we can find a path $\boldsymbol{s}'$ from s covering $s_{p'}$ in $\mathcal{N}_{i+1}$ with $|\boldsymbol{s}'| \leq \mathit{d}_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right)$. Since $\mathcal{N}_{i+1}$ is a WSTS and $s_{p'} \leq_{\mathit{Config}} \boldsymbol{s}'(|\boldsymbol{s}'|)$, we can replay $s_{p'} \cdot \boldsymbol{s}_2$ from $\boldsymbol{s}'(|\boldsymbol{s}'|)$ yielding a path $\boldsymbol{s}'(|\boldsymbol{s}'|) \cdot \boldsymbol{s}''$ in $\mathcal{N}_{i+1}$ such that $|\boldsymbol{s}''| = |\boldsymbol{s}_2|$ and $\boldsymbol{s}''$ covers s_{cov}. Thus $\boldsymbol{s}'\boldsymbol{s}''$ is a covering path in $\mathcal{N}_{i+1}$ for s_{cov} with $|\boldsymbol{s}'\boldsymbol{s}''| \leq \mathit{d}_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right) + \rho_{\mathcal{N}_i}(s_{\text{cov}})$. Hence we can conclude that
$\mathit{dist}_{\mathcal{N}_{i+1}}(s, s_{\text{cov}}) \leq \mathit{d}_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right) + \rho_{\mathcal{N}_i}(s_{\text{cov}})$.

Since the inequality holds in all cases and s was arbitrary we can conclude that

$$\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) \leq \mathit{d}_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right) + \rho_{\mathcal{N}_i}(s_{\text{cov}}).$$

□

Lemma 10. Let $i \leq n_S$, $B \in \mathbb{N}$. Suppose $s, s' \in \mathit{Config}^\infty$ such that both $\|s\|_{\mathcal{N}_i}, \|s'\|_{\mathcal{N}_i} < \infty$, then $\alpha_{i,B}(s \oplus s') = \alpha_{i,B}(s) + \alpha_{i,B}(s')$ and if $s \ominus s' \in \mathit{Config}^\infty$ then $\alpha_{i,B}(s \ominus s') = \alpha_{i,B}(s) - \alpha_{i,B}(s')$.

Proof. It is clear that for all $1 \leq j \leq i$

$$\begin{aligned}\alpha_{i,B}(s \oplus s')(j) &= |(s \oplus s')(p_{(j)})| = (s \oplus s')(p_{(j)})(\bullet) = s(p_{(j)})(\bullet) + s'(p_{(j)})(\bullet) \\ &= |s(p_{(j)})| + |s'(p_{(j)})| = \alpha_{i,B}(s)(j) + \alpha_{i,B}(s')(j)\end{aligned}$$

and since $s \ominus s' \in \mathit{Config}^\infty$ we know $|s(p_{(j)})| - |s'(p_{(j)})| \geq 0$ and hence

$$\begin{aligned}\alpha_{i,B}(s \ominus s')(j) &= |(s \ominus s')(p_{(j)})| = (s \ominus s')(p_{(j)})(\bullet) = s(p_{(j)})(\bullet) - s'(p_{(j)})(\bullet) \\ &= |s(p_{(j)})| - |s'(p_{(j)})| = \alpha_{i,B}(s)(j) - \alpha_{i,B}(s')(j)\end{aligned}$$

Further let $1 \leq j_{n_{col}} \leq n_C$ and $(j_0, \ldots, j_{n_{col}-1}) \in \mathbb{N}^{\leq B}$. Then

$$\begin{aligned}\alpha_{i,B}(s \oplus s')\left(i + \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k\right) &= \sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} (s \oplus s')(p'_{(j_{n_{col}})})(m^{col}) \\ = \sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} &\left(s(p'_{(j_{n_{col}})})(m^{col}) + s'(p'_{(j_{n_{col}})})(m^{col})\right) \\ = \left(\sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} s(p'_{(j_{n_{col}})})(m^{col})\right) &+ \left(\sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} s'(p'_{(j_{n_{col}})})(m^{col})\right)\end{aligned}$$

$$= \alpha_{i,B}(s)\left(i + \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k\right) + \alpha_{i,B}(s')\left(i + \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k\right)$$

and since $s \ominus s' \in \mathit{Config}^\infty$ we know $s(p'_{(j_{n_{col}})})(m^{col}) - s'(p'_{(j_{n_{col}})})(m^{col}) \geq 0$ and hence

$$\alpha_{i,B}(s \ominus s')\left(i + \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k\right) = \sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} (s \ominus s')(p'_{(j_{n_{col}})})(m^{col})$$

$$= \sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} \left(s(p'_{(j_{n_{col}})})(m^{col}) - s'(p'_{(j_{n_{col}})})(m^{col})\right)$$

$$= \left(\sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} s(p'_{(j_{n_{col}})})(m^{col})\right) - \left(\sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} s'(p'_{(j_{n_{col}})})(m^{col})\right)$$

$$= \alpha_{i,B}(s)\left(i + \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k\right) - \alpha_{i,B}(s')\left(i + \sum_{k=0}^{n_{col}} j_k \cdot (B+1)^k\right)$$

Hence we know that for all $1 \leq j \leq \widehat{d}_{i,B}$ $\alpha_{i,B}(s \oplus s')(j) = \alpha_{i,B}(s)(j) + \alpha_{i,B}(s')(j)$ and thus $\alpha_{i,B}(s \oplus s') = \alpha_{i,B}(s) + \alpha_{i,B}(s')$, and if $s \ominus s' \in \mathit{Config}^\infty$ then for $1 \leq j \leq \widehat{d}_{i,B}$ $\alpha_{i,B}(s \ominus s')(j) = \alpha_{i,B}(s)(j) - \alpha_{i,B}(s')(j)$ and thus

$$\alpha_{i,B}(s \ominus s') = \alpha_{i,B}(s) - \alpha_{i,B}(s').$$

□

Lemma 30. Let $1 \leq i \leq n_S$, $B \in \mathbb{N}$. Suppose $p = p'_{(j_{n_{col}})} \in \mathcal{P}^C$, and $j_0, \ldots, j_{n_{col}-1} \in \mathbb{N}^{\leq B}$ such that $m, m' \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1}) \subseteq \mathcal{M}^{colour}$ and $m'' \in \mathcal{M}^{colour}$ then

$$\alpha_{i,B}([p \mapsto [m \oplus m'']]) = \alpha_{i,B}([p \mapsto [m' \oplus m'']])$$

Proof. Let $j'_0, \ldots, j'_{n_{col}-1}$ such that $m \oplus m'' \in \gamma_B^{col}(j'_0, \ldots, j'_{n_{col}-1})$ and write $j'_{n_{col}} = j_{n_{col}}$. Let us consider $m_0 \oplus c$. Let $1 \leq k \leq n_{col}$ and note that since $m, m' \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ we have $\min(m(p_{(k)}^{col}), B) = j_{k-1} = \min(m(p_{(k)}^{col}), B)$; and thus

$$\begin{aligned} j_{k-1} &= \min((m \oplus m'')(p_{(k)}^{col}), B) = \min(m(p_{(k)}^{col}) + m''(p_{(k)}^{col}), B) \\ &= \min(\min(m(p_{(k)}^{col}), B) + m''(p_{(k)}^{col}), B) = \min(\min(m'(p_{(k)}^{col}), B) + m''(p_{(k)}^{col}), B) \\ &= \min(m'(p_{(k)}^{col}) + m''(p_{(k)}^{col}), B) \end{aligned}$$

from which we can conclude that $m' \oplus m'' \in \gamma_B^{col}(j'_0, \ldots, j'_{n_{col}-1})$. Further

$$\alpha_{i,B}([p \mapsto [m \oplus m'']])\left(i + \sum_{k=0}^{n_{col}} j'_k \cdot (B+1)^k\right) = \sum_{m^{col} \in \gamma_B^{col}(j'_0,\ldots,j'_{n_{col}-1})} ([p \mapsto [m \oplus m'']])(p)(m^{col})$$

$$= \left([p \mapsto [m \oplus m'']]\right)(p)(m \oplus m'') = 1 = \left([p \mapsto [m' \oplus m'']]\right)(p)(m \oplus m'')$$
$$= \sum_{m^{col} \in \gamma_B^{col}(j'_0,\ldots,j'_{n_{col}-1})} \left([p \mapsto [m' \oplus m'']]\right)(p)(m^{col})$$
$$= \alpha_{i,B}\left([p \mapsto [m' \oplus m'']]\right)\left(i + \sum_{k=0}^{n_{col}} j'_k \cdot (B+1)^k\right)$$

For any $p'_{(j''_{n_{col}})} \in \mathcal{P}^C$, and $j''_0, \ldots, j''_{n_{col}-1} \in \mathbb{N}^{\leq B}$ such that $i + \sum_{k=0}^{n_{col}} j'_k \cdot (B+1)^k \neq i + \sum_{k=0}^{n_{col}} j''_k \cdot (B+1)^k$ it is clear that $m \oplus m'', m' \oplus m'' \notin \gamma_B^{col}(j''_0, \ldots, j''_{n_{col}-1})$ and so we have

$$\alpha_{i,B}\left([p \mapsto [m \oplus m'']]\right)\left(i + \sum_{k=0}^{n_{col}} j''_k \cdot (B+1)^k\right) = \sum_{m^{col} \in \gamma_B^{col}(j''_0,\ldots,j''_{n_{col}-1})} \left([p \mapsto [m \oplus m'']]\right)(p'_{(j''_{n_{col}})})(m^{col})$$
$$= \sum_{m^{col} \in \gamma_B^{col}(j''_0,\ldots,j''_{n_{col}-1})} 0 = 0$$
$$= \sum_{m^{col} \in \gamma_B^{col}(j''_0,\ldots,j''_{n_{col}-1})} \left([p \mapsto [m' \oplus m'']]\right)(p'_{(j''_{n_{col}})})(m^{col})$$
$$= \alpha_{i,B}\left([p \mapsto [m' \oplus m'']]\right)\left(i + \sum_{k=0}^{n_{col}} j''_k \cdot (B+1)^k\right)$$

and also for all $1 \leq j \leq n_S$

$$\alpha_{i,B}\left([p \mapsto [m \oplus m'']]\right)(j) = |\left([p \mapsto [m \oplus m'']]\right)\left(p_{(j)}\right)| = 0 = |\left([p \mapsto [m' \oplus m'']]\right)\left(p_{(j)}\right)|$$
$$= \alpha_{i,B}\left([p \mapsto [m' \oplus m'']]\right)(j)$$

Hence we can conclude that

$$\alpha_{i,B}\left([p \mapsto [m \oplus m'']]\right) = \alpha_{i,B}\left([p \mapsto [m' \oplus m'']]\right)$$

□

Lemma 31. Let $1 \leq i \leq n_S$, $B \in \mathbb{N}$. Suppose $p = p'_{(j_{n_{col}})} \in \mathcal{P}^C$, and $j_0, \ldots, j_{n_{col}-1} \in \mathbb{N}^{\leq B}$ such that $m, m' \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1}) \subseteq \mathcal{M}^{colour}$ and $P \subseteq \mathcal{P}^{col}$ then

$$\alpha_{i,B}\left([p \mapsto [m \restriction \overline{P}]]\right) = \alpha_{i,B}\left([p \mapsto [m' \restriction \overline{P}]]\right)$$

where we write $\overline{P} = \mathcal{P}^{col} \setminus P$.

