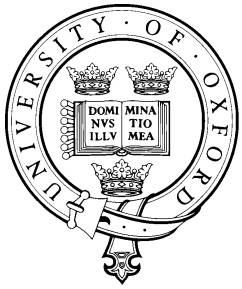
Oxford University Computing Laboratory

Model Checking Systems with Replicated Components using CSP

Tomasz Mazur
St Catherine's College



A thesis submitted for the degree of
Doctor of Philosophy

Trinity Term, 2010

Declaration of Authorship

I hereby confirm that, unless otherwise stated, the work presented in this thesis is my own, except for the following, which should be credited to Gavin Lowe:

- Figures 4.2, 4.3 and 5.1;
- Examples 4.2.2, 4.3.3, 5.2.4, and 5.2.29;
- Remark 5.2.17;
- the basic structure of the proof of Theorem 5.2.24;
- the construction used in the proof of Theorem 6.1.3;
- Lemma 6.3.7.

In addition, the core ideas of Chapter 3 were published, in a simplified version, in a paper co-authored with Gavin Lowe [ML09]. My input into the paper included most of the technical development, with guidance from Gavin Lowe, who also contributed some of the text.

To Hanna,
for making me laugh.

Model Checking Systems with Replicated Components using CSP

Tomasz Mazur
St Catherine's College, University of Oxford

A thesis submitted for the degree of *Doctor of Philosophy*
Trinity Term, 2010

The Parameterised Model Checking Problem asks whether an implementation $Impl(t)$ satisfies a specification $Spec(t)$ for all instantiations of parameter t. In general, t can determine numerous entities: the number of processes used in a network, the type of data, the capacities of buffers, etc. The main theme of this thesis is automation of uniform verification of a subclass of PMCP with the parameter of the first kind, using techniques based on counter abstraction. Counter abstraction works by counting how many, rather than which, node processes are in a given state: for nodes with k local states, an abstract state $(c_1, \ldots, c_k)$ models a global state where c_i processes are in the i-th state. We then use a threshold function z to cap the values of each counter. If for some i, counter c_i reaches its threshold, z_i, then this is interpreted as there being z_i *or more* nodes in the i-th state. The addition of thresholds makes abstract models independent of the instantiation of the parameter.

We adapt standard counter abstraction techniques to concurrent reactive systems modelled using the CSP process algebra. We demonstrate how to produce abstract models of systems that do not use node identifiers (i.e. where all nodes are indistinguishable). Every such abstraction is, by construction, refined by all instantiations of the implementation. If the abstract model satisfies the specification, then a positive answer to the particular uniform verification problem can be deduced.

We show that by adding node identifiers we make the uniform verification problem undecidable. We demonstrate a sound abstraction method that extends standard counter abstraction techniques to systems that make full use of node identifiers (in specifications and implementations). However, on its own, the method is not enough to give the answer to verification problems for all parameter instantiations. This issue has led us to the development of a type reduction theory, which, for a given verification problem, establishes a function ϕ that maps all (sufficiently large) instantiations T of the parameter to some fixed type $\hat{T}$ and allows us to deduce that if $Spec(\hat{T})$ is refined by $\phi(Impl(T))$, then $Spec(T)$ is refined by $Impl(T)$. We can then combine this with our extended counter abstraction techniques and conclude that if the abstract model satisfies $Spec(\hat{T})$, then the answer to the uniform verification problem is positive.

We develop a symbolic operational semantics for CSP processes that satisfy certain normality requirements and we provide a set of translation rules that allow us to concretise symbolic transition graphs. The type reduction theory relies heavily on these results. One of the main advantages of our symbolic operational semantics and the type reduction theory is their generality, which makes them applicable in other settings and allows the theory to be combined with abstraction methods other than those used in this thesis.

Finally, we present TomCAT, a tool that automates the construction of counter abstraction models and we demonstrate how our results apply in practice.

Acknowledgements

Writing a doctoral thesis is undoubtedly a one-of-a-kind experience. It can often be overwhelming and stressful. I feel the need to thank the people who contributed to the successful completion of my adventure as a graduate student.

First and foremost, I would like to thank Gavin Lowe, my tutor and supervisor. K. Patricia Cross once said:

> *The task of the excellent teacher is to stimulate "apparently ordinary" people to unusual effort. The tough problem is not in identifying winners: it is in making winners out of ordinary people.*

To Gavin I owe achieving something I never imagined I could be capable of. Gavin's constant support on technical, stylistic and personal matters has been invaluable. No problem has ever been too small or too big to discuss. Every time I left Gavin's office I left inspired and willing to work even harder. I could not have asked for a better advisor.

My journey through the world of model checking began with an undergraduate course on concurrency at Oxford. It was the encouragement and patience of Philippa Hopcroft that allowed me to see the beauty of the subject. I would like to express my gratitude for all the time she took to teach me CSP.

I would like to thank Joël Ouaknine, Bill Roscoe, Thomas Wahl and especially Ben Worrell for many technical discussions and useful comments on my work.

Thanks to Marta Kwiatkowska, Ranko Lazić, Joël Ouaknine and Ben Worrell for acting as the examiners at various stage of my D.Phil. degree.

Thanks to the anonymous reviewers of [ML09] and [Maz08]. Their feedback greatly helped me to improve clarity.

I would like to thank Richard Thompson for choosing my work as the basis for his final year undergraduate project and for reporting bugs in TomCAT.

The work presented in this thesis has been supported by an EPSRC grant, as well as scholarships from the Letherseller's Company and the Misys Charitable Foundation, for all of which I am extremely grateful.

I want to thank my parents for their love, for teaching me the value of hard work and for giving me every chance to excel in my life.

Finally, special thanks to my wonderful fiancée, Hanna, who has contributed to the creation of this thesis more than one could possibly imagine. Not only did she take excellent care of me, allowing me to focus on my work, but her constant love, belief in me and words of encouragement in times of crisis allowed me not to give up somewhere along the way. I hope that one day I will get a chance to show her how much I appreciate what she has done for me.

"Computer Science is a science of abstraction — creating the right model for a problem and devising the appropriate mechanizable techniques to solve it."

Alfred Aho and Jeffrey Ullman

Contents

Chapter 1

Introduction

1.1 Motivation

Even though we would not tolerate regular failures of mechanical devices, we seem to have come to terms with crashing operating systems, electronic mail not being delivered, power outages, "secure" areas being broken into, etc. Computer systems are, by far, the most complex creation of mankind. This is why errors within them are all too common. Up until recently the primary method of correctness verification was *testing*, which, given an input, checks the produced output against the expected outcome. This approach suffers from two main problems. Firstly, it is almost always impossible to test every possible input and execution path. Secondly, testing works only for completed implementations. This makes it particularly unsuitable for verification of safety-critical systems; it is highly unlikely that someone would ever want to perform testing to verify that a nuclear power plant never blows up, for example.

In contrast to the above, formal verification methods concentrate on *proving* the correctness of a given system. This is achieved using either *theorem proving* or *model checking* (see Section 1.3). In the former, a mathematical proof about what the system can or cannot do is constructed (a task which usually requires a lot of ingenuity from the user). In the latter, a model of the system is built and its state space is exhaustively explored with each state verified against a given specification. However, both of these methods are very tedious when performed by hand. This is why computers have been employed in automating these tasks.

Computer-Aided Formal Verification has been an active research field since the beginning of 1980's. It is possible to identify a number of reasons (some of which overlap) that have caused an elevated interest in this area of theoretical computer science in the last two decades. The following are the main ones.

Scalability With the exponential growth of the size of computer systems, hardware manufacturers and software houses realised that the traditional approach of creating implementations and running suites of test cases does not scale very well. With the ever-increasing demand for better quality products and ever-decreasing time allowances for testing, alternative workflow solutions had to be investigated.

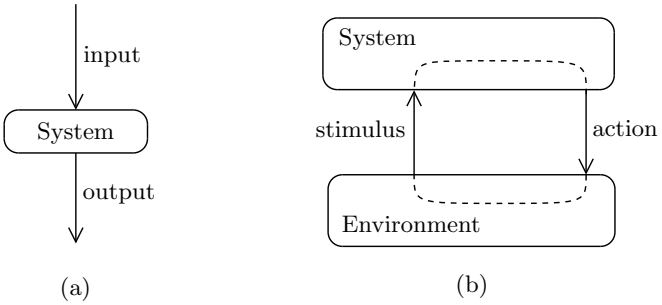


Figure 1.1: Behavioural difference between: (a) computation programs and (b) reactive systems.

Safety-critical systems There is a widespread use of computer systems in different areas of human life (e.g. medicine, aviation and astronautics, entertainment, infrastructure, transportation and power production), where a failure of such system could result in large-scale consequences. The Therac-25 radiation therapy machine accidents [LT93], Ariane 5 rocket explosion [Lan96, Kun97], London Ambulance Service computer-aided dispatch system crash [van00] and the 2003 Blackout in the USA and Canada [AE04] are probably the best-known examples of failures of critical systems caused by insufficient testing.

Security protocols Computer networks have become an essential part of human lives. More and more often they are used in situations where privacy, secrecy and authentication are of utmost importance. Security protocols are some of the shortest programs available and, at the same time, some of the most difficult to analyse for correctness. Not uncommon are cases where a protocol had been regarded as fully secure for a number of years and then an attack was found using formal methods [Low96].

1.2 Concurrent reactive systems

A fundamental question that can be asked in computer science is "What is a computer program?". One answer would be that it is a function $f : I \to O$ defined on a set of inputs I and with return values in a set O. This answer is, however, only partially true, as it does not say anything about how the program interacts with its environment. Even though the sole purpose of a lot of programs is performing computations, returning the result and terminating, there is an equally large (if not larger) group of systems, which, in fact, never terminate. Their task is a constant *interaction* with the environment, i.e. responding to *stimuli* with *actions*. Examples of such systems include Internet routers, operating systems, telephone networks, sensors, and microcode in domestic appliances. Figure 1.1 shows the difference between reactive and non-reactive systems based on the type of their interaction with the environment.

Concurrency is the study of simultaneous execution and interaction of multiple programs and has recently become a hugely important topic. The behaviour of systems

like networks, transportation infrastructure, protocols, databases, multicore chips or distributed algorithms is most naturally described by models composed of a number of interacting processes.

For the purpose of this thesis, we propose the following definition (based on the definitions from [HP85] and [MP95]).

Definition 1.2.1. A physical or logical entity is called a *concurrent synchronous reactive system* if it satisfies the following two properties:

(i) it is designed to respond to external stimuli with actions affecting the environment within which it operates, and

(ii) it is composed of one or more processes, possibly interacting with each other and co-existing in the same time frame.

1.3 Theorem proving and model checking

In general, there are two main approaches to formal verification of concurrent reactive systems: *theorem proving* and *model checking*.

Theorem proving is a deductive method, where the fact that an implementation is satisfied by a specification is expressed as a formula in some logic. Then, the validity of the formula (theorem) has to be proven. Such proofs would normally have to be performed by hand, but large parts can, in fact, be automated. The biggest advantage of theorem proving is its applicability to a vast range of systems (including infinite-state ones). However, a lot of problems require substantial user ingenuity in order to construct the necessary proofs and the scope for automation is, therefore, limited. We do not use theorem proving in this thesis, but a good introduction to the subject can be found e.g. in [Fit96] and [RV01].

The principle behind model checking can be outlined as follows. Suppose that *Impl* is a model of a system to be verified and *Spec* is a specification that the model should satisfy, both expressed using some suitable mathematical formalism. Verification then occurs by exploring (explicitly or symbolically) all states of *Impl* and checking if they satisfy the specification. The greatest advantage of model checking is a large scope for automation, at the cost of being applicable only to finite-state systems and a few families of infinite systems. In addition, if the implementation fails to satisfy the specification, then model checking can produce a counterexample (a behaviour of the implementation that is not allowed by the specification) that can be used for debugging purposes. On the other hand, this approach to formal verification suffers from the *state explosion problem*: the time complexity of verification algorithms depends on the size of the implementation, which is typically exponential in the size of its description. This means that standard model checking algorithms can only work in cases where the system to be verified is of finite and (relatively) small size.

In the past, model checking has proved very successful in verification of hardware designs [MC85, BCMD90, MS91, Bar93, CGH$^+$95, HLP01, Ben01]. This success has been made possible by the fact that hardware tends to be less complex and

more structured. However, with more sophisticated model checking algorithms and increased computing power, formal verification of software has become one of the most active areas of research [God97, BR01, CGP02, BR02a, HJMS02, CCG+03].

One approach to model checking, highly popularised by Clarke, Emerson[1] and Grumberg [CE81, CES86, CGL94, CGP99] is based on temporal logics, where specifications are formulated as expressions in a linear time logic (e.g. LTL [Pnu77]) or a branching time logic (e.g. CTL [BAMP81]). Another approach defines a partial order $\sqsubseteq$ on the set of all expressable systems. The intuitive meaning of $P \sqsubseteq Q$ (pronounced "Q refines P") for systems P and Q is that Q is in some sense "better" than P (e.g. it is more deterministic, less abstract or contains more implementation details). In this approach *Spec* and *Impl* are modelled using the same formalism and we define *Impl* to satisfy *Spec* if and only if $Spec \sqsubseteq Impl$. An immediate advantage of refinement checking over temporal logic formulae satisfaction is the fact that what constitutes a specification for a given implementation in one context can be treated as its abstraction in another. This is a very useful feature when working with compositional construction of implementations.

In this thesis we use the refinement-based approach to model checking, where all implementations and specifications are modelled using the CSP process algebra (see Section 1.5) and refinement checks are performed automatically using the FDR model checker (see Section 1.5.3).

1.4 Parameterised Model Checking Problem

It is often the case that specifications or implementations contain free variables. These can be parameters that affect the topology of the system (e.g. the number of nodes in a network or the number of users of a system), the types of data variables (e.g. datatypes of database records or memory contents), performance parameters (e.g. bandwidths, response times, clock speeds), or capacities of buffers or queues used.

A positive verification of, for example, a database where some number data field ranges from 1 to 10 does not mean that the specification is still satisfied when we let the allowed range of this field be $\{1 \,..\, 11\}$. We are, therefore, often interested in the *uniform verification* of a given parameterised pair of a specification *Spec* and an implementation *Impl*, i.e. we want to check whether *Impl* satisfies *Spec* for *all* instantiations of the parameters. Given such *Spec* and *Impl*, the *Parameterised Verification Problem* (*PVP*) asks whether *Impl* can be uniformly verified against *Spec*.

The *Parameterised Model Checking Problem* (*PMCP*) is a subclass of the Parameterised Verification Problem, where the verification is done via model checking. PMCP is, in general, undecidable [AK86], as the Halting Problem [Sip96] can be shown to reduce to it by observing that a sequential, parameterised process $Proc(t)$ can be used to simulate $\#t$ steps of a given Turing machine. Traditional model checkers can only work with a single instantiation of the parameters at a time. There is, therefore, no hope to directly solve any PMCP instance in a finite amount of time if

[1] The 2007 Turing Award was given to Edmund M. Clarke, E. Allen Emerson and Joseph Sifakis for their contributions to making model checking a successful computer-aided formal verification method.

any of the parameters is unbounded. However, even for finite parameters, the number of separate verification checks (e.g. k^n for a system with n variables, each taking values from a set of size k) usually makes the complete verification impractical. This problem has been an active area of research over the last two decades (see Chapter 2) and is also going to be the main topic of this thesis.

1.5 Introduction to CSP

A *process algebra* (sometimes also called a *process calculus*) is a mathematical theory for modelling and analysing concurrent systems that consists of:

- a language suitable for formalisation of concurrent processes, including their interaction and synchronisation, and
- a calculus (a set of algebraic laws) for process definition transformations and verification of process order or equivalence.

Process algebras are designed primarily to model systems with communication between processes based on message passing (as opposed to shared memory communication used in languages like Java and C#), whether synchronous or asynchronous, but can also be used to model shared variable programs via a suitable emulation of variables using processes [Ros01, RH07].

We recommend [Bae05] as an introduction to process algebras . The main process calculi include: CSP [Hoa85, Ros97], CCS [Mil80, Mil89], ACP [BK84, BW90], and π-calculus [MPW89, SW01] (a comparison of various process algebras can be found in [van97] and [BB06]). In our work use the first of these.

CSP (Concurrent Sequential Processes) was first introduced by Hoare in 1978 [Hoa78] as a blackboard concurrent programming language. It was then extended with a formally defined semantics and developed into a process algebra by Brookes, Hoare and Roscoe in 1984 [BHR84] and described in detail in Hoare's book[2] in 1985 [Hoa85]. Roscoe's book [Ros97] is currently the most comprehensive reference on the subject.

In CSP processes interact with each other and the environment within which they operate by communicating *events* along *channels*; for example, $c.a.3$ is an event over channel c, passing data a and 3. We assume that all channels have fixed types (i.e. a fixed number of pieces of data can be passed on each channel and the type of the data passed in each position is also fixed). We use the notation $\{|\ c\ |\}$ to mean the set of events passed over channel c.

The set of all observable (visible) events that a process P can communicate is called the *alphabet* of P and denoted by $\Sigma(P)$. Whenever the process is clear from the context, we simply write Σ to mean the alphabet of that process. A special, internal event, denoted by τ, is never included in any alphabet, so for convenience we write Σ^τ to mean $\Sigma \cup \{\tau\}$. We also write Σ^* to mean the set of all finite sequences (possibly of zero length) of events from Σ.

[2] At the time of writing the thesis Hoare's book is ranked as the second most cited computer science publication of all times, with almost 3000 citations [cit10].

In the rest of this section we present the syntax of the CSP language, two different denotational models (traces and stable failures) and we conclude with a survey of available tool support for CSP.

1.5.1 Syntax

As with any process algebra [Pie96], the language of CSP consists of a small number of primitives and a selection of operators that allow us to construct more complex models.

The CSP syntax that we use in this thesis is the following.

$Proc ::=$

$STOP$	*deadlock*
$\alpha \rightarrow Proc$	*prefix*
$Proc \Box Proc$	*external choice*
$\Box\, i \in \mathcal{I} \bullet Proc(i)$	*replicated external choice*
$Proc \sqcap Proc$	*internal choice*
$\sqcap\, i \in \mathcal{I} \bullet Proc(i)$	*replicated internal choice*
$Proc \triangleright Proc$	*sliding choice (timeout)*
if b then $Proc$ else $Proc$	*conditional choice*
$b \;\&\; Proc$	*positive conditional choice*
$Proc \setminus X$	*hiding*
$Proc[\![{}^{b}/_{a}]\!]$	*renaming by substitution*
$Proc[\![\mathcal{R}]\!]$	*relational renaming*
$Proc \;{}_{X}\|\|_{Y}\; Proc$	*alphabetised parallel*
$\Vert\, i \in \mathcal{I} \bullet [A(i)]\; Proc(i)$	*replicated alphabetised parallel*
$Proc \underset{X}{\Vert} Proc$	*generalised parallel*
$\underset{X}{\Vert}\; i \in \mathcal{I} \bullet Proc(i)$	*replicated generalised parallel*
$Proc \;\vert\vert\vert\; Proc$	*interleaving*
$\vert\vert\vert\, i \in \mathcal{I} \bullet Proc(i)$	*replicated interleaving*
X	*process identifier*

The process $STOP$ is a synonym of deadlock, i.e. it is the process that cannot engage in any communication with the environment and cannot perform any events on its own.

The process $\alpha \rightarrow P$ can perform any event that the construct α describes, and then subsequently behaves like P. The construct α is an expression of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$, where

- c is a channel name,
- $\S_i \in \{\$, ?, !\}$ is an input/output symbol[3],
- if $\S_i \in \{\$, ?\}$, then x_i is an input variable, otherwise it is an output value, and
- if $\S_i \in \{\$, ?\}$, then X_i is a type parameter or type of input, otherwise it is *null*.

The ! symbol denotes an output; ? denotes an input; \$ denotes a nondeterministic choice (which we sometimes call a nondeterministic input or nondeterministic selection). The ? and \$ operators both bind variables to concrete values. For example, the process $in\$x{:}\{0,1\}?y{:}2,3!4 \rightarrow out!(x+y) \rightarrow STOP$ nondeterministically chooses a value $v \in \{0,1\}$ and binds the variable x to it; it is then willing to perform any event of the form $in.v.w.4$ for $w \in \{2,3\}$, and binds the variable y to the value w; it then performs the event $out.(v+w)$ and deadlock. For constructs where $\S_i = !$ for every i, we use the more traditional . output symbol instead, e.g. we write $c.v_1.v_2.v_3$ to mean $c!v_1!v_2!v_3$. Whenever X_i is *null*, we omit it in practice, e.g. we write $c!v$ instead of $c!v{:}null$. The only way a process can communicate a visible event is via a prefix construct.

For two processes P and Q, the *external* (or *deterministic*) choice $P \,\Box\, Q$ is a process that offers the environment the choice of performing any initial event of P or Q; if an initial event of P is performed, then the choice is resolved to P, and if an initial event of Q is performed, then the choice is resolved to Q. The $\Box$ operator is commutative and associative, so we can define an external choice between more than two processes. To do so, we use $\Box$, the replicated version of the operator: $\Box\, i \in \mathcal{I} \bullet P(i)$ is an external choice between processes $P(i)$ for each i in some finite, non-empty[4] indexing set $\mathcal{I}$. We write $?a{:}A \rightarrow P(a)$ as syntactic sugar for $\Box\, a \in A \bullet a \rightarrow P(a)$. $P \sqcap Q$ represents an *internal* (or *nondeterministic*) choice, where the process behaves either like P or like Q, where the choice is made by some invisible mechanism that we do not model and which cannot be influenced by the environment. Similarly to the external choice operator, $\sqcap$ is commutative and associative, so its replicated version, $\bigsqcap$, exists: $\bigsqcap i \in \mathcal{I} \bullet P(i)$ is an internal choice between processes $P(i)$ for each i in some finite and non-empty indexing set $\mathcal{I}$. We define the *sliding* choice (or *timeout*) $P \triangleright Q$ to be a process which behaves like P for a nondeterministically long

[3]Standard CSP commonly also uses the . symbol, but this is only syntactic sugar and can always be replaced by one of \$, ?, !.

[4]The non-emptiness of the indexing set is not strictly necessary in the case of external choice since the $\Box$ operator has a unit (the $STOP$ process: $P \,\Box\, STOP = P$ for all P), so the external choice between zero processes is well-defined. However, the $\sqcap$ operator does not have a unit, so the non-emptiness of the indexing set is strictly required in that case. For consistency we decide to assume that indexing sets of all replicated operators are non-empty.

period of time, but if the environment does not engage in any activity with P within this time, it switches to behaving like Q. The $\triangleright$ operator is not commutative, since its second argument is not initially switched on (no binary operator without both of its arguments treated symmetrically can ever be commutative), so no replicated version of sliding choice can be defined.

The process if b then P else Q is a conditional choice between processes P and Q. If b evaluates to *True*, then this process behaves like P; otherwise it behaves like Q. The process $b \mathbin{\&} P$ is syntactic sugar for if b then P else $STOP$, i.e. P is enabled if and only if guard b is true. We say "a conditional choice on t" to mean a conditional choice whose boolean condition involves only variables and/or values of type t.

For any set $X \subseteq \Sigma$, $P \setminus X$ is a process which behaves like P except that whenever P would normally communicate an event from set X, $P \setminus X$ performs the internal action, τ, instead. Given visible events a and b, $P[\![{}^{b}/_{a}]\!]$ is a process that behaves like P except that whenever P would normally perform a, the renamed process performs b instead. Similarly, $P[\![\mathcal{R}]\!]$ is a process that for every pair of visible events a and b such that $a \mathcal{R} b$, behaves like P except that whenever P would normally perform a, the renamed process performs b instead.

The notion of parallel composition of processes is key to CSP, allowing one to model concurrency. The process $P \ {}_X\|_Y \ Q$ is a parallel composition of P and Q, where P is allowed to communicate only members of the set of visible events X, Q is allowed to communicate only members of the set of visible events Y and the synchronisation occurs on all common events (i.e. those in $X \cap Y$). The binary alphabetised parallel operator is commutative and associative, hence we can define its replicated version: $\Big\|\ i \in \mathcal{I} \bullet [A(i)]\ P(i)$ is the parallel composition of processes $P(i)$ indexed over a finite and non-empty set $\mathcal{I}$, where each $P(i)$ is allowed to perform only events from $A(i)$ and synchronises on event e in $A(i)$ with each process $P(j)$ such that $e \in A(j)$. The process $P \underset{X}{\|} Q$ is the parallel composition of P and Q with handshaken synchronisation on all the members of the set of visible events X. (It may be tempting to say that $P \underset{X \cap Y}{\|} Q = P \ {}_X\|_Y \ Q$, but this is only true when P and Q do not communicate events that are not in X and Y, respectively.) The replicated version of the generalised parallel operator is denoted by $\Big\|_X$; the process $\Big\|_X\ i \in \mathcal{I} \bullet P(i)$ is a parallel composition of the processes $P(i)$ for each i in some finite and non-empty indexing set $\mathcal{I}$, where all $P(i)$ synchronise on the events in X, while each of them is free to perform all other events independently of the rest. Finally, $P \ ||| \ Q$ is the interleaving of P and Q: the processes run in parallel, but do not synchronise on any event (note that this is equivalent to $P \underset{\{\}}{\|} Q$). The replicated version of the interleaving operator, $\Big|||$, is defined to be identical to the replicated generalised parallel over the empty set of events.

Processes are defined by means of equations, such as $P = a \rightarrow P$. We assume a global environment E, mapping identifiers to process definitions, capturing these equations. When a process identifier X is encountered in syntax, E is used to look up which process definition should be substituted for X. Throughout this thesis

we assume that within every process there must be at least one visible or invisible communication occurring between any two lookups in E. This means that process definition like $P = P$ are not allowed.

A discussion of CSP syntax terms not included in the language introduced above can be found in Section 8.3.

So far we have used the term "process" loosely. We now concretise it by making an important distinction between *process syntaxes* (also called process definitions) and *concrete processes*. A process syntax is an open CSP term (i.e. one with free variables). On the other hand, every closed CSP term represents a process. For example, if $Proc(t)$ is a term where t is free, then it is a process syntax; it represents a family of processes $Proc(T)$, one for each concrete instantiation T.

For convenience we define a process $Run(A)$, which always offers to communicate every event in a set A:

$$Run(A) = ?a{:}A \rightarrow Run(A).$$

Similarly, we define $Chaos(A)$ to be a process that can always nondeterministically choose to either to offer or reject any event in a set A:

$$Chaos(A) = STOP \sqcap ?a{:}A \rightarrow Chaos(A).$$

1.5.2 Denotational models and refinement

The purpose of a process algebra, like CSP, is to model and *verify* correctness of concurrent systems. CSP specifications are expressed in the same formalism as implementations, i.e. as processes. Then, an implementation *Impl* satisfies a specification *Spec* if it *refines* it, which we denote by writing $Spec \sqsubseteq Impl$. Intuitively, process Q refines process P (or P is *refined by* Q) if Q does not exhibit any behaviour that is not a behaviour of P. The type of behaviour that identifies a CSP process depends on the denotational model that is used. Such models are obtained from denotational semantics [Ros97, Chapter 8], which are functions from syntax of programs to abstract objects that represent their *meaning* (or *denotations*). The union of all denotations for a fixed semantics forms a denotational model. There are three canonical models for CSP: *traces*, *stable failures* and *failures/divergences*. Some of the newer models also include *infinite traces* [Ros97, Chapter 10], *refusal traces* [LO06] and *stable revivals* [RRS07, Ros08b]. In this thesis we use the traces and stable failures models, discussed below. For a good treatment of the failures/divergences model see [Ros97, Chapter 8].

Traces model $\mathcal{T}$

In this model, each process P is described by a set of finite traces (written $traces(P)$), which contains all the finite sequences of visible communications in which P can engage.

In the traces model the notion of refinement we defined above becomes the *traces refinement*:

$$P \sqsubseteq_{\mathrm{T}} Q \Leftrightarrow traces(Q) \subseteq traces(P).$$

If $P \sqsubseteq_{\mathrm{T}} Q$ and $Q \sqsubseteq_{\mathrm{T}} P$, then we say that P and Q are *traces equivalent*, denoted $P \equiv_{\mathrm{T}} Q$.

Example 1.5.1. Let

$$P = a \rightarrow a \rightarrow P \,\Box\, b \rightarrow P$$

and

$$Q_1 = STOP, \qquad Q_2 = a \rightarrow STOP, \qquad Q_3 = a \rightarrow b \rightarrow STOP.$$

Then

$$traces(P) = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, a\rangle, \langle a, a, b\rangle, \langle a, a, a\rangle, \ldots\}$$

and

$$traces(Q_1) = \{\langle\rangle\},$$
$$traces(Q_2) = \{\langle\rangle, \langle a\rangle\},$$
$$traces(Q_3) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}.$$

Hence, $P \sqsubseteq_{\mathrm{T}} Q_1$ and $P \sqsubseteq_{\mathrm{T}} Q_2$, but $P \not\sqsubseteq_{\mathrm{T}} Q_3$.

End of example.

Using traces, we can define some useful notation. Given a process P, we let $initials(P)$ be the set of all the initially available visible events of P, i.e.

$$initials(P) = \{a \mid \langle a\rangle \in traces(P)\}.$$

In addition, if tr is a trace of P, then P/tr (pronounced "P after tr") is the process that P becomes after it performs tr. So, in particular,

$$traces(P/tr) = \{tr' \mid tr \hat{} tr' \in traces(P)\}.$$

Stable failures model $\mathcal{F}$

In this model, a process P is identified by the set of its traces (defined as above) together with the set of its *failures* (written $failures(P)$). A failure is a pair (tr, X), where $tr \in traces(P)$ and $X \subseteq \Sigma$, and represents the behaviour where P can perform trace tr to reach a stable state P' (i.e. τ is not available in P'), in which it can refuse the whole of X (i.e. none of the events in X is available), denoted P' *ref* X.

When refinement is interpreted over the stable failures model, we get the notion of *stable failures refinement*:

$$P \sqsubseteq_{\mathrm{F}} Q \Leftrightarrow traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P).$$

If $P \sqsubseteq_{\mathrm{F}} Q$ and $Q \sqsubseteq_{\mathrm{F}} P$, then we say that P and Q are *stable failures equivalent*, denoted $P \equiv_{\mathrm{F}} Q$.

Example 1.5.2. Let

$$P = a \rightarrow STOP \,\Box\, b \rightarrow STOP$$

and

$$Q = a \rightarrow STOP \sqcap b \rightarrow STOP.$$

Then

$$traces(P) = traces(Q) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$$

and

$$\begin{aligned} failures(P) = \{&(\langle\rangle, \{\}),\\ &(\langle a\rangle, \{\}), (\langle a\rangle, \{a\}), (\langle a\rangle, \{b\}), (\langle a\rangle, \{a, b\}),\\ &(\langle b\rangle, \{\}), (\langle b\rangle, \{a\}), (\langle b\rangle, \{b\}), (\langle b\rangle, \{a, b\})\}, \end{aligned}$$

$$\begin{aligned} failures(Q) = \{&(\langle\rangle, \{\}), (\langle\rangle, \{a\}), (\langle\rangle, \{b\}),\\ &(\langle a\rangle, \{\}), (\langle a\rangle, \{a\}), (\langle a\rangle, \{b\}), (\langle a\rangle, \{a, b\}),\\ &(\langle b\rangle, \{\}), (\langle b\rangle, \{a\}), (\langle b\rangle, \{b\}), (\langle b\rangle, \{a, b\})\}, \end{aligned}$$

where the last equation follows from the fact that initially process Q is not stable, but after performing an internal action that resolves the internal choice, it may be in one of two states: one, where it offers only a and the other, where it offers only b. Hence, $Q \sqsubseteq_{\mathrm{F}} P$, but $P \not\sqsubseteq_{\mathrm{F}} Q$. In addition, observe that $P \sqsubseteq_{\mathrm{T}} Q$ and $Q \sqsubseteq_{\mathrm{T}} P$, so P and Q can be distinguished in the stable failures model, but not in the traces model (it is true, in fact, that the stable failures model always identifies less than the traces model).

End of example.

As mentioned above, every denotational representation of a CSP process P (including $traces(P)$ and $failures(P)$) can be obtained using the rules of denotational semantics, which can be found for example in [Ros97, Chapter 8]. An alternative approach (and the one we take most of the time in this thesis) is to extract denotational values from an a labelled transition system representing P, obtained by applying an operational semantics. We describe this method in more detail in Section 4.2.3.

1.5.3 Tool support

All currently available tools related to CSP require scripts to be written in (a variant of) the machine-readable version of CSP, CSP_M (see e.g. [Sca98] or [Ros97, Appendix B]), which contains (in addition to the blackboard CSP constructs) an implementation of a rich functional language for added expressiveness.

A basic way to analyse CSP processes is to use an animator like ProBE (Process Behaviour Explorer[5]) [For03]. The user can interact with a chosen process P

[5] Available from `http://www.fsel.com/probe_download.html`.

by playing the role of P's environment. By taking decisions about which events are communicated (but also which way nondeterminism is resolved), any possible execution path can be explored. ProBE allows interaction with infinite-state processes (although, obviously, one can never see the complete behaviour).

The FDR (Failures/Divergences Refinement) model checker[6] [For99] allows one to automatically perform refinement checks. When a CSP_M script with process definitions, say P and Q, is loaded, FDR can automatically test for refinement $P \sqsubseteq_{\text{M}} Q$ in a given denotational model M. FDR can also check for determinism, divergence-freedom and deadlock-freedom. The overall operation of the tool is quite simple [For99]. However, the implemented optimisation mechanisms are rather complex and their knowledge is quite essential for efficient model checking using this tool (details of the operation of FDR can be found in [Ros94] and [Ros97, Appendix C]). The backend of FDR can also be used on its own in order to, among others, obtain state machines of processes. We will use this feature in our tool (see Section 7.1) that implements the techniques we develop in Chapter 3 and Chapter 6.

There are also other interesting tools related to verification using CSP, including model checkers ProB[7] [LP07, LF08], ARC[8] [PY96] and PAT[9] [SLD08], but their usefulness for our work is limited.

1.6 Global assumptions

In this section we state conditions that we assume all processes used in this thesis satisfy. We introduce these for technical and simplification reasons. Clauses (i) and (ii) of the following assumption do not reduce expressiveness, while the reduction of expressiveness due to clause (iii) turns out to be minimal in practice.

Assumption 1.6.1. A process syntax $Proc(t)$, parameterised by a type t, satisfies this assumption if:

(i) all guards of conditional choices within $Proc(t)$ contain either only variables of type t, or only variables and values of types other that t,

(ii) in (binary or replicated) external and sliding choices, $Proc(t)$ contains no name clashes between type t nondeterministic-selection variables of one argument and free variables of another argument, e.g. $c\$x{:}t \rightarrow STOP \,\Box\, d.x \rightarrow STOP$ is not allowed,

(iii) constructs of $Proc(t)$ do not contain multiple occurrences of some input variable of type t, e.g. $c!x!x$ and $c?x{:}X!x$ for X not related to t are allowed, but $c\$x{:}t?x{:}t$ and $c?x{:}t!x$ are not.

[6] Available from `http://www.fsel.com/fdr2_download.html`.

[7] Available from `http://www.stups.uni-duesseldorf.de/ProB/index.php5/Download`.

[8] Available from `http://www.cs.adelaide.edu.au/~esser/arc.html`. The model checking capabilities of ARC are restricted at the moment of writing this thesis.

[9] Available from `http://www.comp.nus.edu.sg/~pat`.

Clause (i) of Assumption 1.6.1 simplifies our treatment of conditionals when working with symbolic representations of processes (see Section 4.3). Observe that the guard of every conditional can be expressed using predicates that involve only types other than t, and predicates that involve only t, combined together using conjunction and disjunction. The conjunctions and disjunctions can be eliminated using the laws

$$\text{if } P \vee P' \text{ then } Q \text{ else } R \;\equiv\; \text{if } P \text{ then } Q \text{ else (if } P' \text{ then } Q \text{ else } R),$$

$$\text{if } P \wedge P' \text{ then } Q \text{ else } R \;\equiv \text{if } P \text{ then (if } P' \text{ then } Q \text{ else } R) \text{ else } R.$$

Hence, any process can be rewritten to satisfy clause (i) of Assumption 1.6.1.

We have introduced clause (ii) as we will later store assignments of values to variables explicitly; clashes of variables names could introduce undesirable updates of values in such assignments. For example, consider the syntax

$$in_1\$x{:}t \rightarrow (out.x \rightarrow STOP \;\Box\; in_2\$x{:}t \rightarrow STOP).$$

Then, the value of x that is output using construct $out.x$ should be the value that is assigned to variable x at the time the nondeterministic selection on channel in_1 is resolved. However, unless the output variable x is immediately substituted with the correct value, the nondeterministic selection on channel in_2 can be resolved before the output is performed, leading to the value of x being overwritten. Using alpha-conversion, every process definition that fails clause (ii) can be easily rewritten into a form that satisfies all of them.

Assumption (iii) ensures that values of all outputs of type t have to be previously stored within process's memory. This simplifies some of our proofs, and does not greatly reduce expressiveness.