Proof. Let $j'_0, \ldots, j'_{n_{col}-1}$ such that $m \restriction \overline{P} \in \gamma_B^{col}(j'_0, \ldots, j'_{n_{col}-1})$ and write $j'_{n_{col}} = j_{n_{col}}$. Let us consider $m' \restriction \overline{P}$. Let $1 \leq k \leq n_{col}$ and note that since $m, m' \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ we have $\min(m(p^{col}_{(k)}), B) = j_{k-1} = \min(m(p^{col}_{(k)}), B)$; and thus if $p^{col}_{(k)} \notin P$

$$j_{k-1} = \min(m \restriction \overline{P}(p^{col}_{(k)}), B) = \min(m(p^{col}_{(k)}), B) = \min(m'(p^{col}_{(k)}), B) = \min(m' \restriction \overline{P}(p^{col}_{(k)}), B)$$

and for $p^{col}_{(k)} \in P$

$$j_{k-1} = \min(m \restriction \overline{P}(p^{col}_{(k)}), B) = \min(0, B) = 0 = \min(m' \restriction \overline{P}(p^{col}_{(k)}), B)$$

from which we can conclude that $m \restriction \overline{P}, m' \restriction \overline{P} \in \gamma^{col}_B(j'_0, \ldots, j'_{n_{col}-1})$. Lemma 30 then gives

$$\alpha_{i,B}\left(\left[p \mapsto \left[m \restriction \overline{P}\right]\right]\right) = \alpha_{i,B}\left(\left[p \mapsto \left[m' \restriction \overline{P}\right]\right]\right)$$

□

Lemma 11. Suppose $s \in \mathit{Config}^\infty$, $i \leq n_S$, $B \in \mathbb{N}$ such that $|s(p_{(j)})| = \infty$ for all $i < j \leq n_S$.
(i) If $s \rightarrow_{\mathbb{B}_B[\mathcal{N}_i]} s'$ then we have $\alpha_{i,B}(s) \rightarrow_{\mathcal{V}_{i,B}} \alpha_{i,B}(s')$.
(ii) If $\alpha_{i,B}(s) \rightarrow_{\mathcal{V}_{i,B}} \alpha_{i,B}(s) + r$ then there exists an $s' \in \mathit{Config}^\infty$ such that $\alpha_{i,B}(s') = \alpha_{i,B}(s) + r$ and $s \rightarrow_{\mathcal{N}_i} s'$.

Proof. We will first prove (i). Suppose $s \xrightarrow{r}_{\mathbb{B}_B[\mathcal{N}_i]} s'$ for some $r \in \mathcal{R}$ and let us write (a) $r = (I, O)$ if $r \in \mathit{SimpleRules}$; (b) $r = ((p, I), (p', c, O))$ if $r \in \mathit{ComplexRules}$; and (c) $r = ((p, I), (p', P, O))$ if $r \in \mathit{TransferRules}$.

Since $s \xrightarrow{r}_{\mathcal{N}_i} s'$ we know for all $p_{(j)} \in \mathcal{P}^S$ we have $|s(p_{(j)})| \geq |I(p_{(j)})|$ and for all $p' \in \mathcal{P}^C$ $s(p') \supseteq I(p')$. Hence $s \ominus I \in \mathit{Config}^\infty$ and $\alpha_{i,B}(s)(i) = |s(p_{(j)})| \geq |I(p_{(j)})| = \alpha_{i,B}(I)(j)$.

- *Case:* $s \xrightarrow{r}_{\mathcal{N}_i} s', r = (I, O) \in \mathit{SimpleRules}$.
  In this case we can see that for all $p'_{(j)} \in \mathcal{P}^C$

  $$\alpha_{i,B}(s)\left(i + j\,(B+1)^{n_{col}}\right) = s(p'_{(j)})(\mathbf{0}) \geq I(p'_{(j)})(\mathbf{0}) = \alpha_{i,B}(I)\left(i + j\,(B+1)^{n_{col}}\right).$$

  We see that $\alpha_{i,B}(r)$ is enabled at $\alpha_{i,B}(s)$ and hence there is a transition $\alpha_{i,B}(s) \rightarrow_{\mathcal{V}_{i,B}} \alpha_{i,B}(s) + \widehat{r}$. We know that $s' = (s \ominus I) \oplus O$ so since $s \ominus I \in \mathit{Config}^\infty$ Lemma 10 gives us $\alpha_{i,B}(s') = \left(\alpha_{i,B}(s) - \alpha_{i,B}(I)\right) + \alpha_{i,B}(O)$ and clearly $\widehat{r} = (-\alpha_{i,B}(I)) + \alpha_{i,B}(O)$ so $\alpha_{i,B}(s') = \alpha_{i,B}(s) + \widehat{r}$ and thus $\alpha_{i,B}(s) \rightarrow_{\mathcal{V}_{i,B}} \alpha_{i,B}(s')$ which is what we wanted to prove.

- *Case:* $s \xrightarrow{r}_{\mathcal{N}_i} s', r = ((p, I), (p', c, O)) \in \mathit{ComplexRules}$.
  In this case we know that for some $m \in s(p)$ we have $s \ominus [p \mapsto [m]] \ominus I \in \mathit{Config}^\infty$ and

  $$s' = (s \ominus [p \mapsto [m]] \ominus I) \oplus \left[p' \mapsto [m \oplus c]\right] \oplus O$$

  Suppose $p = p'_{(j_{n_{col}})}$, and $j_0, \ldots, j_{n_{col}-1} \in \mathbb{N}^{\leq B}$ such that $m \in \gamma^{col}_B(j_0, \ldots, j_{n_{col}-1})$ then we know that

  $$\alpha_{i,B}(s)\left(i + \sum_{k=0}^{n_{col}} j_k\,(B+1)^k\right) = \sum_{m^{col} \in \gamma^{col}_B(j_0, \ldots, j_{n_{col}})} s(p)(m^{col})| \geq s(p)(m) \geq 1.$$

  We can thus see that for some $m_0 \in \gamma^{col}_B(j_0, \ldots, j_{n_{col}-1})$ such that $\|m_0\|_I \leq B$ the rule $\widehat{r}_{m_0}$ is enabled at $\alpha_{i,B}(s)$ and thus $\alpha_{i,B}(s) \xrightarrow{\widehat{r}_{m_0}} \mathcal{V}_{i,B}\ \alpha_{i,B}(s) + \widehat{r}_{m_0}$. Since $s \ominus [p \mapsto [m]] \ominus I \in \mathit{Config}^\infty$ Lemma 10 gives us

  $$\alpha_{i,B}(s') = \alpha_{i,B}(s) - \alpha_{i,B}\left([p \mapsto [m]]\right) - \alpha_{i,B}\left(I\right) + \alpha_{i,B}\left(\left[p' \mapsto [m \oplus c]\right]\right) + \alpha_{i,B}(O)$$

Since $m, m_0 \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ Lemma 30 applies to give both

$$\alpha_{i,B}\left([p \mapsto [m]]\right) = \alpha_{i,B}\left([p \mapsto [m_0]]\right)$$
$$\alpha_{i,B}\left(\left[p' \mapsto [m \oplus c]\right]\right) = \alpha_{i,B}\left(\left[p' \mapsto [m_0 \oplus c]\right]\right)$$

and hence

$$\begin{aligned}\alpha_{i,B}(s') &= \alpha_{i,B}(s) - \alpha_{i,B}\left([p \mapsto [m_0]]\right) - \alpha_{i,B}\left(I\right) + \alpha_{i,B}\left(\left[p' \mapsto [m_0 \oplus c]\right]\right) + \alpha_{i,B}(O) \\ &= \alpha_{i,B}(s) + \widehat{r}_{m_0}\end{aligned}$$

and thus $\alpha_{i,B}(s) \xrightarrow{\widehat{r}_{m_0}}_{\mathcal{V}_{i,B}} \alpha_{i,B}(s')$ which is what we wanted to prove.

- *Case:* $s \xrightarrow{r}_{\mathcal{N}_i} s', r = ((p, I), (p', P, O)) \in$ *TransferRules*.
In this case we know that for some $m \in s(p)$ we have $s \ominus [p \mapsto [m]] \ominus I \in$ *Config*$^\infty$ and

$$s' = (s \ominus [p \mapsto [m]] \ominus I) \oplus \left[p' \mapsto \left[m_{\overline{P}}\right]\right] \oplus O \oplus \left(m_P \circ \zeta^{-1}\right)$$

where $m_P = m \restriction P$ and $m_{\overline{P}} = m \restriction \{\mathcal{P}^{col} \setminus P\}$. Since $\|s'\|_{\mathcal{N}_i} < B$ we can conclude that for all $p \in \zeta(P)$ we can bound $|\left(m_P \circ \zeta^{-1}\right)(p)| < B$ which implies that

$$\max_{p \in P}(|m(p)|) < B.$$

Suppose $p = p'_{\left(j_{n_{col}}\right)}$, and $j_0, \ldots, j_{n_{col}-1} \in \mathbb{N}^{\leq B}$ such that $m \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ then we know that

$$\alpha_{i,B}(s)\left(i + \sum_{k=0}^{n_{col}} j_k \left(B+1\right)^k\right) = \sum_{m^{col} \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}})} s(p)(m^{col}) \geq s(p)(m) \geq 1.$$

Further for all k such that $p^{col}_{(k)} \in P$ it is the case that $j_k < B$. Hence for some $m_0 \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ such that $\|m_0\|_{col} \leq B$ and $\max_{p \in P}(|m_0(p)|) < B$ the rule $\widehat{r}_{m_0}$ is enabled at $\alpha_{i,B}(s)$ and so $\alpha_{i,B}(s) \xrightarrow{\widehat{r}_{m_0}}_{\mathcal{V}_{i,B}} \alpha_{i,B}(s) + \widehat{r}_{m_0}$. Since $s \ominus [p \mapsto [m]] \ominus I \in$ *Config*$^\infty$ Lemma 10 yields

$$\begin{aligned}\alpha_{i,B}(s') = \alpha_{i,B}(s) &- \alpha_{i,B}\left([p \mapsto [m]]\right) - \alpha_{i,B}(I) + \alpha_{i,B}\left(\left[p' \mapsto \left[m_{\overline{P}}\right]\right]\right) \\ &+ \alpha_{i,B}(O) + \alpha_{i,B}(\left(m_P \circ \zeta^{-1}\right)).\end{aligned}$$

Since $m, m_0 \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ we can apply Lemma 30 and Lemma 31 to get

$$\alpha_{i,B}\left([p \mapsto [m]]\right) = \alpha_{i,B}\left([p \mapsto [m_0]]\right)$$
$$\alpha_{i,B}\left(\left[p' \mapsto \left[m_{\overline{P}}\right]\right]\right) = \alpha_{i,B}\left(\left[p' \mapsto \left[m_{0\overline{P}}\right]\right]\right)$$

where $m_{0\overline{P}} = m_0 \restriction (\mathcal{P}^{col} \setminus P)$. Since $\max_{p \in P}(|m(p)|) < B$ and $\max_{p \in P}(|m_0(p)|) < B$ we can see that

$$m(p_{(k)}) = \min(m(p_{(k)}), B) = j_{k-1} = \min(m_0(p_{(k)}), B) = m_0(p_{(k)})$$

for all $1 \leq k \leq \mathcal{P}^{col}$ such that $p^{col}_{(k)} \in P$. Hence we can see that

$$m_P \circ \zeta^{-1} = m_{0P} \circ \zeta^{-1}$$

where $m_{0P} = m_0 \restriction P$ which implies

$$\begin{aligned}\alpha_{i,B}(s') &= \alpha_{i,B}(s) - \alpha_{i,B}\left([p \mapsto [m_0]]\right) - \alpha_{i,B}(I) + \alpha_{i,B}\left(\left[p' \mapsto \left[m_{0\overline{P}}\right]\right]\right) \\ &\quad + \alpha_{i,B}(O) + \left(m_{0P} \circ \zeta^{-1}\right) \\ &= \alpha_{i,B}(s) + \widehat{r}_{m_0}\end{aligned}$$

and hence $\alpha_{i,B}(s) \xrightarrow{\widehat{r}_{m_0}}_{\mathcal{V}_{i,B}} \alpha_{i,B}(s')$ which is what we wanted to prove.

Hence in all cases $\alpha_{i,B}(s) \rightarrow_{\mathcal{V}_{i,B}} \alpha_{i,B}(s')$ thus (i) holds.

For (ii) suppose $\alpha_{i,B}(s) \rightarrow_{\mathcal{V}_{i,B}} \alpha_{i,B}(s) + r$ for some $r \in \hat{F}_{i,B}$. We know that either (a) for some $r_0 = (I, O) \in$ *SimpleRules* we have $r = \widehat{r_0}$; or (b) for some $m \in \mathcal{M}^{colour}$ such that $\|m\|_{col} \leq B$, $r = \widehat{r_0}_m$ for some $r_0 = ((p, I), (p', c, O)) \in$ *ComplexRules*; or (c) for some $m \in \mathcal{M}^{colour}$ such that $\|m\|_{col} \leq B$ and for all $\max_{p \in P}(|m(p)|) < B$, $r = \widehat{r_0}_m$ for some $r_0 = ((p, I), (p', P, O)) \in$ *TransferRules*.

Before we do a case analysis on r_0 let us have a look at the simple places. First of all $1 \leq j \leq i$, $p_{(j)} \in \mathcal{P}^S$ we know that $\alpha_{i,B}(s)(j) + r(j) \geq 0$, i.e. $|s(p_{(j)})| \geq |I(p_{(j)})|$ and we know that $|s(p_{(j)})| = \infty$ for $i < j \leq n_S$. Hence clearly $s \ominus (I \restriction \mathcal{P}^S) \in$ *Config*$^\infty$.

We proceed the proof by a case analysis on r.

- *Case:* $r = \widehat{r_0}$ for $r_0 \in$ *SimpleRules*.
  In this case we can see that for all $p'_{(j)} \in \mathcal{P}^C$

  $$s(p'_{(j)})(\mathbf{0}) = \alpha_{i,B}(s)\left(i + j\,(B+1)^{n_{col}}\right) \geq \alpha_{i,B}(I)\left(i + j\,(B+1)^{n_{col}}\right) = I(p'_{(j)})(\mathbf{0}).$$

  Hence $s \ominus I \in$ *Config*$^\infty$. Further we know that

  $$\alpha_{i,B}(s) + r = \alpha_{i,B}(s) - \alpha_{i,B}(I) + \alpha_{i,B}(O)$$

  and since by above $s \ominus I \in$ *Config*$^\infty$ we get

  $$\alpha_{i,B}(s) + r = \alpha_{i,B}(s \ominus I \oplus O)$$

  if we define $s' = s \ominus I \oplus O$ then clearly $s \rightarrow_{\mathcal{N}_i} s'$ which is what we wanted to prove.

- *Case:* $r = \widehat{r_0}_m$ for $r_0 \in$ *ComplexRules*.
  Suppose $p = p'_{(j_{n_{col}})}$ and $j_0, \ldots, j_{n_{col}-1} \in \mathbb{N}^{\leq B}$ such that $m \in \gamma^{col}_B(j_0, \ldots, j_{n_{col}-1})$ then by definition $\widehat{r_0}_m(i + \sum_{k=0}^{n_{col}} j_k (B+1)^k) = -1$, since $\alpha_{i,B}(s) \rightarrow_{\mathcal{V}_{i,B}} \alpha_{i,B}(s) + r$ we must have that

  $$\alpha_{i,B}(s)\,(\theta) + \widehat{r_0}_m\,(\theta) \geq 0$$

where $\theta = i + \sum_{k=0}^{n_{col}} j_k \, (B+1)^k$. Hence we can deduce that

$$\sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} |s(p'_{(j_{n_{col}})})(m^{col})| \geq 1$$

Hence there exists a $m_0 \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ such that $|s(p'_{(j_{n_{col}})})(m_0)| \geq 1$.

It is easy to see that $s \ominus I \ominus [p \mapsto [m_0]] \in \mathit{Config}^\infty$

$$s \to_{\mathcal{N}_i} (s \ominus I \ominus [p \mapsto [m_0]]) \oplus O \oplus \left[p' \mapsto [m_0 \oplus c]\right] =: s'$$

We can then use Lemma 10 to see that

$$\alpha_{i,B}(s') = \alpha_{i,B}(s) - \alpha_{i,B}(I) - \alpha_{i,B}\left([p \mapsto [m_0]]\right) + \alpha_{i,B}(O) + \alpha_{i,B}\left(\left[p' \mapsto [m_0 \oplus c]\right]\right)$$

Since $m, m_0 \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ and Lemma 30 yields

$$\begin{aligned} \alpha_{i,B}\left([p \mapsto [m_0]]\right) &= \alpha_{i,B}\left([p \mapsto [m]]\right) \\ \alpha_{i,B}\left(\left[p' \mapsto [m_0 \oplus c]\right]\right) &= \alpha_{i,B}\left(\left[p' \mapsto [m \oplus c]\right]\right) \end{aligned}$$

Hence we can see that

$$\begin{aligned} \alpha_{i,B}(s') &= \alpha_{i,B}(s) - \alpha_{i,B}(I) - \alpha_{i,B}\left([p \mapsto [m]]\right) + \alpha_{i,B}(O) + \alpha_{i,B}\left(\left[p' \mapsto [m \oplus c]\right]\right) \\ &= \alpha_{i,B}(s) + r \end{aligned}$$

which is what we wanted to prove.

- *Case:* $r = \widehat{r_0}_m$ for $r_0 \in \mathit{TransferRules}$.
Suppose $p = p'_{(j_{n_{col}})}$ and $j_0, \ldots, j_{n_{col}-1} \in \mathbb{N}^{\leq B}$ such that $m \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ then by definition $\widehat{r_0}_m(i + \sum_{k=0}^{n_{col}} j_k (B+1)^k) = -1$, since $\alpha_{i,B}(s) \to_{\mathcal{V}_{i,B}} \alpha_{i,B}(s) + r$ we must have that

$$\alpha_{i,B}(s)\,(\theta) + \widehat{r_0}_m\,(\theta) \geq 0.$$

where $\theta = i + \sum_{k=0}^{n_{col}} j_k \, (B+1)^k$. Hence we can deduce that

$$\sum_{m^{col} \in \gamma_B^{col}(j_0,\ldots,j_{n_{col}-1})} |s(p'_{(j_{n_{col}})})(m^{col})| \geq 1$$

Hence there exists a $m_0 \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ such that $|s(p'_{(j_{n_{col}})})(m_0)| \geq 1$.

Hence it is easy to see that $s \ominus I \ominus [p \mapsto [m_0]] \in \mathit{Config}^\infty$ and

$$s \to_{\mathcal{N}_i} s'$$

where

$$s' = s \ominus [p \mapsto [m_0]] \ominus I \oplus \left[p' \mapsto \left[m_{0\overline{P}}\right]\right] \oplus O \oplus \left(m_{0P} \circ \zeta^{-1}\right)$$

where $m_{0P} = m_0 \restriction P$ and $m_{0\overline{P}} \restriction (\mathcal{P}^{col} \setminus P)$. Since $s \ominus [p \mapsto [m_0]] \ominus I \in \mathit{Config}^\infty$ Lemma 10 yields

$$\alpha_{i,B}(s') = \alpha_{i,B}(s) - \alpha_{i,B}\left([p \mapsto [m_0]]\right) - \alpha_{i,B}(I) + \alpha_{i,B}\left(\left[p' \mapsto [m_0\,[p_0 \mapsto 0 \mid p_0 \in P]]\right]\right) + \alpha_{i,B}(O) + \alpha_{i,B}(((m_0 \restriction P) \circ \zeta^{-1} + \mathbf{0})).$$

Since $m, m_0 \in \gamma_B^{col}(j_0, \ldots, j_{n_{col}-1})$ we can apply Lemma 30 and Lemma 31 to get

$$\alpha_{i,B}\left([p \mapsto [m]]\right) = \alpha_{i,B}\left([p \mapsto [m_0]]\right)$$
$$\alpha_{i,B}\left(\left[p' \mapsto \left[m_{\overline{P}}\right]\right]\right) = \alpha_{i,B}\left(\left[p' \mapsto \left[m_{0\overline{P}}\right]\right]\right)$$

where $m_{\overline{P}} = m \restriction (\mathcal{P}^{col} \setminus P)$. Further we notice by assumption $\max p \in P |m(p)| < B$ and thus for all $1 \leq k \leq \mathcal{P}^{col}$ such that $p^{col}_{(k)} \in P$ we have

$$m(p_{(k)}) = \min(m(p_{(k)}), B) = j_{k-1} = \min(m_0(p_{(k)}), B) = m_0(p_{(k)})$$

and hence we can see that

$$\left(m_P \circ \zeta^{-1}\right) = \left(m_{0P} \circ \zeta^{-1} + \mathbf{0}\right)$$

where $m_P = m \restriction P$ which implies

$$\begin{aligned}\alpha_{i,B}(s') &= \alpha_{i,B}(s) - \alpha_{i,B}\left([p \mapsto [m]]\right) - \alpha_{i,B}(I) + \alpha_{i,B}\left(\left[p' \mapsto \left[m_{\overline{P}}\right]\right]\right) \\ &\quad + \alpha_{i,B}(O) + \alpha_{i,B}(m_P \circ \zeta^{-1}) \\ &= \alpha_{i,B}(s') + r\end{aligned}$$

which is what we wanted to prove.

Thus in all cases also (ii) holds and hence we can conclude the proof. □

Proposition 7. The relation $\{\left(s, \alpha_{i,B}(s)\right) \mid s \in \mathbb{B}_B[\mathit{Config}^\infty], |s(p_{(j)})| = \infty, i < j\}$ is a bisimulation between the labelled transition systems $\mathbb{B}_B[\mathcal{N}_i]$ and $\mathbb{B}_B[\mathcal{V}_{i,B}]$.

Proof. Let $\mathcal{B} = \{\left(s, \alpha_{i,B}(s)\right) \mid s \in \mathbb{B}_B[\mathit{Config}^\infty], |s(p_{(j)})| = \infty, i < j\}$ and suppose $(s, \widehat{s}) \in \mathcal{B}$. By definition we know that $\widehat{s} = \alpha_{i,B}(s)$. Suppose $s \xrightarrow{s'}_{\mathbb{B}_B[\mathcal{N}_i]} s'$ is a transition then Lemma 11 yields that $\alpha_{i,B}(s) \xrightarrow{s'}_{\mathcal{V}_{i,B}} \alpha_{i,B}(s')$. Clearly $(s, \alpha_{i,B}(s')) \in \mathcal{B}$ since $s' \in \mathbb{B}_B[\mathit{Config}^\infty]$ which also implies that for all $1 \leq j \leq i$ we have $|\alpha_{i,B}(s')(j)| = |s'(p_{(j)})| < B$ and thus $\left\|\alpha_{i,B}(s')\right\|_{\mathcal{V}_{i,B}} < B$. Hence $\alpha_{i,B}(s) \xrightarrow{s'}_{\mathbb{B}_B[\mathcal{V}_{i,B}]} \mathcal{V}_{i,B}\alpha_{i,B}(s')$ is transition and so we can conclude that $\mathcal{B}$ is a simulation relation.

Let us now consider $\mathcal{B}^{-1}$. Suppose $(s, \widehat{s}) \in \mathcal{B}$, again we know that $\widehat{s} = \alpha_{i,B}(s)$. Suppose $\alpha_{i,B}(s) \xrightarrow{u}_{\mathbb{B}_B[\mathcal{V}_{i,B}]} \alpha_{i,B}(s)+r$ is a transition for some $r \in R_{i,B}$. Lemma 11 then yields that there exists $s' \in \mathit{Config}^\infty$ such that $\alpha_{i,B}(s') = \alpha_{i,B}(s) + r$ and $s \xrightarrow{s'}_{\mathcal{N}_i} s'$. Hence we can conclude $u = s'$. Further since $\left\|\alpha_{i,B}(s) + r\right\|_{\mathcal{V}_{i,B}} < B$ we know that $\left\|\alpha_{i,B}(s')\right\|_{\mathcal{V}_{i,B}} < B$ which implies that that for all $1 \leq i \leq j$ we have $|s'(p_{(j)})| = \alpha_{i,B}(s')(i) < B$ and thus $\|\mathcal{N}_i\|\, s' < B$. Hence $(s, \alpha_{i,B}(s')) \in \mathcal{B}$ and so we know that $\mathcal{B}^{-1}$ is a simulation.