1.7 Aims and contributions

Our work focuses on uniform verification of parameterised systems, where the parameter describes the type of identities of node processes forming a network. For the rest of this thesis we let t denote this free identifier, which from now on we call the *distinguished type*. This does not stop processes from having other parameters in their syntax, but their value must be known and fixed at the time of writing the process definition or an additional technique for handling parameters (e.g. data independence) must be used for complete analysis.

For simplicity reasons, we present our techniques using processes with a single distinguished type parameter. However, our theory can be extended to processes with multiple parameters (see, for example, Section 5.3).

Throughout this thesis we assume that every instantiation T of type t is non-empty and finite. In addition, without loss of generality we assume that every instantiation T of type t is equal to the set of the first $\#T$ natural numbers, i.e. if $\#T = n$, then $T = \{0\,..\,n-1\}$. Our results and techniques extend to all set types T, isomorphic to the set of first n natural numbers via simple bijections from $\{0\,..\,n-1\}$ to T.

Model checking works very effectively as a means for finding bugs in systems (some prime examples can be found in [Low96, HLP01, Ben01]). Given an instance of PMCP (see Section 1.4) with a specification $Spec(t)$ and an implementation $Impl(t)$, one can check if $Spec(T) \sqsubseteq Impl(T)$ for some finite number of instantiations T with a hope that if there is a bug present in some instantiation of the implementation, then it would manifest itself in one of these checks. However, once no more bugs can be found using this approach, one often wants to prove correctness for all instantiations of the parameter. The main aim of this thesis is to find automated methods, based on the techniques of counter abstraction (see below and Section 2.2), to aid such uniform verification of systems.

In our work, all node processes are generated from a single template and run in parallel (with non-trivial synchronisations allowed), possibly within some general context (usually in the form of a controller composed in parallel with the nodes with hiding or renaming of some events). The interaction between processes occurs through synchronous message passing. Specifications are described in the same formalism as implementation models (as opposed to e.g. formulae of a temporal logic) and correctness is verified using refinement checking. Finally, we use both traces and failures in system analysis (except for Chapter 6, where only traces are used), in order to allow verification of specifications that talk not only about safety of the implementation (i.e. what the system can do), but also about the availability of events (i.e. what the system cannot refuse to do).

In Section 1.4 we noted that the Parameterised Model Checking Problem is, in general, undecidable. However, the generality of this result tells us very little about the decidability of uniform verification of systems that we consider in this thesis. In our work we discuss the decidability of the subclasses of PMCP that satisfy the characteristics we presented above. In particular, we describe a family of systems for which the problem becomes decidable. The rest of the thesis then focuses on sound (but necessarily incomplete) abstraction methods for the undecidable problems.

Counter abstraction is a well-known abstraction method for systems with many replicated components that can be traced back to the work of Lubachevsky [Lub84]. It is primarily based on using non-negative integer counters to count these components: for nodes with k local states, an abstract state $(c_1, \ldots, c_k)$ models a global state where c_i processes are in the i-th state. This allows us to keep only the information about *how many* node processes are in a given state and discard the information about *which* nodes are in this state. We then use a threshold function z to cap the values of each counter. If for some i, counter c_i reaches its threshold, z_i, then this is interpreted as there being z_i or more nodes in the i-th state. The addition of thresholds makes abstract models independent of the instantiation of the parameter.

In our work, we adapt canonical counter abstraction techniques for modelling concurrent reactive systems using CSP. Such techniques work only for identical node processes, which means that we need to ban node identifiers (therefore making all nodes indistinguishable), i.e. the implementations we initially consider are of the form

$$Impl(t) = C\left[\left\|\!\!\left\|_{A_{Sync}} \; i \in t \bullet \mathcal{N}\right.\right],$$

where:

- $\mathcal{N}$ models a single, finite-state node process,
- A_{Sync} is a set of events on which all nodes have to synchronise, and
- $C[\cdot]$ is some general, t-independent context.

We demonstrate how to produce a counter abstraction model $Abstr_z$, based on a threshold function z, that is t-independent and is such that, by construction,

$$Abstr_z \sqsubseteq Impl(T)$$

for all sufficiently large T. Then, by testing whether

$$Spec \sqsubseteq Abstr_z \tag{1.1}$$

(e.g. using FDR) we can deduce that $Spec \sqsubseteq Impl(T)$ for all sufficiently large T, where $Spec$ is t-independent and both of the above refinements are either in the traces or stable failures model. Observe that in (1.1) we use the same, unmodified specification that is provided in the original verification problem. Unique to our work is the use of a threshold function that allows different threshold values for different counters, resulting in the ability to fine-tune abstractions with a greater accuracy compared to techniques that use a single threshold for all counters.

We show that the addition of node identifiers makes the problem of uniform verification undecidable. We provide a sound, but necessarily incomplete, verification method based on an extension of standard counter abstraction techniques that can be applied to systems that make full use of node identifiers (within specifications, node processes and implementation contexts). The implementations we consider are of the form

$$Impl(t) = C_t \left[\left|\left| \right| i \in t \bullet [A(i,t)]\, \mathcal{N}_i(t) \right] ,$$

where:

- $\mathcal{N}_i(t)$ models a single, finite-state node with identity i and with awareness of all node identities in t,
- $A(i,t)$ is the set of all visible events that $\mathcal{N}_i(t)$ can communicate (its alphabet), and
- $C_t[\cdot]$ is some CSP context, for example that places nodes in parallel with a controller (possibly parameterised by t) and may hide some communication.

We demonstrate how to produce a counter abstraction model $Abstr_z$, based on a threshold function z, that is t-independent and is such that, by construction,

$$Abstr_z \sqsubseteq_{\mathrm{T}} \phi\,(Impl(T))$$

for all sufficiently large T, where ϕ is a function that maps each such T to some fixed type $\hat{T}$. Then, we can use FDR to check whether

$$Spec(\hat{T}) \sqsubseteq_{\mathrm{T}} Abstr_z,$$

and hence deduce that

$$Spec(\hat{T}) \sqsubseteq_{\mathrm{T}} \phi(Impl(T)). \tag{1.2}$$

However, this, on its own, is not enough to allow us to infer that $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$ for all T. This issue has led us to the development of a type reduction theory, which allows us to make the above inference by establishing the size of a fixed type to which ϕ maps all sufficiently large instantiations of the distinguished type and such that

$$Spec(\hat{T}) \sqsubseteq \phi(Impl(T)) \text{ implies that } Spec(T) \sqsubseteq Impl(T), \tag{1.3}$$

with both refinements in either the traces or the stable failures model. We develop a symbolic operational semantics for CSP processes that satisfy certain normality requirements and we present a set of translation rules that allow us to concretise symbolic transition graphs. The ability to describe operations of specifications in a symbolic way allows us to use known behaviours of one instantiation to deduce behaviours of another one. Our type reduction theory relies heavily on these results. One of the main advantages of the symbolic operational semantics and the type reduction theory is their generality, which makes them applicable in other settings and allows the theory to be combined with abstraction methods other than those used in this thesis.

Finally, we have implemented TomCAT, a tool that automates the construction of counter abstraction models.

1.8 Thesis overview

We have structured this thesis as follows.

Chapter 2 presents an overview of various techniques used in model checking of parameterised systems. In particular, we describe work that uses counter abstraction, a method that we will use throughout this thesis.

In Chapter 3 we introduce the ideas of counter abstraction and demonstrate how they transfer to processes that are modelled using CSP and verified through refinement checking. These techniques require that all node processes are identical and that specifications and contexts within which nodes run are independent of the instantiation of the distinguished type (i.e. the only influence of the instantiation of type t is on the number of nodes present within the system). This means that node identifiers cannot appear within definitions of nodes, specifications or implementation contexts. We also present decidability results for the problem of uniform verification of the systems that we consider.

Chapter 4 is devoted to operational semantics for CSP. The main part of this chapter presents a symbolic operational semantics. It allows us to reason about families of parameterised specifications without the need for instantiating the parameter,

which is a prerequisite for our work in Chapter 5. We begin the chapter with preliminaries, the most important of which is the **SeqNorm** condition, which enforces certain regularity (normality) of all process that satisfy it. Then, we present a fairly standard operational semantics. Next, we introduce a symbolic operational semantics for processes that satisfy **SeqNorm**. The characteristic property of the transition graphs it generates is a symbolic form of all the parts of constructs that relate to the distinguished type, and a concrete form of all their other parts. Finally, we present a set of translation rules from the symbolic transition graphs to their concrete instantiations and we show that these rules, combined with our symbolic operational semantics, forms an operational semantics that is congruent to the standard one. We also relate concrete and symbolic traces.

In Chapter 5 we present our type reduction theory. We begin with a number of regularity results that are consequences of the **SeqNorm** condition, paving the way to the main type reduction theorems for both the traces and stable failures model. The theorems establish a function ϕ that maps all sufficiently large instantiations T of the distinguished type to some fixed type $\hat{T}$ such that (1.3) holds. This, when combined with a suitable abstraction method, allow us to perform a small number of refinement checks in order to deduce the answer to a given uniform verification problem (although termination is not guaranteed).

One such abstraction method is studied in Chapter 6, where we extend the standard counter abstraction method from Chapter 3 to processes with awareness of their own identities as well as identities of other nodes. We present a sound method which can automatically create finite-state abstractions of parameterised implementations. We consider refinement checks in the traces model. In addition, we prove that uniform verification of families of systems that use node identifiers is, in general, undecidable.

In Chapter 7 we present TomCAT, a tool that automates the production of counter abstraction models based on the techniques of both Chapter 3 and Chapter 6. In addition, we provide a case study, in which we demonstrate how our results apply in practice by verifying a variation of the producer-consumers problem.

Finally, we conclude and discuss possible directions in which our work can be developed further in Chapter 8.

Chapter 2

Related work

Parametric model checking has been a very active area of scientific research in recent years. In this chapter we give an overview of various approaches to verification of parameterised systems.

As PMCP is, in general, undecidable [AK86], all attempts to solve uniform verification problems fall into two categories: introducing some restrictions and providing a sound and complete decision algorithm for the obtained decidable subclass of PMCP, or providing a sound, but incomplete verification method. The following are the contributions with the most general results in the former category. In [GS92] German and Sistla prove decidability of uniform verification of systems with identical node processes and a single controller with CCS-like synchronisations [Mil80, Mil89] (out of all of the references quoted in this section this model of concurrency is probably the closest to ours). The provided decision algorithm is based on a reduction of the verification problem to the reachability problem over vector addition systems with states (see Section 3.1), which is similar to the approach we take to prove decidability of a subclass of the uniform verification problem considered in this thesis (see Theorem 3.1.1). The algorithm is double exponential in the size of a node and the controller (however, it is polynomial for systems without controllers). A very similar result is presented in [EN96]. Emerson and Namjoshi use a synchronous model that is more general than that in [GS92], allow transition guards to use tests over states of all processes (in addition to tests over single nodes) and provide a decision algorithm that is polynomial in the size of a node and the controller. However, their techniques suffer from a limitation that makes it impossible to verify, for example, mutual exclusion algorithms. In [EK00] Emerson and Kahlon prove decidability of uniform verification of asynchronous systems that consist of many categories of processes (some of which can be controllers). The decision algorithm is based on reducing verification of systems of all sizes to verification of systems of sizes up to some small and fixed cutoff size. A similar reduction is used in proving decidability of verification of ring-based system in [EN95, EK04]. This is also the approach that we combine with abstraction techniques in this thesis. However, we apply such combinations to systems with more general synchronisations, while sacrificing completeness.

Most research that focuses on sound, but incomplete verification methods falls into one of the following categories.

Predicate abstraction This overapproximation abstraction method creates Boolean Programs [BR00b] (i.e. programs that use variables only of the Boolean type) from a given implementation by specifying logical predicates and defining which of these predicates need to hold in a given state. Then, any two concrete states are identified under the abstraction if and only if exactly the same predicates hold in them.

Counter abstraction This is an abstraction method applicable to systems with the parameter determining the number of (similar) node processes forming a network. Counter abstraction works by counting, possibly only up to some finite threshold, how many nodes are in a given state and discarding the information about which local state each of the processes is in. Counter abstraction will play a significant role throughout this thesis.

Data independence This technique concentrates only on the subclass of PMCP where the distinguished parameter is a datatype. Suppose that a given system can input and output values of the parameter type, but cannot use them to influence the control flow. Then, data independence implies the existence of a fixed (and usually small) size of an instantiation T of the parameter such that the system satisfies a given specification for all datatypes larger than T if and only if it does so for T.

Induction This deductive proof method can be viewed as a generalisation of the traditional mathematical induction. A predicate is proved to hold in states of an unboudedly large system by deducing an infinite number of statements of the form $\psi(T) \Rightarrow \psi(T \cup \{x\})$ (one for each T) from solutions of a finite number of finite-state problems. Structural induction is a particular form of induction, applicable to recursively-defined systems. The proof always follows the same pattern. Firstly, we show that a given property is satisfied by the system with a basic structure (the base case). We then show that a system of an arbitrary size satisfies the property, provided all of its subsystems do. From this we can conclude that the property is satisfied by all systems, regardless of their structure.

Counterexample-guided abstraction refinement When verifying an implementation using a sound, but incomplete abstraction method, a model *Abstr* may fail to meet a specification, even though the implementation satisfies it. One of the advantages of model checking is the amount of information present when a verification check results in a negative answer. Most model checking algorithms produce a counterexample that indicates the behaviour of the tested model that is not allowed by the specification. The method of counterexample-guided abstraction refinement exploits this by using the "abstract, verify and refine" approach, where we initially start with a very coarse abstraction of the implementation and whenever we obtain a counterexample, we check it against the specification. If it proves to be a valid behaviour in the concrete system, then the overall verification yields a negative result. However, when the counterexample is an artefact of the coarseness of the abstract model, we refine (extend)

the abstraction in a way that eliminates the behaviour that the counterexample represents. We then go back to the verification stage.

In the rest of this chapter we discuss the research that has been performed in each of the above areas in more detail.

2.1 Predicate abstraction

Predicate abstraction is a general abstraction method (first introduced by Graf and Saïdi in 1997 [GS97]) for reducing large (possibly unbounded) systems into Boolean Programs [BR00b] (i.e. programs that use variables only of the Boolean type) that can be efficiently model checked against a given specification.

Predicate abstraction can be viewed as a combination of the theorem proving and model checking (see Section 1.3). Its operation can be outline as follows. Firstly, a set of logical predicates needs to be created. Then, any two concrete states are identified under the abstraction if and only if exactly the same predicates hold in them, i.e. the evaluation of the predicates on the set of all concrete states induces a partition on the state space. Finally, specifications are captured by defining which of the predicates need to hold in a given state.

The choice of predicates is crucial. In the original paper [GS97], they are supplied by the user. When a spurious counterexample is produced, the user needs to redefine the set of predicates and perform the check again. Even though there are heuristics that help with defining the "right" predicates (e.g. they use guards from conditionals), this process requires ingenuity and can be time-consuming. A lot of effort has been therefore put into automatic generation of sensible predicates. In [DD01] an interesting method combining predicate abstraction and counterexample-guided abstraction refinement (see Section 2.5) is discussed. Suppose that we are given an abstract spurious trace $s_1, s_2, \ldots, s_k$. Let γ be the concretisation function, so that $\gamma(s_i)$ is the set of all concrete states corresponding to the abstract state s_i. Now, because our trace is spurious, there exists $i \in \{2\,..\,k-1\}$ such that there is no concrete state within $\gamma(s_i)$ such that it is the end point of a transition originating from $\gamma(s_{i-1})$ and the starting point for a transition to some state in $\gamma(s_{i+1})$. In order to eliminate the spurious counterexample we need to split $\gamma(s_i)$ into two sets: one that contains the successors of the concrete states in $\gamma(s_{i-1})$ and one that contains the predecessors of states in $\gamma(s_{i+1})$. In order to perform the cut, predicates that distinguish these two types of states have to be defined. The authors propose to use the CVC theorem prover [SBD02, BB04] to check satisfiability of specifications expressed as logical formulas, where the output of the tool is used to automatically generate the new predicates. This approach is also shown to work in the case of quantified predicates (e.g. when unbounded parameters are used).

In the past decade or so, predicate abstraction has been applied to a wide range of (often parameterised) problems. In [PB05], Pek and Bogunović show how predicate abstraction can be used in protocol verification (the Bakery algorithm and Fischer's mutual exclusion protocol are studied as examples). Similarly, Das et al. [DDP99] show how predicate abstraction can be used in automated verification of concurrent

protocols (e.g. cache coherence or a concurrent garbage collector) described in a simplified version of the *Mur*ϕ language [DDHY92, Dil96].

Microsoft has extensively used various forms of predicate abstraction in its SLAM project [BR01, BR02a]. While many uses of the method involved programs defined using languages based on guarded commands, Ball et al. [BMMR01] show how predicate abstraction can be used in verification of parameterised C programs. This task introduces new challenges (due to, for example, a presence of pointers and procedure calls), but the solution is of high practical value (e.g. it allowed Microsoft to perform automatic verification of Windows device drivers).

An interesting (from the perspective of our work) application of predicate abstraction can be found in [LHR01], where Lesens and Saïdi consider parameterised networks of (possibly infinite-state) sequential node processes with synchronisation using shared variables. The systems under consideration can have two kinds of parameters: one that specifies the topology of a system (i.e. the number of node processes running in parallel) and one that determines the types of data variables. In order to tackle the unbounded systems, the authors construct a two-level abstraction: first, a single (infinite state) node process is abstracted and then an abstract network (composed of the abstracted node processes) is obtained. The resulting system is finite-state and can, therefore, be model checked directly. With each top-level abstract state, associated is an additional set of variables, where each (i, j)-th variable records how many node processes are in the i-th state and satisfy the j-th predicate. This idea is very similar to that behind counter abstraction, which we will introduce in Section 2.2 and which we will use in Chapter 3 and Chapter 6.

Example 2.1.1. Suppose that we model a standard 8×8 chessboard with a single rook placed in square (1,7) (see Figure 2.1). The rook can attack all positions in the first column and the seventh row. We also place a single king figure in square (8,1). The king is allowed to move in 8 directions to any neighbouring position, without ever entering a square that is under attack by the rook and without going off the board. We assume that the it cannot attack the rook. In addition, without loss of generality under the above assumptions, we assume that the king can only move left, right, up and down (since any diagonal move can be composed using two such moves).

Consider an implementation $Impl = King(8, 1) \setminus Moves$, where

$$\begin{aligned} King(x, y) \quad = \quad & y > 1 \;\&\; down \rightarrow King(x, y - 1) \\ & \Box\; y < 6 \;\&\; up \rightarrow King(x, y + 1) \\ & \Box\; x > 2 \;\&\; left \rightarrow King(x - 1, y) \\ & \Box\; x < 8 \;\&\; right \rightarrow King(x + 1, y) \\ & \Box\; x = 1 \vee y \geq 7 \rightarrow error \rightarrow STOP \end{aligned}$$

and $Moves = \{left, right, up, down\}$.

Suppose that we want to verify that the king can never reach square (8,8). If the implementation allowed it, then $\langle error \rangle$ would be a trace of *Impl*. So to test this, we let

$$Spec = STOP.$$

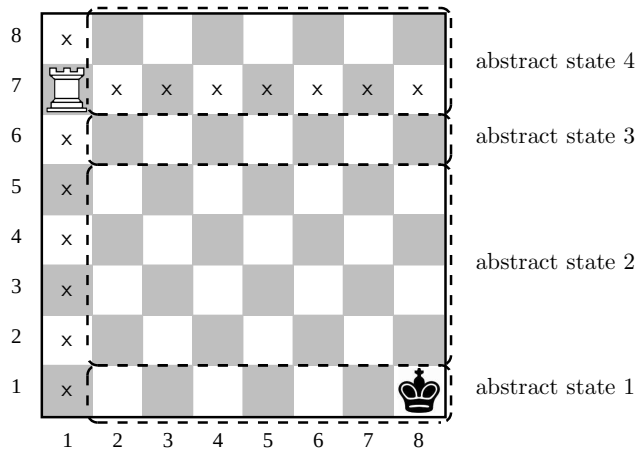


Figure 2.1: Illustrating predicate abstraction with a chess example.

The implementation, defined as above, has 64 states (the position of the king on the board can, theoretically, be anywhere between (1,1) and (8,8)), but 22 of them are unreachable. However, it is easy to see that the horizontal movement of the king is irrelevant for our specification and so is any vertical movement between rows 2 and 5. Hence, we create a predicate abstraction of our system using the following four predicates.

$P_1 \equiv y = 1,$
$P_2 \equiv y > 1 \wedge y < 6,$
$P_3 \equiv y = 6,$
$P_4 \equiv y \geq 7.$

This abstraction has four states (see Figure 2.1), one of which is unreachable. The i-th state corresponds to all the concrete states where P_i is true. Let

$$\begin{aligned}
\widetilde{King}(1) &= up \rightarrow \widetilde{King}(2),\\
\widetilde{King}(2) &= up \rightarrow (\widetilde{King}(2) \sqcap \widetilde{King}(3))\\
&\quad \Box\\
&\quad down \rightarrow (\widetilde{King}(2) \sqcap \widetilde{King}(1)),\\
\widetilde{King}(3) &= down \rightarrow \widetilde{King}(2),\\
\widetilde{King}(4) &= error \rightarrow STOP.
\end{aligned}$$

Then $\widetilde{Impl} = \widetilde{King}(1) \setminus Moves$ is a CSP process that models our predicate abstraction. Observe that when the concrete system is in a state where P_2 is true, then making an up move can result in the king staying in the region of the board where P_2 is still true; alternatively it can take the king to a field where P_3 is true. Similarly, a down move can result in the king moving to a field where P_2 is still true, but it may also take the king to a field where P_1 is now true.

We have that *Spec* $\sqsubseteq_{\mathrm{T}}$ *Impl* if and only if *Spec* $\sqsubseteq_T \widetilde{Impl}$. However, the number of states that need to be explored in the latter check is 14 times smaller than in the former one. In addition, if we consider a more general, $n \times 8$ chessboard (i.e. each row is n fields long), then a direct uniform verification would require infinitely many checks (one for each n), while the procedure given above proves the result for all n using a single refinement check.

End of example.

2.2 Counter abstraction

Counter abstraction is a simple and well-known abstraction method. In its general form, it is a special case of predicate abstraction [GS97, Das03, LB04], discussed in Section 2.1, which, in turn, is a special case of *finitary abstraction* [KP00], a framework for transforming infinite-state system into finite-state abstractions.

Counter abstraction applies to uniform verification problems where the parameter specifies how many similar *node processes* are present in the system. The main idea is to replace the concrete state space by an abstract state space, where each state is a tuple of integer counters $(c_1, \ldots, c_k)$ such that for each i, c_i counts how many node processes are currently in the i-th state. This is called *counter abstraction with unbounded counters* (or *counter abstraction without thresholds*). Such abstract models still depend on the number of nodes present in the system, so they are not suitable for use in uniform verification. However, they help reduce the effects of the *global state explosion problem*, because symmetrically equivalent global states are identified.

The idea of abstracting systems consisting of a fixed number of similar processes by counting the processes in their control states can be traced back to the work of Lubachevsky [Lub84]. Based on exactly the same ideas is *symmetry reduction*. However, unlike counter abstraction, formalisations of symmetry reduction are usually based on a group theoretic approach, which is very different from the process algebraic approach we use in this thesis. In addition, just as counter abstraction without thresholds, symmetry reduction assumes an arbitrary, but fixed number of node processes, which means its usefulness for uniform verification is limited. For these reasons we do not directly relate to symmetry reduction techniques within this thesis. However, some interesting work on this topic can be found in [ID96, ES93, CEFJ96, ES97, AHI98, ET98, ET99, GS99, EHT00, EW03, BG05, DM05, EW05, WGC05, Wah07].

In this thesis we use *counter abstraction with thresholds*, which extends basic counter abstraction with unbounded counters with a (often constant) threshold function that determines the maximum attainable value for every counter. If the value of the i-th counter reaches the corresponding threshold, z_i, then we interprete this as there being z_i *or more* processes in the i-th state. Abstract models created in such a way from systems of sufficiently large size are independent of the number of node processes. This means that a single refinement check solves the verification problem for all values of the parameter (possibly except for some small ones).

We feel that the idea of counter abstraction with thresholds is best presented in [PXZ02] (interestingly, to the best of our knowledge, this is also the first published work that uses the term "counter abstraction" explicitly). In their paper,

Pnueli et al. (and also Xu in [Xu05]) show how to verify a general temporal formula on a concurrent system with unboundedly many similar processes running in parallel. The formalism used for expressing implementations is the *Fair Transition Systems* [MP95] in which processes are interleaved and communicate with each other via shared variables only. Each process may also be given its own local variables (whose values may depend on the process's identity). However, this destroys the full symmetry of the system. Before the verification procedure is performed, the user has to replace the local variables by some appropriate global variables that will model the same behaviour. This task may require a significant level of ingenuity. In addition, it is shown how, given a concrete state space and a temporal formula, to create a corresponding abstract machine and abstract temporal formula. In addition, some heuristics are provided for extending the abstract formula so that different liveness properties can be verified. The threshold function that is used is a constant function $z = 2$, which means that processes are counted within the $\{0, 1, many\}$ equivalence classes. This can be used only for verification of properties that allow presence of at most one process in a given location (e.g. mutual exclusion). A very similar approach with the same threshold function is used for verification of cache coherence protocols by Pong and Dubois in [PD95].

In [CG87], Clarke and Grumberg show that if $Impl(T) \parallel Abstr(Impl(t))$ and $Impl(T \cup \{i\}) \parallel Abstr(Impl(t))$ are equivalent under an appropriate relation, then it is enough to consider only the instances of the implementation of size less than or equal to $\#T$ in order to deduce whether a given ICTL* formula holds *for all* sizes of the system. A method similar to counter abstraction with a constant threshold $z = 1$ is used to construct $Abstr(Impl(t))$. However, in order for the method to work, the user needs to provide a pair of functions that match certain non-trivial conditions. This task usually requires some ingenuity, so the scope for automation is limited.

The work published in [Ver94] is based on counter abstraction techniques similar to those we present in Chapter 3. Vernier considers systems with a similar structure to the one we use and with synchronous communication. The verification algorithm consists of two parts. The first creates a finite abstraction that is independent of the number of processes present in the concrete system. This is similar to our method, but instead of using a fixed threshold function, appropriate thresholds are determined on-the-fly (at the expense of compilation complexity). The second part of the algorithm evaluates CTL-X specifications on the abstract state machine.

Counter abstraction creates an interesting problem when verifying liveness properties of a parameterised concurrent system. Such checks usually require some kind of fairness assumption. Unfortunately, standard counter abstraction techniques suffer from the problem of lost node identifiers (i.e. all processes have to be identical), which means that expressing fairness conditions over such models becomes non-trivial. Research on using counter abstraction for model checking with fairness can be found e.g. in [PXZ02, Xu05, SLR$^+$09]. In addition, we will present an extension of standard counter abstraction techniques that preserves node identifiers in Chapter 6.

Finally, it is worth mentioning work that, similarly to our considerations, is based on the CSP process algebra and refinement-oriented approach to model checking, and which, even though does not use counter abstraction explicitly, uses abstract models

equivalent to those obtained using counter abstraction. In [Low04] Lowe considers a verification of the Adapted Needham Schroeder Public Key Protocol [Low95, Low96], where agent identities and nonces are partitioned into certain equivalence classes. Each such partition is used to create an abstraction of the implementation. The partitions are iteratively made finer using a CEGAR-based approach (see Section 2.5). Each such abstraction can be viewed as a combination of several counter abstractions of protocol session participants, one for each of the equivalence classes.

For examples of an application of counter abstraction with unbounded counters see Chapter 3. Examples of an application of counter abstraction with thresholds can be found in Chapter 6 and Chapter 7.

2.3 Data independence

It is often the case that a system to be verified is parameterised by some datatype t. Model checkers (like FDR) can handle one instantiation T of type t at a time (and even this may be impractical if T is large). However, in most cases we are interested in uniform verification, i.e. we want to check whether a given result holds for all sizes of T. The theory of data independence seeks to identify the systems whose behaviour is not affected by what the values of the datatype T actually are (or is affected in a way that does not influence the property to be verified) and to provide theorems that allow us to create an abstraction *Abstr* of an implementation *Impl* that is exact, i.e. such that *Impl* satisfies a given specification *if and only if* *Abstr* does.

Informally, we say that a given system is *data independent* in datatype parameter t if values of this type can only be input, stored, output or compared for equality. In particular, operations on values of type t that can influence what instantiation of this type is used are banned. We provide a formal definition of data independence in Section 4.1.1 (also see [Ros97, Chapter 15] and [Laz99]). A lot of the work on data independence considers either a stronger definition, where no equality tests between values of type t are allowed, or its more general version, where functions with co-domains equal to t are additionally allowed.

The concept of data independence was introduced by Wolper in 1986 [Wol86] in relation to concurrent network protocols. He considered the effect of functions that change data values within programs on the behaviour of the overall system. He defined a program $P(t)$ to be data independent if for every function f such that $\mathrm{dom}(f) = t$, if σ is a trace of $P(t)$, then $f(\sigma)$ is a trace of $P(f(t))$. Using this definition, Wolper showed that it is possible to verify safety of a transmission of messages taken from an unboundedly large set. Using the same technique, Sabnani verified the Alternating Bit Protocol [BSW69] in [Sab88]. Hojati and Brayton showed in [HB95] how data independence (which they call *data insensitivity*) can be used in verification of hardware systems. In [Qad03] Qadeer proved that it is possible to automatically verify unbounded concurrent multiprocessor architectures with shared memory by using data independence.

2.3.1 Data independence and CSP

The most important results in the area of data independence from the point of view of this thesis come from the work of Lazić [Laz99]. In his research, a new language was created (a mixture of a subset of CPM_M and typed λ-calculus), for which a collection of theorems was derived; the theorems establish existence of datatype *thresholds* B such that the outcome of a refinement check is guaranteed to be the same for all sizes of the datatype greater than or equal to B. The results have since been extended to the standard CSP_M [Ros97, Chapter 15].

So far the most successful application of data independence in CSP has been in the area in security (network protocols in particular). Roscoe suggested a method of applying data independence to verification of network protocols in [Ros98, RB99]. Extensions, automation and integration with Lowe's Casper security protocol verification tool [Low98] are further discussed in [BLR00, Bro01, BR02b]. Some more examples of the application of CSP and data independence in security can be found in [Low04, RL05].

Another large family of systems where verification using data independence and CSP has had a lot of success is processes that use arrays, for example databases, caches and storage devices [LR98, RL01, New03, LNR04b, LNR04a].

Data independence has also proved useful in inductive model checking of parameterised systems that are built in a recursive manner (see Section 2.4).

Example 2.3.1. Suppose that we model a pipeline into which we can input data d (of type t) together with a read-only flag ro. If $ro = no$, then the system is allowed to modify d, otherwise the data is to be treated in a read-only way. The data is then used within the pipeline (in a way that can possibly modify it, if allowed) and output. Formally, we let

$$Impl(t) = Pipeline(t) \setminus \{\!|\ use\ |\!\},$$

where

$$\begin{aligned} Pipeline(t) = {} & input?d{:}t?ro{:}\{yes, no\} \rightarrow use!d!ro\$c{:}\{yes, no\} \\ & \rightarrow \text{if } c = no \text{ then } output.d \rightarrow Pipeline(t) \\ & \qquad \text{else } output?d'{:}t \rightarrow Pipeline(t). \end{aligned}$$

Here, $use!d!ro\$c{:}\{yes, no\}$ models a series of actions (possibility modifying the data) that can happen in the pipeline between the input and the output. We assume that this series of actions treats t data independently and that it uses no equality tests between variables of type t. Flag c indicates whether the data has been changed (in which case $c = yes$) or not ($c = no$).

We let the specification, $Spec(t)$, say that whenever some piece of data is input and the read-only flag is set to yes, then the same piece of data must be output; otherwise we assume nothing about the output (in particular we allow it to be different from the input).

$$\begin{aligned} Spec(t) = {} & input?d{:}t!no \rightarrow output?d'{:}t \rightarrow Spec(t) \\ & \Box \\ & input?d{:}t!yes \rightarrow output.d \rightarrow Spec(t). \end{aligned}$$

Our verification problem is to check whether $Spec(T) \sqsubseteq Impl(T)$ for all non-empty datatypes T. We can use FDR to perform the checks for single instances of T. However, if we test that $Spec(T_0) \sqsubseteq Impl(T_0)$ for T_0 of size 1 and 2, then the result for the general case follows from Theorem 15.2.2 in [Ros97, Chapter] (checking that $Spec(t)$ and $Impl(t)$ satisfy the hypothesis of the theorem is easy).

End of example.

2.4 Induction

Induction methods for parameterised systems aim to prove predicates of the form $\forall\, t \bullet \psi(t)$ by showing that

$$\psi(\{\}) \wedge \forall\, T, x \notin T \bullet \psi(T) \Rightarrow \psi(T \cup \{x\}). \tag{2.1}$$

In [MQS00] McMillan has shown how inductive methods can be combined with model checking to verify parameterised system of unbounded sizes (e.g. an N-process bakery algorithm [Lam74] is verified for all N). The main idea is to prove that (2.1) holds by abstracting natural numbers using a finite number of intervals (the results easily generalise to all types isomorphic to natural numbers). In the most basic case, given a natural number n, four intervals are used: $(0, n-1)$, $\{n-1\}$, $\{n\}$, (n, ∞), but additional singleton intervals can be considered for constants used within implementations. The inference from a verification for a single value of n to natural numbers can be then made by observing that the equivalence classes induced by this abstraction (the individual internals) are isomorphic for all values of n.

Support for these techniques (as well as for temporal case splitting, symmetry-based simplification and data type reducion [McM98, McM99], from which the inductive methods have evolved) has been implemented within the Cadance SMV proof assistant (unfortunately, considerable user input is often still required); this functionality was used in [KNS01] to verify the non-probabilistic component of a complex randomised distributed consensus protocol (while the probabilistic part was verified using the PRISM model checker [PRI]).

Structural induction is a collective name for all methods where induction is applied over a recursive composition. It is particularly suited for verification of multi-agent systems, parameterised by the number of components (e.g. networks, buses, distributed systems, etc.). Structural induction used in PMCP verification is probably easiest described in a framework where the set of all processes forms a lattice with some refinement order $\sqsubseteq$ defined between them (e.g. the CSP process algebra). Suppose that $\mathcal{N}$ is a node process and that we can model a given parameterised implementation as $Impl(t) = (\ldots(\mathcal{N} \oplus \mathcal{N}) \oplus \ldots) \oplus \mathcal{N}$ (with $\#t$ copies of $\mathcal{N}$), where $\oplus$ is an operator that is monotonic with respect to $\sqsubseteq$ (in the majority of cases $\oplus$ is the parallel composition operator, possibly with an some renaming or hiding). Observe that by using renaming we can make the node processes distinguishable. Now suppose that for some instantiation T of type t the following holds.

$$Inv(1) \sqsubseteq \mathcal{N} \tag{2.2}$$

$$\forall\, n \in \{1 \,..\, \#T - 1\} \bullet Inv(n+1) \sqsubseteq Inv(n) \oplus \mathcal{N} \tag{2.3}$$

$$Spec \sqsubseteq Inv(T) \tag{2.4}$$

We can then deduce that $Spec \sqsubseteq Impl(T)$. Even though it may seem that this does not bring us any closer to the goal (of verifying the result for all instantiations of type t), there are two different directions in which we can proceed now. The first one requires that structure invariants are independent of their parameters (i.e. $Inv(i) = Inv(j)$ for all i, j) [KM89, WL90, RJDR98, CR99a]. Provided that such an invariant exists and that the refinements in (2.2) and (2.3) hold, testing for $Spec \sqsubseteq Inv$ is enough to deduce the result for all sizes of the implementation. However, the construction of such invariants is often far from obvious and the scope for automation is very limited. In fact, it is shown in [WL90] that there are problems where suitable invariants cannot be found and even deciding their existence, is in general, undecidable.

A different approach is taken in the work of Creese and Roscoe [CR98, CR99b, CR99c, CR00, Cre01]. The induction is based on data independence techniques (see Section 2.3). Suppose that specification *Spec* and every node process is data independent in t. Since the implementation uses a replicated operator indexed over the whole of t to construct the network, it is not data independent in t (see Definition 4.1.1). This means that we cannot directly use data independence results to prove that $Spec \sqsubseteq Impl(T)$ for all T. However, provided that we can establish the truth of the refinements in (2.2), (2.3) and (2.4) for some instantiation T_0 of type t, whose size is equal to a certain threshold value, then we can deduce that the checks are true for all types T such that $\#T \geq \#T_0$. This provides us with unboundedly many inductive proofs (one for each T). This family of proofs (together with some direct checks for the cases when the instantiation of the parameter is smaller than T_0) allows us to perform uniform verification in a finite (and usually small) number of steps. The challenge, again, is to find a suitable process *Inv*, which is data independent in t.

For a number of interesting examples of structural induction see e.g. [KM89, WL90, Cre01].