We can thus conclude that $\mathcal{B}$ is a bisimulation. □

Proposition 8. The relation $\{(\alpha_{i,B}(s), \alpha_{i,B}(s)) \mid s \in \mathit{Config}^\infty\}$ is a simulation relation for $\mathcal{V}_{i,B}$ and $\mathcal{V}^{\leq}_{i,B}$. The relation $\{(\alpha_{i,B}(s), s') \mid s \leq_{\mathit{Config}} s', s, s' \in \mathit{Config}^\infty, |s(p_{(j)})| = \infty, i < j\}$ is a weak simulation for $\mathcal{V}^{\leq}_{i,B}$ and $\mathcal{N}_i$.

Proof. That $\{(\alpha_{i,B}(s), \alpha_{i,B}(s)) \mid s \in \mathit{Config}^\infty\}$ is a simulation relation for $\mathcal{V}_{i,B}$ and $\mathcal{V}^{\leq}_{i,B}$ is obvious, since if $v \xrightarrow{\sim}_{\mathcal{V}_{i,B}} v'$ then $v \xrightarrow{\sim}_{\mathcal{V}^{\leq}_{i,B}} v'$.

For the other direction let $\mathcal{W} = \{(\alpha_{i,B}(s), s') \mid s \leq_{\mathit{Config}} s', s, s' \in \mathit{Config}^\infty, |s(p_{(j)})| = \infty, i < j\}$. Suppose $(\alpha_{i,B}(s), \alpha_{i,B}(s')) \in \mathcal{W}$ then we know that $s \leq_{\mathit{Config}} s'$. Suppose that $\alpha_{i,B}(s) \xrightarrow{\sim}{}^*_{\mathcal{V}^{\leq}_{i,B}} v$ then we may split this transition up $\alpha_{i,B}(s) \xrightarrow{\epsilon}_{\mathcal{V}^{\leq}_{i,B}} v_1 \xrightarrow{\epsilon}_{\mathcal{V}^{\leq}_{i,B}} \cdots \xrightarrow{\epsilon}_{\mathcal{V}^{\leq}_{i,B}} v_{n-1} \xrightarrow{\sim}_{\mathcal{V}^{\leq}_{i,B}} v_n \xrightarrow{\epsilon}_{\mathcal{V}^{\leq}_{i,B}} \cdots \xrightarrow{\epsilon}_{\mathcal{V}^{\leq}_{i,B}} v$.

Let us prove a Lemma:

Lemma. If $\alpha_{i,B}(s_0) \xrightarrow{r}_{\mathcal{V}_{i,B}} \alpha_{i,B}(s_0) + r$ with $r \in \hat{F}^{\leq}_{i,B}$ then there exists $s_1 \in \mathit{Config}^\infty$ such that $\alpha_{i,B}(s_1) = \alpha_{i,B}(s_0) + r$ and $s_1 \leq_{\mathit{Config}} s_0$.

Proof. Since $r \in \hat{F}^{\leq}_{i,B}$ we know $r = \alpha_{i,B}(M_{p,m'}) - \alpha_{i,B}(M_{p,m})$ for some $m, m' \in \mathcal{M}^{\mathit{colour}}$, such that both $\|m\|_{\mathit{col}}, \|m'\|_{\mathit{col}} \leq B$ and $m >_{\mathcal{M}^{\mathit{colour}}} m'$. Suppose $p = p'_{(j_{n_{\mathit{col}}})}$ and $j_0, \ldots, j_{n_{\mathit{col}}-1} \in \mathbb{N}^{\leq B}$ such that $m \in \gamma^{\mathit{col}}_B(j_0, \ldots, j_{n_{\mathit{col}}-1})$ then by definition

$$r\left(i + \sum_{k=0}^{n_{\mathit{col}}} j_k \left(B+1\right)^k\right) = -1,$$

since $\alpha_{i,B}(s_0) \rightarrow_{\mathcal{V}^{\leq}_{i,B}} \alpha_{i,B}(s_0) + r$. we must have that

$$\alpha_{i,B}(s_0)\left(\theta\right) + r\left(\theta\right) \geq 0$$

where $\theta = i + \sum_{k=0}^{n_{\mathit{col}}} j_k \left(B+1\right)^k$. Hence we can deduce that

$$\sum_{m^{\mathit{col}} \in \gamma^{\mathit{col}}_B(j_0,\ldots,j_{n_{\mathit{col}}-1})} |s_0(p'_{(j_{n_{\mathit{col}}})})(m^{\mathit{col}})| \geq 1$$

Thus there exists a complex token $m_0 \in \gamma^{\mathit{col}}_B(j_0, \ldots, j_{n_{\mathit{col}}-1})$ such that $|s_0(p'_{(j_{n_{\mathit{col}}})})(m_0)| \geq 1$.

Since $\|m\|_{\mathit{col}} \leq B$ we know that

$$m(p^{\mathit{col}}_{(k)}) = \min(m(p^{\mathit{col}}_{(k)}), B) = j_{k-1} = \min(m_0(p^{\mathit{col}}_{(k)}), B) \leq m_0(p^{\mathit{col}}_{(k)})$$

for all $1 \leq k \leq n_{\mathit{col}}$ and hence $m' <_{\mathcal{M}^{\mathit{colour}}} m \leq_{\mathcal{M}^{\mathit{colour}}} m_0$.

Clearly $s_0 \ominus s_{p,m_0} \in \mathit{Config}^\infty$ and hence Lemma 10 gives

$$\alpha_{i,B}(s_0 \oplus s_{p,m'} \ominus s_{p,m_0}) = \alpha_{i,B}(s_0) + \alpha_{i,B}(s_{p,m'}) - \alpha_{i,B}(s_{p,m_0})$$

since $m, m_0 \in \gamma^{\mathit{col}}_B(j_0, \ldots, j_{n_{\mathit{col}}-1})$ Lemma 30 gives

$$\alpha_{i,B}(s_0 \oplus s_{p,m'} \ominus s_{p,m_0}) = \alpha_{i,B}(s_0) + \alpha_{i,B}(s_{p,m'}) - \alpha_{i,B}(s_{p,m}) = \alpha_{i,B}(s_0) + r$$

It is easy to see that $s_1 := s_0 \oplus s_{p,m'} \ominus s_{p,m_0} \leq_{\mathit{Config}} s$ which concludes the proof of the Lemma. □

The Lemma allows us to conclude that there are $s_j \in \mathit{Config}^\infty$ such that $v_j = \alpha_{i,B}(s_j)$ for all $j \in \langle n-1\rangle$, $|s_j(p_{(j')})| = \infty$ for all $i < j'$ and $s_j \leq_{\mathit{Config}} s_{j-1} \leq_{\mathit{Config}} s$ for $1 < j < n$. Inspecting the transition $\alpha_{i,B}(s_{n-1}) \xrightarrow{\sim}_{\mathcal{V}^{\leq}_{i,B}} v_n$ then due to the labelling we know that $\alpha_{i,B}(s_{n-1}) \rightarrow_{\mathcal{V}_{i,B}} v_n$. Lemma 11 then yields that there is a $s'' \in \mathit{Config}^\infty$ such that $s_{n-1} \rightarrow_{\mathcal{N}_i} s''$ and $v_n = \alpha_{i,B}(s'')$. Since $s \leq_{\mathit{Config}} s'$ and $\mathcal{N}_i$ is a WSTS we obtain $\exists s''' \in \mathit{Config}^\infty$ such that $s' \xrightarrow{\sim}_{\mathcal{N}_i} s'''$ and $s'' \leq_{\mathit{Config}} s'''$. Applying the above lemma further we can see that $v = \alpha_{i,B}(s'''')$ for some $s''' \leq_{\mathit{Config}} s''$ which implies $s'''' \leq_{\mathit{Config}} s'''$ and hence that $(\alpha_{i,B}(s''''), s''') \in \mathcal{W}$ which is what we wanted to prove.

Thus $\mathcal{W}$ is a weak simulation. □

Corollary 12. Let $1 \leq i \leq n_S$, $B \in \mathbb{N}$. Suppose $s, s' \in \mathit{Config}^\infty$ such that there exists a covering path from s for s' in $\mathbb{B}_B[\mathcal{N}_i]$ then there exists a covering path from $\alpha_{i,B}(s)$ for $\alpha_{i,B}(s')$ in $\mathbb{B}_B[\mathcal{V}^{\leq}_{i,B}]$.

Proof. Let $\boldsymbol{s}$ be a covering path for s' from s in $\mathbb{B}_B[\mathcal{N}_i]$. Then Proposition 7 yields a bisimulation for $\mathbb{B}_B[\mathcal{N}_i]$ and $\mathbb{B}_B[\mathcal{V}_{i,B}]$. Hence $\alpha_{i,B}(\boldsymbol{s}) = \alpha_{i,B}(\boldsymbol{s}(1)) \cdots \alpha_{i,B}(\boldsymbol{s}(|\boldsymbol{s}|))$ is a path in $\mathbb{B}_B[\mathcal{V}_{i,B}]$ from $\alpha_{i,B}(s)$ to $\alpha_{i,B}(\boldsymbol{s}(|\boldsymbol{s}|))$.

It is immediate that $\alpha_{i,B}(\boldsymbol{s})$ is also a path in $\mathbb{B}_B[\mathcal{V}^{\leq}_{i,B}]$.

Since $\boldsymbol{s}$ is a covering path for s' in $\mathbb{B}_B[\mathcal{N}]$ we have $s' \leq_{\mathit{Config}} \boldsymbol{s}(|\boldsymbol{s}|)$ and hence for all $1 \leq j \leq i$ we have $|\boldsymbol{s}(|\boldsymbol{s}|)(p_{(j)})| \geq |s'(p_{(j)})|$, further for all $1 \leq j \leq n_C$ we have $\boldsymbol{s}(\boldsymbol{s})(p'_{(j)}) \leq_{\mathbb{M}[\mathcal{M}^{\mathit{colour}}]} s'(p'_{(j)})$. Hence we know that for all $1 \leq j \leq n_C$

$$s'(p'_{(j)}) = \left[m^j_1, \ldots, m^j_{k_j}\right] \qquad \text{and} \qquad \boldsymbol{s}(\boldsymbol{s})(p'_{(j)}) = \left[{m'}^j_1, \ldots, {m'}^j_{k'_j}\right]$$

such that for all $1 \leq j \leq n_C$ and $1 \leq j' \leq k_j$ we have $m^j_{j'} \leq_{\mathcal{M}^{\mathit{colour}}} {m'}^j_{j'}$ (where we wlog assume the injection $h(j') = j'$).

We will now extend $\alpha_{i,B}(\boldsymbol{s})$ to a new path $\alpha_{i,B}(\boldsymbol{s})\boldsymbol{s}'$ so that $\boldsymbol{s}'(|\boldsymbol{s}'|)$ covers $\alpha_{i,B}(s')$ in $\mathcal{V}_{i,B}$. This will be achieved by using rules in $\hat{F}^{\leq}_{i,B}$ that will allow us to swap the counter abstraction representation of the complex tokens ${m'}^j_{j'}$ for $m^j_{j'}$.

Let us write $t_0 = \boldsymbol{s}(\boldsymbol{s})$ using the rules in $\hat{F}^{\leq}_{i,B}$

$$\left(\alpha_{i,B}(M_{p_{(1)},m^1_{j'}}) - \alpha_{i,B}(M_{p_{(1)},{m'}^1_{j'}})), \alpha_{i,B}(M_{p_{(1)},{m'}^1_{j'}}))\right)$$

for $1 \leq j \leq k_1$ we can see using Lemma 10 that

$$\begin{aligned}\alpha_{i,B}(t_0) &\rightarrow_{\mathcal{V}^{\leq}_{i,B}} \alpha_{i,B}(t_0[p_{(1)} \mapsto \left(s(p_{(1)}) \ominus [{m'}^1_1] \oplus [m^1_1]\right)]) \\ &\rightarrow_{\mathcal{V}^{\leq}_{i,B}} \cdots \rightarrow_{\mathcal{V}^{\leq}_{i,B}} \alpha_{i,B}(t_0[p_{(1)} \mapsto (s(p_{(1)}) \ominus [{m'}^1_1, \ldots, {m'}^1_{k_1}] \oplus [m^1_{j'}, \ldots, m^1_{k_1}])])\end{aligned}$$

Clearly

$$\begin{aligned}&t_0\left[p_{(1)} \mapsto \left(s(p_{(1)}) \ominus \left[{m'}^1_1, \ldots, {m'}^1_{k_1}\right] \oplus \left[m^1_{j'}, \ldots, m^1_{k_1}\right]\right)\right] \\ &\quad = t_0\left[p_{(1)} \mapsto \left[m^1_1, \ldots, m^1_{k_1}, {m'}^1_{k_1+1}, \ldots, {m'}^1_{k'_1}\right]\right] =: t_1\end{aligned}$$

and we note that $s' \leq_{\mathbb{B}_B[\mathcal{N}]} t_1$. We can simply carry on in this fashion and obtain

$$\alpha_{i,B}(t_1) \to^*_{\mathcal{V}^{\leq}_{i,B}} \alpha_{i,B}(t_2) \to^*_{\mathcal{V}^{\leq}_{i,B}} \cdots \to^*_{\mathcal{V}^{\leq}_{i,B}} \alpha_{i,B}(t_{n_C})$$

and for all $1 \leq j \leq n_C$ we have

$$t_j = t_{j-1}\left[p_{(j)} \mapsto \left[m^j_1, \ldots, m^j_{k_j}, m'^j_{k_j+1}, \ldots, m'^j_{k'_j}\right]\right]$$

By construction we know that $\alpha_{i,B}(\boldsymbol{s}(|\boldsymbol{s}|)) \to^*_{\mathcal{V}^{\leq}_{i,B}} \alpha_{i,B}(t_{n_C})$ and so there is a path $\boldsymbol{s}'$ from $\alpha_{i,B}(\boldsymbol{s}(|\boldsymbol{s}|))$ to $\alpha_{i,B}(t_{n_C})$ and we further have for all $p \in \mathcal{P}^C$, $m \in \mathcal{M}^{\textit{colour}}$, $|s'(p)(m)| \leq |t_{n_C}(p)(m)|$, for all $p \in \mathcal{P}^S$, $|s'(p)| \leq |t_{n_C}(p)|$.

Thus it is routine to check that $\alpha_{i,B}(s') \leq_{\mathbb{N}^{\widehat{d}_{i,B}}} \alpha_{i,B}(t_{n_C})$ and thus $\alpha_{i,B}(\boldsymbol{s})\boldsymbol{s}'$ is a covering path for $\alpha_{i,B}(s')$ from $\alpha_{i,B}(s)$ and we note that clearly $\left\|\alpha_{i,B}(\boldsymbol{s})\boldsymbol{s}'\right\|^*_{\mathcal{V}_{i,B}} < B$.

Hence we can conclude that $\textit{dist}_{\mathbb{B}_B[\mathcal{N}]}(s, s') > 0 \implies \textit{dist}_{\mathbb{B}_B[\mathcal{V}^{\leq}_{i,B}]}(\alpha_{i,B}(s), \alpha_{i,B}(s')) > 0$. □

Corollary 6. Let $i \leq n_S$, $B \in \mathbb{N}$. For all $(s, s') \in S_{i,B}$ we have

$$\textit{dist}_{\mathcal{N}_i}(s, s') \leq \textit{dist}_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s), \alpha_{i,B}(s')).$$

Proof. Since $(s, s') \in S_{i,B}$ we know that there exists a covering path $\boldsymbol{s}$ in $\mathcal{N}_i$ from s for s' with norm $\|\boldsymbol{s}\|^*_{\mathcal{N}_i} < B$, i.e. $\boldsymbol{s}$ is a path in $\mathbb{B}_B[\mathcal{N}_i]$.

Corollary 12 then yields that there exists a covering path $\boldsymbol{s}_0$ from $\alpha_{i,B}(s)$ for $\alpha_{i,B}(s')$ in $\mathbb{B}_B[\mathcal{V}^{\leq}_{i,B}]$

In particular $\boldsymbol{s}_0$ is a path in $\mathcal{V}^{\leq}_{i,B}$ from $\alpha_{i,B}(s)$ that covers $\alpha_{i,B}(s')$. Hence we can find a path $\boldsymbol{s}_1$ in $\mathcal{V}^{\leq}_{i,B}$ from $\alpha_{i,B}(s)$ that covers $\alpha_{i,B}(s')$ such that $|\boldsymbol{s}_1| = \textit{dist}_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s), \alpha_{i,B}(s'))$.

Proposition 8 then gives us that $\mathcal{W}$ is a weak simulation for $\mathcal{V}^{\leq}_{i,B}$ and $\mathcal{N}_i$, and since clearly $(\alpha_{i,B}(s), s) \in \mathcal{W}$ an easy induction on the length of $\boldsymbol{s}_1$ gives us a path $\boldsymbol{s}'$ in $\mathcal{N}_i$ such that $|\boldsymbol{s}'| \leq |\boldsymbol{s}_1| = \textit{dist}_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s), \alpha_{i,B}(s'))$ and $(\alpha_{i,B}(s''), \boldsymbol{s}'(|\boldsymbol{s}'|)) \in \mathcal{W}$ where $\boldsymbol{s}_1(|\boldsymbol{s}|_1) = \alpha_{i,B}(s'')$ which by definition means $s'' \leq_{\textit{Config}} \boldsymbol{s}(|\boldsymbol{s}|)$

Since $\alpha_{i,B}(s'') \geq_{\mathbb{N}^{\widehat{d}_{i,B}}} \alpha_{i,B}(s')$ we know that for all $1 \leq j \leq n_S$, $|s''(p_{(j)})| \geq |s'(p_{(j)})|$. In order to compare the complex places let us fix $p \in \mathcal{P}^C$ and enumerate all elements of $\left(\mathbb{N}^{\leq B}\right)^{n_{\textit{col}}}$ by $\boldsymbol{j}_1, \ldots, \boldsymbol{j}_N$ where $N = |\left(\mathbb{N}^{\leq B}\right)^{n_{\textit{col}}}|$. Since $\alpha_{i,B}(s'') \geq_{\mathbb{N}^{\widehat{d}_{i,B}}} \alpha_{i,B}(s')$ we know that for all $1 \leq k \leq N$

$$\sum_{m^{\boldsymbol{j}_k} \in \Xi} s'(p)(m^{\boldsymbol{j}_k}) \leq \sum_{m^{\boldsymbol{j}_k} \in \Xi} s''(p)(m^{\boldsymbol{j}_k})$$

where we abbreviate $\Xi = \gamma^{\textit{col}}_B(\boldsymbol{j}_k(1), \ldots, \boldsymbol{j}_k(n_{\textit{col}}))$. Hence we can conclude that

$$s'(p) = \left[m^{\boldsymbol{j}_1}_1, \ldots, m^{\boldsymbol{j}_1}_{n_1}, \ldots, m^{\boldsymbol{j}_N}_1, \ldots, m^{\boldsymbol{j}_N}_{n_N}\right]$$
$$s''(p) = \left[m'^{\boldsymbol{j}_1}_1, \ldots, m'^{\boldsymbol{j}_1}_{n'_1}, \ldots, m^{\boldsymbol{j}_N}_1, \ldots, m^{\boldsymbol{j}_N}_{n'_N}\right]$$

where for all $1 \leq k \leq N$ and $1 \leq k' \leq n_k$, $m^{\boldsymbol{j}_k}_{k'}, m'^{\boldsymbol{j}_k}_{k'} \in \gamma^{col}_B(\boldsymbol{j}_k(1), \ldots, \boldsymbol{j}_k(n_{col}))$ and

$$n_k = \sum_{m^{\boldsymbol{j}_k} \in \Xi} s'(p)(m^{\boldsymbol{j}_k}), \qquad n'_k = \sum_{m^{\boldsymbol{j}_k} \in \Xi} s''(p)(m^{\boldsymbol{j}_k})$$

and hence $n_k \leq n'_k$. Let us pair up $m^{\boldsymbol{j}_k}_{k'}$ and $m'^{\boldsymbol{j}_k}_{k'}$ for all $1 \leq k \leq N$ and $1 \leq k' \leq n_k$. Since $\|s'\|_{\mathcal{N}_i;C} \leq B$ we know that $\left\|m^{\boldsymbol{j}_k}_{k'}\right\|_{col} \leq B$ and hence since $m^{\boldsymbol{j}_k}_{k'}, m'^{\boldsymbol{j}_k}_{k'} \in \gamma^{col}_B(\boldsymbol{j}_k(1), \ldots, \boldsymbol{j}_k(n_{col}))$ we can deduce that $m^{\boldsymbol{j}_k}_{k'} \leq_{\mathcal{M}^{colour}} m'^{\boldsymbol{j}_k}_{k'}$. Hence we can deduce that $s'(p) \leq_{\mathbb{M}[\mathcal{M}^{colour}]} s''(p)$ for all $p \in \mathcal{P}^C$ and thus $s' \leq_{Config} s''$. Since $s'' \leq_{Config} \boldsymbol{s}'(|\boldsymbol{s}'|)$ we can conclude that $s' \leq_{Config} \boldsymbol{s}'(|\boldsymbol{s}'|)$.

Hence $\boldsymbol{s}'$ is a path in $\mathcal{N}_i$ from s that covers s' and $|\boldsymbol{s}'| \leq dist_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s), \alpha_{i,B}(s'))$ from which we can conclude that $dist_{\mathcal{N}_i}(s, s') \leq dist_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s), \alpha_{i,B}(s'))$. □

Corollary 7. Let $i \leq n_S$, $B \in \mathbb{N}$. Then

$$d_{\mathcal{N}_i}\left(S_{(i,B)}\right) \leq \sup\left\{\rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s')) : \left\|s'\right\|_{\mathcal{N}_i;C} \leq B\right\}.$$

Proof. Let $(s, s') \in S_{(i,B)}$ then Corollary 6 gives that

$$dist_{\mathcal{N}_i}(s, s') \leq dist_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s), \alpha_{i,B}(s')) \leq \rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s'))$$

because $\alpha_{i,B}(s) \in \mathbb{N}^{\widehat{d}_{i,B}}$. Taking max over $S_{(i,B)}$:

$$\begin{aligned} d_{\mathcal{N}_i}\left(S_{(i,B)}\right) &= \max\left\{dist_{\mathcal{N}_i}(s, s') \mid (s, s') \in S_{(i,B)}\right\} \leq \max\left\{\rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s') \mid (s, s') \in S_{i,B})\right\} \\ &\leq \max\left\{\rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s')) : \left\|s'\right\|_{\mathcal{N}_i;C} < B\right\} \end{aligned}$$

which concludes the proof. □

Corollary 8. For $i \leq n_S$ and $B \in \mathbb{N}$ we have

$$d_{\mathcal{N}_i}\left(S_{(i,B)}\right) \leq \left(6 \max\{R, B, 1\} \max\left\{R', B\right\}\right)^{\left(\widehat{d}_{i,B}+1\right)!}.$$

Proof. Let s be such that $\|s\|_{\mathcal{N}_i;C} \leq B$, then Lemma 12 [BFP12, Lemma 12] applies to give us $\rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s)) \leq (6R_{\mathcal{V}^{\leq}_{i,B}} R'_{\mathcal{V}^{\leq}_{i,B}})^{(\widehat{d}_{i,B}+1)!}$.