2.5 Counterexample-guided abstraction refinement

As mentioned in Section 1.3, one of the biggest problems of model checking is the state explosion problem. One (and very common) way to deal with it is to abstract implementations. There are two types of abstractions: *exact* and *conservative* (the latter also called *overapproximations*). In the former, an implementation satisfies a specification if and only if the abstraction does. In the latter, the "only if" part may not be true; it may happen that a verification of an abstraction yields a negative result, but the original implementation is in fact correct. If this happens, then the abstraction is too coarse and a more detailed one needs to be constructed. However, it is then important to know which behaviours of the implementation were absent in the abstraction (and therefore caused it to fail to meet the specification) and should now be added back in. In this section we consider a mechanisation process for construction of conservative abstractions.

One of the biggest advantages of model checking over theorem proving is its ability to provide (often elaborate) counterexamples when an implementation fails to satisfy a specification. *Counterexample-guided abstraction refinement* (*CEGAR*) exploits this

by performing the verification in the following four stages.

Abstracting the implementation We start by creating an initial, coarse abstraction of the implementation.

Abstraction verification We use a model checker to verify whether the abstraction satisfies the specification. If the result is positive, we conclude the correctness of the implementation. If the result is negative, we record the counterexample returned by the model checker.

Counterexample analysis Given the counterexample, we check whether it corresponds to valid a behaviour of the implementation; if so, then we have found a bug and the verification terminates with a negative result, otherwise we have got a *spurious counterexample* (one that is a result of the abstraction being too coarse).

Refining the abstraction We modify the abstraction in a way that eliminates the possibility of the spurious counterexample we go back to the abstraction verification step.

Kurshan [Kur94] as well as Balarin and Sangiovanni-Vincentelli [BSV93] proposed similar methods (which Kurshan calls *localisation reduction*), based on the scheme presented above, that automatically create successive, more detailed abstractions. They use the *L*-language [Kur94] to model concurrent systems composed of a number of similar processes. Each state of a process is represented by two sets of variables and *L*-processes communicate using shared variables. Two types of abstraction are used. One abstracts away all the variables of some processes whose behaviour is deemed to be unrelated to a given specification. This is achieved by replacing them by nondeterministic selections over all their possible values. The second type abstracts away the details of what some processes communicate. This is done by replacing their output variables by nondeterministic selections over all possible outputs. When a spurious counterexample is encountered, a minimum set of variables that were previously abstracted away is added back in. This approach does not actually reduce the number of states to be verified (and significantly adds to the number of transitions) at any stage, so its usefulness in model checking with explicit state traversal (e.g. as done by FDR) is very limited. However, it increases compactness of BDD representations, which greatly aids symbolic model checking.

Clarke et al. [CGJ$^+$00, CGJ$^+$03] used the same general idea as presented above, but their approach starts with a small-sized abstract model that grows with each iteration of the algorithm. By using more levels of abstraction for each variable used, each iteration allows for much finer changes between the successive overapproximations. In the quoted papers, the authors also proved that the problem of finding the minimal change in the abstraction that eliminates a given spurious counterexample is NP-hard. However, they showed that in practice it is often sufficient to use a heuristic algorithm that finds a small enough, but possibly suboptimal, successive abstraction.

Techniques based on the counterexample-guided abstraction refinement paradigm have been successfully used in many applications. Some examples include: electronic

circuits [CGKS02], hybrid systems (i.e. systems that use both discrete and continuous variables) [CFH+03, ADI06], software code [BR00a, DD02, DGL06] (including the SLAM [BR02a] and BLAST [HJMS02] projects), network protocols [LBBO01] and security [Low04].

In Section 3.3.3 we will use a CEGAR-based algorithm with heuristics for finding small, but possibly suboptimal counter abstraction models (see Section 2.2).

2.6 Conclusions

In this section we summarise the contributions of this thesis against the background of the related work presented in this chapter.

The main abstraction methods presented in this thesis (Chapter 3 and Chapter 6) are based on counter abstraction. Compared with the work referenced in Section 2.2, our techniques are characterised by the following.

- Interaction between processes occurs through communication of events.
- Specifications are provided as process definitions (as opposed to temporal logic formulas) and are verified through refinement checking.
- Nodes are allowed to run in fairly general contexts.
- Stable failures are used for systems without node identifiers to allow verification that talks not only about safety (i.e. what a system can do), but also about the availability of events (i.e. what the system cannot refuse to do).
- In safety verification problems:
    - Nodes run in alphabetised parallel, with alphabets parameterised by node identities and their type.
    - Nodes can use their own identities to model private communication and identities.
    - Nodes can use their own identities, together with identities of other nodes to model two-way synchronisation.
    - Nodes can receive identities of other nodes, store them, and use them in subsequent communications.
    - A renaming function, ϕ, is used to collapse all node identities to a finite (and usually small) set of values.

Counter abstraction can be viewed as a particular instance of predicated abstraction (Section 2.1). The predicates in such a case have the form "$c_i = 0$", "$c_i = 1$", ..., "$c_i \geq z_i$", where c_i is the number of processes in the i-th state and z_i is a threshold that limits the values that c_i can attain. The main difference between our approach and general predicate abstraction methods (defined as in [GS97, Das03]) is the specification verification paradigm; while predicate abstraction specifications determine which predicates are evaluated to *True* in what states, we have chosen a refinement

based approach (see under "Specifications based on counter values" in Section 8.2 for a discussion on an approach much closer related to that of predicate satisfiability). In addition, predicate discovery is usually a manual process, while our methods are fully automatic.

The same problem is exhibited within techniques based on induction (Section 2.4). They attempt to solve similar problems to those we consider in this thesis and have the advantage of applicability to a wide range of formalisms and the ability to use existing tools without modification, but require considerable user input in constructing deductive proofs or invariants.

Our type reduction theory, presented in Chapter 5 is in many ways similar to the data independence theories described in Section 2.3. The main differences include:

- instead of being a datatype, the distinguished type determines the number of processes forming a network in our case, and
- the type reduction theory does not directly allow one to solve parameterised verification problems, but requires a suitable abstraction method (however, it can be combined with a number of different abstraction techniques).

In addition, our symbolic operational semantics (presented in Section 4.3) is based on the same ideas as the symbolic operational semantics used by Lazić to prove a data independence theory for CSP [Laz99]. However, we aim for conciseness and simplicity in our operational semantics, while sacrificing some degree of generality.

Finally, we note the use of a CEGAR approach (Section 2.5) in the process of determining small values of thresholds to be used within counter abstraction (see Section 3.3.3).

Chapter 3

Counter abstraction of systems with no identity awareness

The problem we attempt to solve in this chapter is a subclass of the Parameterised Model Checking Problem (see Section 1.4) and can be specified as follows.

> *Given a concurrent system* $Impl(T)$*, consisting of* $\#T$ *similar processes, and a specification* $Impl(T)$*, is it true that* $Impl(T)$ *satisfies* $Spec(T)$ *for all sizes of* T*?*

In this chapter we assume that specifications do not depend on T, so we will omit the parameter from now on. The implementations under consideration consist of a number of identical *node processes*. The general form of such systems is

$$Impl(t) = C[Nodes(t)],$$

where

- $C[\cdot]$ is a CSP context, independent of t,
- $Nodes(t) = \|_{A_{sync}} \; i \in t \bullet \mathcal{N}$,
- $\mathcal{N}$ models a single, finite-state node process, and
- A_{sync} denotes the set of events on which all node processes synchronise.

Here we make the simplifying assumption that all nodes, specifications and contexts used to build implementations do not use node identifiers in their definitions (i.e. are independent of the instantiation T of type t). We will extend the techniques of this chapter to processes that use node identifiers in Chapter 6. Example 3.0.1 presents a verification problem matching the above requirements, which we attempt to solve in this chapter.

Example 3.0.1. Suppose we model a process scheduling mechanism in an operating system for a multiprocessor machine. The implementation consists of a number of nodes, each representing a single process requesting CPU access, and a scheduler.

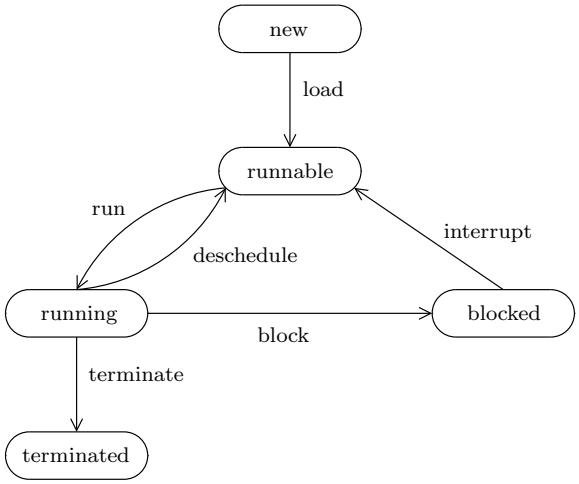


Figure 3.1: State diagram of a basic model for processes within an operating system.

We adopt the common model of process states (e.g. [Tan87]), as shown in Figure 3.1. We model nodes as below.

$$\mathcal{N} = Node(new)$$
$$Node(new) = load \rightarrow Node(runnable)$$
$$Node(runnable) = run \rightarrow Node(running)$$
$$\begin{aligned} Node(running) = {} & deschedule \rightarrow Node(runnable) \\ & \Box\ block \rightarrow Node(blocked) \\ & \Box\ terminate \rightarrow STOP \end{aligned}$$
$$Node(blocked) = interrupt \rightarrow Node(runnable)$$

The process *Core*, below, models a single CPU resource. When *Core* is idle it can *run* and become busy; and when busy, any of the events that imply that a process no longer needs CPU time (i.e. *deschedule*, *block* or *terminate*) brings it back to the idle state.

$$Core(idle) = run \rightarrow Core(busy)$$
$$\begin{aligned} Core(busy) = {} & deschedule \rightarrow Core(idle) \\ & \Box\ block \rightarrow Core(idle) \\ & \Box\ terminate \rightarrow Core(idle) \end{aligned}$$

Let *cores* be the number of processors present in the system (later on, we verify the system for a single value of *cores*). The task of the scheduler is to divide CPU time between the different processes. It consists of *cores* interleaved processes *Core*, each representing a single processor resource. We abstract away the details of the algorithm that the scheduler uses to decide which process should be given access to a

CPU next (letting it pick any available process nondeterministically), and hence our analysis holds for *all* scheduling algorithms.

$$Scheduler = |||\ i \in \{1\,..\,cores\} \bullet Core(idle)$$

In a real system, processes are allowed to load at any time. However, for simplicity and to better demonstrate our method, we assume that all processes are loaded simultaneously. Therefore, we put all the node processes in parallel, synchronising only on the *load* event.

$$Nodes(t) = \underset{\{load\}}{||}\ i \in t \bullet \mathcal{N}$$

To create the implementation, we compose all the nodes in parallel with the scheduler process, synchronising on all the events other than *load*. Finally we rename all the events that imply that a process no longer needs CPU time (i.e. *deschedule*, *block* or *terminate*) to a single event *stopRun*, since, for specification verification purposes, we need not distinguish why a process no longer needs CPU time.

$$Impl(t) = (Nodes(t) \underset{Alpha}{||} Scheduler) \\ [\![^{stopRun,stopRun,stopRun}/_{deschedule,block,terminate}]\!]$$

$$Alpha = \{|\ run, deschedule, block, terminate\ |\}$$

Finally, we would like our specification to say that we never have more than *cores* processes in the *running* state, i.e. the number of *run* events minus the number of *stopRun* events never exceeds *cores*. Further, if there is at least one process running, then it can stop running, so the event *stopRun* is not refused. Finally, the event *interrupt* may or may not be available. Hence, we define

$$Spec = load \rightarrow Spec'(0)$$

$$Spec'(x) = x < cores\ \&\ (run \rightarrow Spec'(x+1) \sqcap STOP) \\ \Box\ x > 0\ \&\ stopRun \rightarrow Spec'(x-1) \\ \Box\ (interrupt \rightarrow Spec'(x) \sqcap STOP).$$

We fix the value of *cores*. Our verification problem is then:

$$\forall\ T \bullet Spec \overset{?}{\sqsubseteq}_{\mathrm{F}} Impl(T). \tag{3.1}$$

End of example.

In our counter abstraction techniques we represent processes using *labelled transition systems*, defined as follows.

Definition 3.0.2. A *labelled transition system (LTS)* is a tuple $\mathcal{L} = (S, s_0, L, \longrightarrow)$, where S is a set of states, $s_0 \in S$ is an initial node, L is a set of labels, and $\longrightarrow \subseteq S \times L \times S$ is a transition relation. We let $\hat{\mathcal{L}} = S$ denote the set of nodes of $\mathcal{L}$.

Throughout this chapter (and also Chapter 6) we will often use a special kind of an LTS, a *counter state machine*, to represent counter abstracted models[1]:

Definition 3.0.3. A k-dimensional *counter state machine* is a state machine whose states are k-tuples of integer counters. Whenever k is clear from context we refer to any such machine simply as a counter state machine.

Counter abstraction works by transforming a concrete model of node processes into an abstract counter state machine, independent of T. The main idea is to abstract away the information about *which* nodes are currently in a given state and only record the *number* of processes in that state. Therefore, counter c_i of an abstract state $(c_1, \ldots, c_k)$ counts how many node processes are currently in the i-th state within the concrete parallel composition of all nodes, $Nodes(T)$. Then, each counter c_i is given a threshold z_i and we interpret $c_i = z_i$ to mean that there are z_i *or more* processes in the i-th state.

The rest of this chapter is structured as follows. In Section 3.1 we discuss decidability of uniform verification of implementations defined as above. Section 3.2 describes the technique of counter abstraction with unbounded integer counters (we use the word "unbounded" to mean that we do not put any cap on the values of the counters; for every given type T, the values of the counters are obviously bounded by the size of T, the number of processes present in the system). It is observed that, even though this may dramatically reduce the number of states in the state machine representation of the system, the state space still depends on T (since each counter is allowed to take values between 0 and $\#T$). We prove that such a system is (strongly) bisimilar to the concrete parallel composition of node processes, and therefore it is traces and stable failures equivalent to $Nodes(T)$. The abstraction is improved further by adding thresholds to the counters in Section 3.3. We show that this creates abstract models that form anti-refinements of the original system and that are independent of the number of node processes. This allows us to perform a single refinement check against *Spec* in order to conclude that $Impl(T)$ refines *Spec* for all but some small types T. We demonstrate our techniques by applying them to verification problem (3.1) from Example 3.0.1. In Section 3.3.3 we discuss how thresholds that are too small can lead to spurious counterexamples and we present an algorithm for finding threshold functions that avoid spurious counterexamples. Finally, we conclude in Section 3.4.

3.1 Decidability results

In Section 1.4 we noted that PMCP is, in general, undecidable. However, the reduction of the Halting Problem in [AK86] does not imply undecidability of the subclass of PMCP that we consider in this chapter; node processes used by Apt and Kozen strongly depend on the instantiation of the distinguished type, which we do not allow here. In this section we discuss decidability of uniform verification of the families

[1] Our notion of counter state machines is similar, but more general than that of a vector addition systems (VAS) [HP76] with labelled transitions, since a counter state machine may test for zero, while a VAS cannot.

of systems defined in the introduction of this chapter. In particular, the results we present below demonstrate the existence of a decidability boundary between systems in which node processes are interleaved and systems in which they can all synchronise on some common events.

Firstly, we provide a restriction of the family of systems that we consider in this chapter for which the problem of uniform verification becomes decidable. We say that a context $C[\cdot]$ is finite-state, if for every finite-state P, $C[P]$ is finite state.

Theorem 3.1.1. *The problem of verifying if Spec* $\sqsubseteq$ $C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right.\right]$ *for all* T*, where* $C[\cdot]$*,* $\mathcal{N}$ *and Spec are all finite-state and* t*-independent, and the refinement is either in the traces or the stable failures model, is decidable.*

The proof of this theorem, below, is based on simulating CSP processes using vector addition systems with states, which we define as follows.

Definition 3.1.2. A *k-dimensional vector addition system with states* (*VASS*) is a tuple

$$\mathcal{V} = (Q, V, \delta, q_0, v_0),$$

where:

- Q is a finite state of control states,
- $V \subseteq \mathbb{Z}^k$ is a set of addition vectors,
- $\delta \subseteq Q \times V \times Q$ is a transition relation,
- q_0 is the initial control state, and
- $v_0 \in \mathbb{N}^k$ is the initial vector.

The configurations of $\mathcal{V}$ are pairs (q, v) in $Q \times V$. A configuration (q, v) can evolve to a configuration $(q', v + w)$, provided that $(q, w, q') \in \delta$ and $v + w \geq 0$.

Definition 3.1.3. Given a configuration (q, v) of a VASS $\mathcal{V}$, the *coverability* problem asks whether a configuration (q, v') such that $v' \geq v$ is reachable within $\mathcal{V}$.

Theorem 3.1.4. *The coverability problem over vector addition systems with states is decidable.*

Proof: It is shown in [KM69] and [Rac78] that the coverability problem over vector addition systems (without states) (VAS) is decidable. However, every k-dimensional VASS can be simulated by a $(k + 3)$-dimensional VAS [HP76], so the result follows. ∎

The proof of Theorem 3.1.1 is in many ways similar to that in [GS92], where German and Sistla show decidability of model checking of systems with replicated components, running in parallel with a controller, against specifications expressed in a temporal logic. Their proof uses a construction of a VASS from three automata:

one for a node process, one for a controller and one that is equal to the complement of the automaton that accepts precisely the inputs that satisfy a given specification formula. In our proof, instead of complementing the specification automaton, we use a watchdog specification transformation [RGM$^+$03, Ros10] that "moves" a specification from the left-hand side of a refinement check to its right-hand side. To do so, we use a process WDT (which is finite-state if *Spec* is), which allows all the traces that *Spec* does, but after performing a trace that *Spec* does not allow, it performs the $fail\_$ event. We then have the following result (from [RGM$^+$03]).

Lemma 3.1.5. *Given a finite-state specification Spec and an implementation definition Impl(t), we have that for all T,*

$$Spec \sqsubseteq_{\mathrm{T}} Impl(T) \quad \textit{iff} \quad Chaos(\Sigma \setminus \{fail\_\}) \sqsubseteq_{\mathrm{T}} Impl(T) \underset{\Sigma\setminus\{fail\_\}}{\parallel} WDT.$$

A similar technique exists for the stable failures model. This time, however, in addition to the traces refinement check using the WDT process, we also[2] use the watchdog specification transformation process WDF (which is finite-state if *Spec* is) from [RGM$^+$03] and perform a deadlock-freedom check[3]. We then have the following result (based on a discussion in [RGM$^+$03]).

Lemma 3.1.6. *Given a finite-state specification Spec and an implementation definition Impl(t), we have that for all T,*

$$Spec \sqsubseteq_{\mathrm{F}} Impl(T) \quad \textit{iff} \quad \begin{pmatrix} Chaos(\Sigma \setminus \{fail\_\}) \sqsubseteq_{\mathrm{T}} Impl(T) \underset{\Sigma\setminus\{fail\_\}}{\parallel} WDT \\ \wedge\ Impl(T) \underset{\Sigma}{\parallel} WDF \text{ is deadlock-free} \end{pmatrix}.$$

Proof (of Theorem 3.1.1): We first prove the result for the traces model. Let WDT be the watchdog specification transformation process for the traces model. Then, on letting $Spec' = Chaos(\Sigma \setminus \{fail\_\})$, Lemma 3.1.5 implies that for all T,

$$Spec \sqsubseteq_{\mathrm{T}} C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right.\right] \quad \text{iff} \quad Spec' \sqsubseteq_{\mathrm{T}} C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right.\right] \underset{\Sigma\setminus\{fail\_\}}{\parallel} WDT. \quad (3.2)$$

Let $st_1, \ldots, st_k$ be the states of $\mathcal{N}$, where st_1 is the initial state. We simulate $C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right.\right] \underset{\Sigma\setminus\{fail\_\}}{\parallel} WDT$ for all non-empty and finite T using a single k-dimensional VASS $\mathcal{V}(\mathcal{N}, C, WDT)$ using the following construction (a similar construction, with more details, can be found in [GS92]):

[2]It is shown in [RGM$^+$03] that the same can be achieved using a single deadlock-freedom check using only the WDF process, but the construction uses the CSP interrupt operator, $\triangle$, which is not included in the language considered in this thesis (see Section 8.3 for discussion on adding $\triangle$ to our language).

[3]Every deadlock-freedom check can be expressed using a stable failures refinement check: a process P is deadlock-free if and only if $DF(\Sigma) \sqsubseteq_{\mathrm{F}} P$, where $DF(A) = \prod a \in A \bullet a \rightarrow DF(A)$ is the most nondeterministic process with alphabet A.

- We want to talk about the states of context $C[\cdot]$. Formally, we mean these to be the states of $C[Run(\Sigma \setminus \{fail\_\})]$, i.e. where we allow all transitions of the context and ignore the states of the nodes. Then, we define the control states of $\mathcal{V}(\mathcal{N}, C, WDT)$ to be pairs of states of $C[\cdot]$ and WDT (observe that there are only finitely many such pairs), extended with some distinct initial state q_0.
- Every transition of $C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right] \underset{\Sigma\setminus\{fail\_\}}{\|} WDT$, labelled with an event a, can be considered as $((st_c, st_w), i) \xrightarrow{a} ((st'_c, st'_w), j)$, which indicates that the context changes its state from st_c to st'_c, the watchdog process changes its state from st_w to st'_w and some node process moves from state st_i to state st_j (observe that since all processes are identical, we do not lose any information by not recording which node performs this transition), where $i = j$ if no node participates in the transition. We simulate each such transition in $\mathcal{V}(\mathcal{N}, C, WDT)$ using a transition $((st_c, st_w), (v_1, \ldots, v_k), (st'_c, st'_w))$ such that $(v_1, \ldots, v_k)$ indicates between which states (if any) a node process moves, i.e. for all h different from i and j, $v_h = 0$, and if $i \neq j$, then $v_i = -1$ and $v_j = 1$, otherwise $v_i = v_j = 0$.
- We let the initial vector be $(0, \ldots, 0)$.
- We allow on-the-fly node process generation by including VASS transitions $(q_0, (1, 0, \ldots, 0), q_0)$ and $(q_0, (0, \ldots, 0), (st_0^c, st_0^w))$, where st_0^c and st_0^w are the initial states of $C[Run(\Sigma \setminus \{fail\_\})]$ and WDT, respectively. These transitions allow us to initially introduce an arbitrary number of node processes, say n, all of which are in the initial state, st_1 and then evolve a state corresponding the initial state of $C\left[\left|\left|\left| i \in \{0 \,..\, n-1\} \bullet \mathcal{N}\right.\right.\right] \underset{\Sigma\setminus\{fail\_\}}{\|} WDT$.

We have that $C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right] \underset{\Sigma\setminus\{fail\_\}}{\|} WDT$ can perform some trace and reach a state in which the event $fail\_$ is available if and only if $\mathcal{V}(\mathcal{N}, C, WDT)$ has a reachable state $((st_c, st_w), v)$ such that st_w can communicate $fail\_$. This means that the problem of deciding if $Spec' \sqsubseteq_{\mathrm{T}} C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right] \underset{\Sigma\setminus\{fail\_\}}{\|} WDT$ for all T is equivalent to coverability of all the states $((st_c, st_w), (0, .., 0))$ such that st_w can communicate $fail\_$ (observe that there may be only finitely many such pairs (st_c, st_w)). Then, Theorem 3.1.4, combined with (3.2), proves the result for the traces model.

The proof for the stable failures is an extension of the above. In addition to using the WDT process, we now use the watchdog specification transformation process WDF, defined as in [RGM+03]. Then, Lemma 3.1.6 implies that for all T,

$$Spec \sqsubseteq_{\mathrm{F}} C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right] \text{ iff } \begin{pmatrix} Spec' \sqsubseteq_{\mathrm{T}} C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right] \underset{\Sigma\setminus\{fail\_\}}{\|} WDT \wedge \\ C\left[\left|\left|\left| i \in T \bullet \mathcal{N}\right.\right.\right] \underset{\Sigma}{\|} WDF \text{ is deadlock-free} \end{pmatrix}, \quad (3.3)$$

where $Spec'$ is as before. We have already shown that verifying if $Spec' \sqsubseteq_{\mathrm{T}} Impl(T) \underset{\Sigma\setminus\{fail\_\}}{\|} WDT$ for all T is decidable. We can now simulate

$C\left[\left|\left|\right| i \in T \bullet \mathcal{N}\right] \underset{\Sigma}{\|} WDF$ using a VASS (constructed as above). Since the deadlock-freedom problem is decidable over vector addition systems[4] [EN94], the problem of testing if $C\left[\left|\left|\right| i \in T \bullet \mathcal{N}\right] \underset{\Sigma}{\|} WDF$ for all T is decidable. Therefore, (3.3) implies the result for the stable failures model. ∎

The simulation of implementations used in the above proof works, because an interleaving of processes cannot model a counter with the ability to test it for the zero value. However, if we allow the nodes to be composed in generalised parallel, i.e. we consider implementations of the form

$$C\left[\underset{A_{Sync}}{\|} \; i \in t \bullet \mathcal{N}\right],$$

where $C[\cdot]$ and $\mathcal{N}$ are as before and A_{Sync} is some set of events, then the simulation breaks. This is because vector addition systems cannot model zero testing. In fact, we can show that the addition of synchronisations between nodes makes the problem of uniform verification undecidable. We do so by proving a stronger result for the case where nodes run without the presence of any context.

Theorem 3.1.7. *The problem of verifying if Spec* $\sqsubseteq \underset{A_{Sync}}{\|} \; i \in T \bullet \mathcal{N}$ *for all* T*, where Spec and* $\mathcal{N}$ *are finite-state and t-independent, and where the refinement is either in the traces model or in the stable failures model, is undecidable.*

The proof of this theorem, below, is based on using CSP processes to simulate two-counter machines [Min67] and reducing the Halting Problem (well-known to be undecidable [Sip96]) to the problem of uniform verification of $\underset{A_{Sync}}{\|} \; i \in T \bullet \mathcal{N}$.

Informally, a two-counter machine is a Turing machine [Sip96] with two unbounded, non-negative integer registers (counters) in place of the usual work tape. The head of a two-counter state machine reads inputs from a single read-only input tape (after which it can move to the left, right or stay in place) and a control automaton performs actions from the following instruction set:[5]: increment a counter, decrement a counter, or test if a counter is 0 and change the control state accordingly. The following definition captures this formally.

Definition 3.1.8. A *two-counter machine* is a tuple

$$\mathcal{M} = (Q, \Gamma, \delta, q_0, Q_A),$$

where

[4]Even though Esparza gives decidability over Petri nets [Pet62], it is a well-known fact that Petri nets are equivalent to vector addition systems with states [Hac74, KM82].

[5]Two-counter machines can also be defined with alternative definitions of the instruction set. Here, we use the model due to Minsky [Min67] with three instructions: increment, decrement and test for zero. Other models use instructions that can clear registers, copy the value of a register to another register, compare values of registers, etc. Some of many such alternative models can be found in [Min61, Lam61, SS63, ER64, BBJ02].

- Q is a finite set of control states,
- Γ is a finite alphabet,
- $\delta : (Q \times \Gamma \times \{True, False\}^2) \to (Q \times \{*, +, -\} \times \{1, 2\})$ is a transition function that reads four inputs: the current control state, the input symbol and two Boolean flags that indicate whether the values of the two counters are equal to 0 (in which case the appropriate flag equals *True*; otherwise it equals *False*), and returns a new control state and an instruction to leave counter unmodified ($*$), increment the indicated counter ($+$) or decrement the indicated counter ($-$); in addition, the transition function is restricted to returning only the increment instruction for non-zero counters only, i.e. if $\delta(q, a, f_1, f_2) = (q', pm, i)$, where $f_j = 0$ for some $j \in \{1, 2\}$ and $pm = -$, then $i \neq j$,
- q_0 is the initial state, and
- Q_A is a set of accepting states.

The configurations of $\mathcal{M}$ are triples $(q, c_1, c_2) \in Q \times \mathbb{N} \times \mathbb{N}$. Given an input a from Γ, a configuration (q, c_1, c_2) can evolve to a configuration (q', c_1', c_2'), provided that $\delta(q, a, f_1, f_2) = (q', pm, i)$, where for j in $\{1, 2\}$, $f_j = True$ if and only if $c_j = 0$ (i.e. the flags represent the values of counters correctly), if $pm = -$, then $f_i = False$ (i.e. the transition does not decrease a counter whose value is 0), if $pm = +$, then $c_i' = c_i + 1$, if $pm = -$, then $c_i' = c_i - 1$ and if $pm = *$, then $c_i' = c_i$ (i.e. depending on pm, counter i is incremented, decremented or left unmodified), and $c_{3-i}' = c_{3-i}$ (i.e. the other counter remains as before).

Theorem 3.1.9. *The Halting Problem is undecidable over two-counter machines.*

Proof: Follows from the fact that two-counter machines are Turing-equivalent [Min67], and that the Halting Problem is undecidable over Turing machines [Sip96]. ∎

Proof (of Theorem 3.1.7): Since refinement in the traces model reduces to refinement in the stable failures model, it is enough to prove undecidability in the traces model. Let $\mathcal{M}$ be an arbitrary two-counter machine. Let

$$\begin{aligned} Cell(i) &= up.i \to Cell'(i) \\ &\quad \Box \\ &\quad zero.i \to Cell(i) \\ Cell'(i) &= down.i \to Cell(i). \end{aligned}$$

Then,

$$Counter(i, t) = \|_{\{zero.i\}} \; id \in t \bullet Cell(i)$$

models a single register (counter), initialised to 0, with non-negative integer values no greater than $\#t$ and which has a test for zero (as $Counter(i, t)$ can communicate the

zero.i event if an only the value of the counter it models is 0). Also, let *Ctrl* model the finite-state control automaton of $\mathcal{M}$ in a way that the event *halt* is communicated if and only if $\mathcal{M}$ is in an accepting state. Then, for all T,

$$\begin{array}{ll} & \exists\, tr \bullet Ctrl \underset{\{|zero,up,down|\}}{\parallel} (Counter(1,T) \,|||\, Counter(2,T)) \xRightarrow{tr\,\hat{}\,\langle halt\rangle} \\ \Leftrightarrow & \mathcal{M} \text{ halts with both counters never exceeding } \#T. \end{array} \tag{3.4}$$

If we let

$$\mathcal{N} = Cell(1) \,|||\, Cell(2),$$

then for all T,

$$Counter(1,T) \,|||\, Counter(2,T) \equiv_{\mathrm{T}} \Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N}.$$

Hence, (3.4) implies that for all T,

$$\begin{array}{ll} & \exists\, tr \bullet Ctrl \underset{\{|zero,up,down|\}}{\parallel} \Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N} \xRightarrow{tr\,\hat{}\,\langle halt\rangle} \\ \Leftrightarrow & \mathcal{M} \text{ halts with both counters never exceeding } \#T. \end{array}$$

Let $Spec = Chaos(\Sigma \setminus \{halt\})$. Then, the above implies that

$$\begin{array}{ll} & Spec \not\sqsubseteq_{\mathrm{T}} Ctrl \underset{\{|zero,up,down|\}}{\parallel} \Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N} \\ \Leftrightarrow & \mathcal{M} \text{ halts with both counters never exceeding } \#T. \end{array}$$

This means that

$$\begin{array}{ll} & \forall\, T \bullet Spec \sqsubseteq_{\mathrm{T}} Ctrl \underset{\{|zero,up,down|\}}{\parallel} \Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N} \\ \Leftrightarrow & \mathcal{M} \text{ does not halt.} \end{array} \tag{3.5}$$

Finally, let *Spec*′ be like *Spec*, except that after every behaviour not allowed by *Ctrl*, it allows arbitrary behaviour. Then, for every T we have that

$$Spec \sqsubseteq_{\mathrm{T}} Ctrl \underset{\{|zero,up,down|\}}{\parallel} \Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N}$$

if and only if

$$Spec' \sqsubseteq_{\mathrm{T}} \Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N}.$$

This, combined with (3.5) implies that

$$\begin{array}{ll} & \forall\, T \bullet Spec' \sqsubseteq_{\mathrm{T}} \Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N} \\ \Leftrightarrow & \mathcal{M} \text{ does not halt.} \end{array}$$

Since $\mathcal{M}$ is arbitrary, deciding if it halts over two-counter machines is undecidable (Theorem 3.1.9). Hence, the above implies that the problem of uniform verification of $\Big\|_{\{|zero|\}} i \in T \bullet \mathcal{N}$ is undecidable. ∎

3.2 Counter abstraction with unbounded counters

In this section we present a transformation method that generates a counter state machine bisimilar to $Nodes(T)$. Every state in this system is a tuple of integer counters, each counting how many node processes are in a given concrete state.

Let a node process $\mathcal{N}$ be represented using an LTS $(S, s_0, \Sigma^\tau, \longrightarrow)$. Suppose that each node process has exactly k local states, say $st_1, \ldots, st_k$. We assume, without loss of generality, that $st_1 = s_0$. Let T be a fixed type. Then each state of the state machine of $Nodes(T)$ can be represented as a tuple of the form $(st_{f(1)}, \ldots, st_{f(\#T)})$ for some function $f : T \to \{1 \,..\, k\}$ that maps every node identity to the index of the state in which the nodes is. Then $(st_{f(1)}, \ldots, st_{f(\#T)})$ corresponds to the process $\Big\|_{A_{Sync}} \; i \in T \bullet st_{f(i)}$ (i.e. the i-th node process is in the state $st_{f(i)}$).

3.2.1 Constructing the counter state machine

We now define a method that, using the LTS of $\mathcal{N}$, generates a counter state machine

$$\zeta_\infty^T(\mathcal{N}) = (A_\infty(T), a_\infty(T), \Sigma_\infty^\tau, \longrightarrow_\infty(T))$$

of an abstract model $\zeta_\infty^T(\mathcal{N})$ that is bisimilar to $Nodes(T)$. Whenever T is clear from context, we omit it from the right-hand side and simply write $(A_\infty, a_\infty, \Sigma_\infty^\tau, \longrightarrow_\infty)$.

Let T be fixed. The states in A_∞ are k-tuples of non-negative integer counters

$$(c_1, \ldots, c_k)$$

such that $\Sigma_{j=1}^k c_j = \#T$. This means that $(0, \ldots, 0)$ is not a state in A_∞, as we assumed that $\#T > 0$ (see Section 1.7). A state $(c_1, \ldots, c_k)$ corresponds to the concrete states

$$(st_{f(1)}, \ldots, st_{f(\#T)})$$

for functions f from T to $\{1 \,..\, k\}$ and where each c_i counts the number of nodes in state st_i, i.e.

$$\forall\, i \in \{1 \,..\, k\} \bullet c_i = \#\{j \in T \mid st_{f(j)} = st_i\}.$$

The initial state a_∞ is the tuple $(\#T, 0, \ldots, 0)$. This naturally corresponds to the situation where all the node processes are in their initial states.

To define the set of labels of the counter machine, Σ_∞^τ, it is enough to observe that the transformed system can perform only those events that a single node process can. Hence, we let

$$\Sigma_\infty^\tau = \Sigma^\tau.$$

In addition, we let $\Sigma_\infty = \Sigma_\infty^\tau \setminus \{\tau\}$.

Finally, we define the transition relation $\longrightarrow_\infty$. Let $(c_1, \ldots c_k)$ and $(c'_1, \ldots c'_k)$ be two states in A_∞. We perform a case analysis on the type of events in Σ_∞^τ.

When a is a private visible event of a node process or a τ, there is an a-transition between $(c_1, \ldots c_k)$ and $(c'_1, \ldots c'_k)$ if and only if there exist state indexes i and j in $\{1 \,..\, k\}$ such that

- there is an a-transition of some node process from state st_i to state st_j, with c_i having a value of at least 1 (i.e. at least one node process is in state st_i), and
- the corresponding counters in the abstract state change correctly: if st_i and st_j are different states, then c_i is decreased by 1, c_j is increased by 1 and all the other counters remain unchanged; if, however, st_i and st_j are the same state, then all the counters remain unchanged.

Formally, for every a in $\Sigma^{\tau}_{\infty} \setminus A_{Sync}$,

$$\begin{aligned} & (c_1, \ldots, c_k) \xrightarrow{a}_{\infty} (c'_1, \ldots, c'_k) \\ \Leftrightarrow \quad & \exists\, i, j \in \{1 \,..\, k\} \bullet st_i \xrightarrow{a} st_j \wedge c_i \geq 1 \\ & \qquad \wedge (i \neq j \wedge c'_i = c_i - 1 \wedge c'_j = c_j + 1 \\ & \qquad\qquad \wedge (\forall\, h \in \{1 \,..\, k\} \setminus \{i, j\} \bullet c'_h = c_h) \\ & \qquad \vee\ i = j \wedge \forall\, h \in \{1 \,..\, k\} \bullet c'_h = c_h). \end{aligned} \tag{3.6}$$

In the second case, when e is a visible event shared between all the node processes, we enable an e-labelled transition between $(c_1, \ldots c_k)$ and $(c'_1, \ldots c'_k)$ if and only if

- for every i in $\{1 \,..\, k\}$, there are precisely c_i node processes that are in state st_i and can perform e to reach some target states tgt^i_j for j in $\{1 \,..\, c_i\}$, and
- for every i in $\{1 \,..\, k\}$, precisely c'_i of all the target states are equal to st_i.