Further it is easy to see that $R_{\mathcal{V}^{\leq}_{i,B}} \leq \max\{R, B, 1\}$ and $R'_{\mathcal{V}^{\leq}_{i,B}} \leq \max\{R', B\}$ and hence we obtain $\rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s)) \leq (6 \max\{R, B, 1\} \max\{R', B\})^{(\widehat{d}_{i,B}+1)!}$. Invoking Corollary 7 yields:

$$\begin{aligned} d_{\mathcal{N}_i}\left(S_{(i,B)}\right) &\leq \max\left\{\rho_{\mathcal{V}^{\leq}_{i,B}}(\alpha_{i,B}(s')) \mid \left\|s'\right\|_{\mathcal{N}_i;C} \leq B\right\} \\ &\leq \max_{\|s'\|_{\mathcal{N}_i;C} \leq B}((6 \max\{R, B, 1\} \max\{R', B\})^{(\widehat{d}_{i,B}+1)!}) \\ &\leq \left(6 \max\{R, B, 1\} \max\left\{R', B\right\}\right)^{(\widehat{d}_{i,B}+1)!} \end{aligned}$$

where the last inequality is justified since the argument of max does not depend on s' only on B. □

Theorem 12. *For all $i \leq n_S$ the following three inequalties hold:*

$$\rho_{\mathcal{N}_0}(s_{cov}) \leq 2 \uparrow\uparrow 2 \operatorname{slog}(48 n_C n_S n_{col} R') \tag{i}$$

$$\rho_{\mathcal{N}_{i+1}}(s_{cov}) \leq 2^{2^{((\max\{\rho_{\mathcal{N}_i}(s_{cov}),2\})^{48 n_{col} n_S n_C R'})}} \quad \textit{and} \tag{ii}$$

$$\rho_{\mathcal{N}}(s_{cov}) \leq 2 \uparrow\uparrow 2n_S + 2 \operatorname{slog}(48(n_S+1) n_{col} n_S n_C R'). \tag{iii}$$

Proof. For claim (i) applying Proposition 6 yields that

$$\rho_{\mathcal{N}_0}(s_{\text{cov}}) \leq d_{\mathcal{N}_0}\left(S_{(0,R')}\right)$$

Applying Corollary 8 then gives us:

$$\rho_{\mathcal{N}_0}(s_{\text{cov}}) \leq (6 \max\left\{R, R', 1\right\} R')^{(\widehat{d}_{i,R'}+1)!} + \rho_{\mathcal{N}_i}(s_{\text{cov}})$$

Clearly $R' = \max\{R, R', 1\}$. Hence

$$\rho_{\mathcal{N}_0}(s_{\text{cov}}) \leq (6R'^2)^{(\widehat{d}_{0,R'}+1)!} \leq 2^{\log_2(6R'^2)(\widehat{d}_{0,R'}+1)!}$$

Further

$$\log_2(6R'^2)(\widehat{d}_{0,R'}+1)! + 1 \leq (4 + 2\log_2(R'))(\widehat{d}_{0,R'}+1)! \leq (\widehat{d}_{0,R'} + 6 + R')!$$

since $4 + 2\log_2(R') \leq \widehat{d}_{0,R'} + 6 + R'$, and hence

$$\begin{aligned}\log_2(6R'^2)(\widehat{d}_{0,R'}+1)! + 1 &\leq 2^{\sum_{k=1}^{\widehat{d}_{0,R'}+6+R'} \log_2(k)} \leq 2^{(\widehat{d}_{0,R'}+6+R')^2} \\ &= 2^{((R'+1)((n_C+1)(R'+1)^{n_{col}-1}-1)+6+R')^2} = 2^{(((n_C+1)(R'+1)^{n_{col}})+5)^2}\end{aligned}$$

And

$$\begin{aligned}2\log_2(((n_C+1)(R'+1)^{n_{col}})+5) &\leq 2(\log_2(n_C+1) + n_{col}\log_2(R'+1) + \log_2(5)) \\ &\leq 2n_C + 2n_{col}R' + 6 \leq 48 n_C n_S n_{col} R'\end{aligned}$$

Putting it all together gives

$$\rho_{\mathcal{N}_0}(s_{\text{cov}}) \leq 2 \uparrow\uparrow (2 \operatorname{slog}(48 n_C n_S n_{col} R'))$$

For the second claim we know from Proposition 6 that

$$\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) \leq d_{\mathcal{N}_{i+1}}\left(S_{(i+1,B_i)}\right) + \rho_{\mathcal{N}_i}(s_{\text{cov}}),$$

where $B_i = R' \times \rho_{\mathcal{N}_i}(s_{\text{cov}})$. Corollary 8 then tells us that

$$\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) \leq (6 \max\{R, B, 1\} \max\left\{R', B\right\})^{(\widehat{d}_{i,B}+1)!} + \rho_{\mathcal{N}_i}(s_{\text{cov}})$$

Then since we know $B_i + 1 \geq \max\{R, R', 1\}$ the above implies

$$\begin{aligned}\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) &\leq (6(B_i+1)^2)^{(\widehat{d}_{i+1,B_i}+1)!} + \rho_{\mathcal{N}_i}(s_{\text{cov}}) \leq 2^{\log_2(6(B_i+1)^2)(\widehat{d}_{i+1,B_i}+1)!} + 2^{\log_2\left(\rho_{\mathcal{N}_i}(s_{\text{cov}})\right)} \\ &\leq 2^{\log_2(6(B_i+1)^2)(\widehat{d}_{i+1,B_i}+1)!+1}\end{aligned}$$

where the last line is justified since the inequality $\log_2(6(B_i+1)^2) \geq \log_2\left(\rho_{\mathcal{N}_i}(s_{\text{cov}})\right)$ holds. Further

$$\log_2(6(B_i+1)^2)(\widehat{d}_{i+1,B_i}+1)!+1 \leq (4+2\log_2(B_i+1))(\widehat{d}_{i+1,B_i}+1)! \leq (\widehat{d}_{i+1,B_i}+6+B_i)!$$

since $4 + 2\log_2(B_i+1) \leq \widehat{d}_{i+1,B_i} + 6 + B_i$, and hence

$$\log_2(6(B_i+1)^2)(\widehat{d}_{i+1,B_i}+1)! + 1 \leq 2^{\sum_{k=1}^{\widehat{d}_{i+1,B_i}+6+B_i}\log_2(k)} \leq 2^{(\widehat{d}_{i+1,B_i}+6+B_i)^2}$$

Expanding $\widehat{d}_{i+1,B_i}$ then gives

$$\begin{aligned}\widehat{d}_{i+1,B_i} + 6 + B_i &= i + (B_i+1)\left((n_C+1)(B_i+1)^{n_{col}-1} - 1\right) + 6 + B_i + 1 \\ &= i + (n_C+1)(B_i+1)^{n_{col}} + 6 \leq 2^{\log_2(n_S)} + 2^{\log_2(n_C+1)}(B_i+2)^{n_{col}} + 2^3 \\ &\leq 2^{3+\log_2(n_S)+\log_2(n_C+1)}(B_i+2)^{n_{col}} \leq (B_i+2)^{n_{col}+3+\log_2(n_S)+\log_2(n_C+1)}\end{aligned}$$

Expanding B_i

$$\begin{aligned}B_i + 2 &\leq R' \times \rho_{\mathcal{N}_i}(s_{\text{cov}}) + 2 \leq (R'+1)\max\left\{\rho_{\mathcal{N}_i}(s_{\text{cov}}), 2\right\} \leq 2^{\log_2(R'+1)}\max\left\{\rho_{\mathcal{N}_i}(s_{\text{cov}}), 2\right\} \\ &\leq \left(\max\left\{\rho_{\mathcal{N}_i}(s_{\text{cov}}), 2\right\}\right)^{\log_2(R'+1)+1}\end{aligned}$$

Putting it all together

$$\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) \leq 2^{2^{\left(\left(\left(\max\left\{\rho_{\mathcal{N}_i}(s_{\text{cov}}),2\right\}\right)^{\log_2(R'+1)+1}\right)^{n_{col}+3+\log_2(n_S)+\log_2(n_C+1)}\right)^2}}.$$

And we can see

$$\begin{aligned}2\left(\log_2(R'+1)+1\right)\left(n_{col}+3+\log_2(n_S)+\log_2(n_C+1)\right) &\leq 2\left(R'+1\right)(3+n_{col}+n_S+n_C) \\ &\leq 48 n_{col} n_S n_C R'\end{aligned}$$

since $n_{col}, n_S, n_C, R' \geq 1$ and hence we can conclude

$$\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) \leq 2^{2^{\left(\left(\max\left\{\rho_{\mathcal{N}_i}(s_{\text{cov}}),2\right\}\right)^{48 n_{col} n_S n_C R'}\right)}}.$$

We can prove claim (iii) by a simple induction. The base case is given by (i). The inductive step is:

$$\begin{aligned}\rho_{\mathcal{N}_{i+1}}(s_{\text{cov}}) &\leq 2^{2^{\left(\max\left\{\rho_{\mathbb{B},[\mathcal{N}_i]2^{(}}\right\}\right)^{48 n_{col} n_S n_C R'}}} \leq 2^{2^{\left(2 \uparrow\uparrow (2\cdot i + 2\,\text{slog}(48(i+1)n_{col}n_S n_C R'))\right)^{48 n_{col} n_S n_C R'}}} \\ &\leq 2 \uparrow\uparrow (2\cdot(i+1) + 2\,\text{slog}(48(i+2)n_{col}n_S n_C R'))\end{aligned}$$

which concludes the proof. □

Corollary 9. Coverability for NNCTs is decidable and in Tower.

Proof. Let $\mathcal{N}$ be an NNCT with n_S places, n_C complex places, n_{col} colours and rules $\mathcal{R}$ and a coverability query Q giving rise to a bound R' and

$$B(n_S, n_C, n_{col}, R') = 2 \uparrow\uparrow (2n_S + 2 \operatorname{slog}(48(n_S + 1)n_{col}n_S n_C R')).$$

Then the theorem above tells us that along a covering path a simple place cannot contain more than $R \cdot B(n_S, n_C, n_{col}, R')$ tokens, a complex tokens cannot have more that $R \cdot B(n_S, n_C, n_{col}, R')$ tokens of a particular colour and there are at most $R \cdot B(n_S, n_C, n_{col}, R')$ complex tokens. Hence along a covering path simple place can be represented using $\log_2(R \cdot B(n_S, n_C, n_{col}, R'))$ bits, the state of complex token can be represented using $n_{col} \log_2(R \cdot B(n_S, n_C, n_{col}, R'))$ bits and hence a complex place may be represented by $n_C n_{col} R \cdot B(n_S, n_C, n_{col}, R') \log_2(R \cdot B(n_S, n_C, n_{col}, R'))$ bits. Hence a non-deterministic Turing machine requiring at most $O(B(n_S, n_C, n_{col}, R'))$ space can decide the coverability problem. Using Savitch's theorem we know there is a deterministic Turing machine deciding coverability in $O(B(n_S, n_C, n_{col}, R')^2)$ space and using an exponential to obtain a time bounded Turing machine we find that the coverability problem can be decided in time $O(2^{B(n_S, n_C, n_{col}, R')^2})$ and is thus clearly in Tower.