Formally, for every e in A_{Sync}, we define

$$\begin{aligned} & (c_1, \ldots, c_k) \xrightarrow{e}_{\infty} (c'_1, \ldots, c'_k) \\ \Leftrightarrow \quad & \exists\, tgt : \mathbb{N} \times \mathbb{N} \to S \bullet \\ & \qquad (\forall\, i \in \{1 \,..\, k\}, j \in \{1 \,..\, c_i\} \bullet st_i \xrightarrow{e} tgt^i_j) \\ & \qquad \wedge \forall\, l \in \{1 \,..\, k\} \bullet \\ & \qquad\qquad c'_l = \#\{(i, j) \mid i \in \{1 \,..\, k\} \wedge j \in \{1 \,..\, c_i\} \wedge tgt^i_j = st_l\}. \end{aligned} \tag{3.7}$$

Observe that the two parts of the definition of $\longrightarrow_{\infty}$ are disjoint, so together they combine into a well-formed definition of the abstract transition relation.

Example 3.2.1. Recall the $\mathcal{N}$ process syntax we defined in Example 3.0.1. For any given T we can use the method described above to construct $\zeta^T_{\infty}(\mathcal{N})$ using the LTS of $\mathcal{N}$ (Figure 3.1). Figure 3.2 shows $\zeta^{\{1,2\}}_{\infty}(Node)$, where the states of $\mathcal{N}$ have been numbered in the following order: *new*, *runnable*, *running*, *blocked*, *terminated* (e.g. state $(0, 1, 0, 1, 0)$ indicates that there is a single node in the *runnable* state and a single node in the *blocked* state and no nodes in any of the three remaining states).

End of example.

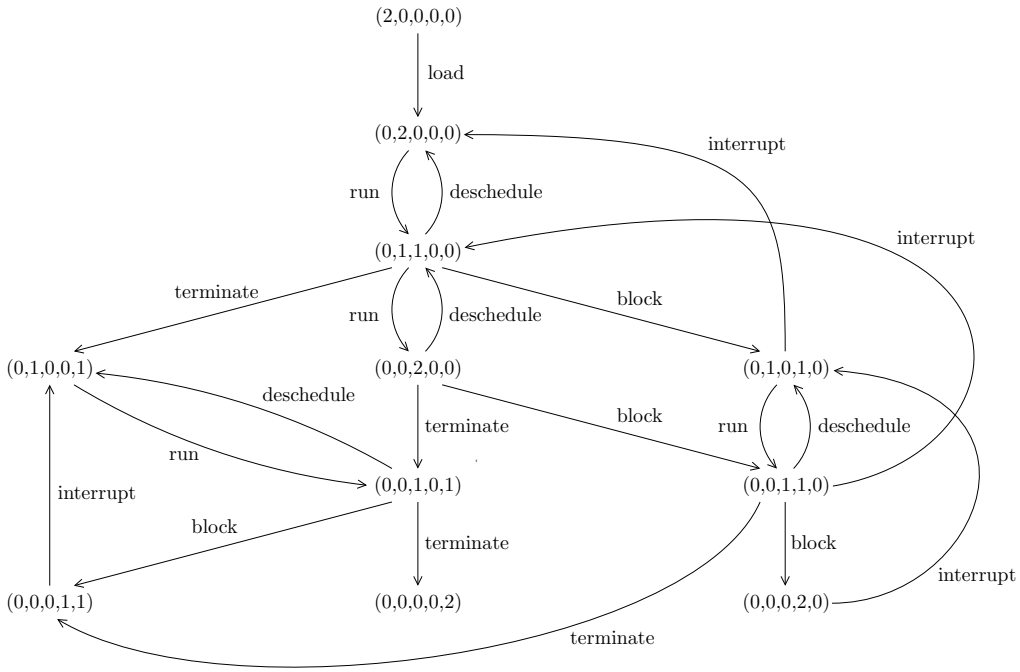


Figure 3.2: Counter state machine $\zeta_{\infty}^{\{1,2\}}(\mathcal{N})$ for $\mathcal{N}$ as in Example 3.0.1.

3.2.2 Denotational equivalence results

In this section we prove that, given a concrete instantiation T of type t, the counter state machine $\zeta_{\infty}^{T}(\mathcal{N})$, built using the method presented in the previous section, is traces and stable failures equivalent to $Nodes(T)$. We do so by showing that $Nodes(T)$ and $\zeta_{\infty}^{T}(\mathcal{N})$ are strongly bisimilar. This results comes as no surprise, as:

- the $\|$ operator is commutative and associative, so the behaviour of $Nodes(T)$ does not depend on the order of the nodes,
- no node identifiers are used, so the behaviour of two distinct nodes that are in the same state is identical, and
- counter abstraction with unbounded counters factors the original state space by node ordering (i.e. it identifies symmetrically equivalent global states).

Proposition 3.2.2. *For every instantiation T of type t, $Nodes(T)$ and $\zeta_{\infty}^{T}(\mathcal{N})$ are strongly bisimilar.*

Proof sketch: By showing that

$$\mathcal{B} \mathrel{\widehat{=}} \left\{ \left(\Big\|_{A_{Sync}} \; j \in T \bullet st_{f(j)}, (c_1, \ldots, c_k) \right) \mid f : T \to \{1 \,..\, k\} \right.$$
$$\left. \wedge \, \forall \, i \in \{1 \,..\, k\} \bullet c_i = \#\{j \in T \mid f(j) = i\} \right\}$$

is a strong bisimulation relation between the states of $Nodes(T)$ and the states of $\zeta_\infty^T(\mathcal{N})$. The bisimilarity of $Nodes(T)$ and $\zeta_\infty^T(\mathcal{N})$ follows then immediately by observing that their initial states ($\underset{A_{Sync}}{\|}\ j \in T \bullet s_0$ and $(\#T, 0, \ldots, 0)$, respectively) are related under $\mathcal{B}$. ∎

Corollary 3.2.3. *For every instantiation T of type t, we have that*

- $\zeta_\infty^T(\mathcal{N}) \equiv_{\mathrm{T}} Nodes(T)$, *and*
- $\zeta_\infty^T(\mathcal{N}) \equiv_{\mathrm{F}} Nodes(T)$.

3.3 Counter abstraction with finite thresholds

In the previous section we showed how to create a state machine based on integer counters, which is traces and stable failures equivalent to the parallel composition of all node processes. Even though such a method offers a dramatic decrease in the number of states by factoring the states with respect to bisimulation, the state machine is still dependent on T (as each counter can take values between 0 and $\#T$). In this section we present an abstraction method, where we introduce a *threshold function* z that defines upper bounds on the values that counters can take.

3.3.1 Constructing the counter state machine

Recall that $(S, s_0, \Sigma^\tau, \longrightarrow)$ is the state machine of $\mathcal{N}$, where $S = (st_1, \ldots st_k)$. Let z be a function from $\{1 \,..\, k\}$ to $\mathbb{N}$. For every i in $\{1 \,..\, k\}$, z_i is a threshold for the counter corresponding to st_i. Our aim is to create a counter state machine

$$\zeta_z(\mathcal{N}) = (A_z, a_z, \Sigma_z^\tau, \longrightarrow_z),$$

independent of T, which forms an abstraction of the original parallel composition of all node processes.

Definition 3.3.1. We define the states in A_z to be tuples of counters $(c_1, \ldots, c_k)$ such that

(i) for every i in $\{1 \,..\, k\}$, c_i is an integer between 0 and z_i, and

(ii) either there exists i in $\{1 \,..\, k\}$ such that $c_i = z_i$ or $\sum_{i=1}^k c_i \geq z_1$.

Observe that clause (ii) of the above definition implies that we only allow tuples of counters that imply a possibility of presence of at least z_1 nodes. This, together with the definition of the initial state, below, ensures that at no point the counters imply presence of fewer process than we started with. Although this is not a necessary requirement, it allows us to avoid some spurious counterexamples.

The addition of the threshold function requires a new definition for the initial state. Throughout this section we assume that $\#T \geq z_1$ (the thresholds are usually small, so all the refinement checks for $\#T < z_1$ can be performed directly) and we let

$$a_z = (z_1, 0, \ldots, 0).$$

The alphabet of the abstract counter state machine is identical to that of $\zeta_\infty^T(\mathcal{N})$, i.e.

$$\Sigma_z^\tau = \Sigma_\infty^\tau = \Sigma^\tau.$$

In addition, we let $\Sigma_z = \Sigma_z^\tau \setminus \{\tau\}$.

Finally, we define the transition relation, $\longrightarrow_z$. Let $(c_1, \ldots c_k)$ and $(c'_1, \ldots c'_k)$ be two states in A_z, as defined in Definition 3.3.1. Intuitively, there is an a-transition available between $(c_1, \ldots c_k)$ and $(c'_1, \ldots c'_k)$ if

- there are an appropriate number (possibly exceeding the thresholds) of processes that need to participate in a in states in which they can execute a,
- the values of the counters of $(c_1, \ldots c_k)$ indicate that there are enough of such processes to perform a, and
- the counters of $(c'_1, \ldots c'_k)$ are updated to reflect the number of processes in corresponding states after a happens, without exceeding their thresholds.

For each kind of event that a can be this translates to the following.

Let a be an event in $\Sigma_z^\tau \setminus A_{Sync}$. Then there is a transition labelled with a between $(c_1, \ldots c_k)$ and $(c'_1, \ldots c'_k)$, if and only if

- there is a local state st_i in S in which a can be executed to reach some state st_j,
- given i in $\{1 \,..\, k\}$, if $z_i > 0$, then the counter corresponding to st_i is at least 1; when $z_i = 0$, there can always be an arbitrary number of processes in st_i, so we always enable the transition (forming a traces anti-refinement), and
- the counters $c'_1, \ldots, c'_k$ are updated correctly: if st_i and st_j are different states then we decrease the counter corresponding to st_i, i.e. c_i, by 1, without going below 0, to get c'_i, and increase the counter corresponding to st_j, i.e. c_j, by 1, without going over threshold z_j, to get c'_j, with both operations happening atomically; in addition, if c_i equals its threshold, z_i, then a second transition is available in the abstract model, since there could be more than z_i nodes in state st_i, so after a happens there would still be at least z_i nodes in state st_i (i.e. $c'_i = c_i$) and c_j gets increased by 1, without going over threshold z_j, to get c'_j (this forms a traces and stable failures anti-refinement); if, however, st_i and st_j are the same state, then all the counters remain unchanged.

Formally, for every a in $\Sigma_z^\tau \setminus A_{Sync}$ we define

$$\begin{aligned}
& (c_1, \ldots, c_k) \xrightarrow{a}_z (c'_1, \ldots, c'_k) \\
\Leftrightarrow \quad & \exists\, i, j \in \{1 \,..\, k\} \bullet \\
& \quad st_i \xrightarrow{e} st_j \wedge (c_i \geq 1 \vee c_i \geq z_i) \\
& \quad \wedge ((i \neq j \wedge (c'_i = \max\{c_i - 1, 0\} \wedge c'_j = \min\{c_j + 1, z_j\} \\
& \qquad\qquad \vee\ c_i = z_i = c'_i \wedge c'_j = \min\{c_j + 1, z_j\}) \\
& \qquad \wedge (\forall\, h \in \{1 \,..\, k\} \setminus \{i, j\} \bullet c'_h = c_h)) \\
& \qquad \vee\ i = j \wedge \forall\, h \in \{1 \,..\, k\} \bullet c'_h = c_h).
\end{aligned} \tag{3.8}$$

The following example illustrates the above definition and the rationale behind clause (ii) of Definition 3.3.1.

Example 3.3.2. Let st_1, st_2 and st_3 be states of a node process such that

$$st_1 \xrightarrow{a} st_2 \qquad \text{and} \qquad st_2 \xrightarrow{b} st_3.$$

Also, let $z_1 = z_3 = 2$ and $z_2 = 1$. Then, using (3.8), we have that

$$(2,0,0) \xrightarrow{a}_z \xrightarrow{a}_z (0,1,0) \xrightarrow{b}_z (0,1,1).$$

However,

$$(2,0,0) \xrightarrow{a}_z \xrightarrow{a}_z (0,1,0) \not\xrightarrow{b}_z (0,0,1), \tag{3.9}$$

as, according to Definition 3.3.1, $(0,0,1)$ is not a state in A_z. Since $z_3 = 1$, state $(0,0,1)$ implies the presence of exactly one node in the system, while state $(2,0,0)$ indicates the presence of at least two nodes. Hence, the b-transition in (3.9) corresponds to a behaviour that changes the number of processes, which is never possible in a concrete implementation.

End of example.

Now, let e be an event in A_{Sync}. We want $(c_1, \ldots c_k)$ and $(c'_1, \ldots c'_k)$ to be related by $\xrightarrow{e}_z$ if and only if there exist e-transitions that allow simultaneous movement of *all* node processes between states of S such that for every i in $\{1 \,..\, k\}$, a number of processes indicated by c_i can move from state st_i and a number of processes indicated by c'_i can move into state st_i. Recall that if for some i in $\{1..k\}$, the counter value c_i is equal to its threshold z_i, then the actual number of processes in the state st_i, call it v_i, is greater or equal to c_i. Observe that, since $\#T \geq z_1$ (and we assumed that $\#T \geq 1$ in Section 1.7), it must be that $\sum_{i=1}^{k} v_i \geq \max\{z_1, 1\}$. Therefore, a movement of all the processes can happen if and only if there exists $v : \{1 \,..\, k\} \to \mathbb{N}$ such that

(i) $\sum_{i=1}^{k} v_i \geq \max\{z_1, 1\}$,

(ii) if, given i in $\{1 \,..\, k\}$, c_i has not reached its threshold (i.e. if $c_i < z_i$), then $v_i = c_i$ and if $c_i = z_i$, then $v_i \geq c_i$, and

(iii) there exists a concrete model such that for every i in $\{1 \,..\, k\}$, precisely v_i node processes within the model are in state st_i, where they can perform e to reach some target states such that if c'_i has not reached its threshold, then precisely c'_i of all the target states are equal to st_i and if $c'_i = z_i$, then at least c'_i of all the target states are equal to st_i.

Formally, for every a in A_{Sync}, we define

$$
\begin{array}{ll}
 & (c_1,\ldots,c_k) \xrightarrow{a}_z (c'_1,\ldots,c'_k) \\
\Leftrightarrow & \exists\, v : \{1\,..\,k\} \to \mathbb{N} \bullet \\
 & \quad \exists\, tgt : \mathbb{N}\times\mathbb{N} \to S \bullet \\
 & \qquad \textstyle\sum_{i=1}^{k} v_i \geq \max\{z_1, 1\} \\
 & \qquad \wedge\ (\forall\, i \in \{1\,..\,k\} \bullet v_i = c_i < z_i \vee c_i = z_i \leq v_i) \\
 & \qquad \wedge\ (\forall\, i \in \{1\,..\,k\}, j \in \{1\,..\,v_i\} \bullet st_i \xrightarrow{a} tgt^i_j) \\
 & \qquad \wedge\ \forall\, l \in \{1\,..\,k\} \bullet \\
 & \qquad\qquad c'_l = \min\{z_l, \#\{(i,j) \mid i \in \{1\,..\,k\} \wedge j \in \{1\,..\,v_i\} \\
 & \qquad\qquad\qquad\qquad \wedge\ tgt^i_j = st_l\}\}.
\end{array}
\tag{3.10}
$$

Observe that the two parts of the definition of $\longrightarrow_z$ are disjoint, so together they combine into a well-formed definition of the abstract transition relation.

The way (3.10) is formulated requires one to check infinitely many possibilities for v in order to deduce transitions in an abstract model. This is a significant practical problem, which we solve by providing upper bounds for the values of v that need to be considered.

Proposition 3.3.3. *For every i in $\{1\,..\,k\}$, $\max\{\sum_{j\in targets(i)} c'_j, z_i, z_1, 1\}$ is an upper bound for the values of v_i that need to be considered in the body of (3.10), where $targets(i) = \{j \in \{1\,..\,k\} \mid st_i \xrightarrow{e} st_j\}$.*

Proof: Suppose for a contradiction that clauses (i)–(iii) on page 47 hold only for functions v for which there exists I in $\{1\,..\,k\}$ such that

$$v_I > \max\{\textstyle\sum_{j\in targets(I)} c'_j, z_I, z_1, 1\}. \tag{3.11}$$

Let v be a minimal such function and let I be an index such that the above holds, i.e. for every function w like v, except that for some i in $\{1\,..\,k\}$, $w_i = v_i - 1$ at least one of (i)–(iii) fails to hold. Let w be a function like v except that $w_I = v_I - 1$. From (3.11) we have that $v_I > z_1$ and $v_I > 1$. Hence $w_I \geq \max\{z_I, 1\}$. In addition, for all i, $w_i \geq 0$, so w satisfies clause (i). Next, (3.11) implies that $v_I > z_I$. However, no counter value can exceed its threshold, so $z_I \geq c_I$. Therefore $v_I > c_I$. Hence, clause (ii) for v implies that $c_I = z_I$. In addition, $v_I > c_I$ implies that $w_I \geq c_I$, so w satisfies clause (ii). Finally, (3.11) implies that $v_I > \sum_{j\in targets(I)} c'_j$, so there must be a target state $st_{I'}$ such that there are more than $c'_{I'}$ node processes that move from st_I to $st_{I'}$. If we remove a single e-transition from st_I to $st_{I'}$, there are still at least $c'_{I'}$ target states equal to $st_{I'}$. Hence, w satisfies clause (iii). Therefore, we have reached a contradiction, as required. ∎

Example 3.3.4. Recall the $\mathcal{N}$ process syntax from Example 3.0.1. For any given threshold function z we can use the method described above to construct $\zeta_z(\mathcal{N})$ using the LTS of $\mathcal{N}$ (Figure 3.1). Figure 3.3 shows $\zeta_z(\mathcal{N})$ with $z_1 = z_2 = z_3 = z_4 = 1$ and $z_5 = 0$, and where the states of $\mathcal{N}$ have been numbered in the following order: *new*, *runnable*, *running*, *blocked*, *terminated* (e.g. state $(0, \mathbf{1}, 0, \mathbf{1}, \mathbf{0})$ indicates

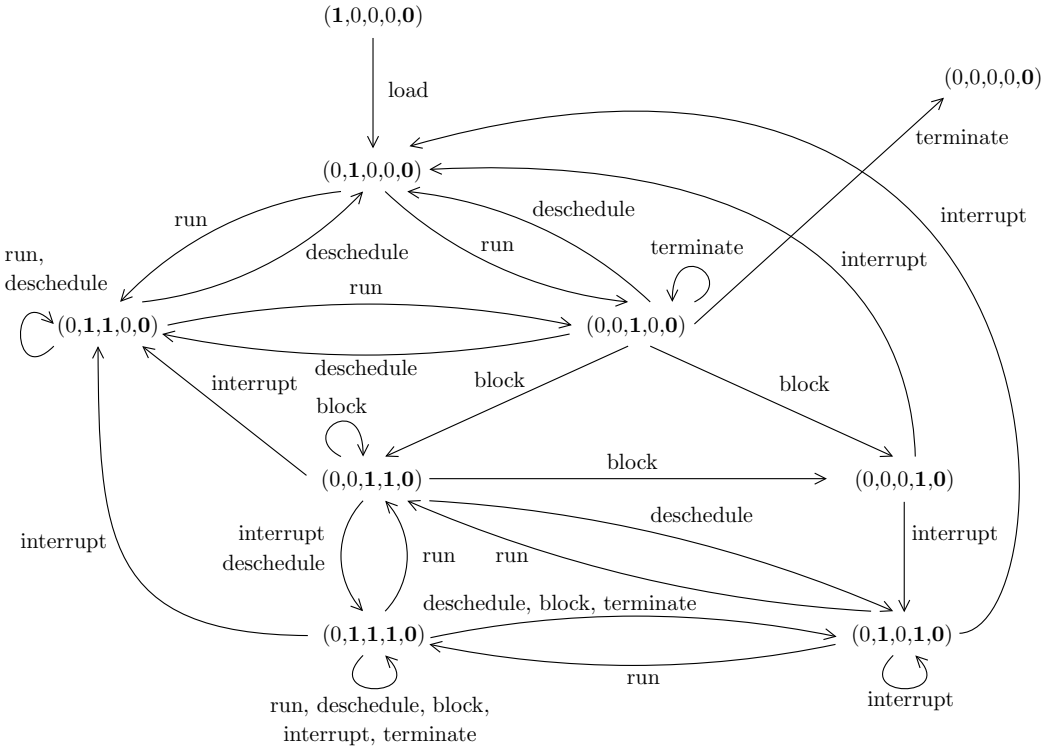


Figure 3.3: The abstract counter state machine $\zeta_z(\mathcal{N})$ where $\mathcal{N}$ is as in Example 3.0.1, $z_1 = z_2 = z_3 = z_4 = 1$ and $z_5 = 0$. Counters that have attained their thresholds are indicated using bold font.

that there are: no nodes in states *new* and *running*, at least one node in state *runnable*, at least one node in state *blocked*, and an unspecified number of nodes in state *terminated*). The figure uses bold font to indicate counters that have attained their thresholds.

End of example.

3.3.2 Refinement results

It is intuitive to expect that counter abstracting a process using thresholds creates an anti-refinement of a counter machine with counters without thresholds of the same process. We prove this formally in this section. This, combined with the results of Section 3.2.2 and thanks to monotonicity of CSP operators, gives us the key result of this chapter, Theorem 3.3.11, which says that if a counter abstraction model put in the same context as used in an implementation satisfies a given specification, then all but some small sizes of the implementation satisfy the specification.

We begin with a definition that, given some type T, relates the states of $\zeta_\infty^T(\mathcal{N})$ and $\zeta_z(\mathcal{N})$. Intuitively, given a threshold function z and two states, s_1 and s_2, in $A_\infty(T)$ and A_z, respectively, we have that s_1 and s_2 are z-corresponding if s_2 can be obtained by capping the counters of s_1 using the thresholds defined by z. The following captures this formally.

Definition 3.3.5. Let T be an instantiation of type t and let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Suppose $s_1 = (c_1, \ldots, c_k)$ and $s_2 = (d_1, \ldots, d_k)$ are two states in $A_\infty(T)$ and A_z, respectively. Then s_1 and s_2 are *z-corresponding*, written $cor_z(s_1, s_2)$, if[6]

$$\forall\, i \in \{1\,..\,k\} \bullet d_i = c_i \sqcap z_i.$$

The following lemma says that, given a threshold function z, two states in $\mathcal{A}_z$ are related by $\xrightarrow{a}_z$ if and only if there exist two z-corresponding states in $\mathcal{A}_\infty(T)$, related by $\xrightarrow{a}_\infty(T)$, for some T of size at least z_1.

Lemma 3.3.6. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let d, d' be two states in A_z. Then, for all events a in Σ_z^τ we have that*

$$\begin{array}{ll} & d \xrightarrow{a}_z d' \\ \Leftrightarrow & \exists\, T \bullet \#T \geq z_1 \\ & \qquad \wedge\, \exists\, c, c' \in A_\infty(T) \bullet cor_z(c, d) \wedge cor_z(c', d') \wedge c \xrightarrow{a}_\infty(T)\, c'. \end{array}$$

Proof: It is easy to see that the result holds for all events in $\Sigma_z^\tau \setminus A_{Sync}$ by comparing definition (3.6) with definition (3.8). So, let a be an event in A_{Sync}. Then, by comparing definition (3.7) with definition (3.10), we get that

$$\begin{array}{ll} & d \xrightarrow{a}_z d' \\ \Leftrightarrow & \exists\, T \bullet \#T \geq \max\{z_1, 1\} \\ & \qquad \wedge\, \exists\, c, c' \in A_\infty(T) \bullet cor_z(c, d) \wedge cor_z(c', d') \wedge c \xrightarrow{a}_\infty(T)\, c'. \end{array} \tag{3.12}$$

However, since we only deal with non-empty instantiations T of type t (see Section 1.7), it must be that $\#T \geq 1$ for all T. Hence, (3.12) implies the result. ∎

Corollary 3.3.7. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let T be an instantiation of type t such that $\#T \geq z_1$. Then, for all events a in $\Sigma_z^\tau = \Sigma_\infty^\tau$,*

$$(c_1, \ldots c_k) \xrightarrow{a}_\infty(T)\, (c'_1, \ldots, c'_k),$$

then

$$(c_1 \sqcap z_1, \ldots, c_k \sqcap z_k) \xrightarrow{a}_z (c'_1 \sqcap z_1, \ldots, c'_k \sqcap z_k).$$

Proposition 3.3.8. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let T be an instantiation of type t such that $\#T \geq z_1$. Then*

$$\zeta_z(\mathcal{N}) \sqsubseteq_{\mathrm{T}} \zeta_\infty^T(\mathcal{N}).$$

Proof: By an easy induction on the number of transitions corresponding to a trace tr, we can prove, using Corollary 3.3.7, that if

$$(\#T, 0, \ldots 0) \xRightarrow{tr} (c'_1, \ldots, c'_k),$$

[6] We write $\sqcap$ to mean the binary minimum operator.

then

$$(\#T \sqcap z_1, 0, \ldots, 0) \stackrel{tr}{\Longrightarrow} (c'_1 \sqcap z_1, \ldots, c'_k \sqcap z_k).$$

Observe that $(\#T, 0, \ldots 0)$ is the initial state of $\zeta^T_\infty(\mathcal{N})$. In addition, we assumed that $\#T \geq z_1$, so $\#T \sqcap z_1 = z_1$. However, $(z_1, 0, \ldots, 0)$ is the initial state of $\zeta_z(\mathcal{N})$. Therefore, tr is in $traces(\zeta_z(\mathcal{N}))$, which implies the result. ∎

Lemma 3.3.9. *Let z be a function from $\{1 \ldots k\}$ to $\mathbb{N}$ and let a be an event in Σ^τ_z. Suppose that*

(i) if $a \notin A_{Sync}$, then for all i in $\{1 \ldots k\}$, if $st_i \xrightarrow{a}$, then $z_i \geq 1$,

(ii) if $a \in A_{Sync}$, then for all i in $\{1 \ldots k\}$, $z_i \geq 1$, and

(iii) for all i in $\{1 \ldots k\}$, $c_i \sqcap z_i = d_i \sqcap z_i$.

Then,

$$(c_1, \ldots, c_k) \xrightarrow{a}_\infty \Leftrightarrow (d_1, \ldots, d_k) \xrightarrow{a}_\infty .$$

Intuitively, clauses (i) and (ii) of the above lemma say that if a is an event on which all nodes synchronise, then all thresholds need to be non-zero, else we allow zero thresholds for counters corresponding to states where a is not available.

Proof: In order to prove the result, we consider the membership of a in A_{Sync}.

Case 1. Suppose that a is in $\Sigma^\tau_z \setminus A_{Sync}$.
Then, by (3.6), a is available in $(c_1, \ldots, c_k)$ if and only if there exists i in $\{1 \ldots k\}$ such that a is available in st_i and $c_i > 0$. Similarly, a is available in $(d_1, \ldots, d_k)$ if and only if there exists i in $\{1 \ldots k\}$ such that a is available in st_i and $d_i > 0$. In addition, for all i such that a is available in st_i we have that $z_i \geq 1$ (by assumption (i)) and $c_i \sqcap z_i = d_i \sqcap z_i$ (by assumption (iii)), so $c_i > 0 \Leftrightarrow d_i > 0$. Hence a is available in $(c_1, \ldots, c_k)$ if and only if it is available in $(d_1, \ldots, d_k)$.

Case 2. Suppose that a is in A_{Sync}.
Then, by (3.7), a is available in $(c_1, \ldots, c_k)$ if and only if for every i in $\{1 \ldots k\}$, if $c_i > 0$, then a is available in st_i. Similarly, a is available in $(d_1, \ldots, d_k)$ if and only if for every i in $\{1 \ldots k\}$, if $d_i > 0$, then a is available in st_i. However, for all i in $\{1 \ldots k\}$, $z_i \geq 1$ (by assumption (ii)) and $c_i \sqcap z_i = d_i \sqcap z_i$ (by assumption (iii)), so $c_i > 0 \Leftrightarrow d_i > 0$. Hence a is available in $(c_1, \ldots, c_k)$ if and only if it is available in $(d_1, \ldots, d_k)$.

Therefore, the result holds for all a in Σ^τ_z. ∎

Proposition 3.3.10. *Let z be a function from $\{1..k\}$ to $\mathbb{N}$. Let T be an instantiation of type t such that $\#T \geq z_1$. Suppose that*

(i) if $A_{Sync} = \{\}$, then for all i in $\{1 \ldots k\}$, either $z_i \geq 1$ or $st_i = STOP$,

(ii) if $A_{Sync} \neq \{\}$*, then for all* i *in* $\{1 \,..\, k\}$*,* $z_i \geq 1$*.*

Then

$$\zeta_z(\mathcal{N}) \sqsubseteq_{\mathrm{F}} \zeta_\infty^T(\mathcal{N}).$$

Proof: By Proposition 3.3.8 we have that $traces(\zeta_\infty^T(\mathcal{N})) \subseteq traces(\zeta_z(\mathcal{N}))$.
Let $(tr, X) \in failures(\zeta_\infty^T(\mathcal{N}))$. Then, there exist $c_1, \ldots c_k$ such that

$$\zeta_\infty^T(\mathcal{N}) \stackrel{tr}{\Longrightarrow} (c_1, \ldots, c_k) \quad \text{and} \quad (c_1, \ldots, c_k) \; ref \; X.$$

Let $a \in X \cup \{\tau\}$. Then the above implies that

$$(c_1, \ldots, c_k) \not\stackrel{a}{\longrightarrow}_\infty . \tag{3.13}$$

By Proposition 3.3.8 we have that

$$\zeta_z(\mathcal{N}) \stackrel{tr}{\Longrightarrow} (c_1 \sqcap z_1, \ldots, c_k \sqcap z_k).$$

Let $c_1', \ldots, c_k'$ be arbitrary integers such that $c_i \sqcap z_i = c_i' \sqcap z_i$ for all i in $\{1 \,..\, k\}$. If $a \in A_{Sync}$, then $A_{Sync} \neq \{\}$, so by assumption (ii), $z_i \geq 1$ for all $i \in \{1 \,..\, k\}$ and if $a \notin A_{Sync}$, then assumptions (i) and (ii) imply that for all $i \in \{1 \,..\, k\}$, either $z_i \geq 1$ or $st_i = STOP$. Hence, from (3.13) we can infer using Lemma 3.3.9 that

$$(c_1', \ldots, c_k') \not\stackrel{a}{\longrightarrow}_\infty .$$

Therefore, Lemma 3.3.6 implies that

$$(c_1 \sqcap z_1, \ldots, c_k \sqcap z_k) \not\stackrel{a}{\longrightarrow}_z .$$

Since a is an arbitrary event in $X \cup \{\tau\}$, we have that $(c_1 \sqcap z_1, \ldots, c_k \sqcap z_k)$ stably refuses X. Thus, $(tr, X) \in failures(\zeta_z(\mathcal{N}))$. ∎

The following theorem joins together the results from this section and from Section 3.2.2, and extends them to parallel compositions put in T-independent CSP contexts $C[\cdot]$. Since *Spec* and $C[\zeta_z(\mathcal{N})]$ are all independent of T, the theorem allows us to perform a single refinement in order to deduce the truth of the refinement $Spec \sqsubseteq Impl(T)$ for all but some small sizes of the implementation.

Theorem 3.3.11. *Let* T *be an instantiation of type* t *such that* $\#T \geq z_1$ *and let* $C[\cdot]$ *be independent of* T*. Then*

(i) if z *is a function from* $\{1 \,..\, k\}$ *to* $\mathbb{N}$ *and*

$$Spec \sqsubseteq_{\mathrm{T}} C[\zeta_z(\mathcal{N})],$$

then

$$Spec \sqsubseteq_{\mathrm{T}} C[Nodes(T)],$$

(ii) if z is a function from $\{1\,..\,k\}$ to $\mathbb{N}$ such that

- *if $A_{Sync} = \{\}$, then for all i in $\{1\,..\,k\}$, either $z_i \geq 1$ or $st_i = STOP$,*
- *if $A_{Sync} \neq \{\}$, then for all i in $\{1\,..\,k\}$, $z_i \geq 1$,*

and

$$Spec \sqsubseteq_{\mathrm{F}} C[\zeta_z(\mathcal{N})],$$

then

$$Spec \sqsubseteq_{\mathrm{F}} C[Nodes(T)].$$

Proof: For all T of size at least z_1, we have that if z is a function from $\{1\,..\,k\}$ to $\mathbb{N}$, then Proposition 3.3.8, Corollary 3.2.3 and transitivity of refinement imply that

$$\zeta_z(\mathcal{N}) \sqsubseteq_{\mathrm{T}} Nodes(T).$$

Hence, by monotonicity of the CSP operators, we have that

$$C[\zeta_z(\mathcal{N})] \sqsubseteq_{\mathrm{T}} C[Nodes(T)].$$

Therefore, if $Spec \sqsubseteq_{\mathrm{T}} C[\zeta_z(\mathcal{N})]$, then, by transitivity of refinement,

$$Spec \sqsubseteq_{\mathrm{T}} C[Nodes(T)],$$

which proves clause (i). Proving clause (ii) is similar, using Proposition 3.3.10 in place of Proposition 3.3.8. ∎

Example 3.3.12. We now return to the verification problem from Example 3.0.1. Suppose that we deal with a dual-core machine (i.e. $cores = 2$). Let z be a given threshold function such that $z_1 = z_2 = z_4 = z_5 = 1$ and $z_3 = 2$. We use the method described in Section 3.3.1 to construct $\zeta_z(\mathcal{N})$, which we represent using a CSP process, called $CANodes_z$. Below are the first three branches of the definition; the other branches are similar and can be found in Appendix A.

$$CANodes_z = CANodes'_z(1,0,0,0,0)$$

$$\begin{aligned}
&CANodes'_z(c_1,c_2,c_3,c_4,c_5) = \\
&\quad c_1 \geq 1 \;\&\; load \rightarrow CANodes'_z(0,1,c_3,c_4,c_5) \\
&\quad \Box \\
&\quad c_2 \geq 1 \;\&\; run \rightarrow (CANodes'_z(c_1,0,(c_3+1)\sqcap 2,c_4,c_5) \\
&\qquad\qquad \sqcap\; CANodes'_z(c_1,1,(c_3+1)\sqcap 2,c_4,c_5)) \\
&\quad \Box \\
&\quad c_3 \geq 1 \;\&\; deschedule \rightarrow \\
&\qquad (\text{if } c_3 = 2 \text{ then } CANodes'_z(c_1,(c_2+1)\sqcap 1,1,c_4,c_5) \\
&\qquad\qquad \sqcap\; CANodes'_z(c_1,(c_2+1)\sqcap 1,2,c_4,c_5) \\
&\qquad\quad \text{else } CANodes'_z(c_1,(c_2+1)\sqcap 1,0,c_4,c_5)) \\
&\quad \Box \\
&\quad \ldots
\end{aligned}$$

$\#T$	2	3	4	5	6	7	8	9	10
states	23	91	253	1324	4861	17497	61966	216514	747955

Table 3.1: Number of states of $Impl(T)$ for various sizes of T.

The nondeterminism present above comes from the fact that whenever counter c_i is equal to its threshold, z_i, it means that there are either exactly z_i or strictly more than z_i node processes in state st_i.

Representing counter state machines using CSP processes like the one above aids readability, but is inefficient to compile in FDR, as it uses a single sequential process with $\prod_{i=1}^{k}(z_i+1)$ states (minus some unreachable ones). In Chapter 7 we will present a tool that automatically produces more efficient implementations based on parallel compositions of k separate processes, one for each counter, that together give a total of $\sum_{i=1}^{k}(z_i+1)$ states (minus some unreachable ones) to compile.

Let

$$C[X] = (X \underset{Alpha}{\parallel} Scheduler)[\![^{stopRun,stopRun,stopRun}/_{deschedule,block,terminate}]\!],$$

where

$$Alpha = \{| \ run, deschedule, block, terminate \ |\}.$$

We can now use FDR to verify that

$$Spec \sqsubseteq_{\mathrm{F}} C[CANodes_z],$$

with the check taking less than 1 second.

We have that the synchronisation of node processes is over a non-empty set and for all i in $\{1..5\}$, $z_i \geq 1$, so Theorem 3.3.11 implies that, given T such that $\#T \geq z_1 = 1$ (i.e. given any T),

$$Spec \sqsubseteq_{\mathrm{F}} C[Nodes(T)] = Impl(T),$$

which solves our verification problem (3.1).

It is interesting to note that the number of states for the concrete state machine grows exponentially[7] in $\#T$ (Table 3.1). On the other hand, the number of states of $C[CANodes_z]$ is fixed at 63 and is (obviously) independent of T. More generally, the number of abstract states is $O(\prod_{i=1}^{k}(z_i+1))$, which is exponential in the size of a single node; however, in most practical situations many of these states are unreachable, so the typical case is much better.