□

C.4 Proofs for Section 5.4

Theorem 13. *Simple coverability, boundedness, and termination for total-transfer NNCT is* Tower*-hard.*

Proof. We can deduce Tower-hardness by showing that given a deterministic bounded two-counter machine $\mathcal{M}$ of size n with counters that are $2 \uparrow\uparrow n$-bounded, we can construct an NNCT $\mathcal{N}_{\mathcal{M}}$ in polynomial-time that weakly bisimulates $\mathcal{M}$. The machine $\mathcal{M}$ can use the following operations: x++, x−−, *reset*(x), *iszero*(x), and *ismax*(x) for each counter x.

The NNCT $\mathcal{N}_{\mathcal{M}}$. Each simulation state of $\mathcal{N}_{\mathcal{M}}$ represents a valuation v of $6n + 2$ *active* and *inactive* counters, and n arrays. In addition to the counters x, y of $\mathcal{M}$, the NNCT $\mathcal{N}_{\mathcal{M}}$ simulates the auxiliary counters s_i, p_i, p'_i, c_i, c'_i and an auxiliary array a_i for each $i \leq n$. To simplify notation, let us write C_i for the set of counters $\{s_i, p_i, p'_i, c_i, c'_i\}$ when $i < n$, and C_n for the set of counters $\{s_n, p_n, p'_n, c_n, c'_n, x, y\}$. Each active counter $d \in C_i$ is $2 \uparrow\uparrow i$-bounded, each inactive counter has an undefined value. For each i the array a_i has length exactly $2 \uparrow\uparrow i + 1$ and carries values $a_i(j) \in \{0, 1, 2\}$ for $j \leq 2 \uparrow\uparrow i$. The NNCT $\mathcal{N}_{\mathcal{M}}$ has two simple places $p[d]$ and $\overline{p}[d]$ for each counter $d \in C_i$, three complex places $p[a_i]$, $\overline{p}[a_i]$, and $p[aux_i]$, and two colours $p[j_i]$ and $\overline{p}[j_i]$ for each array a_i. Further, $\mathcal{N}_{\mathcal{M}}$ has a complex *'sink'* place $p[disc]$. The colour mapping ζ of $\mathcal{N}_{\mathcal{M}}$ maps $p[j_i]$ to $p[p'_i]$ and $\overline{p}[j_i]$ to $\overline{p}[p'_i]$. Lastly, $\mathcal{N}_{\mathcal{M}}$ has a number (polynomial in n) of simple places encoding the control of $\mathcal{M}$ and the internal control of $\mathcal{N}_{\mathcal{M}}$. Further, $\mathcal{N}_{\mathcal{M}}$'s transfer rules will all be total, hence $\mathcal{N}_{\mathcal{M}}$ will be a total-transfer NNCT.

Representing a Valuation. A valuation v is represented by a configuration s as follows:

- For each i and $d \in C_i$, if d is active then there are exactly $v(d)$ •-tokens in $p[d]$ and $2 \uparrow\uparrow i - v(d)$ •-tokens in $\overline{p}[d]$.
- For each i and $d \in C_i$, if d is inactive then both places $p[d]$, $\overline{p}[d]$ are empty.
- For each i the array a_i is represented as follows. For each $k \leq 2 \uparrow\uparrow i$, let $m_{(i,k)}$ be a complex token such that $m_{(i,k)}$ contains exclusively tokens of colours $p[j_i]$ and $\overline{p}[j_i]$: k tokens of colour $p[j_i]$ and $2 \uparrow\uparrow i - k$ tokens of colour $\overline{p}[j_i]$. Then, to represent a_i, there are exactly $v(a_i(k))$ complex tokens $m_{(i,k)}$ in $p[a_i]$ and $2 - v(a_i(k))$ complex tokens $m_{(i,k)}$ in $\overline{p}[a_i]$.
- For each i the place $p[aux_i]$ is empty.

Reducing from $\mathcal{M}$'s Halting Problem. The question whether $\mathcal{M}$ halts, i.e. whether $\mathcal{M}$ reaches a halting control state from its initial state, can then be answered by performing a simple coverability query on $\mathcal{N}_{\mathcal{M}}$'s simple places encoding $\mathcal{M}$'s finite control. Assuming that only $\mathcal{M}$'s halting control states have no successors, $\mathcal{M}$'s halting problem also reduces to the termination problem for $\mathcal{N}_{\mathcal{M}}$. Augmenting $\mathcal{N}_{\mathcal{M}}$ with an additional simple place that is incremented with every transition shows that the halting problem for $\mathcal{M}$ reduces to deciding boundedness of $\mathcal{N}_{\mathcal{M}}$.
Let $d \in \bigcup_{i \in \langle n \rangle} C_i$ we will implement operation d++ as follows

> *Add a •-token to $p[d]$*
> *Remove a •-token from $\overline{p}[d]$*

Suppose d is $2 \uparrow\uparrow i$-bounded. Clearly if configuration s represents valuation v, d is active and $v(d) < 2 \uparrow\uparrow i$ then there are $v(d)$ •-tokens in $p[d]$ and $2 \uparrow\uparrow i - v(d)$ •-tokens in $\overline{p}[d]$. Hence there is at least one •-token in $\overline{p}[d]$. Performing the operation d++ yields a new configuration

Move a complex-token from $\overline{p}[a_i]$ *to* $p[\mathit{aux}_i]$*;*
deactivate(p'_i);
Eject the contents of a complex-token m *in* $p[\mathit{aux}_i]$ *and its remains into* $p[\mathit{disc}]$*;*
isequal(p_i, p'_i); *reset*(p'_i);
Create an empty complex-token in $p[\mathit{aux}_i]$*;*
while $(p_i \neq p'_i)$ **do**
  Inject a $p[j_i]$*-coloured token into a complex token in* $p[\mathit{aux}_i]$*;*
  p'_i++;
while $(\neg(\mathit{ismax}(p'_i)))$ **do**
  Inject a $\overline{p}[j_i]$*-coloured token into a complex token in* $p[\mathit{aux}_i]$*;*
  p'_i++;
Move a complex-token from $p[\mathit{aux}_i]$ *to* $p[a_i]$*;*
reset(p'_i);

Figure C.1: The implementation of $a_i(p_i)$++. Swapping $p[a_i]$ for $\overline{p}[a_i]$ in Line 1 and $\overline{p}[a_i]$ for $p[a_i]$ in Line 12 yields an implementation of $a_i(p_i)$−−.

s' where there are $v(d)+1$ •-tokens in $p[d]$ and $2 \uparrow\uparrow i - (v(d)+1)$ •-tokens in $\overline{p}[d]$ and all other (non-control) places are unchanged. The configuration s' then represents the valuation $v\,[d \mapsto v(d)+1]$. Further if $v(d) = 2 \uparrow\uparrow i$ we note $\overline{p}[d]$ is empty and thus the simulation of d++ blocks at the attempt to remove a token from $\overline{p}[d]$.
We note that we can implement d−− by

Remove a •*-token from* $p[d]$
Add a •*-token to* $\overline{p}[d]$

Similarly to the reasoning on d++ we can see that if configuration s represents valuation v, d is active and $v(d) > 0$ we can successfully simulate d−− and obtain a configuration s' that represents the valuation $v\,[d \mapsto v(d)-1]$. However if $v(d) = 0$ simulating d−− will get stuck.

Auxilliary Operations. In addition to $\mathcal{M}$'s operations, we implement further instructions to simplify and improve readability. The NNCT $\mathcal{N}_\mathcal{M}$ will simulate *activate*(d), *deactivate*(d) for $d \in \bigcup_{i\in\langle n\rangle} C_i$, and operations *reset*$(d)$, *iszero*$(d)$, and *ismax*$(d)$ for $d \in \bigcup_{i\in\langle n\rangle} C_i \setminus \{s_i\}$.

Further for each $i \in \langle n\rangle$, we implement *ismax*&*reset*(s_i) and the counter-specialised operations:

isequal(p_i, p'_i), $a_i(p_i)$++, $a_i(p_i)$−−, *reset*$(a_i(p_i))$, *iszero*$(a_i(p_i))$, *ismax*$(a_i(p_i))$, *activate*(a_i).

All above operations are only guaranteed to succeed if the counter in question is active at the start of the operation.

An Inductive Construction. The counters in C_1 are 2-bounded so implementing operations on them is trivial. For $i < n$, operations on a_i are simulated using s_i, p_i, p'_i, c_i, c'_i and operations on s_{i+1}, p_{i+1}, p'_{i+1}, c_{i+1}, c'_{i+1} are simulated using operations on p_i, p'_i, c_i, c'_i and a_i. A cornerstone of our encoding scheme is the capability to represent arrays and operate on arrays. They are a key ingredient of our implementation of the operation *ismax*&*reset*(s_{i+1}) which is central in the Lipton-style construction. Thus, we start with the array operations.

```
for p_i := 0 to 2 ↑↑ i do
  for z := 0 to 1 do
    Create an empty complex-token in p[aux_i];
    reset(p'_i);
    while (p_i ≠ p'_i) do
      Inject a p[j_i]-coloured token into a complex token in p[aux_i];
      p'_i++;
    while (¬(ismax(p'_i))) do
      Inject a p̄[j_i]-coloured token into a complex token in p[aux_i];
      p'_i++;
    Move a complex-token from p[aux_i] to p̄[a_i];
```

Figure C.2: The implementation of *activate*(a_i).

(i) In Figure C.1 we show how to implement $a_i(p_i)$++. The operation $a_i(p_i)$++ can only succeed if p_i, p_i' are active.

Suppose $a_i(p_i)$++ is executed in a configuration s that represents valuation v and p_i, p_i' are active. If $v(a_i(v(p_i))) < 2$ then we know there exists a complex token $m_{(i,v(p_i))}$ in $\overline{p}[a_i]$ and all complex tokens in $\overline{p}[a_i]$ are of the form $m_{(i,k)}$ for some $k \leq 2 \uparrow\uparrow i$. The move of one $m_{(i,k)}$ complex token from $\overline{p}[a_i]$ to $p[aux_i]$ results in $p[aux_i]$ containing just $m_{(i,k)}$, since by assumption $p[aux_i]$ was empty before. After deactivating p_i' we know that both $p[p_i']$ and $\overline{p}[p_i']$ are empty. Ejecting the contents of $m_{(i,k)}$ removes $m_{(i,k)}$ from $p[aux_i]$, inserts k $\bullet$-tokens into $p[p_i']$, $2 \uparrow\uparrow i - k$ $\bullet$-tokens into $\overline{p}[p_i']$, and places the remaining (now empty) complex token into $p[disc]$. We can see that, disregarding a_i, the configuration we have reached represents a partial valuation v' that sets $v'(p_i') = k$ and $v'(p_i) = v(p_i)$. After executing *isequal*(p_i, p_i') the simulation only succeeds if $k = v(p_i)$. Hence $\overline{p}[a_i]$ now contains $2 - (v(a_i(v(p_i))) + 1)$ complex tokens $m_{(i,v(p_i))}$ and the same number of other complex tokens $m_{(i,k)}$. The two following while loops carefully inject $p[j_i]$-coloured tokens and $\overline{p}[j_i]$-coloured tokens into the newly created token at $p[aux_i]$ to yield a new $m_{(i,v(p_i))}$ located in $p[aux_i]$ which we move to $p[a_i]$. Thus $p[a_i]$ now contains $v(a_i(v(p_i))) + 1$ complex tokens $m_{(i,v(p_i))}$ and the same number of other complex tokens $m_{(i,k)}$ as before. The configuration s' we have reached thus represents a valuation v'' such that $v''(a_i(j)) = v(a_i(j))$ for all $0 \leq j \leq 2 \uparrow\uparrow i$ and $j \neq v(p_i)$, and $v''(a_i(v(p_i))) = v(a_i(v(p_i))) + 1$. Considering the case that $v(a_i(v(p_i))) = 2$ initially, we see that the simulation either blocks while attempting to move a complex token $m_{(i,k)}$ from $\overline{p}[a_i]$ to $p[aux_i]$ or on the execution of *isequal*(p_i, p_i') since it is impossible to obtain $k = v(p_i)$. By swapping $p[a_i]$ and $\overline{p}[a_i]$ in Figure 5.6 we obtain an implementation of $a_i(p_i)$--.

(ii) In Figure C.2 we show how to implement the operation *activate*(a_i) which can only succeed if both counters p_i and p_i' are active.

The interior for-loop simply repeats its body twice and can be considered syntactic sugar. Suppose *activate*(a_i) is invoked in a configuration s such that in s the places $p[a_i]$ and $\overline{p}[a_i]$ are empty. We can then follow our analysis in the second part of $a_i(p_i)$++'s implementation to see that the interior for-loop places two complex tokens $m_{i,k}$ into $\overline{p}[a_i]$ for all $k \leq 2 \uparrow\uparrow i$. Hence when the outer for-loop terminates we have reached a configuration representing a valuation v

such that $v(a_i(k)) = 0$ for all k.

(iii) The following shows how to implement *iszero*$(a_i(p_i))$.

$a_i(p_i)$++; $a_i(p_i)$++;
$a_i(p_i)$--; $a_i(p_i)$--;

Suppose *iszero*$(a_i(p_i))$ is executed in a configuration s that represents valuation v and p_i, p_i' are active. If $v(a_i(v(p_i))) = 0$ we can clearly execute $a_i(p_i)$++ twice and undo the former by executing $a_i(p_i)$-- twice. We thus end up at a configuration representing v. If however $v(a_i(v(p_i))) > 0$ then either $v(a_i(v(p_i))) = 2$ initially or after the first invocation of $a_i(p_i)$++ and our analysis above assures us that the subsequent invocation of $a_i(p_i)$++ blocks. Hence *iszero*$(a_i(p_i))$ only succeeds if $v(a_i(v(p_i))) = 0$. We obtain an implementation of *ismax*$(a_i(p_i))$ by swapping $a_i(p_i)$++ for $a_i(p_i)$-- and vice versa. Analogous reasoning to above then yields that *iszero*$(a_i(p_i))$ only succeeds if $v(a_i(v(p_i))) = 2$.

(iv) The following shows how to implement *reset*$(a_i(p_i))$.

while $(a_i(p_i) \neq 0)$ **do**
  $a_i(p_i)$--

It is trivial to see that the while loop only terminates once a configuration is reached representing a valuation v that sets $v(a_i(v(p_i))) = 0$. In every iteration $v(a_i(v(p_i)))$ is decremented by 1, so termination is ensured.

(v) The following shows how to implement *ismax*&*reset*(s_{i+1}).

for $p_i := 0$ **to** $2 \uparrow\uparrow i$ **do** (*reset*$(a_i(p_i))$;)
while (*iszero*$(a_i(2 \uparrow\uparrow i))$) **do**
  s_{i+1}--; *reset*(p_i); $a_i(p_i)$++;
  while *ismax*$(a_i(p_i))$ **do**
    $a_i(p_i)$--; p_i++; $a_i(p_i)$++;

We know that after the for-loop a_i is the array such that $a_i(x) = 0$ for all $x \leq 2 \uparrow\uparrow i$. The array a_i is meant to be binary representation of a number between 0 and $2 \uparrow\uparrow (i+1)$. This number is initially 0 and we can see that the outer while loop performs a long addition of 1 for each iteration. If $v(a_i(v(p_i))) = 2$ then $v(p_i)$ is an index representing a carry bit in the long addition computation. For each number represented by a_i we perform s_{i+1}--. Hence, if initially $v(s_{i+1}) = 2 \uparrow\uparrow (i+1)$ then, after performing *ismax*&*reset*(s_{i+1}), we reach a configuration that represents a valuation v' such that $v'(s_{i+1}) = 0$. However, if $v(s_{i+1}) < 2 \uparrow\uparrow (i+1)$ holds initially, then after $v(s_{i+1})$ iterations the resulting valuation v' would set $v'(s_{i+1}) = 0$ and a_i would represent the number $v(s_{i+1})$. Since $v(s_{i+1}) < 2 \uparrow\uparrow (i+1)$ this implies that $a_i(2 \uparrow\uparrow i) = 0$ and hence the body of the outer while loop is executed again leading to an invocation of s_{i+1}-- which will block. Hence *ismax*&*reset*(s_{i+1}) will block when executed in a configuration representing v such that $v(s_{i+1}) < 2 \uparrow\uparrow (i+1)$.

We modify the implementation of *ismax*&*reset*(s_{i+1}) by replacing s_{i+1}-- by *Add a* $\bullet$*-token to* $\overline{p}[d]$ we obtain an implementation of *activate*(d). If d is inactive then $p[d]$ and $\overline{p}[d]$ are empty. By an analogous argument to above we can see that invoking *activate*(d) will add $2 \uparrow\uparrow (i+1)$ $\bullet$-token to $\overline{p}[d]$ yielding a configuration where d is active and is valued 0. We can similarly obtain an implementation for *deactivate*(d). First modify *ismax*&*reset*(s_{i+1}) by replacing s_{i+1}-- by *Remove a* $\bullet$*-token from* $\overline{p}[d]$ which gives us an implementation of an intermediate operation *deactivate*$'(d)$. If d is active and valued 0 then *deactivate*$'(d)$ clearly succeeds and removes all

tokens from $\overline{p}[d]$. Hence we can implement *deactivate*(d) by *reset*(d); *deactivate*$'(d)$; which succeeds if d is active.

(vi) The following shows how to implement *iszero*(d) for $d \in C_{i+1} \setminus \{s_{i+1}\}$ and in the case that $i+1 = n$: $d = x$ or y. The precondition for *iszero*(d) is that s_{i+1} is 0.

while $(*)$ **do** $(d{+}{+}; s_{i+1}{+}{+};)$; *ismax&reset*$(s_{i+1})$;
while $(*)$ **do** $(d{-}{-}; s_{i+1}{+}{+};)$; *ismax&reset*$(s_{i+1})$

Note that the only operations that modify s_{i+1} are *iszero*(d) and *ismax&reset*(s_{i+1}). Further, *ismax&reset*(s_{i+1}) is only invoked by *iszero*(d). Thus, except when executing *iszero*(d) we maintain that the counter s_{i+1} is 0.

Suppose *iszero*(d) is started in a configuration s_0 that represents valuation v_0 such that $v_0(s_i) = 0$. After the first while-loop non-deterministically terminates, just before invoking *ismax&reset*(s_{i+1}) we reach a configuration that represents a valuation v such that $v(s_{i+1}) = k$ and $v(d) = v_0(d) + k$. Since neither $d{+}{+}$ nor $s_{i+1}{+}{+}$ blocked we know that $0 \leq k \leq 2 \Uparrow (i+1) - v_0(d)$. In fact for all $0 \leq k \leq 2 \Uparrow (i+1) - v_0(d)$ it is possible to reach such a configuration where $v(s_{i+1}) = k$ and $v(d) = v_0(d) + k$. Invoking *ismax&reset*(s_{i+1}) only succeeds if $k = 2 \Uparrow (i+1)$. Thus all other configurations where $k < 2 \Uparrow (i+1)$ *ismax&reset*(s_{i+1}) blocks. Hence if $v_0(d) > 0$ it will be the case that $k < 2 \Uparrow (i+1)$ and thus *ismax&reset*(s_{i+1}) blocks. If $v_0(d) = 0$ then there is one configuration that we can reach which represents a valuation v_1 such that $v_1(s_{i+1}) = 2 \Uparrow (i+1)$, $v_1(d) = v_0(d) + 2 \Uparrow (i+1) = 2 \Uparrow (i+1)$ and we can successfully invoke *ismax&reset*(s_{i+1}) to yield a new configuration representing a valuation v_2 $v_2(s_{i+1}) = 0$, $v_2(d) = 2 \Uparrow (i+1)$. The second while loop then again non-deterministically decrements d and increments s_{i+1} in the same fashion as above. Again similar reasoning to above yields that there is only one configuration that we can reach that represents a valuation v_3 which values $v_3(s_{i+1}) = 2 \Uparrow (i+1)$ and consequently $v_3(d) = 0$ and which guarantees *ismax&reset*(s_{i+1}) succeeds to yield a configuration that represents a valuation v_4 such that $v_4(s_{i+1}) = v_4(d) = 0$. Hence *iszero*$(d)$ succeeds only if started in a configuration that values d as 0.

We can modify the above to obtain an implementation *ismax*(d). We simply swap the first for the second while loop and analogous reasoning to *iszero*(d) then yields that *ismax*(d) succeeds only when d is valued at $2 \Uparrow (i+1)$ and s_{i+1} at 0.

(vii) The following shows how to implement *isequal*(p_i, p_i') for $d \in C_{i+1} \setminus \{s_{i+1}\}$ if both p_i and p_i' are active and c_i is zero.

while $\big(p_i \neq 0$ **or** $p_i' \neq 0\big)$ **do**
  $p_i{-}{-}$; $p_i'{-}{-}$; $c_i{+}{+}$;
while $(c_i \neq 0)$ **do**
  $p_i{+}{+}$; $p_i'{+}{+}$; $c_i{-}{-}$;

If we invoke *isequal*(p_i, p_i') in a configuration that represents a valuation v such that $v(p_i) = k$, $v(p_i') = k'$ and $v(c_i) = 0$ then for each iteration of the first while loop we obtain a configuration representing a valuation v' such that $v'(p_i) = k - v'(c_i)$, $v'(p_i') = k' - v'(c_i)$. The inequality $v'(c_i) \leq \min(k, k')$ must hold at the end of each iteration as otherwise either $p_i{-}{-}$ or $p_i'{-}{-}$ would have blocked. Since the loop-guard is false only if both $v'(p_i) = 0$ and $v'(p_i') = 0$ we can see that the while loop only terminates successfully once $v'(c_i) = k = k'$ i.e. if initially $v(p_i) = v(p_i')$. Otherwise the loop guard would never be false and thus at some point the invocation of $p_i{-}{-}$

or p_i'-- blocks. Thus after the successful breaking out of the while loop $v'(c_i) = v(p_i) = v(p_i')$ and it is easy to see that the second while loop simply copies the contents of c_i back to p_i and p_i'. Hence invoking *isequal*(p_i, p_i') only succeeds if started in a configuration where $v(p_i) = v(p_i')$ and $v(c_i) = 0$.

(viii) The following shows how to implement *reset*(d) for $d \in C_{i+1} \setminus \{s_{i+1}\}$.

while $(d \neq 0)$ **do**
  d--;

As in the implementation of *reset*$(a_i(p_i))$ the while-loop simply decrements d until d has value 0. Hence *reset*(d) works as expected.

Lastly, in order to set up $\mathcal{N}_\mathcal{M}$ to simulate the initial configuration of $\mathcal{M}$ from its initial state, we set $\mathcal{N}_\mathcal{M}$ to execute *activate*$(-)$ for all counters and arrays in turn, in order of the bound size and length:

activate(s_1); *activate*(p_1); *activate*(p_1');
activate(c_1); *activate*(c_1'); *activate*(a_1);
...
activate(s_{n-1}); *activate*(p_{n-1}); *activate*(p_{n-1}');
activate(c_{n-1}); *activate*(c_{n-1}'); *activate*(a_{n-1});
activate(s_n); *activate*(p_n); *activate*(p_n'); *activate*(c_n);
activate(c_n'); *activate*(x); *activate*(y);

After executing these operations we reach a configuration s in which all counters are active and which represents a valuation v such that $v(d) = 0$ for all counters d and $v(a_i(k)) = 0$ for all $0 \leq k \leq 2 \uparrow\uparrow i$. From this point on, $\mathcal{N}_\mathcal{M}$ weakly bisimulates $\mathcal{M}$. Thus, we can reduce the halting problem for $\mathcal{M}$ to simple coverability, termination, or boundedness of $\mathcal{N}_\mathcal{M}$.

□