End of example.

[7] If T is such that $\#T \geq cores$, then the exact number of states that FDR checks is given by the formula $1 + \sum_{i=0}^{cores} \binom{cores}{i}\binom{\#T}{i}3^{\#T-i}$.

3.3.3 Threshold function cull

The values of the threshold function used in Example 3.3.12 were chosen so that $Spec \sqsubseteq_{\mathrm{F}} C[CANodes_z]$ holds. However, if we let, for example, $z_1 = z_2 = z_3 = z_4 = z_5 = 1$, then

$$(\langle load, run, run, stopRun\rangle, \Sigma) \in failures(C[CANodes_z]),$$

but

$$(\langle load, run, run, stopRun\rangle, \Sigma) \notin failures(Spec).$$

Then, there is no T such that $(\langle load, run, run, stopRun\rangle, \Sigma)$ is a stable failure of $Impl(T)$, so we have found a spurious counterexample. Such spurious counterexamples are results of *discontinuities* in the abstract model, where a discontinuity (in the i-th counter) is defined to be a behaviour where:

- $\zeta_z(\mathcal{N})$ performs a sequence of events that implies that n processes move into the i-th state,
- $z_i = n$, and
- $\zeta_z(\mathcal{N})$ subsequently performs a sequence of events that implies the presence of more than n processes in state st_i.

Note that every discontinuity can be eliminated by increasing appropriate values of z. This leads us to an interesting open question: is there a family of verification problems for which there always exists a finite function z such that all discontinuities that lead to spurious counterexamples are eliminated? Since uniform verification of the family of systems considered in this chapter is undecidable (see Theorem 3.1.7), the existence of a finite z such that $Spec \sqsubseteq C[\zeta_z(\mathcal{N})]$ cannot, in general, be guaranteed. However, we conjecture that for all problems where the nodes run in an interleaving and the specification and the context are finite-state (for which the problem of uniform verification is decidable; see Theorem 3.1.1), there exists a finite threshold function z such that the abstract model satisfies the specification (also, see Section 8.3).

Conjecture 3.3.13. *If Spec and $C[\cdot]$ are finite-state and $A_{Sync} = \{\}$, then there exists a finite threshold function z such that $Spec \sqsubseteq C[\zeta_z(\mathcal{N})]$.*

However, such z may fail to exist if we allow specifications or contexts to be infinite-state, as the following example illustrates.

Example 3.3.14. Suppose a node process performs an *in* event, followed by an *out* event and then deadlocks and suppose an implementation is the interleaving of $\#T$ such nodes for every instantiation T of type t, i.e.

$$\mathcal{N} = in \rightarrow out \rightarrow STOP$$
$$Impl(t) = |||\, i \in t \bullet \mathcal{N}.$$

Suppose that we want to verify that the implementation never performs more *out*'s than *in*'s. Hence, we let

$$
\begin{aligned}
&\mathit{Spec} = \mathit{Spec}'(x)\\
&\mathit{Spec}'(x) = \mathit{in} \rightarrow \mathit{Spec}'(x+1)\\
&\qquad\qquad \Box\\
&\qquad\qquad x > 0 \;\&\; \mathit{out} \rightarrow \mathit{Spec}'(x-1).
\end{aligned}
$$

Observe that *Spec* is infinite-state.

The counter state machine $\zeta_z(\mathcal{N})$ can be represented using a CSP process defined as follows.

$$
\begin{aligned}
&\mathit{CANodes}_z = (z_1, 0, 0)\\
&\mathit{CANodes}'_z(c_1, c_2, c_3) =\\
&\quad c_1 \geq z_1 \vee c_1 \geq 1 \;\&\; \mathit{in} \rightarrow\\
&\qquad (\text{if } c_1 = z_1 \text{ then } \mathit{CANodes}'_z((c_1 - 1) \sqcup 0, (c_2 + 1) \sqcap z_2, c_3)\\
&\qquad\qquad\qquad \sqcap\ \mathit{CANodes}'_z(c_1, (c_2 + 1) \sqcap z_2, c_3)\\
&\qquad\qquad \text{else } \mathit{CANodes}'_z(c_1 - 1, (c_2 + 1) \sqcap z_2, c_3))\\
&\quad \Box\\
&\quad c_2 \geq z_2 \vee c_2 \geq 1 \;\&\; \mathit{out} \rightarrow\\
&\qquad (\text{if } c_2 = z_2 \text{ then } \mathit{CANodes}'_z(c_1, c_2 - 1) \sqcup z_2, (c_3 + 1) \sqcap z_3)\\
&\qquad\qquad\qquad \sqcap\ \mathit{CANodes}'_z(c_1, c_2, (c_3 + 1) \sqcap z_3)\\
&\qquad\qquad \text{else } \mathit{CANodes}'_z(c_1, c_2 - 1, (c_3 + 1) \sqcap z_3))
\end{aligned}
$$

Clearly, $\mathit{Spec} \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T)$ for all T, but there is no threshold function z such that $\mathit{Spec} \not\sqsubseteq_{\mathrm{T}} \mathit{CANodes}_z$ as $\mathit{CANodes}_z$ can always communicate some finite number, say n, of *in*'s and reach a state where the c_2 counter has reached its threshold, so now an arbitrary (and therefore possibly greater than n) number of *out*'s can be communicated.

End of example.

For a given verification problem, if there exists a minimal finite threshold function z^{min} such that $\mathit{Spec} \sqsubseteq \zeta_{z^{min}}(\mathcal{N})$, then it is desirable to find a threshold function z which is as small as possible and covers z^{min}, i.e. is such that $z_i \geq z_i^{min}$ for all i. Below we present a framework of a CEGAR-style (see Section 2.5) algorithm to do just that. Lines 1–7 initialise z to have the smallest values allowed for a given denotational model ($\mathcal{M}$ is either the traces or the stable failures model). Then, we start a loop attempting the verification of the counter abstraction with z against the specification. If refinement holds, then the verification problem is solved with a positive answer (line 16). Otherwise, we check if the counterexample is a valid counterexample of the implementation of some size (line 10). We now argue that this can be done in a finite amount of time both for the trace and stable failures models, since only finitely many sizes of the implementation need to be checked.

Firstly, let *ce* be a trace *tr* and let[8] $n = \#(tr \upharpoonright (\Sigma \setminus A_{Sync}))$, where Σ is the alphabet of a node. We prove that if *tr* is a trace of $Impl(T)$ for some T, then it is a trace of $Impl(T)$ for T such that $\#T \leq n$. Suppose that there is T such that $Impl(T)$ can perform *tr*. In *tr*, at most n nodes do events in $\Sigma \setminus A_{Sync}$ (others do only events in A_{Sync}). Without loss of generality we assume that these nodes have identities in $\{1 \,..\, m\}$, where $1 \leq m \leq n$. Then $Impl(\{1 \,..\, m\})$ can perform *tr*.

Now, let *ce* be a failure (tr, X) and let $n = \#(tr \upharpoonright (\Sigma \setminus A_{Sync})) + \#X$, where Σ is the alphabet of a node. Observe that, thanks to the assumption about finiteness of alphabets, $\#X$ is finite. We prove that if (tr, X) is a failure of $Impl(T)$ for some T, then it is a failure of $Impl(T)$ for T such that $\#T \leq n$. Suppose that there is T such that $Impl(T)$ can perform *tr* to reach a stable state where it refuses X. In *tr*, at most $\#(tr \upharpoonright (\Sigma \setminus A_{Sync}))$ nodes do events in $\Sigma \setminus A_{Sync}$ (others do only events in A_{Sync}) and at most $\#X$ nodes are responsible for the refused events in X. Without loss of generality we assume that these nodes have identities in $\{1 \,..\, m\}$, where $1 \leq m \leq n$. Then (tr, X) is a failure of $Impl(\{1 \,..\, m\})$.

If *ce* turns out to be an admissible counterexample, then the verification problem is solved with a negative answer (line 11). Else, we use *ce* and some heuristic Θ to increase some or all thresholds within z (line 13) and loop back. An example of a simple heuristic is one that increases all thresholds by 1; we call this heuristic Θ_1. A slightly better, but more expensive heuristic is one that increases by 1 all the thresholds that were attained by counters within the abstract model when behaviour *ce* was formed; we call this heuristic Θ_2. An even better (but even more expensive) heuristic, call it Θ_3, is like Θ_2, but it only increases by 1 the thresholds of counters in which a discontinuity occurred when behaviour *ce* was formed.

Algorithm 1 Threshold function cull

```
 1: for i ← 1, k do
 2:     if st_i = STOP or M = T then
 3:         z_i ← 0
 4:     else
 5:         z_i ← 1
 6:     end if
 7: end for
 8: while Spec ⋤_M ζ_z(N) do
 9:     ce ← counterexample
10:     if ∃ T • admissible(ce, Impl(T)) then
11:         return no
12:     else
13:         z ← Θ(ce, z)
14:     end if
15: end while
16: return yes
```

[8] Given $tr \in \Sigma$ and provided that $A \subseteq \Sigma$, $tr \upharpoonright A$ (pronounced "*tr restricted* to A") denotes the trace like *tr*, but with all events not in A removed.

In the cases where a finite threshold function z such that $Spec \sqsubseteq C\,[\zeta_z(\mathcal{N})]$ exists, termination of Algorithm 1 depends on the choice of heuristic Θ. For example, if Θ always increases only the first threshold in a setting where $z_2 \geq 2$ is needed to find a true counterexample, then the algorithm does not terminate. However, termination is always achieved for heuristics Θ_1, Θ_2 and Θ_3, presented above. To see this, let Z be the optimal threshold function. Let z^j be the threshold function used in the j-th iteration of the while loop of the algorithm (in particular, z^1 is the initial threshold function generated by the for loop of the algorithm). In addition, for every j, let $f(z^j) = \sum_{i=1}^{k} \max(Z_i - z_i^j, 0)$. We say that a heuristic Θ is *decreasing* if f is decreasing when Θ is used. The algorithm terminates when we find a finite j such that for all i in $\{1\,..\,k\}$, $z_i^j \geq Z_i$, i.e. when $f(z^j) = 0$. For this to happen, it is enough if the used heuristic Θ is decreasing, since $f(z^1)$ is always a finite, non-negative integer. If $\Theta = \Theta_1$, then for all j, $z_i^{j+1} = z_i^j + 1$ for all i in $\{1\,..\,k\}$, which means that $f(z^{j+1}) < f(z^j)$. Similarly, if $\Theta = \Theta_2$ or $\Theta = \Theta_3$, then at every iteration j there must be at least one i in $\{1\,..\,k\}$ such that $z_i^j < Z_i$ and the i-th counter attains z_i^j, which means that $z_i^{j+1} = z_i^j + 1$, i.e. $f(z^{j+1}) < f(z^j)$ (and discontinuity occurs in the i-th counter when Θ_3 is used). Therefore, all three heuristics are decreasing.

3.4 Conclusions

In this chapter we described how standard counter abstraction techniques can be transferred into the CSP/FDR setting.

We described the family of allowed implementations, each of which consists of a (generalised) parallel composition of unboundedly many identical nodes processes, running in a general CSP context that is independent of the distinguished type. We also presented a running example for the chapter — a model of a multicore machine, where processes require access to CPUs.

We showed that, in general, uniform verification of the family of systems we considered in this chapter is undecidable and we provided a restriction for which the problem becomes decidable.

We presented how an abstract counter state machine based on unbounded integer counters, which models parallel compositions of all nodes, can be constructed using the state machine of a single node process. We showed that such a state machine is strongly bisimilar to the parallel composition of all nodes. We noted that even though such an abstraction leads to a significant reduction in the number of states (thanks to factoring the state space with respect to bisimulation), the abstract models still depend on the instantiation T of type t.

We improved the abstraction techniques by introducing threshold functions z. We presented how abstract counter state machines based on bounded integer counters can be constructed. Also, we proved that such constructions can be achieved with only a finite number of operations. We proved that, for T large enough, the abstract models formed traces and stable failures anti-refinements of parallel compositions of node processes, with the stable failures model requiring the values of certain thresholds to be non-zero. We then used monotonicity of CSP operators to extend the results to node processes running in general, t-independent CSP contexts. The results allow us

to perform a single refinement check (using FDR) in order to conclude that an implementation refines a specification for all but some small types T, with the remaining small types being verifiable directly. We demonstrated our techniques by creating a counter abstraction model for the running example and performing verification using FDR (more case studies using the methods presented in this chapter can be found in [Tho10]).

Finally, we discussed how the choice of the threshold may affect the presence of spurious counterexamples. We also presented an outline of an iterative algorithm for finding small threshold functions, including three heuristics for deciding how to increase thresholds at every iteration.

Chapter 4

Operational Semantics

The main usefulness of a process algebra (like CSP) comes from the fact that it allows us to reason about programs and processes rigorously. Such a formalism is, obviously, not possible without a well defined syntax structure for expressing implementations and specifications (see Section 1.5.1 for CSP syntax). However, even more important is the ability to rigorously capture the meaning of a process definition, i.e. its *semantics*. There are three main ways to describe the behaviour of CSP processes using mathematical terms: *operational semantics*, *denotational semantics* and *algebraic semantics*. In this chapter we look into the first of these, but a detailed description of the other two can be found in [Ros97, Chapters 8 and 11].

The aim of an operational semantics is to provide a precise step-by-step description of how processes execute. It describes state changes when actions are performed. It does so by representing processes using LTSs (*labelled transition systems*; see Definition 3.0.2).

In order to obtain an LTS representing a given process or process syntax in accordance to some operational semantics, a set of transition rules is needed. In 1981, Plotkin [Plo81] introduced *Structured Operational Semantics*, an intuitive and functional way of defining the transition rules through a set of logical inference rules (also called firing rules) that allow us to deduce which transitions are available in each state of a given process. The term "structured" reflects the fact that transition rules are defined in a compositional manner; the overall behaviour is inductively derived from the behaviour of smaller parts. In the case of the CSP process algebra, this means that whenever we define a specification of an operational semantics, we only need to provide inference rules for each of the operators and syntax terms.

The various operational semantics we present in this thesis do not aim to be complete. Their main purpose is to formalise the foundations for the results regarding specifications, presented in Chapter 5. This is why we describe only the minimal operational semantics that allow us to generate transition graphs of the processes that we consider in that chapter. The fragment of CSP that we use when working with specifications consists of: $STOP$, prefixing, external, internal, sliding and conditional choice, and general recursion (through binding). Most CSP specifications one uses in practice lie within this fragment of CSP and others can be rewritten into this form using algebraic laws. We stress, though, that implementation processes can be written

using the full syntax of CSP (in Section 4.2 we present standard operational semantics transition rules for replicated external and internal choice, parallel composition, renaming and hiding).

Operational semantics can be defined at different levels of abstraction. In Section 4.2 we present an operational semantics at the lowest, implementation level. It generates LTSs from process syntax with no free variables. This means that concrete values must be substituted for all parameters before the transition rules can be used. One of the main shortcomings of such an operational semantics, when working with parameterised systems, is the need for repetitive application of the transition rules for each instantiation of the parameters. Lazić addressed this problem in [Laz99] by defining a symbolic operational semantics (for a language similar to CSP, but with an addition of certain lambda calculus terms), where the variables related to the parameters are never instantiated, but rather left as symbols, when an LTS is generated. The advantage of such an approach is that, given a parameterised process syntax, a single symbolic LTS is generated and each of the concrete LTSs can be easily obtained from it by an assignment of values[1]. Such a symbolic LTS can be viewed as a formal structure that captures the essence of the behaviour of a process; it hides the details of the data values, concentrating on the control states between which a process can move by executing actions. This sort of symbolic structure is precisely what we need for our work in Chapter 5. However, the assumptions we make about the processes with which we work cause the application of Lazić's work to be unnecessarily complex for our needs. This is why we define *Semi-Symbolic Operational Semantics* (SSOS), a symbolic operational semantics similar to the one from [Laz99], in Section 4.3. The states of the resulting *semi-symbolic LTSs* (SSLTSs) can be viewed as the control states of families of concrete processes. In order to fully concretise them, it is enough to provide a map of variable names to concrete values (such a map will be called an *environment*). In Section 4.4 we describe *Concrete Operational Semantics with Environments* (COSE), a concrete operational semantics which, for a fixed instantiation of the distinguished type, creates LTSs whose states are triples consisting of a symbolic state (or a modification of such), an environment, and the instantiation of the distinguished type. The specification of COSE is provided as a set of translation rules from SSOS, rather than a set of transition rules. We also show that the combination of SSOS and the translation rules of COSE is, in fact, congruent to the standard one (Section 4.5). Finally, we define the relationship between symbolic traces and concrete traces in Section 4.6 and conclude the chapter in Section 4.7.

4.1 Preliminaries

In the introduction to this chapter we mentioned that we restrict our operational semantics to a fragment of the CSP language when working with specifications. We aim to develop mathematical machinery to prove (in Chapter 5) useful results about specifications that satisfy a certain normality condition, which we define in Section 4.1.2.

[1] In Lazić's work individual concrete LTSs are, in fact, never generated. Instead, the relationships with denotational semantics are established and the denotational values are derived directly from symbolic LTSs.

Earlier, in Section 4.1.1 we define data independence, a crucial part of normality. We will strongly rely on our normality condition when defining our Semi-Symbolic Operational Semantics (Section 4.3) and when deriving type reduction theory results in Chapter 5.

4.1.1 Data independence

Intuitively, we say that a process syntax treats type t data independently if it inputs and outputs values of type t, possibly storing them for later use, but does not perform any operations on these values that could influence either its control flow or the instantiations of type t that can be used. The following definition of a data independent process is based on the one from [Ros97].

Definition 4.1.1. We say that a CSP process syntax is *data independent* with respect to type t if it does not contain:

(i) replicated constructs indexed over any set depending on t, except for replicated nondeterministic choice ($\sqcap$) indexed over the whole of t; however, we allow the use of deterministic and nondeterministic input selections, ? and \$,

(ii) conditional choices on t, except for equality and inequality tests,

(iii) constants of type t,

(iv) functions whose domains or co-domains involve type t,

(v) operations on t, including polymorphic operations, (e.g. tupling or lists),

(vi) selections from sets involving t, unless the selection is over the whole of t, and

(vii) any operations that would extract information about t, e.g. $card(t)$.

Our definition of data independence is slightly stronger than the one from [Ros97, Chapter 15], since we do not allow polymorphic operations on t (e.g. tupling, lists). However, the effect of all finite size structure building functions that do not extract any information about t (e.g. no lists whose length depends on t) can be recovered by using multiple inputs or outputs in the same prefix. For example, we can rewrite the process syntax

$$in?x{:}(t \times t \times t) \rightarrow out.snd(x) \rightarrow STOP,$$

where $snd(x_1, x_2, x_3) = x_2$, as

$$in?x{:}t?y{:}t?z{:}t \rightarrow out.y \rightarrow STOP.$$

The following example shows a process that satisfies data independence.

Example 4.1.2. Let

$$PAIR(t) = input?a{:}\mathrm{t} \rightarrow \mathrm{input?b{:}t} \rightarrow \mathrm{if\ a} \neq \mathrm{b\ then}\ outputPair.a.b \rightarrow PAIR(t) \\ \mathrm{else}\ outputSingle.a \rightarrow PAIR(t).$$

Then $PAIR(T)$ inputs two values in T and, provided they are different, outputs them as a pair (but using two separate outputs for the reasons discussed above). If the inputs a and b are equal, it outputs a as a single element.

End of example.

Remark 4.1.3. Clauses (v) and (vi) of Definition 4.1.1 together imply that, for all constructs $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ of a given data independent process syntax, each X_i is either a type not related to t or precisely the type parameter t, unless $\S_i = !$, in which case $X_i = null$.

4.1.2 The SeqNorm condition

When working with specification processes, it is desirable to ensure their clarity and conformance to a certain standard (normality) to make analyses of their behaviours easier. The **SeqNorm** condition, defined below, achieves this without a major expressiveness reduction.

Given a sequential, data independent process syntax P, we define $Channels(P)$ to be the set of channel names of the initial constructs of P. Formally,

$$
\begin{array}{l}
Channels(STOP) \mathrel{\widehat{=}} \{\}, \\
Channels(c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \rightarrow P) \mathrel{\widehat{=}} \{c\}, \\
Channels(P \,\Box\, Q) \mathrel{\widehat{=}} Channels(P) \cup Channels(Q), \\
Channels(P \sqcap Q) \mathrel{\widehat{=}} Channels(P) \cup Channels(Q), \\
Channels(P \triangleright Q) \mathrel{\widehat{=}} Channels(P) \cup Channels(Q), \\
Channels(X) \mathrel{\widehat{=}} Channels(P), \quad \text{if } E(X) = P.
\end{array}
$$

Definition 4.1.4. A process syntax $Proc(t)$ satisfies **SeqNorm** if

(i) it is data independent,

(ii) it is sequential and contains no renaming or hiding,

(iii) it contains no replicated external or nondeterministic choice (but we do allow nondeterministic selections through the use of the \$ symbol),

(iv) for all external choices $P(t) \,\Box\, Q(t)$, internal choices $P(t) \sqcap Q(t)$ and sliding choices $P(t) \triangleright Q(t)$ within $Proc(t)$ we have that

- $Channels(P(t)) \cap Channels(Q(t)) = \{\}$
- every conditional choice on t in $P(t)$ and $Q(t)$ is after a prefix,

Our definition of **SeqNorm** is similar to definitions of **Norm** used in the CSP literature [Ros97, Laz99], with the main difference being the inclusion of data independence (clause (i)) and sequentiality (clause (ii)) in **SeqNorm**.

We have decided to include data independence as a part of the definition of **SeqNorm**, since we always use **SeqNorm** with data independence.

We have decided to include sequentiality as part of the definition of **SeqNorm**, since we mainly assume **SeqNorm** only for *specification* processes, which are nearly

always sequential; when a specification is not sequential, it can be rewritten into a sequential form using algebraic equivalences [Ros97]. Further, this assumption greatly simplifies our proofs.

If we ignore data independence and sequentiality, then compared to the definition of **Norm** in [Laz99], we place fewer restrictions on branches of external, internal and sliding choices (for example, we allow branches of external choices to be nondeterministic choices), but we ban renaming. However, **SeqNorm** is stronger than **Norm** defined in [Ros97], as we ban replicated nondeterministic choice over any type, rather than just the distinguished type, and we require that not only the initial events of the arguments of the $\Box$, $\sqcap$ and $\triangleright$ operators are disjoint, but so are their channels. Example 4.1.5, below, shows an example of a process that satisfies Roscoe's definition, but not ours.

Example 4.1.5. Let

$$Proc(t) = in?x{:}t?y{:}t \rightarrow \text{ if } x = y \text{ then } STOP \text{ else } out.x \rightarrow STOP \,\Box\, out.y \rightarrow STOP.$$

Then, once $Proc(t)$ reaches the negative branch of the conditional choice, we are guaranteed that x and y are not equal, so the initial events of the external choice are different (but the channels are obviously not).

End of example.

In practice, however, most useful processes have the property that if the initial events of the branches of $\Box$, $\sqcap$ and $\triangleright$ are disjoint, then so are the channels of those events.

Throughout this thesis we assume that each process satisfying **SeqNorm** is finite-state, which means that such processes cannot use replicated choices indexed over an infinite type. However, every replicated choice indexed over a finite type can be expressed as a finite number of binary choices, so our ban of all replicated external choices (introduced only for technical reasons) does not reduce expressibility for the processes that we assume to satisfy **SeqNorm** in this thesis.

Remark 4.1.6. **SeqNorm** aims to remove all nondeterminism whose effect is not immediately observable. One way to ban such nondeterminism is to allow the use of replicated nondeterministic choice ($\sqcap$) only where the values chosen are used in the immediately performed events. While defining such a restriction is certainly possible, clause (iii) of the definition simplifies things by removing the $\sqcap$ operator altogether, while allowing nondeterministic inputs. This is sufficient for most practical situations where replicated nondeterministic choice with immediately observable effects would normally be used. The only processes (with all nondeterminism immediately observable) that are no longer expressible are those where the values being chosen via the $\sqcap$ operator are used in more than one event, e.g. $\sqcap x \in X \bullet (c_1.x \rightarrow STOP \,\Box\, c_2.x \rightarrow STOP)$.

Remark 4.1.7. If a particular process syntax fails **SeqNorm** because of the second subclause of clause (iv), then the following algebraic laws (distribution of $\Box$, $\sqcap$ and $\triangleright$

over conditional choice) can be used to convert it to an equivalent process definition, satisfying this assumption:

$$P \bowtie (\text{if } b \text{ then } Q \text{ else } R) \equiv \text{if } b \text{ then } (P \bowtie Q) \text{ else } (P \bowtie R),$$
$$(\text{if } b \text{ then } P \text{ else } Q) \bowtie R \equiv \text{if } b \text{ then } (P \bowtie R) \text{ else } (Q \bowtie R),$$

where $\bowtie$ is one of $\Box, \sqcap$ or $\triangleright$.

SeqNorm may be seen as a rather strong condition and, in fact, it would be inappropriate to assume that it is satisfied by most implementation processes. However, in this chapter (and Chapter 5) we will only ever require that *specification* processes satisfy this condition. In practice, almost all sequential processes that are regarded as useful and well-defined specifications meet the requirements of **SeqNorm**.

4.1.3 General definitions and notation

Before we start describing the operational semantics for CSP, it is beneficial to present some notation and definitions that will often be used in the following sections. Additional pieces of notation and local definitions will be introduced in the relevant parts of this chapter.

Prefix constructs may depend on the distinguished type, so, in theory, they should be decorated with parameter t, e.g. $Proc(t) = \alpha(t) \rightarrow Proc'(t)$. However, for brevity, we omit the parameter in all cases where it is indifferent or clear from the context whether we deal with a prefix syntax or its instantiated form with the instantiation of type t also being clearly implied.

For any construct α of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$, we define a number of useful functions that return index sets of variables and values within α, based on their type and the kind of input or output they model:

$$\$^t(\alpha) \mathrel{\widehat{=}} \{i \in \{1 \,..\, k\} \mid \S_i = \$ \wedge X_i = t\},$$
$$\$^{non\text{-}t}(\alpha) \mathrel{\widehat{=}} \{i \in \{1 \,..\, k\} \mid \S_i = \$ \wedge X_i \neq t\},$$
$$\$(\alpha) \mathrel{\widehat{=}} \$^t(\alpha) \cup \$^{non\text{-}t}(\alpha),$$

$$?^t(\alpha) \mathrel{\widehat{=}} \{i \in \{1 \,..\, k\} \mid \S_i = ? \wedge X_i = t\},$$
$$?^{non\text{-}t}(\alpha) \mathrel{\widehat{=}} \{i \in \{1 \,..\, k\} \mid \S_i = ? \wedge X_i \neq t\},$$
$$?(\alpha) \mathrel{\widehat{=}} ?^t(\alpha) \cup ?^{non\text{-}t}(\alpha),$$

$$!^t(\alpha) \mathrel{\widehat{=}} \{i \in \{1 \,..\, k\} \mid \S_i = ! \wedge x_i \text{ is of type } t\},$$
$$!^{non\text{-}t}(\epsilon) \mathrel{\widehat{=}} \{i \in \{1 \,..\, k\} \mid \S_i = ! \wedge x_i \text{ is not of type } t\},$$
$$!(\alpha) \mathrel{\widehat{=}} !^t(\alpha) \cup !^{non\text{-}t}(\alpha).$$

Note that if α is a construct within a process definition that satisfies clauses (iii)–(vi) of data independence (Definition 4.1.1), then for all $\dagger$ in $\{\$, ?, !\}$,

$$\dagger(\alpha) = \{i \in \{1 \,..\, k\} \mid \S_i = \dagger\}.$$

Sometimes we will want to modify constructs. The following functions make it easier. Let $\alpha = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and let $\dagger$ be one of either t or *non-t*. Then, we define:

- $Replace^{\dagger}_{\$\mapsto!}(\alpha)$ to be a construct like α, but where for every i in $\$^{\dagger}(\alpha)$ the $\S_i$ symbol (which must be a \$) is replaced by a ! and X_i is replaced by *null*, and
- $Replace_{\$\mapsto!} \mathrel{\widehat{=}} Replace^{t}_{\$\mapsto!} \circ Replace^{non\text{-}t}_{\$\mapsto!}$.

Example 4.1.8. Let

$$\epsilon = c\$x_1{:}t?x_2{:}t\$x_3{:}X!x_4,$$

where X it a type not related to t and x_4 is some output variable. Then,

$$Replace^{t}_{\$\mapsto!}(\epsilon) = c!x_1?x_2{:}t\$x_3{:}X!x_4,$$
$$Replace^{non\text{-}t}_{\$\mapsto!}(\epsilon) = c\$x_1{:}t?x_2{:}t!x_3!x_4,$$

and

$$Replace_{\$\mapsto!}(\epsilon) = c!x_1?x_2{:}t!x_3!x_4.$$

End of example.

Substitution will play an important role in defining the operational semantics in the following sections of this chapter. We use square brackets to denote substitution: for a variable x and a value v, $P[v/x]$ is like P, but with every free occurrence of x replaced with v (here, P can be a process, a definition of a set, a definition of a relation, etc). Substitution is different from renaming, since renaming is a function or relation from values to values, while substitution is a function from variables to values.

Finally, we define

- *Value* to be the set of all values,
- *Var* to be the set of all variable names (we assume $Var \cap Value = \{\}$), and
- $FV(P)$ to be the set of free variables of process or process syntax P.

4.2 Standard CSP operational semantics

In this section we present a standard operational semantics for CSP. It generates LTSs from syntax without free variables. This means that, when dealing with parameterised processes, all parameters have to be assigned concrete values before the transitions rules can be applied. The specification of the operational semantics is presented in Plotkin's style [Plo81]. The inference rules in Section 4.2.1 are based on the ones from [Ros97, Chapter 7], but we introduce some important modifications in Section 4.2.2.

We distinguish two types of transitions: *visible* and *invisible* (also called *internal*). An invisible transition, labelled with τ, represents an event that can be performed by a process without any interaction from the environment and which is not observable by the environment. A visible transition, on the other hand, is labelled with an event that is observable by the environment and requires its synchronisation in order to be performed. We write $P \xrightarrow{a} Q$ to mean that there is an a-labelled transition from state P to state Q and we write $P \xrightarrow{a}$ to mean that there exists some state Q such that $P \xrightarrow{a} Q$. In addition, we write $P \xrightarrow{a}{}^{*} Q$ to mean that there exist states $P_0 = P, P_1, \ldots, P_{n-1}, P_n = Q$, for $n \geq 0$, such that for all i in $\{0 \,..\, n-1\}$, $P_i \xrightarrow{a} P_{i+1}$. In particular, $P \xrightarrow{a}{}^{*} P$ (taking $n = 0$). Finally, $P \xrightarrow{a}{}^{*}$ means that there exists some state Q such that $P \xrightarrow{a}{}^{*} Q$.

Throughout this section we assume that all process definitions satisfy clauses (iii)–(vi) of data independence (Definition 4.1.1) and we let T be a fixed instantiation of type t.

4.2.1 Common firing rules

We now present the inference rules that are identical to those presented in, for example, [Ros97, Chapter 7]. We include them here for reference purposes only.

STOP
STOP is a synonym of deadlock, so there cannot be any inference rules related to it.

External choice
Internal events do not resolve external choice and for $P(T) \,\Box\, Q(T)$, both $P(T)$ and $Q(T)$ are switched on, so any τ's available immediately in $P(T)$ or $Q(T)$ must be promoted, leading to the following rules.

$$\frac{P(T) \xrightarrow{\tau} P'(T)}{P(T) \,\Box\, Q(T) \xrightarrow{\tau} P'(T) \,\Box\, Q(T)} \qquad \frac{Q(T) \xrightarrow{\tau} Q'(T)}{P(T) \,\Box\, Q(T) \xrightarrow{\tau} P(T) \,\Box\, Q'(T)}$$

Any other event, on the other hand, resolves external choice. Hence we have the following two rules.

$$\frac{P(T) \xrightarrow{a} P'(T)}{P(T) \,\Box\, Q(T) \xrightarrow{a} P'(T)} [\, a \neq \tau \,] \qquad \frac{Q(T) \xrightarrow{a} Q'(T)}{P(T) \,\Box\, Q(T) \xrightarrow{a} Q'(T)} [\, a \neq \tau \,]$$

Internal choice
Internal choice is always immediately resolved to one of its branches, producing a single invisible transition.

$$\frac{}{P(T) \sqcap Q(T) \xrightarrow{\tau} P(T)} \qquad \frac{}{P(T) \sqcap Q(T) \xrightarrow{\tau} Q(T)}$$

Sliding choice (timeout)
A sliding choice may be, at any time, resolved to the second argument by a single internal action:

$$\frac{}{P(T) \triangleright Q(T) \xrightarrow{\tau} Q(T)}$$

The first argument of the $\triangleright$ operator is always switched on, so we need to promote all internal actions that it may have available:

$$\frac{P(T) \xrightarrow{\tau} P'(T)}{P(T) \triangleright Q(T) \xrightarrow{\tau} P'(T) \triangleright Q(T)}.$$

The second argument, however, is not switched on, so there is no symmetric rule for $Q(T)$. Finally, the choice may be resolved to the first argument by any of its visible transitions:

$$\frac{P(T) \xrightarrow{a} P'(T)}{P(T) \triangleright Q(T) \xrightarrow{a} P'(T)} \left[\, a \neq \tau \,\right].$$

Parallel composition
Since generalised parallel can be used to express both interleaving and alphabetised parallel, we only need to provide firing rules for the generalised parallel operator. Both arguments of every parallel composition are switched on, so there are two rules to promote all internal actions of the arguments.

$$\frac{P(T) \xrightarrow{\tau} P'(T)}{P(T) \underset{X(T)}{\parallel} Q(T) \xrightarrow{\tau} P'(T) \underset{X(T)}{\parallel} Q(T)}$$

$$\frac{Q(T) \xrightarrow{\tau} Q'(T)}{P(T) \underset{X(T)}{\parallel} Q(T) \xrightarrow{\tau} P(T) \underset{X(T)}{\parallel} Q'(T)}$$

Since an argument of a parallel composition is free to perform visible events that are not in the synchronisation set without any participation of the other argument, we have the following two rules.

$$\frac{P(T) \xrightarrow{e} P'(T)}{P(T) \underset{X(T)}{\parallel} Q(T) \xrightarrow{e} P'(T) \underset{X(T)}{\parallel} Q(T)} \left[\, e \in \Sigma \setminus X(T) \,\right]$$

$$\frac{Q(T) \xrightarrow{e} Q'(T)}{P(T) \underset{X(T)}{\parallel} Q(T) \xrightarrow{e} P(T) \underset{X(T)}{\parallel} Q'(T)} \big[\, e \in \Sigma \setminus X(T) \,\big]$$

Finally, all events in the synchronisation set require participation of both arguments in order to be performed.

$$\frac{P(T) \xrightarrow{e} P'(T) \wedge Q(T) \xrightarrow{e} Q'(T)}{P(T) \underset{X(T)}{\parallel} Q(T) \xrightarrow{e} P'(T) \underset{X(T)}{\parallel} Q'(T)} \big[\, e \in X(T) \,\big]$$

Hiding
Events that are not in the set of events to be hidden remain unchanged:

$$\frac{P(T) \xrightarrow{a} P'(T)}{P(T) \setminus X(T) \xrightarrow{a} P'(T) \setminus X(T)} \big[\, a \notin X(T) \,\big] .$$

However, every visible event that is in the set of events to be hidden is turned into an invisible action:

$$\frac{P(T) \xrightarrow{e} P'(T)}{P(T) \setminus X(T) \xrightarrow{\tau} P'(T) \setminus X(T)} \big[\, e \in X(T) \,\big] .$$

Renaming
Renaming only affects visible events, so every internal action needs to be preserved, leading to the following rule.

$$\frac{P(T) \xrightarrow{\tau} P'(T)}{P(T)[\![\mathcal{R}(T)]\!] \xrightarrow{\tau} P'(T)[\![\mathcal{R}(T)]\!]}$$

On the other hand, every visible event is renamed to a value indicated by the renaming relation:

$$\frac{P(T) \xrightarrow{e} P'(T)}{P(T)[\![\mathcal{R}(T)]\!] \xrightarrow{e'} P'(T)[\![\mathcal{R}(T)]\!]} \big[\, e \mathcal{R}(T) e' \,\big] .$$

Observe that the above rule implies that a single visible transition may lead to multiple transitions if the renaming is a one-to-many map.

4.2.2 Modified and new firing rules

In this section we present operational semantics firing rules that either are not explicitly stated in [Ros97] or are different from those presented in Roscoe's book. The main differences are within the rules for prefix (due to the addition of nondeterministic selections to our language) and recursion (which we handle using identifier bindings rather than fixed points).

Prefix
We have included the nondeterministic selection symbol, \$, in the syntax of the CSP language we consider (see Section 1.5.1). Since this symbol is absent from the language for which the standard operational semantics is defined in [Ros97, Chapter 7], we present a set of modified transition rules for prefix to accommodate this change. The rules for other operators are not affected by the introduction of \$.

Let α be a construct of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ for the rest of this section. The standard inference rule for prefix in the CSP language without the \$ operator (as presented in [Ros97, Chapter 7]) is as follows:

$$\frac{}{\alpha \to P(T) \xrightarrow{c.v_1 \ldots v_k} P(T)[v_i/x_i \mid i \in ?(\alpha)]} \left[\, c.v_1 \ldots v_k \in \mathit{Comms}(\alpha) \,\right],$$

where $\mathit{Comms}(\alpha)$ is the set of concrete events that α describes; formally:

$$\begin{aligned}&\mathit{Comms}(c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k) = \\ &\quad \{c.v_1 \ldots v_k \mid \forall\, i \in \{1\,..\,k\} \bullet (\S_i = ? \wedge v_i \in X_i) \vee (\S_i = ! \wedge v_i = x_i)\}.\end{aligned}$$

In order to define the prefix transition rules for the language with nondeterministic selections added in, we proceed in two steps. Firstly, we deal with constructs with no nondeterministic selections. We do so by adding an additional side condition to the above rule, so that it is never applied to prefixes that contain a nondeterministic selection.

Prefix Rule 1 (prefixes with no nondeterministic selections)

$$\frac{}{\alpha \to P(T) \xrightarrow{c.v_1 \ldots v_k} P(T)[v_i/x_i \mid i \in ?(\alpha)]} \left[\begin{array}{l} c.v_1 \ldots v_k \in \mathit{Comms}(\alpha) \\ \wedge\ \#\$(\alpha) = 0 \end{array}\right]$$

The second step involves deriving transitions from prefix constructs with at least one nondeterministic selection, producing invisible transitions that resolve the choices, and substituting the chosen values for the variables of the choices. The natural way to define such a transition rule would be the following.

$$\frac{\begin{array}{c}\mathrm{dom}(v) = \$(\alpha) \wedge \forall\, i \in \$(\alpha) \bullet v_i \in X_i \\ \alpha \to P(T)\end{array}}{\xrightarrow{\tau}\ \ (\mathit{Replace}_{\$\mapsto !}(\alpha) \to P(T))\,[v_i/x_i \mid i \in \$(\alpha)]} \left[\, \#\$(\alpha) > 0 \,\right]$$

Observe that this rule is equivalent to defining $\alpha \to P(T)$ as

$$\Box\langle x_i{:}X_i \mid i \in \$(\alpha)\rangle \bullet \mathit{Replace}_{\$\mapsto!}(\alpha) \to P(T),$$

where, given an indexing set $\mathcal{I} = \{i_1, \ldots, i_n\}$, we use $\Box\langle x_i{:}X_i \mid i \in \mathcal{I}\rangle \bullet P(x_{i_1}, \ldots, x_{i_n})$ as a shorthand notation for $\Box(x_{i_1}, \ldots, x_{i_n}) \in X_{i_1} \times \cdots \times X_{i_n} \bullet P(x_{i_1}, \ldots, x_{i_n})$. Here all the nondeterministic selections, irrespectively of their types, are resolved simultaneously. However, for reasons that will become clear later, we prefer to simultaneously resolve all nondeterministic selections over types other than t before simultaneously resolving all nondeterministic selections over type t. This leads to the two followings inference rules.

Prefix Rule 2a (prefixes with nondeterministic selections over non-t types)

$$\frac{\mathrm{dom}(v) = \$^{\textit{non-}t}(\alpha) \wedge \forall\, i \in \$^{\textit{non-}t}(\alpha) \bullet v_i \in X_i}{\begin{array}{ll} & \alpha \to P(T) \\ \xrightarrow{\tau} & \big(\mathit{Replace}^{\textit{non-}t}_{\$\mapsto!}(\alpha) \to P(T)\big)\,[v_i/x_i \mid i \in \$^{\textit{non-}t}(\alpha)] \end{array}} \big[\, \#\$^{\textit{non-}t}(\alpha) > 0 \,\big]$$

Prefix Rule 2b (prefixes with nondeterministic selections only over type t)

$$\frac{v \in \$^t(\alpha) \to T}{\begin{array}{ll} & \alpha \to P(T) \\ \xrightarrow{\tau} & \big(\mathit{Replace}^{t}_{\$\mapsto!}(\alpha) \to P(T)\big)\,[v_i/x_i \mid i \in \$^{t}(\alpha)] \end{array}} \left[\begin{array}{l} \#\$^{\textit{non-}t} = 0 \\ \wedge\, \#\$^t(\alpha) > 0 \end{array}\right]$$

The above two rules are consistent with defining $\alpha \to P(T)$ as

$$\Box\langle x_i{:}X_i \mid i \in \$^{\textit{non-}t}(\alpha)\rangle \bullet \big(\Box\langle x_i{:}T \mid i \in \$^t(\alpha)\rangle \bullet \mathit{Replace}_{\$\mapsto!}(\alpha) \to P(T)\big)\,.$$

One of the consequences of using the above two rules is that the resulting LTSs are not strongly bisimilar to the LTSs obtained if the single, natural transition rule we presented earlier was used instead. However, the only difference between the two families of graphs is that LTSs obtained using Prefix Rule 2a and Prefix Rule 2b may contain a single additional τ before a visible event for every construct that contains nondeterministic selections of both type t and at least one type other than t (see Example 4.2.1). This means that all denotational values calculated from the LTSs in one of the standard models are in both cases identical.

Example 4.2.1. Let $\mathit{Proc}(t) = \mathit{chn}\$x{:}\{a, b\}\$y{:}t \to \mathit{STOP}$. Suppose we use the standard operational semantics to work out the transition graph of $\mathit{Proc}(\{0, 1, 2\})$. Figure 4.1(a) shows the resulting LTS when the single, natural way to resolve nondeterministic selections is used, while Figure 4.1(b) shows the resulting LTS when Prefix Rule 2a and Prefix Rule 2b are used.

End of example.

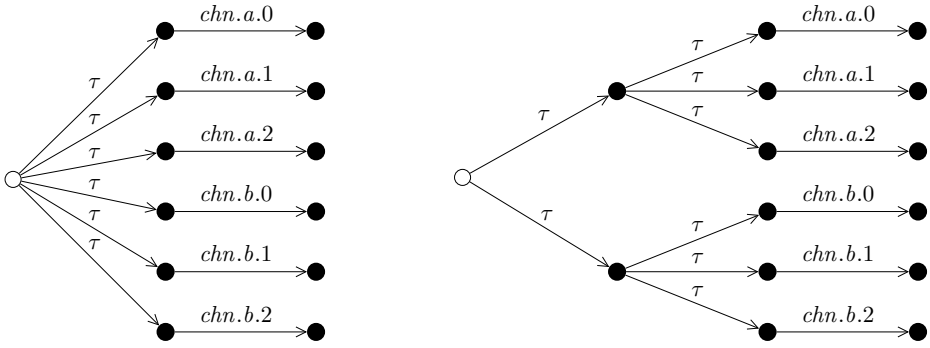


Figure 4.1: Two LTSs of $Proc(\{0,1,2\})$, for $Proc(t) = chn\$x{:}\{a,b\}\$y{:}t \rightarrow STOP$, obtained using: (a) the single, natural way to resolve nondeterministic selections, and (b) combination of Prefix Rule 2a and Prefix Rule 2b.

Conditional choice

The transitions of a conditional choice whose boolean condition evaluates[2] to *True* are precisely those of its positive ("then") branch. Similarly, the transitions of a conditional choice whose boolean condition evaluates to *False* are precisely those of its negative ("else") branch. We add the following two rules to our specification of the operational semantics to capture this formally.

$$\frac{P(T) \xrightarrow{a} P'(T)}{(\text{if } \mathit{True} \text{ then } P(T) \text{ else } Q(T)) \xrightarrow{a} P'(T)}$$

$$\frac{Q(T) \xrightarrow{a} Q'(T)}{(\text{if } \mathit{False} \text{ then } P(T) \text{ else } Q(T)) \xrightarrow{a} Q'(T)}$$

Replicated external choice

Similarly to its binary version, replicated external choice is not resolved by internal events. In addition, all arguments of $\Box$ are switched on. This means that all internal actions of all arguments have to be promoted, leading to the following rule.

$$\frac{P_j(T) \xrightarrow{\tau} P'_j(T)}{\begin{array}{rl} & \Box\, i \in \mathcal{I}(T) \bullet P_i(T) \\ \xrightarrow{\tau} & \Box\, i \in \mathcal{I}(T) \bullet P'_i(T) \end{array}} \left[\begin{array}{l} j \in \mathcal{I}(T) \\ \wedge\, \forall\, i \in \mathcal{I}(T) \setminus \{j\} \bullet P'_i(T) = P_i(T) \end{array}\right]$$

However, visible events performable by any argument of replicated external choice need to resolve the choice, leading to the following rule.

[2]By the time any conditional choice is reached, its boolean condition is guaranteed to have all the variables substituted with concrete values, so the evaluation can be performed immediately.

$$\frac{P_j(T) \xrightarrow{e} P'_j(T)}{\Box\, i \in \mathcal{I}(T) \bullet P_i(T) \xrightarrow{e} P'_j(T)} \big[\, e \neq \tau \wedge j \in \mathcal{I}(T) \,\big]$$

Replicated internal choice
For any non-empty indexing set $\mathcal{I}(T)$, $\sqcap\, i \in \mathcal{I}(T) \bullet P_i(T)$ can nondeterministically choose to behave like any of the $P_i(T)$'s. The internal decision which way this choice is resolved is manifested through a single internal action. Formally:

$$\frac{}{\sqcap\, i \in \mathcal{I}(T) \bullet P_i(T) \xrightarrow{\tau} P_j(T)} \big[\, j \in \mathcal{I}(T) \,\big]\,.$$

Replicated generalised parallel composition
In replicated sharing all processes have to synchronise on the events included in the synchronisation set:

$$\frac{\forall\, i \in \mathcal{I}(T) \bullet P_i(T) \xrightarrow{e} P'_i(T)}{\Big\|_{X(T)}\; i \in \mathcal{I}(T) \bullet P_i(T) \xrightarrow{e} \Big\|_{X(T)}\; i \in \mathcal{I}(T) \bullet P'_i(T)} \big[\, e \in X(T) \,\big]\,.$$

However, each process can perform an internal action or an event not contained in the common synchronisation set without any interaction from other processes, leading to the following rule.

$$\frac{P_j(T) \xrightarrow{a} P'_j(T)}{\Big\|_{X(T)}\; i \in \mathcal{I}(T) \bullet P_i(T) \xrightarrow{a} \Big\|_{X(T)}\; i \in \mathcal{I}(T) \bullet P'_i(T)} \left[\begin{array}{l} a \in \Sigma^\tau \setminus X(T) \wedge j \in \mathcal{I}(T) \\ \wedge\, \forall\, i \in \mathcal{I}(T) \setminus \{j\} \bullet P'_i(T) = P_i(T) \end{array}\right]$$

The above rules also provide the semantics of the replicated interleaving operator, since $\Big|\Big|\Big|\, i \in \mathcal{I}(T) \bullet P_i(T) = \Big\|_{\{\}}\; i \in \mathcal{I}(T) \bullet P_i(T)$.

Replicated alphabetised parallel composition
Since replicated alphabetised parallel cannot (in general) be expressed using the replicated sharing operator (as was the case with the binary versions of these operators), we provide its semantics here. Firstly, each process can perform an internal action without any interaction from other processes:

$$\frac{P_j(T) \xrightarrow{\tau} P'_j(T)}{\begin{array}{rl} & \Big\| \; i \in \mathcal{I}(T) \bullet [A_i(T)]P_i(T) \\ \xrightarrow{\tau} & \Big\| \; i \in \mathcal{I}(T) \bullet [A_i(T)]P'_i(T) \end{array}} \left[\begin{array}{l} j \in \mathcal{I}(T) \\ \wedge \, \forall \, i \in \mathcal{I}(T) \setminus \{j\} \bullet P'_i(T) = P_i(T) \end{array} \right].$$

Secondly, for every visible event e contained in the alphabets of and only of the processes with indexes in some subset $\mathcal{J}(T)$ of an indexing set $\mathcal{I}(T)$, the processes with indexes in $\mathcal{J}(T)$ synchronise on e without any interaction of the processes with indexes in $\mathcal{I}(T) \setminus \mathcal{J}(T)$. Formally:

$$\frac{\forall \, i \in \mathcal{J}(T) \bullet P_i(T) \xrightarrow{e} P'_i(T)}{\begin{array}{rl} & \Big\| \; i \in \mathcal{I}(T) \bullet [A_i(T)]P_i(T) \\ \xrightarrow{e} & \Big\| \; i \in \mathcal{I}(T) \bullet [A_i(T)]P'_i(T) \end{array}} \left[\begin{array}{l} \mathcal{J}(T) \subseteq \mathcal{I}(T) \\ \wedge \; e \in \Big(\bigcap_{i \in \mathcal{J}(T)} A_i(T) \Big) \\ \qquad \setminus \bigcup_{i \in \mathcal{I}(T) \setminus \mathcal{J}(T)} A_i(T) \\ \wedge \, \forall \, j \in \mathcal{I}(T) \setminus \mathcal{J}(T) \bullet \\ \qquad P'_j(T) = P_j(T) \end{array} \right].$$

Binding
Recall that we defined E to be a global environment, i.e. a map from identifiers to process definitions (Section 1.5.1). Whenever a process identifier is encountered, it is enough to look it up in E, the act of which produces a single internal action.

$$\frac{E(X) = P}{X(T) \xrightarrow{\tau} P(T)}$$

Example 4.2.2. Let

$$\begin{aligned} Proc(t) = c\$x{:}\{a, b\}\$y{:}t?z{:}t \rightarrow \text{if } y = z \text{ then } d!x \rightarrow STOP \\ \text{else } STOP. \end{aligned}$$

Figure 4.2 represents the operational semantics for $Proc(T)$ with $T = \{0, 1\}$. We omit part of the semantics because of lack of space. In the figure, we write $Q_{x,y,z}$ as a shorthand for if $y = z$ then $d!x \rightarrow STOP$ else $STOP$. The first τ transitions correspond to Prefix Rule 2a; the second τ transitions correspond to Prefix Rule 2b; the visible transitions correspond to Prefix Rule 1.

End of example.

4.2.3 Calculating denotational values

It is possible to calculate denotational values of processes without resorting to operational semantics. Such a direct way, using denotational semantics, is discussed in [Ros97, Chapter 8]. However, since we will often work with LTSs, it makes sense to derive these values directly from transition graphs. Firstly, we need two definitions

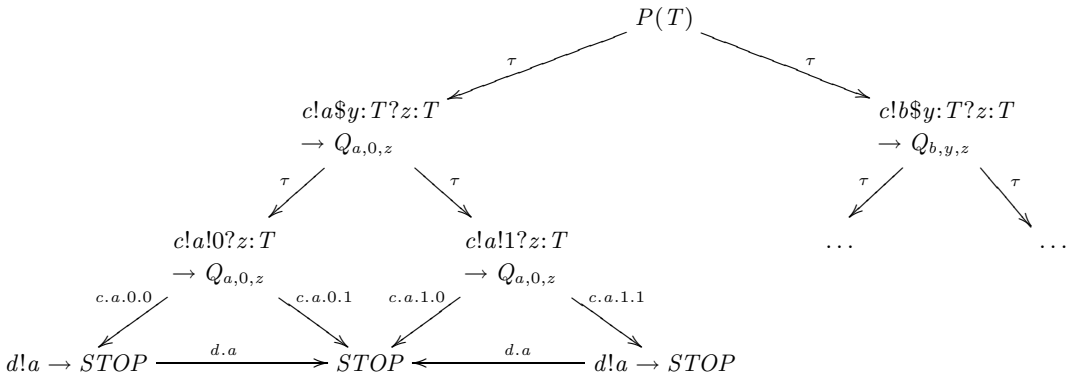


Figure 4.2: LTS of the process from Example 4.2.2.

(from [Ros97, Chapter 7]). Given two states, say $P(T)$ and $Q(T)$, and a sequence of events (visible or invisible) $s = \langle a_i \mid i \in \{1 \,..\, n\}\rangle$ for some $n \geq 0$, we write

$$P(T) \stackrel{s}{\longmapsto} Q(T)$$

if there exist states $P_0(T) = P(T), P_1(T), \ldots, P_n(T) = Q(T)$ such that for all i in $\{0 \,..\, n-1\}$ we have that $P_i(T) \stackrel{a_{i+1}}{\longrightarrow} P_{i+1}(T)$. In addition,

$$P(T) \stackrel{s}{\longmapsto}$$

if there exists some process $Q(T)$ such that $P(T) \stackrel{s}{\longmapsto} Q(T)$. Further, we write

$$P(T) \stackrel{tr}{\Longrightarrow} Q(T)$$

if there is s such that $P(T) \stackrel{s}{\longmapsto} Q(T)$ and tr is the restriction of s to visible events. In addition,

$$P(T) \stackrel{tr}{\Longrightarrow}$$

if there exists some process $Q(T)$ such that $P(T) \stackrel{tr}{\Longrightarrow} Q(T)$.

We are now ready to work out $traces(P)$ and $failures(P)$ of a given process $P(T)$:

$$traces(P(T)) = \{tr \in \Sigma^* \mid \exists\, Q(T) \bullet P(T) \stackrel{tr}{\Longrightarrow} Q(T)\}$$

and

$$failures(P(T)) = \{(tr, X) \in \Sigma^* \times \Sigma \mid \exists\, Q(T) \bullet P(T) \stackrel{tr}{\Longrightarrow} Q(T) \\ \wedge\, X \subseteq \Sigma \wedge Q(T)\ ref\ X\},$$

where $Q(T)$ *ref* X means that $Q(T)$ is in a stable state (i.e. one where it cannot perform any invisible actions) and for all events a in X and all states $Q'(T)$ the transition $Q(T) \stackrel{a}{\longrightarrow} Q'(T)$ is not available.

4.3 Semi-Symbolic Operational Semantics

Symbolic representation of models is often used in model checking (see e.g. [McM92, BCM+92]). In most cases the approach taken is to create a single, compact structure that represents the behaviour of multiple instances of a given system. In such settings a symbolic state is a *propositional formula* that describes sets of concrete states. The specification check is then performed on the symbolic model in order to deduce verification results for all the concretisations this model corresponds to.

In this section we present a symbolic operational semantics for CSP. In contrast to the above, its aim is not to be used to perform abstract refinement checking of processes. The symbolic operational semantics works in a way similar to that of the standard operational semantics (Section 4.2). The main difference is that all parts of systems that involve the distinguished type are left in their original, uninstantiated form. Therefore, the labels of transitions may contain some symbolic parts and some concrete parts; symbolic states are still *process syntaxes*, derived from original process syntaxes. This is why we call each resulting transition graph a *semi-symbolic labelled transition system* (SSLTS). For the same reasons we call our operational semantics *Semi-Symbolic Operational Semantics* (SSOS).

SSLTSs act as a bridge between processes obtained from a single process syntax by taking different instantiations of the distinguished type. They are of interest to us as, later on, they will allow us to use known behaviours of a given instance of the process to deduce facts about the behaviour of other instances of the same process definition.

Throughout this section we assume that all processes satisfy **SeqNorm**.

4.3.1 Symbolic transitions

In order to be able to tell symbolic and standard transitions apart, the symbolic transition relation is denoted by $\longrightarrow_s$, i.e. $P(t) \xrightarrow{\alpha}_s Q(t)$ denotes the fact that there is an α-labelled transition from symbolic state $P(t)$ to symbolic state $Q(t)$. We write $P(t) \xrightarrow{\alpha}_s$ to mean that there exists some symbolic state $Q(t)$ such that $P(t) \xrightarrow{\alpha} Q(t)$. In addition, we write $P(t) \xrightarrow{\alpha}_s^* Q(t)$ to mean that there exist symbolic states $P_0(t) = P(t), P_1(t), \ldots, P_{n-1}(t), P_n(t) = Q(t)$, for $n \geq 0$, such that for all i in $\{0 \,..\, n-1\}$, $P_i(t) \xrightarrow{\alpha}_s P_{i+1}(t)$. In particular, $P(t) \xrightarrow{\alpha}_s^* P(t)$ (taking $n = 0$). Finally, $P(t) \xrightarrow{\alpha}^*$ means that there exists some symbolic state $Q(t)$ such that $P(t) \xrightarrow{\alpha}^* Q(t)$.

We distinguish the following three types of symbolic transitions.

Invisible The invisible symbolic transitions are in a direct correspondence with the standard invisible transitions and are labelled with τ's (which in this setting are be called *invisible* (or *internal*) *symbolic events*).

Visible Visible symbolic transitions are similar to standard visible transitions. The main difference is that while the labels of standard visible transitions contain no input symbols and no variables, labels of visible symbolic transitions may contain nondeterministic selections of type t (i.e. $\$x{:}t$), deterministic inputs of

type t (i.e. $?x{:}t$), outputs of type t (i.e. $!x$, where x is a variable of type t), or outputs of non-t parts (i.e. $!v$, where v is a value not of type t). Formally, each visible symbolic transition is labelled with a *visible symbolic event*, a construct of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ for $k > 0$, where

- c is a channel name,
- $\S_i \in \{\$, ?, !\}$ is an input/output symbol,
- x_i is a variable of type t or a value of type other than t (it can be a value only if it is immediately preceded by the output symbol !),
- X_i is t if and only if the preceding input/output symbol, $\S_i$, is either \$ or ?; otherwise it is *null*.

We let *Visible* denote the set of all visible symbolic events.

Conditional Since variables of type t are not instantiated within SSLTSs, but left in their symbolic form, any boolean condition that contains at least one such variable and is non-trivial (e.g. not "$x = x$") cannot be evaluated to either *True* or *False* at the time of generating a symbolic transition graph. Hence, in order to deal with processes with such conditional choices involving variables of type t, we introduce conditional symbolic transitions. Each such transition is labelled with a *conditional symbolic event*, a boolean expression obtained from the guard of a conditional choice on t or its negation. For example, the syntax "if $x = y$ then P else Q" gives raise to the conditional symbolic event "$x = y$" and "$\neg x = y$". We let *Cond* denote the set of all possible conditional symbolic events. Without loss of generality we assume that the process syntax contains no trivial conditionals such as "$x = x$".

Remark 4.3.1. If ϵ is a visible symbolic event, then $\$^{non\text{-}t}(\epsilon) = ?^{non\text{-}t}(\epsilon) = \{\}$.

Example 4.3.2. Recall the following process definition that we introduced in Example 4.2.1:

$$Proc(t) = chn\$x{:}\{a, b\}\$y{:}t \rightarrow STOP.$$

The visible events (available after resolving the first nondeterministic selection; see the inference rules below) are $chn!a\$y{:}t$ and $chn!b\$y{:}t$.

End of example.

We will usually use α and its derivatives (α', α_1, etc.) to denote labels whose kind is unknown or indifferent and ϵ and its derivatives (ϵ', ϵ_1, etc.) to denote visible symbolic events.

4.3.2 Firing rules

We define the specification of Semi-Symbolic Operational Semantics using a set of inference rules, below. Recall that we are considering only processes that satisfy **SeqNorm**. Therefore, we only provide transition rules for operators that the

condition allows.

STOP
Since *STOP* is a synonym of deadlock, there are no symbolic firing rules associated with it.

Prefix
Let α be a construct of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ for the rest of this section. There are two transition rules for prefix. The first one defines the initial symbolic events of $\alpha \rightarrow P(t)$ in the case when α contains no nondeterministic selections over types other than t. It is similar to Prefix Rule 1 from the standard operational semantics (see Section 4.2.2), except that variables of type t are left in their symbolic form when a transition label is obtained from α.

Symbolic Prefix Rule 1

$$\frac{\epsilon = c\S'_1 x'_1{:}X'_1 \ldots \S'_k x'_k{:}X'_k}{(\alpha \rightarrow P(t)) \xrightarrow{\epsilon}_s P(t)[x'_i/x_i \mid i \in ?^{non\text{-}t}(\alpha)]} \left[\begin{array}{l} \epsilon \in Comms^{non\text{-}t}(\alpha) \\ \wedge\ \#\$^{non\text{-}t}(\alpha) = 0 \end{array}\right],$$

where $Comms^{non\text{-}t}(\alpha)$ is the set of events that α describes (under the assumption that α contains no nondeterministic selections over types other than t), with the parts involving type t left in their symbolic form; formally:

$$\begin{array}{l} Comms^{non\text{-}t}(c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k) = \\ \quad \{c\S'_1 x'_1{:}X'_1 \ldots \S'_k x'_k{:}X'_k \mid \forall\, i \in \{1\,..\,k\} \bullet \\ \qquad \S_i = ? \wedge X_i \neq t \wedge \S'_i = ! \wedge x'_i \in X_i \wedge X'_i = null \\ \qquad \vee\ \S_i = \S'_i \in \{\$, ?\} \wedge X'_i = X_i = t \wedge x'_i = x_i \\ \qquad \vee\ \S_i = \S'_i = ! \wedge X'_i = X_i \wedge x'_i = x_i\}. \end{array}$$

The second transition rule of prefix deals with prefixes that contain at least one nondeterministic selection over a type other than t. It is similar to Prefix Rule 2a from the standard operational semantics (Section 4.2.2), except it uses symbolic transitions instead of standard transitions and process syntaxes with free identifiers instead of concrete processes. All the nondeterministic selections over types other than t are resolved simultaneously, the act of which generates a single τ transition. The values being chosen are substituted for the variables of the selections.

Symbolic Prefix Rule 2

$$\frac{\mathrm{dom}(v) = \$^{non\text{-}t}(\alpha) \wedge \forall\, i \in \$^{non\text{-}t}(\alpha) \bullet v_i \in X_i}{\begin{array}{ll} & \alpha \rightarrow P(t) \\ \xrightarrow{\tau}_s & \left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha) \rightarrow P(t)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)] \end{array}} \left[\, \#\$^{non\text{-}t}(\alpha) > 0 \,\right]$$

External choice
There is very little difference between the transition rules for external choice in SSOS and the standard operational semantics (Section 4.2). One exception is the presence of conditional symbolic transitions, which need to be taken into considerations here. Conditional choice must be allowed to be resolved without any other influence on the overall state of the system, which means that the members of *Cond* must be promoted by the $\Box$ operator in the same way τ's are, leading to the following two firing rules.

$$\frac{P(t) \xrightarrow{\alpha}_s P'(t)}{P(t) \,\Box\, Q(t) \xrightarrow{\alpha}_s P'(t) \,\Box\, Q(t)} \big[\, \alpha \in \mathit{Cond} \cup \{\tau\} \,\big]$$

$$\frac{Q(t) \xrightarrow{\alpha}_s Q'(t)}{P(t) \,\Box\, Q(t) \xrightarrow{\alpha}_s P(t) \,\Box\, Q'(t)} \big[\, \alpha \in \mathit{Cond} \cup \{\tau\} \,\big]$$

Visible symbolic events, however, resolve external choice (in exactly the same way standard visible events do).

$$\frac{P(t) \xrightarrow{\epsilon}_s P'(t)}{P(t) \,\Box\, Q(t) \xrightarrow{\epsilon}_s P'(t)} \big[\, \epsilon \notin \mathit{Cond} \cup \{\tau\} \,\big]$$

$$\frac{Q(t) \xrightarrow{\epsilon}_s Q'(t)}{P(t) \,\Box\, Q(t) \xrightarrow{\epsilon}_s Q'(t)} \big[\, \epsilon \notin \mathit{Cond} \cup \{\tau\} \,\big]$$

Internal choice
Similarly to the standard operational semantics (Section 4.2.1), internal choice is always immediately resolved to one of its branches, producing a single invisible symbolic transition.

$$\frac{}{P(t) \sqcap Q(t) \xrightarrow{\tau}_s P(t)} \qquad \frac{}{P(t) \sqcap Q(t) \xrightarrow{\tau}_s Q(t)}$$

Sliding choice (timeout)
The transitions rules for the $\triangleright$ operator are similar to the ones from the standard operational semantics.

$$\frac{}{P(t) \triangleright Q(t) \xrightarrow{\tau}_s Q(t)}$$

$$\frac{P(t) \xrightarrow{\alpha}_s P'(t)}{P(t) \triangleright Q(t) \xrightarrow{\alpha}_s P'(t) \triangleright Q(t)} \big[\, \alpha \in \mathit{Cond} \cup \{\tau\} \,\big]$$

$$\frac{P(t) \xrightarrow{\epsilon}_s P'(t)}{P(t) \triangleright Q(t) \xrightarrow{\epsilon}_s P'(t)} \big[\, \epsilon \notin \mathit{Cond} \cup \{\tau\} \,\big]$$

Conditional choice
Clause (i) of Assumption 1.6.1 implies that guards of conditional choices may not contain both variables of type t and variables of non-t types. The truth of any boolean condition that contains no variables of type t (i.e. every conditional not in *Cond*) can be fully evaluated at the time of SSLTS generation. Hence, we have the following rules, similar to the standard ones.

$$\frac{P(t) \xrightarrow{\alpha}_s P'(t)}{(\text{if } \mathit{True} \text{ then } P(t) \text{ else } Q(t)) \xrightarrow{\alpha}_s P'(t)}$$

$$\frac{Q(t) \xrightarrow{\alpha}_s Q'(t)}{(\text{if } \mathit{False} \text{ then } P(t) \text{ else } Q(t)) \xrightarrow{\alpha}_s Q'(t)}$$

Every conditional choice with a boolean condition *cond* that involves type t and whose truth cannot be evaluated at the time of SSLTS generation (i.e. every conditional in *Cond*) may either evolve to the positive branch by following a conditional transition labelled with *cond* or it may evolve to the negative branch by following a conditional transition labelled with the negation of *cond*.

$$\frac{}{(\text{if } \mathit{cond} \text{ then } P(t) \text{ else } Q(t)) \xrightarrow{\mathit{cond}}_s P(t)} \big[\, \mathit{cond} \in \mathit{Cond} \big]$$

$$\frac{}{(\text{if } \mathit{cond} \text{ then } P(t) \text{ else } Q(t)) \xrightarrow{\neg \mathit{cond}}_s Q(t)} \big[\, \mathit{cond} \in \mathit{Cond} \big]$$

Binding
Similarly to the standard operational semantics case, whenever a process identifier is encountered, it is enough to look it up in global environment E. Each such look-up produces a single invisible symbolic transition.

$$\frac{E(X) = P}{X(t) \xrightarrow{\tau}_s P(t)}$$

Example 4.3.3. Recall the following process from Example 4.2.2:

$$\mathit{Proc}(t) = c\$x{:}\{a, b\}\$y{:}t?z{:}t \rightarrow \text{if } y = z \text{ then } d!x \rightarrow \mathit{STOP}. \\ \text{else } \mathit{STOP}$$

Figure 4.3 we represent the semi-symbolic operational semantics for $\mathit{Proc}(t)$. In the figure, we write $Q_{x,y,z}$ as a shorthand for if $y = z$ then $d!x \rightarrow \mathit{STOP}$ else STOP. The τ transitions correspond to Symbolic Prefix Rule 2; the visible transitions correspond to Symbolic Prefix Rule 1.

End of example.

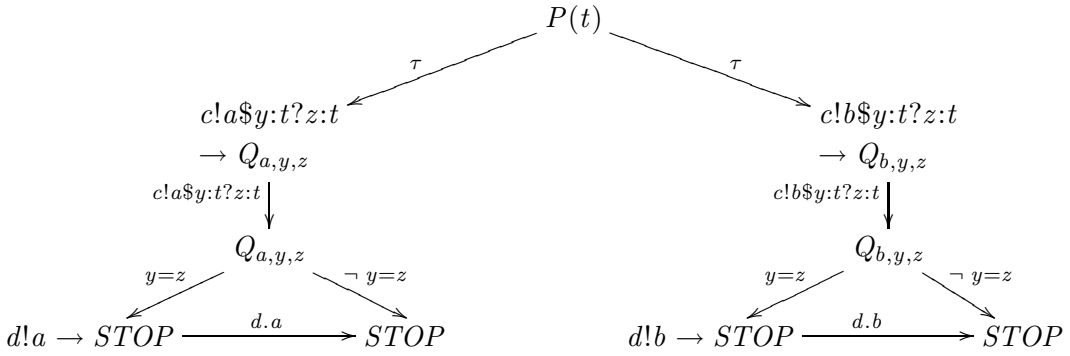


Figure 4.3: SSLTS of the process from Example 4.3.3.

We end this section with a remark that allows us to reduce the size of obtained SSLTSs.

Remark 4.3.4. When building SSLTSs using the firing rules presented above, we treat two symbolic states as identical if their operational semantics are alpha-equivalent, i.e. if their operational semantics are equal modulo renaming of bound variables.

Example 4.3.5. Consider the following three definitions of a one place buffer for items of type t:

$$COPY(t) = in?x{:}t \rightarrow out.x \rightarrow COPY(t),$$
$$COPY'(t) = in?y{:}t \rightarrow out.y \rightarrow COPY'(t),$$
$$COPY''(t) = in?x{:}t \rightarrow out.x \rightarrow in?y{:}t \rightarrow out.y \rightarrow COPY'(t).$$

Then, we consider $COPY(t)$ and $COPY''(t)$ to be identical, but not $COPY(t)$ and $COPY''(t)$, since $COPY(t)$ contains additional τ's that correspond to the lookup of process identifier $COPY(t)$. In addition, symbolic states $out.x \rightarrow COPY(t)$ and $out.y \rightarrow COPY'(t)$ are not identical, since variables x and y are not bound.

End of example.

4.3.3 Symbolic traces

Symbolic traces will play a vital role in the analysis of behaviour of process families based on SSOS. They are similar to ordinary CSP traces (Section 1.5.2), except they contain visible symbolic events instead of ordinary visible events and may contain conditional symbolic events. Formally, we define a symbolic trace as follows. Let $Proc(t)$ be some process syntax and let $\mathcal{S} = (S, s_0, L, \longrightarrow_s)$ be the SSLTS obtained by applying the SSOS to $Proc(t)$. Given two symbolic states $P(t)$ and $Q(t)$ in S and a sequence of symbolic events $\sigma = \langle \alpha_i \mid i \in \{1 \mathinner{..} n\} \rangle$, we say that

$$P(t) \stackrel{\sigma}{\longmapsto}_s Q(t)$$

if there exist symbolic states $P_0(t) = P(t), P_1(t), \ldots, P_n(t) = Q(t)$ such that for all i in $\{0 \,..\, n-1\}$ we have that $P_i(t) \xrightarrow{\alpha_{i+1}}_s P_{i+1}(t)$. We also write

$$P(t) \overset{\sigma}{\longmapsto}_s$$

to mean that there exists a symbolic state $Q(t)$ such that $P(t) \overset{\sigma}{\longmapsto}_s Q(t)$. If $P(t) \overset{\sigma}{\longmapsto}_s$, then σ is called a *symbolic trace* of $P(t)$. Therefore, a symbolic trace of $Proc(t)$ is a sequence of labels of symbolic events that form a path, starting at s_0, through the $\mathcal{S}$. We let $SymbolicTraces(Proc(t))$ denote the set of all symbolic traces of $Proc(t)$. Observe that symbolic traces are quite different from standard traces as they may contain invisible symbolic events (τ's) and conditional symbolic events, while ordinary traces contain only visible events. In Section 4.6 we will study the relationship between symbolic and concrete traces in more detail. We will usually use σ, ρ and their derivatives (σ', ρ_1, etc.) to denote symbolic traces.

4.4 Concrete Operational Semantics with Environments

So far we have defined a concrete and a (semi) symbolic operational semantics for CSP (see Section 4.2 and Section 4.3, respectively). In this section we present a concrete operational semantics which joins the two together. We call it *Concrete Operational Semantics with Environments* (COSE). The states of an SSLTS correspond to the control states of a given process. In order to link the symbolic and concrete states (where the latter contain information not only about program state, but also about data of type t) we need a mechanism for introducing concrete values into symbolic states. We do this through the use of *environments*. The environments defined in this section are a concept very different from that of an environment with which processes communicate, or E, the global map of identifiers to process definitions that we introduced in Section 1.5.1. Intuitively, environments map free variables within process syntaxes to concrete values that were previously bound to such variables through some inputs. The following, formal definition is more general, but, combined with the rules of COSE (Section 4.4.1), will capture our intuition.

Definition 4.4.1. Let $Env(t) \mathrel{\widehat{=}} Var \nrightarrow t$. Then an *environment* is a partial function $\Gamma \in Env(T)$ for some instantiation T of type t.

We adopt the notational convention that for all v in $Value$, $\Gamma(v) = v$. We lift the application of environments to various structures that we use (constructs, processes, sets, relations, etc.) in the natural way: if $X(T)$ is such a structure and Γ is in $Env(T)$, then $\Gamma(X(T))$ is a structure like $X(T)$ but with every free variable x of type t replaced by $\Gamma(x)$ (assuming x is in $\mathrm{dom}(\Gamma)$). In particular, given a process definition $P(t)$ we define

$$P(t)[\Gamma] \mathrel{\widehat{=}} P(t)[\Gamma(x)/x \mid x \in \mathrm{dom}(\Gamma)].$$

Note that for all environments Γ and all symbolic events α we have that

$$\$^\dagger(\Gamma(\alpha)) = \$^\dagger(\alpha) \quad \text{and} \quad ?^\dagger(\Gamma(\alpha)) = ?^\dagger(\alpha)$$

for $\dagger \in \{t, \mathit{non}\text{-}t\}$, which means that

$$\$(\Gamma(\alpha)) = \$(\alpha) \quad \text{and} \quad ?(\Gamma(\alpha)) = ?(\alpha).$$

Let T and T' be two instantiations of type t. Then, given a function $f : T \to T'$ and an environment Γ in $\mathit{Env}(T)$, we define

$$f(\Gamma) \mathrel{\widehat{=}} \{x \mapsto f(v) \mid \Gamma(x) = v\}.$$

Observe that $f(\Gamma)$ is an environment in $\mathit{Env}(T')$.

The states of the LTS $\mathcal{C}$ that COSE generates from a given process syntax $\mathit{Proc}(t)$ are configurations $(P(t), \Gamma, T)$, where:

- $P(t)$ is a symbolic state, equal to or slightly modified from a state of the SSLTS $\mathcal{S}$ of $\mathit{Proc}(t)$,
- Γ is an environment in $\mathit{Env}(T)$, and
- T is a concrete instantiation of type t.

Note that the inclusion of the type instantiation as the third element of a configuration means that each choice of T gives rise to a different LTS.

The initial state of $\mathcal{C}$ is defined to be the configuration $(P_0(t), \{\}, T)$, where $P_0(t)$ is the initial state of $\mathcal{S}$. To emphasise the fact that COSE is a concrete operational semantics we denote the transition relation using the same symbol ($\longrightarrow$) that we used in Section 4.2. Whenever the concrete type T is clear from the context or indifferent, we omit it from the configurations and use pairs $(P(t), \Gamma)$ consisting of a symbolic state $P(t)$ and an environment Γ in $\mathit{Env}(T)$.

We treat two configurations as identical if they describe exactly the same process. Formally, $(P(t), \Gamma, T) = (P'(t), \Gamma', T')$ if and only if:

- $P(t)[\Gamma] \equiv_\alpha P'(t)[\Gamma']$, and
- $T = T'$,

where $\equiv_\alpha$ denotes operational semantics alpha-equivalence, i.e. equality of operational semantics modulo renaming of bound variables.

Example 4.4.2. Recall the following definitions of a one place buffer for items of type t from Example 4.3.5:

$$\mathit{COPY}(t) = \mathit{in}?x{:}t \to \mathit{out}.x \to \mathit{COPY}(t),$$
$$\mathit{COPY}'(t) = \mathit{in}?y{:}t \to \mathit{out}.y \to \mathit{COPY}'(t).$$

Then,

$$(\mathit{COPY}(t), \{\}, T) = (\mathit{COPY}'(t), \{\}, T)$$

for all T, since $\mathit{COPY}(t)[\{\}]$ is equal to $\mathit{COPY}'(t)[\{\}]$ with bound variable y renamed to x. Also, we have that

$$(\mathit{out}.x \to \mathit{COPY}(t), \{x \mapsto 0\}, T) = (\mathit{out}.y \to \mathit{COPY}'(t), \{x \mapsto 0, y \mapsto 0\}, T),$$

since

$$\begin{array}{ll} & (out.x \to COPY(t))[\{x \mapsto 0\}] \\ = & out.0 \to COPY(t)[\{x \mapsto 0\}] \\ \equiv_\alpha & out.0 \to COPY'(t)[\{x \mapsto 0, y \mapsto 0\}] \\ = & (out.y \to COPY'(t))[\{x \mapsto 0, y \mapsto 0\}]. \end{array}$$

End of example.

Remark 4.4.3. Observe that[3] $P(t)[\Gamma] = P(t)[FV(P(t)) \lhd \Gamma]$ for all symbolic states $P(t)$ and all environments Γ. Therefore, configurations $(P(t), \Gamma, T)$ and $(P(t), FV(P(t)) \lhd \Gamma, T)$ are identical. From now on we always assume environment minimality within configurations, which we achieve by restricting the environment Γ of every configuration $(P(t), \Gamma, T)$ to the free variables of $P(t)$.

Throughout the rest of this section we assume that all processes satisfy **SeqNorm**.

4.4.1 Translation rules

We present the specification of COSE using a set of translation rules that show how concrete transition graphs are obtained from SSLTSs. Let T be a fixed instantiation of the distinguished type parameter t for the rest of this section.

Given a symbolic state $P(t)$, we let $Q(t) = Replace^t_{\$\mapsto!}(c, P(t))$ be a symbolic state like $P(t)$, except every transition from $P(t)$ labelled with a visible symbolic transition ϵ on channel c is replaced with an identical transition in $Q(t)$, but labelled with $Replace^t_{\$\mapsto!}(\epsilon)$ instead, i.e. $P(t) \xrightarrow{\epsilon}_s P'(t)$ if and only if $Q(t) \xrightarrow{Replace^t_{\$\mapsto!}(\epsilon)} P'(t)$. We will see later (Corollary 5.1.15) that all such transitions over c result from the same construct.

The first translation rule shows how visible symbolic events that contain a nondeterministic selection over type t get instantiated into ordinary invisible transitions and leave a visible symbolic event with no nondeterministic selection over type t to be dealt with later.

Translation Rule 1

$$\frac{\begin{array}{c} P(t) \xrightarrow{\epsilon}_s Q(t) \\ \epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \wedge v \in \$^t(\epsilon) \to T \end{array}}{\begin{array}{ll} & (P(t), \Gamma, T) \\ \xrightarrow{\tau} & (Replace^t_{\$\mapsto!}(c, P(t)), \Gamma \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon)\}, T) \end{array}} \big[\, \#\$^t(\epsilon) > 0 \,\big].$$

[3] Given a function f and a set $S \subseteq \mathrm{dom}(f)$, $S \lhd f$ denotes the function obtained from f by restricting its domain to the elements of S, i.e.

$S \lhd f = \{x \mapsto f(x) \mid x \in S\}$.

When we defined **SeqNorm** in Section 4.1.2 we made an assumption that there are is never a clash between a nondeterministic input variable of type t from one branch of an external or sliding choice and a free variable present in the other branch. Without this assumption, Translation Rule 1 could produce wrong answers, as demonstrated by the following example.

Example 4.4.4. Let

$$Proc(t) = c_1?x{:}t \rightarrow (c_2\$x{:}t?y{:}t \rightarrow STOP \,\Box\, c_1!x \rightarrow STOP)$$

and let $T = \{0, 1\}$. Then, after performing $c_1.0$ and a τ resolving the nondeterministic selection by choosing $x = 1$, the configuration $(Proc(T), \{\})$ evolves to $(c_2!x?y{:}T \rightarrow STOP \,\Box\, c_1!x \rightarrow STOP, \{x \mapsto 1\})$. Then, by Translation Rule 2 (see below), the event $c_1.1$ is available, which clearly should not be the case.

End of example.

Next, we show how visible symbolic events that contain no nondeterministic selections of type t get instantiated into concrete visible events by substituting values from the environment for all the outputs of type t and choosing the values of all deterministic inputs of type t.

Translation Rule 2

$$\frac{\begin{array}{c} P(t) \xrightarrow{\epsilon}_s Q(t) \\ \epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \\ \left(\begin{array}{c} \mathrm{dom}(v) = \{1\,..\,k\} \wedge (\forall\, i \in ?^t(\epsilon) \bullet v_i \in T) \\ \wedge\, \forall\, i \in !(\epsilon) \bullet v_i = \Gamma(x_i) \end{array}\right) \end{array}}{(P(t), \Gamma, T) \xrightarrow{c.v_1 \ldots v_k} (Q(t), \Gamma \oplus \{x_i \mapsto v_i \mid i \in ?^t(\epsilon)\}, T)} \left[\, \#\$^t(\epsilon) = 0 \,\right].$$

Example 4.4.5. Let

$$Proc(t) = chn?x{:}\{a, b\}\$y{:}t?z{:}t \rightarrow Proc'(t)$$

and $T = \{0, 1\}$. Then Symbolic Prefix Rule 1 implies that

$$Proc(t) \xrightarrow{chn!a\$y:t?z:t}_s Proc'(t)[a/x], \tag{4.1}$$

$$Proc(t) \xrightarrow{chn!b\$y:t?z:t}_s Proc'(t)[b/x]. \tag{4.2}$$

Considering either transition, Translation Rule 1 implies that configuration $(Proc(t), \{\,\}, T)$ can do a τ and become either of

$$conf_0 = (Replace^t_{\$\mapsto!}(c, Proc(t)), \{y \mapsto 0\}, T),$$
$$conf_1 = (Replace^t_{\$\mapsto!}(c, Proc(t)), \{y \mapsto 1\}, T).$$

Now, from (4.1) and (4.2) and the definition of *Replace*,

$$\begin{aligned}
&\mathit{Replace}^{t}_{\$\mapsto !}(c, \mathit{Proc}(t)) \xrightarrow{\mathit{chn}!a!y?z{:}t}_s \mathit{Proc}'(t)[a/x],\\
&\mathit{Replace}^{t}_{\$\mapsto !}(c, \mathit{Proc}(t)) \xrightarrow{\mathit{chn}!b!y?z{:}t}_s \mathit{Proc}'(t)[b/x].
\end{aligned}$$

Hence, using Translation Rule 2, we can deduce

$$\begin{aligned}
&\mathit{conf}_0 \xrightarrow{\mathit{chn}.a.0.0} (\mathit{Proc}'(t)[a/x], \{y \mapsto 0, z \mapsto 0\}, T),\\
&\mathit{conf}_0 \xrightarrow{\mathit{chn}.a.0.1} (\mathit{Proc}'(t)[a/x], \{y \mapsto 0, z \mapsto 1\}, T),\\
&\mathit{conf}_0 \xrightarrow{\mathit{chn}.b.0.0} (\mathit{Proc}'(t)[b/x], \{y \mapsto 0, z \mapsto 0\}, T),\\
&\mathit{conf}_0 \xrightarrow{\mathit{chn}.b.0.1} (\mathit{Proc}'(t)[b/x], \{y \mapsto 0, z \mapsto 1\}, T);
\end{aligned}$$

and similarly for conf_1.

End of example.

Remark 4.4.6. We can combine Translation Rule 1 and Translation Rule 2 to deduce that if

$$\begin{aligned}
&P(t) \xrightarrow{\epsilon}_s Q(t) \wedge \epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k\\
&\wedge \operatorname{dom}(v) = \{1 \,..\, k\} \wedge (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v_i \in T) \wedge \forall\, i \in !(\epsilon) \bullet v_i = \Gamma(x_i),
\end{aligned}$$

then

$$(P(t), \Gamma, T)\ [\xrightarrow{\tau}]\ \xrightarrow{c.v_1 \ldots v_k} (Q(t), \Gamma \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon) \cup ?^t(\epsilon)\}, T),$$

where $[\xrightarrow{\tau}]$ denotes an optional τ transition, present if $\#\$^t(\epsilon) > 0$.

The next translation rule says that when a concrete LTS is obtained from an SSLTS, invisible symbolic transitions are turned into standard invisible transitions.

Translation Rule 3

$$\frac{P(t) \xrightarrow{\tau}_s Q(t)}{(P(t), \Gamma, T) \xrightarrow{\tau} (Q(t), \Gamma, T)}$$

The final translation rule shows how conditional symbolic transitions disappear when an SSLTS is instantiated into a concrete LTS using COSE; the labels are evaluated in the environment, affecting the availability of the subsequent transitions.

Translation Rule 4

$$\frac{\begin{array}{c}P(t) \xrightarrow{\mathit{cond}}_s Q(t)\\ (Q(t), \Gamma, T) \xrightarrow{a} (R(t), \Gamma', T)\end{array}}{(P(t), \Gamma, T) \xrightarrow{a} (R(t), \Gamma', T)} \left[\ \mathit{cond} \in \mathit{Cond} \wedge [\![\mathit{cond}]\!]_\Gamma\ \right],$$

where $[\![\mathit{cond}]\!]_\Gamma$ denotes the truth value of the proposition obtained from *cond* by substituting all free variables of type t with their corresponding values contained within the environment Γ. Note that if $(Q(t), \Gamma, T)$ is deadlocked, then so is $(P(t), \Gamma, T)$.

4.5 Congruence of SSOS and COSE to the standard operational semantics

We will often work with concrete LTSs generated by COSE rather than by the standard operational semantics. It is therefore important that the two operational semantics are congruent so that any denotational values extracted from them are identical. The following theorem proves such a congruence.

Theorem 4.5.1. (Congruence of SSOS and COSE to the standard operational semantics)
*Suppose that $Proc(t)$ is some process syntax that satisfies **SeqNorm** . Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the LTSs generated from $Proc(t)$, for some fixed instantiation T of type t, using COSE and the standard operational semantics, respectively. Then $\mathcal{L}_1$ and $\mathcal{L}_2$ are strongly bisimilar.*

Proof sketch: By showing that

$$\mathcal{B} = \left\{((P(t), \Gamma), P(T)[\Gamma]) \mid (P(t), \Gamma) \in \hat{\mathcal{L}}_1 \wedge P(T)[\Gamma] \in \hat{\mathcal{L}}_2\right\}.$$

is a strong bisimulation relation between $\hat{\mathcal{L}}_1$ and $\hat{\mathcal{L}}_2$, the states of $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively, using structural induction on $P(t)$. ∎

One implication of Theorem 4.5.1 is the fact that we can express denotational values of configurations of LTSs obtained using COSE in terms of the denotational values calculated from states of LTSs generated using standard operational semantics, namely:

$$\mathit{traces}(\mathit{Proc}(t), \Gamma, T) = \mathit{traces}(\mathit{Proc}(T)[\Gamma])$$

and

$$\mathit{failures}(\mathit{Proc}(t), \Gamma, T) = \mathit{failures}(\mathit{Proc}(T)[\Gamma])$$

for every process syntax $Proc(t)$, instantiation T of t and environment Γ in $Env(T)$.

4.6 Relating symbolic and concrete traces

We end this chapter by defining what it means for a concrete trace to be an instantiation of a symbolic trace. We do this by using a ternary relation *generates* that links symbolic traces (Section 4.3.3), environments and concrete traces. The environments are included in the relation, since, in order to relate a symbolic trace to a concrete trace, concrete values need to be substituted for the free variables that can occur within the symbolic trace; these concrete values come from environments.

Given a process syntax $Proc(t)$ and an instantiation T of type t, we define a relation $\mathit{generates} \subseteq \mathit{SymbolicTraces}(\mathit{Proc}(T)) \times \mathit{Env}(T) \times (\Sigma(\mathit{Proc}(T)))^*$ (for convenience written using infix notation: $\sigma\ \mathit{generates}_\Gamma\ tr$, for a symbolic trace σ, an environment Γ and a concrete trace tr) utilising the following rules:

(i) $\langle\rangle$ $generates_\Gamma$ $\langle\rangle$,

(ii) σ $generates_\Gamma$ $tr \Leftrightarrow \langle\tau\rangle\hat{}\sigma$ $generates_\Gamma$ tr,

(iii) σ $generates_\Gamma$ $tr \wedge [\![cond]\!]_\Gamma \Leftrightarrow \langle cond\rangle\hat{}\sigma$ $generates_\Gamma$ tr,

(iv) $e \in Insts_\Gamma(\epsilon) \wedge \sigma$ $generates_{\Gamma \oplus Match(\epsilon,e)}$ $tr \Leftrightarrow \langle\epsilon\rangle\hat{}\sigma$ $generates_\Gamma$ $\langle e\rangle\hat{}tr$,

where if ϵ is a visible symbolic event of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ (recall that, by Remark 4.3.1, $\$^{non\text{-}t}(\epsilon) = ?^{non\text{-}t}(\epsilon) = \{\}$), then

$$Insts_\Gamma(\epsilon) = \{c.v_1 \ldots v_k \mid \forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v_i \in T \wedge \forall\, i \in !(\epsilon) \bullet v_i = \Gamma(x_i)\}$$

and

$$Match(\epsilon, c.v_1 \ldots v_k) = \{x_i \mapsto v_i \mid i \in \$^t(\epsilon) \cup ?^t(\epsilon)\}.$$

Observe that rule (ii) indicates that we essentially ignore any τ's present in the symbolic traces.

For brevity we write σ, σ' $generates_\Gamma$ tr to mean that both σ $generates_\Gamma$ tr and σ' $generates_\Gamma$ tr.

Using the above definition we can describe what it means for a (concrete) visible event to *match* a visible symbolic event. In the following definition we treat two concrete events as essentially different if they are available after different traces.

Definition 4.6.1. An event e available in $Proc(T)$ immediately after a trace tr *matches* a visible symbolic event ϵ if there exists a symbolic trace σ such that $\sigma\hat{}\langle\epsilon\rangle$ is in $SymbolicTraces(Proc(t))$ and $\sigma\hat{}\langle\epsilon\rangle$ $generates_{\{\}}$ $tr\hat{}\langle e\rangle$.

4.7 Conclusions

This chapter focused on operational semantics for CSP. The main motivation for this work was the need for (semi-)symbolic transitions graphs, which we use in Chapter 5 to deduce behaviours of instantiations of a given specification syntax, knowing certain behaviours of another instantiation.

In Section 4.3 we described Semi-Symbolic Operational Semantics (SSOS), which produces semi-symbolic LTSs (SSLTSs). An SSLTS abstracts away the details of what values of type t are used by a given specification, i.e. only the control flow is modelled. For this, separation of details of the instantiation of type t from the control flow is needed. In addition, we want specifications to avoid nondeterminism whose effect is not immediately observable. We formally captured the above, together with some technical and simplifying assumptions, using the **SeqNorm** condition. This condition does not significantly reduce expressiveness, as most useful specifications that one writes in practice satisfy it.

Although symbolic LTSs are useful in their own right, we require the ability to generate concrete transition graphs from them. This is why in Section 4.4 we defined Concrete Operational Semantics with Environments (COSE) by providing a set of

translation rules from SSLTSs to concrete LTSs. A key concept of COSE is that of an environment. Environments instantiate free variables of type t with concrete values from a given instantiation T of type t. Since SSLTSs have only the parts related to type t left in a symbolic form, one cannot simultaneously resolve all nondeterministic selections over arbitrary types. This means that the combination of SSOS and COSE is not congruent to the standard operational operational semantics presented in [Ros97, Chapter 7] (see Example 4.2.1). To overcome this issue, in Section 4.2 we defined a slightly modified standard operational semantics (while preserving denotational values for all processes that we allow). We also included some rules that we use in later chapters and which are not explicitly stated in [Ros97, Chapter 7]. In Section 4.5 we proved congruence between our standard operational semantics and the combination of SSOS and COSE.

Finally, we related symbolic and concrete traces, by instantiating symbolic events into concrete events with the use of environments in Section 4.6.

We will use the work presented in this chapter in our type reduction theory, described in Chapter 5. The symbolic representation of transition graphs will allow us to deduce a number of regularity results for processes (specifications in our case) that satisfy **SeqNorm**. Our rules for translating symbolic states into concrete configurations (see Section 4.4), together with the proof that the operational semantics obtained in such a way is congruent to the standard one (Theorem 4.5.1) will allow us to deduce important relationships between different concrete instances of a given specification. In addition, in Chapter 6 it will be beneficial for us to represent states of node processes using configurations produced by COSE.

Chapter 5

Type reduction theory

As mentioned in the Introduction, the main issue of parameterised model checking (in addition to the state explosion problem) is the unboundedness of parameters. Even if efficient model checking algorithms existed for checking finite models of a fixed size, this would tell us very little about the correctness of models for all instantiations of their parameters. One approach to deal with this problem is to find a finite bound on the size of the parameter (a threshold) such that we can guarantee that extending the type beyond this size does not influence the behaviour of the system in a non-trivial way. This is the approach Lazić used in his data independence theory [LR98]. A main assumption of data independence (see Definition 4.1.1) is that processes do not use replicated operators that are indexed over the distinguished type. However, at the centre of each implementation that we consider is a parallel composition of node processes indexed over the parameter, so Lazić's results do not apply here. In this chapter we present a similar theory, which we call a *type reduction theory*, for dealing with unbounded parameters, with the main exception being that while in data independence

$$Spec(\hat{T}) \sqsubseteq Impl(\hat{T}) \text{ implies that } \forall\, T \supseteq \hat{T} \bullet Spec(T) \sqsubseteq Impl(T),$$

where $\hat{T} = \{0\,..\,B\}$ is a small and fixed type for some non-negative integer B, our theory shows that (provided certain criteria are met)

$$Spec(\hat{T}) \sqsubseteq \phi(Impl(T)) \text{ implies that } Spec(T) \sqsubseteq Impl(T), \tag{5.1}$$

for all T such that $\hat{T} \subseteq T$, where ϕ is a B-collapsing function:

Definition 5.0.1. A *B-collapsing function* is a function $\phi : T \to \{0\,..\,B\}$ such that

- $\phi(v) = v$ for all v in $\{0\,..\,B-1\}$,
- $\phi(v) = B$ for all v in $\{B\,..\,\#T-1\}$.

Our two main results of this chapter, Theorem 5.2.15 and Theorem 5.2.24, prove the above in the traces and stable failures models, respectively. It is important to realise that, on its own, our theory does not resolve the problem of an infinite number of refinement checks needed to solve a given verification problem. This is due to the

fact that process $\phi(Impl(T))$, even though it can only communicate values of type t that are in $\phi(T)$, still depends on T. For example, if

$$Impl(t) = \left|\left|\left| i \in t \bullet c.i \rightarrow STOP\right.\right.\right.$$

and $\phi(v) = 0$ for all v, then for every T, $\phi(Impl(T))$ can only communicate $c.0$, but the number of such events that can be communicated is equal to the size of T. In practice, the theory needs to be combined with an abstraction method that can produce a process $Abstr$ such that

$$Abstr \sqsubseteq \phi(Impl(T))$$

for all sufficiently large T. Then, it is enough to test

$$Spec(\hat{T}) \sqsubseteq Abstr$$

to deduce that

$$Spec(\hat{T}) \sqsubseteq \phi(Impl(T))$$

and then use (5.1). We will present an abstraction method for constructing suitable $Abstr$ processes in Chapter 6.

A significant part of this chapter is devoted to showing how certain behaviours (either in the traces or the stable failures model) of specifications instantiated with uncollapsed types can be inferred from known behaviours of the same specification instantiated with reduced types (justifying the name of our theory).

The rest of this chapter is structured as follows. In Section 5.1 we describe a number of regularity results that are consequences of the **SeqNorm** condition (see Definition 4.1.4). These results ensure that all specification processes that we use exhibit certain clarity in their behaviour. The main part of our theory is contained in Section 5.2, which we begin with a number of conditions for processes (Section 5.2.1). We present and prove type reduction theorems for both the traces and the stable failures model in Section 5.2.2 and Section 5.2.3, respectively.

5.1 Regularity results

In this section we present a series of auxiliary results that are consequences of the **SeqNorm** condition. Our main findings can be summarised as follows:

- There is no ambiguity about what configuration a process reaches after performing a sequence of concrete events that does not end with an internal event (Lemma 5.1.12);
- There is no ambiguity which construct gives rise to a given concrete event that is available after a given trace (Corollary 5.1.15); and
- Every event available in a process syntax instantiated with a collapsed type is also an event available in the same process syntax instantiated with the uncollapsed type, and the target configurations are the same, except for the underlying type (Proposition 5.1.18).

Regularity results will play a vital role in proving the main theorems of the type reduction theory.

In Section 4.6 we defined what it means for a concrete visible event to match a visible symbolic event. In the following sections we will often need to relate concrete and symbolic visible events and syntax constructs that give rise to them. The following definition establishes such relationships formally.

Definition 5.1.1. Given a sequential process syntax $Proc(t)$, let $\alpha_1, \alpha_2, \ldots$ be the prefix constructs of $Proc(t)$ (here, two prefix constructs are regarded as different if they appear in different places in $Proc(t)$). Let T be a given instantiation of type t. Then, for every pair of a trace tr and a visible event e such that $tr \hat{} \langle e \rangle$ is a trace of $Proc(T)$, there must be at least one α_i that gives rise to e immediately after tr. We then say that α_i is *matched* by e (or that e *matches* α_i). We define visible symbolic events to match syntax constructs in an analogous way.

Example 5.1.2. Let

$$\begin{aligned} P(t) = {} & c?x{:}t \rightarrow STOP \\ & \Box \\ & c\$x{:}t \rightarrow c.x \rightarrow STOP. \end{aligned}$$

Then, for $T = \{0, 1\}$, given trace $tr = \langle\rangle$, the event $e = c.1$ matches both the constructs $c?x{:}t$ and $c\$x{:}t$, but not $c.x$ (as $c.x$ may give rise to $c.1$, but only after the trace $\langle c.1 \rangle$ and not the empty trace).

End of example.

Note that the process in the above example does not satisfy **SeqNorm**. We will show (in Corollary 5.1.15) that for processes that do satisfy **SeqNorm**, each event (after a given trace) matches a *unique* construct.

The following lemma shows that (for a process that satisfies **SeqNorm**), each visible or conditional symbolic event leads to a unique symbolic state.

Lemma 5.1.3. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Let ϵ be a visible or conditional symbolic event and suppose that*

$$Proc(t) \stackrel{\sigma_1}{\longmapsto}_s Proc_1'(t) \quad \textit{and} \quad Proc(t) \stackrel{\sigma_2}{\longmapsto}_s Proc_2'(t),$$

where $\sigma_1 = \tau^a \hat{} \langle \epsilon \rangle$ for $a \geq 0$ and $\sigma_2 = \tau^b \hat{} \langle \epsilon \rangle$ for $b \geq 0$. Then $Proc_1'(t) = Proc_2'(t)$.

A main reason for introducing the **SeqNorm** condition is to ban all nondeterminism whose effect is not immediately observable. Without this condition, such nondeterminism can come from various sources, each of which renders Lemma 5.1.3 untrue. For example, consider the process definition

$$Proc(t) = \sqcap\, x \in X \bullet a \rightarrow c.x \rightarrow STOP.$$

Then any sensible firing rule for $\sqcap$ would imply that after performing the symbolic trace $\langle a \rangle$, $Proc(t)$ could be in any of the $\#X$ states $c.x \rightarrow STOP$ (one for each value

of x), contrary to Lemma 5.1.3. Non-immediately observable nondeterminism can also be introduced by any binary choice operator whose sets of initially available events of the branches are not disjoint. Finally, conditional choices on t before prefixes in branches of binary choices can also be sources of nondeterministic behaviour within the symbolic transition graphs. For example, the process definition

$$\begin{aligned} Proc_{x,y}(t) = {} & (\text{if } x = y \text{ then } a \rightarrow STOP \text{ else } STOP) \\ & \Box \\ & \text{if } x = y \text{ then } b \rightarrow STOP \text{ else } STOP \end{aligned}$$

can do the conditional symbolic event "$x = y$'' and reach either the symbolic state $a \rightarrow STOP \ \Box$ (if $x = y$ then $b \rightarrow STOP$ else $STOP$) or the symbolic state (if $x = y$ then $a \rightarrow STOP$ else $STOP$) $\Box$ $b \rightarrow STOP$. This is why we banned all the behaviours described above by assuming that process syntaxes satisfy[1] **SeqNorm**.

Proof of Lemma 5.1.3: We prove the result by a structural induction on $Proc(t)$. The most interesting cases are those for prefix and external choice.

STOP
This case holds trivially.

Prefix
Suppose $Proc(t) = \alpha \rightarrow Proc'(t)$ for some construct $\alpha = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and some process syntax $Proc'(t)$. Clearly ϵ must be a visible symbolic event matching α, say $c\S'_1 x'_1{:}X'_1 \ldots \S'_k x'_k{:}X'_k$. We consider two different cases, corresponding to the number of nondeterministic selections over non-t types of α.

Case 1. Suppose that $\#\$^{non\text{-}t}(\alpha) = 0$.
Then, by Symbolic Prefix Rule 1 (p. 78), it must be that

$$\sigma_1 = \sigma_2 = \langle \epsilon \rangle$$

with ϵ in $Comms^{non\text{-}t}(\alpha)$ and

$$Proc'_1(t) = Proc'_2(t) = Proc'(t)[x'_i/x_i \mid i \in ?^{non\text{-}t}(\alpha)].$$

Case 2. Suppose that $\#\$^{non\text{-}t}(\alpha) > 0$.
Then, Symbolic Prefix Rule 2 (p. 78) implies that the only symbolic transitions in $Proc(t)$ are

$$Proc(t) \xrightarrow{\tau}_s \left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha) \rightarrow Proc'(t)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)]$$

[1] **SeqNorm** requires that every process syntax containing an external, internal or sliding choice has all the conditionals on t in its branches occurring after a prefix. In theory, it would be enough for this lemma if both branches of a binary choice had the sets of the initial conditional symbolic events disjoint. However, the stronger assumption is needed for various other results, so for consistency we choose to use this version of the assumption. This is only a technicality, since any process failing the more restrictive assumption can be easily rewritten in a form that satisfies it, as already noted in Remark 4.1.7.

for each function v with $\mathrm{dom}(v) = \{1 \mathinner{..} k\}$ and such that if i is in $\$^{non\text{-}t}(\alpha)$, then v_i is in X_i. We are guaranteed that

$$\#\$^{non\text{-}t}(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha)) = 0,$$

so, Symbolic Prefix Rule 1 (p. 78) implies that there are two functions v as above, say v^1 and v^2, such that for $j \in \{1, 2\}$,

$$Proc'_j(t) = (Proc'(t)[v^j_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)])[x'_i/x_i \mid i \in ?^{non\text{-}t}(\alpha)]$$

and

$$\epsilon \in Comms^{non\text{-}t}((Replace^{non\text{-}t}_{\$\mapsto !}(\alpha))[v^j_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)]).$$

Then, thanks to the definition of $Comms^{non\text{-}t}$, v^1 and v^2 are equal under domain restriction to $\$^{non\text{-}t}(\alpha)$. Therefore, $Proc'_1(t) = Proc'_2(t)$.

External choice
Suppose that $Proc(t) = P(t) \,\Box\, Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. Since $Proc(t)$ satisfies **SeqNorm**, we know that neither $P(t)$ nor $Q(t)$ contains a conditional choice on t before a prefix. Therefore ϵ cannot be a conditional symbolic event, so must be a visible symbolic event. **SeqNorm** implies that the channels of the initial visible symbolic events of $P(t)$ and $Q(t)$ are disjoint, so we have that either

$$P(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s \quad \text{or} \quad Q(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s,$$

but not both. Without loss of generality we assume the former. Then the inductive hypothesis implies that there is a unique symbolic state $P'(t)$ such that

$$P(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s P'(t).$$

Even though there may be some τ's, contributed by $Q(t)$, in the symbolic trace of $P(t)$ leading to $Proc_1(t)$, the uniqueness of $P'(t)$ implies that $Proc'_1(t) = Proc'_2(t) = P'(t)$.

Internal choice
Suppose that $Proc(t) = P(t) \sqcap Q(t)$ for some process syntax $P(t)$ and $Q(t)$. Then the only symbolic transitions from $Proc(t)$ are

$$Proc(t) \xrightarrow{\tau}_s P(t) \quad \text{and} \quad Proc(t) \xrightarrow{\tau}_s Q(t).$$

Therefore $\sigma_1 = \langle\tau\rangle\hat{}\,\rho_1$ and $\sigma_2 = \langle\tau\rangle\hat{}\,\rho_2$ for some ρ_1, $\rho_2 \in SymbolicTraces(P(t)) \cup SymbolicTraces(Q(t))$ such that $\rho_1, \rho_2 \in \{\tau\}^*\hat{}\,\langle\epsilon\rangle$. The rest is now similar to the case for external choice, above.

Sliding choice (timeout)
Similar to the case for external choice, above.

Conditional choice
Suppose that $Proc(t) = \text{if } cond \text{ then } P(t) \text{ else } Q(t)$ for some boolean condition $cond$ and some process syntaxes $P(t)$ and $Q(t)$. If $cond$ is not in $Cond$, then $cond$ immediately evaluates to $True$ or $False$ and the result is implied by the inductive hypothesis for $P(t)$ or $Q(t)$, respectively. If $cond$ is in $Cond$, then $\sigma_1 = \sigma_2 = \langle cond \rangle$ and $Proc'_1(t) = Proc'_2(t) = P(t)$, or $\sigma_1 = \sigma_2 = \langle \neg cond \rangle$ and $Proc'_1(t) = Proc'_2(t) = Q(t)$.

Binding
Suppose that $Proc(t) = X(t)$ for some process identifier X and suppose that a global environment E maps X to some process definition P (i.e. $E(X) = P$). Then the SSOS firing rule for binding implies that

$$\sigma_1 = \langle \tau \rangle^\frown \rho_1 \quad \text{and} \quad \sigma_2 = \langle \tau \rangle^\frown \rho_2$$

for some ρ_1, ρ_2 in $SymbolicTraces(P(t))$. The result is then implied by the inductive hypothesis for $P(t)$. ∎

Our next result lifts Lemma 5.1.3 to traces that are "similar" in the following sense.

Definition 5.1.4. Let σ and σ' be two symbolic traces. Then σ and σ are *non-τ equivalent*, written $\sigma \equiv_{non\text{-}\tau} \sigma'$, if their restrictions to conditional and visible symbolic events are identical, i.e. if $\sigma \setminus \{\tau\} = \sigma' \setminus \{\tau\}$.

Corollary 5.1.5. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Suppose further that*

$$Proc(t) \overset{\sigma}{\longmapsto}_s P(t) \quad \text{and} \quad Proc(t) \overset{\sigma'}{\longmapsto}_s Q(t).$$

Then if neither σ nor σ' ends with a τ and $\sigma \equiv_{non\text{-}\tau} \sigma'$, then $P(t) = Q(t)$.

Proof: By induction on the number of visible and conditional symbolic events of σ and σ' (which must be equal, since $\sigma \equiv_{non\text{-}\tau} \sigma'$), and using Lemma 5.1.3. ∎

The following lemma relates two initial visible symbolic events on the same channel.

Lemma 5.1.6. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Then, if $Proc(t) \overset{\sigma}{\longmapsto}_s \overset{\epsilon}{\longrightarrow}_s$ and $Proc(t) \overset{\sigma'}{\longmapsto}_s \overset{\epsilon'}{\longrightarrow}_s$, where $\sigma, \sigma' \in (Cond \cup \{\tau\})^*$ are such that $\sigma \equiv_{non\text{-}\tau} \sigma'$ and $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and $\epsilon' = c'\S'_1 x'_1{:}X'_1 \ldots \S'_l x'_l{:}X'_l$ are visible symbolic events, then*

(i) if the channels of ϵ and ϵ' are identical (i.e. $c = c'$), then the parts of ϵ and ϵ' involving type t are equal, i.e.

$$\$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon) = \$^t(\epsilon') \cup ?^t(\epsilon') \cup !^t(\epsilon')$$

and

$$\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon) \bullet \S_i = \S'_i \wedge x_i = x'_i \wedge X_i = X'_i,$$

(ii) if $Insts_\Gamma(\epsilon) \cap Insts_\Gamma(\epsilon') \neq \{\}$ *for some environment* Γ*, then* $\epsilon = \epsilon'$*.*

Proof: Firstly, observe that if $Insts_\Gamma(\epsilon) \cap Insts_\Gamma(\epsilon') \neq \{\}$, then the channels of ϵ and ϵ' must be the same. Hence in both cases $c = c'$. Since every channel has a fixed structure of the communication along it, the number of components of ϵ and ϵ' must be identical, i.e. $k = l$. We now prove both clauses using a structural induction on $Proc(t)$. The most interesting case is that for prefix.

STOP
The result holds trivially in this case.

Prefix
Suppose that $Proc(t) = \alpha \rightarrow Proc'(t)$ for some construct α of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and some process syntax $Proc'(t)$. We perform a case analysis on the number of nondeterministic selections over non-t types of α.

Case 1. Suppose that $\#\$^{non\text{-}t}(\alpha) = 0$.
Then, by Symbolic Prefix Rule 1 (p. 78) (observe that the other symbolic firing rules are not applicable in this case), it must be that

$$\sigma = \sigma' = \langle\rangle$$

and that both ϵ and ϵ' are in $Comms^{non\text{-}t}(\alpha)$. By the definition of $Comms^{non\text{-}t}$ (p. 78), ϵ and ϵ' may differ only in the values of deterministic inputs of non-t types of α. Hence, clause (i) of the lemma holds. To prove clause (ii), we let $c.v_1 \ldots v_k$ be a common member of $Insts_\Gamma(\epsilon)$ and $Insts_\Gamma(\epsilon')$. Then the definition of $Insts_\Gamma$ (p. 88) implies that

$$\forall\, i \in !(\epsilon) \bullet \S_i = \S'_i = ! \wedge x_i = x'_i \wedge v_i = \Gamma(x_i) \wedge X_i = X'_i = null. \tag{5.2}$$

The definition of $Comms^{non\text{-}t}$ implies that $?^{non\text{-}t}(\alpha) \subseteq !(\epsilon)$, so

$$\forall\, i \in ?^{non\text{-}t}(\alpha) \bullet \S_i = \S'_i = ! \wedge x_i = x'_i \wedge X_i = X'_i = null.$$

This, combined with clause (i) of the lemma, (5.2) and the fact that $\$^{non\text{-}t}(\alpha) = 0$, implies that $\epsilon = \epsilon'$.

Case 2. Suppose that $\#\$^{non\text{-}t}(\alpha) > 0$.
Then, by Symbolic Prefix Rule 2 (p. 78) (observe that the other symbolic firing rules are not applicable in this case), the only transitions in $Proc(t)$ are

$$\begin{aligned} Proc(t) \xrightarrow{\tau}_s & \left(Replace^{non\text{-}t}_{\$\mapsto!}(\alpha) \rightarrow Proc'(t)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)] \\ &= Replace^{non\text{-}t}_{\$\mapsto!}(\alpha)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)] \\ &\quad \rightarrow Proc'(t)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)] \end{aligned}$$

for functions v such that $\mathrm{dom}(v) = \$^{non\text{-}t}(\alpha)$, and if i is in $\$^{non\text{-}t}(\alpha)$, then v_i is in X_i. Clearly,

$$\#\$^{non\text{-}t}\left(Replace^{non\text{-}t}_{\$\mapsto!}(\alpha)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)]\right) = 0$$

for every such function v, so Symbolic Prefix Rule 1 (p. 78) implies that

$$\epsilon \in \mathit{Comms}^{\text{non-}t}\left(\left(\mathit{Replace}^{\text{non-}t}_{\$\mapsto !}(\alpha)\right)[v_i/x_i \mid i \in \$^{\text{non-}t}(\alpha)]\right),$$
$$\epsilon' \in \mathit{Comms}^{\text{non-}t}\left(\left(\mathit{Replace}^{\text{non-}t}_{\$\mapsto !}(\alpha)\right)[v'_i/x_i \mid i \in \$^{\text{non-}t}(\alpha)]\right)$$

for some functions v and v' such that $\mathrm{dom}(v) = \mathrm{dom}(v') = \$^{\text{non-}t}(\alpha)$ and if i is in $\$^{\text{non-}t}(\alpha)$, then v_i and v'_i are in X_i. The definition of $\mathit{Comms}^{\text{non-}t}$ (p. 78) implies that the parts of ϵ and ϵ' that involve type t are identical, which proves clause (i) of the lemma. To prove clause (ii), we let $c.v_1 \ldots v_k$ be a common member of $\mathit{Insts}_\Gamma(\epsilon)$ and $\mathit{Insts}_\Gamma(\epsilon')$. Then, the definition of Insts_Γ (p. 88) implies that

$$\forall\, i \in !(\epsilon) \bullet \S_i = \S'_i = ! \wedge x_i = x'_i \wedge v_i = \Gamma(x_i) \wedge X_i = X'_i = \mathit{null}. \tag{5.3}$$

However, the definition of $\mathit{Comms}^{\text{non-}t}$ implies that

$$?^{\text{non-}t}(\alpha) = ?^{\text{non-}t}\left(\left(\mathit{Replace}^{\text{non-}t}_{\$\mapsto !}(\alpha)\right)[v_i/x_i \mid i \in \$^{\text{non-}t}(\alpha)]\right) \subseteq !(\epsilon),$$
$$\$^{\text{non-}t}(\alpha) \subseteq !^{\text{non-}t}\left(\left(\mathit{Replace}^{\text{non-}t}_{\$\mapsto !}(\alpha)\right)[v_i/x_i \mid i \in \$^{\text{non-}t}(\alpha)]\right) \subseteq !(\epsilon).$$

Hence,

$$\forall\, i \in \$^{\text{non-}t}(\alpha) \cup ?^{\text{non-}t}(\alpha) \bullet \S_i = \S'_i \wedge x_i = x'_i \wedge X_i = X'_i.$$

This, combined with clause (i) of the lemma and (5.3), implies that $\epsilon = \epsilon'$.

External choice
Suppose that $\mathit{Proc}(t) = P(t) \,\Box\, Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. Since $\mathit{Proc}(t)$ satisfies **SeqNorm**, we know that neither $P(t)$ nor $Q(t)$ contains a conditional choice on t before a prefix. Hence $\sigma, \sigma \in \{\tau\}^*$. Also, **SeqNorm** implies that the channels of the initial visible symbolic events $P(t)$ and $Q(t)$ are disjoint and we know that the channels of ϵ and ϵ' are the same, so the SSOS firing rules for $\Box$ (p. 79) imply that either

$$P(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s \quad \text{and} \quad P(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon'}_s$$

or

$$Q(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s \quad \text{and} \quad Q(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon'}_s,$$

but not both. Then, the inductive hypothesis for $P(t)$ and $Q(t)$ implies the result in both cases.

Internal choice
Suppose that $\mathit{Proc}(t) = P(t) \sqcap Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. Since $\mathit{Proc}(t)$ satisfies **SeqNorm**, we know that neither $P(t)$ nor $Q(t)$ contains a conditional choice on t before a prefix. Hence $\sigma, \sigma \in \{\tau\}^*$. The SSOS firing rules for $\sqcap$ (p. 79) imply that the only symbolic transitions from $\mathit{Proc}(t)$ are

$$\mathit{Proc}(t) \xrightarrow{\tau}_s P(t) \quad \text{and} \quad \mathit{Proc}(t) \xrightarrow{\tau}_s Q(t).$$

Therefore $\sigma = \langle\tau\rangle^\frown\rho$ and $\sigma' = \langle\tau\rangle^\frown\rho'$ for some $\rho, \rho' \in \{\tau\}^*$. The rest is now similar to the external choice case, above.

Sliding choice (timeout)
Similar to the external choice case, above.

Conditional choice
Suppose that $Proc(t) = \text{if } cond \text{ then } P(t) \text{ else } Q(t)$ for some boolean condition *cond* and some process syntaxes $P(t)$ and $Q(t)$. If *cond* is not in *Cond*, then *cond* immediately evaluates to either *True* or *False* and the result is implied by the inductive hypothesis for $P(t)$ or $Q(t)$, respectively. So suppose that *cond* is in *Cond*. Then, the SSOS firing rules for conditional choice (p. 80) imply that the only symbolic transitions in $Proc(t)$ are

$$Proc(t) \xrightarrow{cond}_s P(t) \quad \text{and} \quad Proc(t) \xrightarrow{\neg cond}_s Q(t).$$

The fact that $\sigma \equiv_{non\text{-}\tau} \sigma'$ implies that either

$$\sigma = \langle cond\rangle^\frown\rho \quad \text{and} \quad \sigma' = \langle cond\rangle^\frown\rho'$$

or

$$\sigma = \langle \neg cond\rangle^\frown\rho \quad \text{and} \quad \sigma' = \langle \neg cond\rangle^\frown\rho'$$

for some $\rho, \rho' \in (Cond \cup \{\tau\})^*$ such that $\rho \equiv_{non\text{-}\tau} \rho'$. Without loss of generality we assume the former. Then

$$P(t) \stackrel{\rho}{\longmapsto}_s \xrightarrow{\epsilon}_s \quad \text{and} \quad P(t) \stackrel{\rho'}{\longmapsto}_s \xrightarrow{\epsilon'}_s .$$

The result is now implied by the inductive hypothesis for $P(t)$.

Binding
Suppose that $Proc(t) = X(t)$ for some process identifier X and suppose that the global environment E maps X to some process definition P (i.e. $E(X) = P$). Then the SSOS firing rule for binding (p. 80) implies that

$$\sigma_1 = \langle\tau\rangle^\frown\rho_1 \quad \text{and} \quad \sigma_2 = \langle\tau\rangle^\frown\rho_2$$

for some $\rho_1, \rho_2 \in SymbolicTraces(P(t)) \cap (Cond \cup \{\tau\})^*$ such that $\rho_1 \equiv_{non\text{-}\tau} \rho_2$. Then, the result is implied by the inductive hypothesis for $P(t)$. ∎

Definition 5.1.7. Let $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and $\epsilon' = c'\S'_1 x'_1{:}X'_1 \ldots \S'_k x'_k{:}X'_k$ be two visible symbolic events. Then ϵ and ϵ' are *non-t equivalent*, written $\epsilon \equiv_{non\text{-}t} \epsilon'$, if the channels and non-t parts of ϵ and ϵ are identical, i.e. if $c = c'$ and

$$\forall\, i \in \$^{non\text{-}t}(\epsilon) \cup ?^{non\text{-}t}(\epsilon) \cup !^{non\text{-}t}(\epsilon) \cup \$^{non\text{-}t}(\epsilon') \cup ?^{non\text{-}t}(\epsilon') \cup !^{non\text{-}t}(\epsilon') \bullet \\ \S_i = \S'_i \wedge x_i = x'_i \wedge X_i = X'_i.$$

We lift $\equiv_{non\text{-}t}$ to sequences of visible symbolic events using pointwise application. Finally, we define two symbolic traces, σ and σ' to be non-t equivalent, written $\sigma \equiv_{non\text{-}t} \sigma'$ if the restrictions of σ and σ' to visible symbolic events are non-t equivalent.

The following lemma shows that symbolic traces that:

- do not contain conditional symbolic events,
- do not end in a τ, and
- are non-t equivalent,

are unique up to τ's, and lead to unique symbolic states.

Lemma 5.1.8. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Suppose further that*

$$Proc(t) \stackrel{\sigma}{\longmapsto}_s P(t) \quad \textit{and} \quad Proc(t) \stackrel{\sigma'}{\longmapsto}_s P'(t),$$

where $\sigma, \sigma \in SymbolicTraces(Proc(t)) \cap (Visible \cup \{\tau\})^*$ do not end with a τ and $\sigma \equiv_{non\text{-}t} \sigma'$. Then $\sigma \equiv_{non\text{-}\tau} \sigma'$ and $P(t) = P'(t)$.

Proof: We prove the result by an induction on the number of visible events in σ and σ' (which must be equal).

Base case. Suppose that $\sigma = \sigma' = \langle\rangle$. Then, clearly $\sigma \equiv_{non\text{-}\tau} \sigma'$ and $P(t) = P'(t) = Proc(t)$.

Inductive case. Suppose that $\sigma = \sigma_0 \hat{} \tau^a \hat{} \langle\epsilon\rangle$ and $\sigma' = \sigma'_0 \hat{} \tau^b \hat{} \langle\epsilon'\rangle$ for some symbolic traces σ_0 and σ'_0 that do not end with a τ, integers $a, b \geq 0$ and visible symbolic events ϵ and ϵ'. Then

$$Proc(t) \stackrel{\sigma_0}{\longmapsto}_s P_0(t) \stackrel{\tau^a \hat{} \langle\epsilon\rangle}{\longmapsto}_s P(t) \quad \text{and} \quad Proc(t) \stackrel{\sigma'_0}{\longmapsto}_s P'_0(t) \stackrel{\tau^b \hat{} \langle\epsilon'\rangle}{\longmapsto}_s P'(t)$$

for some symbolic states $P_0(t)$ and $P'_0(t)$. Since $\sigma \equiv_{non\text{-}t} \sigma'$, we have that $\sigma_0 \equiv_{non\text{-}t} \sigma'_0$ and $\epsilon \equiv_{non\text{-}t} \epsilon'$. Therefore, by the inductive hypothesis $\sigma_0 \equiv_{non\text{-}\tau} \sigma'_0$ and $P_0(t) = P'_0(t)$. In addition, by Lemma 5.1.6, the parts of ϵ and ϵ' involving type t must be identical. Hence, $\epsilon = \epsilon'$. This, combined with the fact that $\sigma_0 \equiv_{non\text{-}\tau} \sigma'_0$, implies that $\sigma \equiv_{non\text{-}\tau} \sigma'$. Finally, Lemma 5.1.3 implies that $P(t) = P'(t)$. ∎

The following lemma shows that if a process can perform a conditional event initially (after only τ's), then *all* its initial events (after τ's) must be that conditional or its negation.

Lemma 5.1.9. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Then, if*

$$Proc(t) \stackrel{\sigma}{\longmapsto}_s \stackrel{cond}{\longrightarrow}_s \quad \textit{and} \quad Proc(t) \stackrel{\sigma'}{\longmapsto}_s \stackrel{\alpha}{\longrightarrow}_s,$$

where $\sigma, \sigma' \in \{\tau\}^*$, $cond \in Cond$ and $\alpha \neq \tau$, then $\alpha \in \{cond, \neg cond\}$.

Proof: Since $Proc(t)$ satisfies **SeqNorm**, we know that there are no conditionals before prefixes in branches of external, internal and sliding choices. Hence one of the following must hold:

(i) $Proc(t)$ is a conditional choice on t, where the boolean condition is equal to $cond$ or $\neg cond$;

(ii) $Proc(t)$ is a process identifier bound by the global environment E to a conditional choice like that in clause (i) or (iii); or

(iii) $Proc(t)$ is a conditional choice whose boolean condition immediately evaluates to *True* or *False* and appropriate branch is a process syntax as in clause (i) or (ii).

This means that the only transitions available in $Proc(t)$ are $Proc(t) \xrightarrow{\tau}{}^*_s \xrightarrow{cond}_s$ and $Proc(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\neg cond}_s$. ∎

The following lemma shows that if two symbolic traces each contain a single visible symbolic event, and each trace can be instantiated in the same environment, then they contain the same conditional events before the visible event, essentially because those conditionals must evaluate to *True* in the initial environment.

Lemma 5.1.10. *Suppose that* $Proc(t)$ *satisfies* ***SeqNorm****. Let* $\sigma \hat{} \langle \epsilon \rangle, \sigma' \hat{} \langle \epsilon' \rangle \in SymbolicTraces(Proc(t))$ *be such that* $\sigma, \sigma' \in (Cond \cup \{\tau\})^*$, $\epsilon, \epsilon' \in Visible$, $\sigma \hat{} \langle \epsilon \rangle$ *generates*$_\Gamma$ $\langle e \rangle$ *and* $\sigma' \hat{} \langle \epsilon' \rangle$ *generates*$_\Gamma$ $\langle e' \rangle$ *for some environment* Γ *and some visible events* e *and* e'*. Then* $\sigma \equiv_{non\text{-}\tau} \sigma'$.

Proof: Let $\kappa : SymbolicTraces(Proc(t)) \to \mathbb{N}$ be a function that returns the number of conditional symbolic events within a given symbolic trace. We prove the result using an induction on $\kappa(\sigma)$.

Base case. Suppose that $\kappa(\sigma) = 0$. Then $\sigma \in \{\tau\}^*$, so $Proc(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s$. Suppose $\kappa(\sigma') > 0$. Then $\sigma' = \tau^a \hat{} \langle cond \rangle \hat{} \rho$ for some $a \geq 0$, some $cond \in Cond$ and some symbolic trace $\rho \in (Cond \cup \{\tau\})^*$. Therefore $Proc(t) \xrightarrow{\tau}{}^*_s \xrightarrow{cond}_s$. By Lemma 5.1.9, $\epsilon \in \{cond, \neg cond\}$. This is a contradiction as $\epsilon \in Visible$. Therefore $\kappa(\sigma') = 0$, which means that $\sigma' \in \{\tau\}^*$. Hence $\sigma \equiv_{non\text{-}\tau} \sigma'$.

Inductive case. Suppose that the result holds for all process syntaxes and all their symbolic traces with exactly k conditional symbolic events. Consider $\kappa(\sigma) = k + 1$. Then $\sigma = \tau^a \hat{} \langle cond \rangle \hat{} \rho$ for some $a \geq 0$, some $cond \in Cond$ and some $\rho \in (Cond \cup \{\tau\})^*$ with $\kappa(\rho) = k$. Arguing similarly as in the base case, $\kappa(\sigma') > 0$. Therefore, $\sigma' = \tau^b \hat{} \langle cond' \rangle \hat{} \rho'$ for some $b \geq 0$, some $cond' \in Cond$ and some $\rho' \in (Cond \cup \{\tau\})^*$. By Lemma 5.1.9, $cond' \in \{cond, \neg cond\}$. Since $\sigma \hat{} \langle \epsilon \rangle$ *generates*$_\Gamma$ $\langle e \rangle$ and $\sigma' \hat{} \langle \epsilon' \rangle$ *generates*$_\Gamma$ $\langle e' \rangle$, $cond$ and $cond'$ must both evaluate to *True* within Γ, because there are no visible symbolic events within σ and σ' before $cond$ and $cond'$, respectively, that could modify the environment Γ. So it must be that $cond = cond'$. By Lemma 5.1.3, there is a unique state $P(t)$ such that $Proc(t) \xrightarrow{\tau}{}^*_s \xrightarrow{cond}_s P(t)$. Hence, $\rho \hat{} \langle \epsilon \rangle$ and $\rho' \hat{} \langle \epsilon' \rangle$ are both symbolic traces of $P(t)$

such that $\rho\hat{}\langle\epsilon\rangle$ *generates*$_\Gamma$ $\langle e\rangle$ and $\rho'\hat{}\langle\epsilon'\rangle$ *generates*$_\Gamma$ $\langle e'\rangle$. In addition, $\kappa(\rho) = k$. Therefore, by the inductive hypothesis, $\rho \equiv_{non\text{-}\tau} \rho'$, which implies that $\sigma \equiv_{non\text{-}\tau} \sigma'$. ∎

The following lemma shows that each concrete visible transition leads to a unique process syntax and environment.

Lemma 5.1.11. *Suppose that Proc(t) satisfies* ***SeqNorm****. Suppose further that*

$$(Proc(t), \Gamma_{init}) \xrightarrow{\tau}{}^* \xrightarrow{e} (P(t), \Gamma)$$

and

$$(Proc(t), \Gamma_{init}) \xrightarrow{\tau}{}^* \xrightarrow{e} (Q(t), \Gamma'),$$

where e is a visible event. Then $P(t) = Q(t)$ *and* $\Gamma = \Gamma'$.

Proof: By the translation rules of COSE (see Section 4.4.1), there must exist symbolic traces σ, σ' and visible symbolic events $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and $\epsilon' = c'\S'_1 x'_1{:}X'_1 \ldots \S'_l x'_l{:}X'_l$ such that

- $Proc(t) \xrightarrow{\sigma}{}_s \xrightarrow{\epsilon}{}_s P(t)$ and $Proc(t) \xrightarrow{\sigma'}{}_s \xrightarrow{\epsilon'}{}_s Q(t)$, and
- $\sigma\hat{}\langle\epsilon\rangle$ *generates*$_{\Gamma_{init}}$ $\langle e\rangle$ and $\sigma'\hat{}\langle\epsilon'\rangle$ *generates*$_{\Gamma_{init}}$ $\langle e\rangle$.

Then it must be that $\sigma, \sigma' \in (Cond \cup \{\tau\})^*$. So, by Lemma 5.1.10, $\sigma \equiv_{non\text{-}\tau} \sigma'$. From the definition of the *generates* relation (p. 87) we have that $e \in Insts_{\Gamma_{init}}(\epsilon) \cap Insts_{\Gamma_{init}}(\epsilon')$, so Lemma 5.1.6 implies that $\epsilon = \epsilon'$. Hence, $\sigma\hat{}\langle\epsilon\rangle \equiv_{non\text{-}\tau} \sigma'\hat{}\langle\epsilon'\rangle$ and so we can infer, using Corollary 5.1.5, that $P(t) = Q(t)$. In addition, the translation rules of COSE (see Section 4.4.1) imply that if $e = c.v_1 \ldots v_k$, then

$$\Gamma = \Gamma_{init} \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon) \cup ?^t(\epsilon)\}$$

and

$$\Gamma' = \Gamma_{init} \oplus \{x'_i \mapsto v_i \mid i \in \$^t(\epsilon') \cup ?^t(\epsilon')\}.$$

However, $\epsilon = \epsilon'$, so $\Gamma = \Gamma'$, as required. ∎

An important property of normality is the lack of ambiguity about what state a process reaches after performing a sequence of concrete events that does not end with a τ. The following lemma establishes this formally.

Lemma 5.1.12. *Suppose that Proc(t) satisfies* ***SeqNorm****. Suppose further that*

$$(Proc(t), \Gamma_{init}) \xrightarrow{s} (P(t), \Gamma) \quad \textit{and} \quad (Proc(t), \Gamma_{init}) \xrightarrow{s'} (Q(t), \Gamma'),$$

where s, s' *do not end with a* τ *and* $s \setminus \tau = s' \setminus \tau$. *Then* $P(t) = Q(t)$ *and* $\Gamma = \Gamma'$.

Proof: By a straightforward induction on the length of $s \setminus \{\tau\}$ and using Lemma 5.1.11. ∎

The following, important result says that **SeqNorm** implies that every two symbolic traces that give rise to the same concrete trace, and are either the empty symbolic trace or both end in a visible symbolic event, are identical up to internal actions.

Proposition 5.1.13. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Then, if $\sigma, \sigma' \in SymbolicTraces(Proc(t))$ are such that σ $generates_\Gamma$ tr and σ' $generates_\Gamma$ tr, and either $\sigma = \sigma' = \langle\rangle$ or both σ and σ' end in a visible symbolic event, then $\sigma \equiv_{non\text{-}\tau} \sigma'$.*

Proof: We prove the result by an induction on the length of tr.

Base case. Suppose that $tr = \langle\rangle$. Then σ $generates_\Gamma$ $\langle\rangle$ and σ' $generates_\Gamma$ $\langle\rangle$. Therefore, by the definition of the *generates* relation (p. 87), σ and σ' cannot contain visible symbolic events. Hence, by the assumptions of this proposition, $\sigma = \sigma' = \langle\rangle$, which means that $\sigma \equiv_{non\text{-}\tau} \sigma'$.

Inductive case. Suppose that the result holds for all traces of length k. Consider a trace tr_{k+1} of length $k+1$. Then there exists a trace tr_k of length k and a visible event e such that $tr_{k+1} = tr_k \hat{} \langle e \rangle$. There must also exist symbolic traces $\sigma_1, \sigma_1', \sigma_2$, and σ_2' such that

- either $\sigma_1 = \sigma_1' = \langle\rangle$ or both σ_1 and σ_1' end in a visible symbolic event,
- $\sigma = \sigma_1 \hat{} \sigma_2$ and $\sigma' = \sigma_1' \hat{} \sigma_2'$, and
- σ_1 $generates_\Gamma$ tr_k and σ_1' $generates_\Gamma$ tr_k.

Then, by the inductive hypothesis, $\sigma_1 \equiv_{non\text{-}\tau} \sigma_1'$. Hence, if $P(t)$ and $Q(t)$ are symbolic states such that $Proc(t) \stackrel{\sigma_1}{\longmapsto}_s P(t)$ and $Proc(t) \stackrel{\sigma_1'}{\longmapsto}_s Q(t)$, then, by Corollary 5.1.5, $P(t) = Q(t)$. Let s be a sequence of events such that $s \setminus \tau = tr_k$ and $(Proc(t), \Gamma) \stackrel{s}{\longmapsto} (P(t), \Gamma)$ for some unique environment Γ (uniqueness being guaranteed by Lemma 5.1.12). We now have that σ_2 $generates_\Gamma$ $\langle e \rangle$ and σ_2' $generates_\Gamma$ $\langle e \rangle$. Hence, $\sigma_2, \sigma_2' \neq \langle\rangle$. Therefore, both σ_2 and σ_2' must end in a visible symbolic event (since they are suffixes of σ and σ'). So, $\sigma_2 = \rho \hat{} \langle \epsilon \rangle$ and $\sigma_2' = \rho' \hat{} \langle \epsilon' \rangle$ for some symbolic traces $\rho, \rho' \in (Cond \cup \{\tau\})^*$ and some visible symbolic events ϵ and ϵ'. Hence, by Lemma 5.1.10, $\rho \equiv_{non\text{-}\tau} \rho'$. Also from the definition of *generates* we have that $e \in Insts_\Gamma(\epsilon) \cap Insts_\Gamma(\epsilon')$, so, by Lemma 5.1.6, $\epsilon = \epsilon'$. Therefore, $\sigma_2 \equiv_{non\text{-}\tau} \sigma_2'$, and hence $\sigma \equiv_{non\text{-}\tau} \sigma'$. ∎

Observe that the part "either $\sigma = \sigma' = \langle\rangle$ or both σ and σ' end in a visible symbolic event" is a strictly necessary assumption of Proposition 5.1.13, as illustrated by the following example.

Example 5.1.14. Let

$$Proc(t) = in?x{:}t?y{:}t \rightarrow \text{if } x = y \text{ then } a \rightarrow STOP \text{ else } STOP.$$

Let $\sigma = \langle in?x{:}t?y{:}t \rangle$ and $\sigma' = \langle in?x{:}t?y{:}t, x = y \rangle$. Then σ and σ' are both symbolic traces of $Proc(t)$, and σ, σ' *generates*$_{\{\}}$ $\langle in.0.0 \rangle$, but $\sigma \not\equiv_{non\text{-}\tau} \sigma'$.

End of example.

The following corollary is an immediate consequence of Proposition 5.1.13 and says that, for a process that satisfies **SeqNorm**, for each visible event that is performed after a given trace, there is never any ambiguity what construct this event matches.

Corollary 5.1.15. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Then, if $tr \,\hat{}\, \langle e \rangle$ is a trace of $(Proc(t), \Gamma, T)$ for some type T and some environment Γ, and e matches constructs α and α', then $\alpha = \alpha'$.*

Proof: Suppose for contradiction that $\alpha \neq \alpha'$. Definition 5.1.1 implies that α and α' give rise to e immediately after tr. By Proposition 5.1.13, if $\sigma \,\hat{}\, \langle \epsilon \rangle$ and $\sigma' \,\hat{}\, \langle \epsilon' \rangle$ are both symbolic traces of $Proc(t)$ such that $\sigma \,\hat{}\, \langle \epsilon \rangle, \sigma' \,\hat{}\, \langle \epsilon' \rangle$ *generates*$_{\{\}}$ $tr \,\hat{}\, \langle e \rangle$ and where ϵ and ϵ' are visible, then $\sigma \equiv_{non\text{-}\tau} \sigma'$ and $\epsilon = \epsilon'$. We now have that ϵ matches both α and α'. Then, the firing rules of SSOS (see Section 4.3.2) imply that α and α' must be constructs in different branches of an external, internal or sliding choice. Since both of these constructs give rise to the same concrete event, e, their channels must be identical. Hence, $Proc(t)$ contains a binary choice with branches sharing a common channel name. This contradicts the fact that $Proc(t)$ satisfies **SeqNorm**, so it must be that $\alpha = \alpha'$. ∎

The following proposition can be viewed as a dual of Proposition 5.1.13. It says that any two symbolic traces, identical up to internal actions, generate precisely the same concrete traces.

Proposition 5.1.16. *Suppose that σ generates$_\Gamma$ tr and $\sigma \equiv_{non\text{-}\tau} \rho$. Then ρ generates$_\Gamma$ tr.*

Proof: By going through the clauses of the definition of the *generates* relation (p. 87). Recall that clause (ii) of that definition says that all τ's in symbolic traces are essentially ignored when they generate concrete traces and observe that the restrictions of σ and ρ to visible and conditional symbolic events are identical. ∎

The following lemma compares corresponding constructs in two processes, one of which refines the other.

Lemma 5.1.17. *Suppose that $P(t)$ and $Q(t)$ satisfy **SeqNorm**. Let T be an instantiation of type t. Suppose that $(Q(t), \Gamma_{init}) \sqsubseteq_T (P(t), \Gamma_{init})$. Then for all visible symbolic events ϵ, ϵ', symbolic traces σ, σ' and traces tr such that*

(i) $tr \,\hat{}\, \langle e \rangle \in traces(P(t), \Gamma_{init}, T)$,

(ii) $\sigma \,\hat{}\, \langle \epsilon \rangle \in SymbolicTraces(P(t))$ generates$_{\Gamma_{init}}$ $tr \,\hat{}\, \langle e \rangle$, and

(iii) $\sigma' \,\hat{}\, \langle \epsilon' \rangle \in SymbolicTraces(Q(t))$ generates$_{\Gamma_{init}}$ $tr \,\hat{}\, \langle e \rangle$,

we have that $!(\epsilon') \subseteq !(\epsilon)$.

Proof: Suppose for a contradiction that there is some $j \in !(\epsilon') \setminus !(\epsilon)$. Then $j \in \$^t(\epsilon) \cup ?^t(\epsilon)$ (since $\$^{non\text{-}t}(\epsilon) = ?^{non\text{-}t}(\epsilon) = \{\}$ by Remark 4.3.1). This means that the j-th variable or value of ϵ' is of type t, so $j \in !^t(\epsilon')$. Let $e = c.v_1 \ldots v_k$ and let $e' = c.v_1 \ldots v_{j-1}.v'_j.v_{j+1} \ldots v_k$, where $v'_j \in T \setminus \{v_j\}$. By Remark 4.1.3, if a process can perform a given event, then it can also perform every other event that differs only in the values of inputs. Therefore, $tr\hat{}\langle e'\rangle \in traces(P(t), \Gamma_{init}, T)$, i.e. e' matches ϵ.

Since $(Q(t), \Gamma_{init}, T) \sqsubseteq_{\mathrm{T}} (P(t), \Gamma_{init}, T)$, we must have that $tr\hat{}\langle e'\rangle \in traces(Q(t), \Gamma_{init})$. Clause (iii), combined with the fact that v'_j is an output for $Q(t)$, different from v_j, implies that $\sigma'\hat{}\langle\epsilon'\rangle$ cannot generate $tr\hat{}\langle e'\rangle$ within Γ_{init}. So let $\rho\hat{}\langle\epsilon''\rangle \in SymbolicTraces(Q(t))$ be such that $\rho\hat{}\langle\epsilon''\rangle$ $generates_{\Gamma_{init}}$ $tr\hat{}\langle e'\rangle$. Also, let $\sigma' = \sigma_1\hat{}\sigma_2$ and $\rho = \rho_1\hat{}\rho_2$, where σ_1 and ρ_1 are either both the empty symbolic trace or both end with visible symbolic events, and $\sigma_2, \rho_2 \in (Cond \cup \{\tau\})^*$. Then, clause (iii) implies that σ_1 $generates_{\Gamma_{init}}$ tr and ρ_1 $generates_{\Gamma_{init}}$ tr, so by Proposition 5.1.13, $\sigma_1 \equiv_{non\text{-}\tau} \rho_1$. Hence, if $Q'(t)$ and $Q''(t)$ are symbolic states such that $Q(t) \overset{\sigma_1}{\longmapsto}_s Q'(t)$ and $Q(t) \overset{\rho_1}{\longmapsto}_s Q''(t)$, then, thanks to Corollary 5.1.5, we have that $Q'(t) = Q''(t)$.

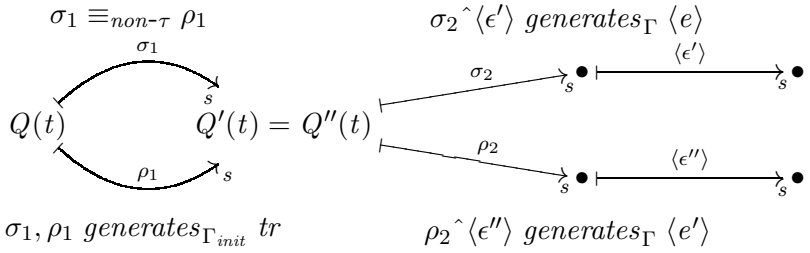


Figure 5.1: Illustration of the proof of Lemma 5.1.17.

Let Γ be an environment reached after tr, i.e. such that $(Q(t), \Gamma_{init}, T) \overset{s}{\longmapsto} (Q'(t), \Gamma, T)$ for some s such that $s \setminus \tau = tr$; by Lemma 5.1.12, Γ is unique. Then, we have that

- $\sigma_2\hat{}\langle\epsilon'\rangle, \rho_2\hat{}\langle\epsilon''\rangle \in SymbolicTraces(Q'(t))$,
- $\langle e\rangle, \langle e'\rangle \in traces(Q'(t), \Gamma, T)$,
- $\sigma_2\hat{}\langle\epsilon'\rangle$ $generates_\Gamma$ $\langle e\rangle$, and
- $\rho_2\hat{}\langle\epsilon''\rangle$ $generates_\Gamma$ $\langle e'\rangle$.

So, by Lemma 5.1.10, $\sigma_2 \equiv_{non\text{-}\tau} \rho_2$. Let $\epsilon' = c\S'_1x'_1{:}X'_1 \ldots \S'_kx'_k{:}X'_k$ and $\epsilon'' = c\S''_1x''_1{:}X''_1 \ldots \S''_kx''_k{:}X''_k$. Then, since the channels of ϵ' and ϵ'' are the same, Lemma 5.1.6 implies that $!^t(\epsilon') = !^t(\epsilon'')$ and $x'_j = x''_j$. Since $\sigma_2\hat{}\langle\epsilon'\rangle$ $generates_\Gamma$ $\langle e\rangle = \langle c.v_1 \ldots v_k\rangle$ and $\rho_2\hat{}\langle\epsilon''\rangle$ $generates_\Gamma$ $\langle e'\rangle = \langle c.v_1 \ldots v_{j-1}.v'_j.v_{j+1} \ldots v_k\rangle$ and $j \in !^t(\epsilon'')$ (as $j \in !^t(\epsilon')$), we have that $x'_j = v_j$ and $x''_j = v'_j$. Hence $v_j = v'_j$. This is a contradiction, so $!(\epsilon') \subseteq !(\epsilon)$. ∎

The following proposition and its corollary form our final consequence of **SeqNorm**. The proposition says that every event available in a process syntax instantiated with some type is also an event available in the same process syntax instantiated with a larger type. In addition, the target configurations are the same (except for the underlying type). Corollary 5.1.19 then extends this observation to traces.

Proposition 5.1.18. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Let T and $\hat{T}$ be instantiations of type t such that $\hat{T} \subseteq T$ and let Γ be an environment in $Env(T)$. Then, if*

$$(Proc(t), \Gamma, \hat{T}) \xrightarrow{a} (Proc'(t), \Gamma', \hat{T}) \tag{5.4}$$

for some symbolic state $Proc'(t)$ and some environment Γ', then

$$(Proc(t), \Gamma, T) \xrightarrow{a} (Proc'(t), \Gamma', T).$$

Proof: Since $\hat{T}$ is a subset of T, we have that $\Gamma, \Gamma' \in Env(\hat{T})$ implies $\Gamma, \Gamma' \in Env(T)$, since every partial function from Var to $\hat{T}$ is also a partial function from Var to T. We now prove the result using an induction on n, the number of times Translation Rule 4 of COSE (p. 86) had to be applied in order to obtain the transition in (5.4).

Base case. Suppose that $n = 0$. We separately consider the cases for a being τ or visible.

Case 1. Suppose that $a = \tau$.
Then, the translation rules of COSE (see Section 4.4.1) imply that the transition in (5.4) can be a result of either Translation Rule 1 (p. 84) or Translation Rule 3 (p. 86). Firstly, we consider the former, where it must be that $Proc(t) \xrightarrow{\epsilon}_s P(t)$ for some visible symbolic event $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ such that $\#\$^t(\epsilon) > 0$ and some symbolic state $P(t)$. In addition, $Proc'(t) = Replace^t_{\$\mapsto!}(c, Proc(t))$ and $\Gamma' = \Gamma \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon)\}$, where v is a function in $\$^t(\epsilon) \to \hat{T}$. Then, v is also a function in $\$^t(\epsilon) \to T$, so Translation Rule 1 implies that

$$\begin{aligned}(Proc(t), \Gamma, T) &\xrightarrow{\tau} (Replace^t_{\$\mapsto!}(c, Proc(t)), \Gamma \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon)\}, T) \\ &= (Proc'(t), \Gamma', T).\end{aligned}$$

If the transition in (5.4) is a result of Translation Rule 3, then it must be that $Proc(t) \xrightarrow{\tau}_s Proc'(t)$ and $\Gamma = \Gamma'$. Hence, the same rule implies that $(Proc(t), \Gamma, T) \xrightarrow{\tau} (Proc'(t), \Gamma', T)$.

Case 2. Suppose that $a = c.v_1 \ldots v_k$ is a visible event.
Then, the translation rules of COSE imply that the transition in (5.4) must be the result of Translation Rule 2 (p. 85). The rule implies that $Proc(t) \xrightarrow{\epsilon}_s Proc'(t)$ for some visible symbolic event $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ such that $\#\$^t(\epsilon) = 0$. In addition, $\Gamma' = \Gamma \oplus \{x_i \mapsto v_i \mid i \in ?^t(\epsilon)\}$, and for all i in $?^t(\epsilon)$, v_i is in $\hat{T}$. However,

since $\hat{T} \subseteq T$, we have that for all i in $?^t(\epsilon)$, v_i is in T. Therefore, Translation Rule 2 implies that

$$(Proc(t), \Gamma, T) \xrightarrow{a} (Proc'(t), \Gamma \oplus \{x_i \mapsto v_i \mid i \in ?^t(\epsilon)\}, T) = (Proc'(t), \Gamma', T).$$

This completes the base case.

Inductive case. Suppose the result holds for some $n = k$, where $k \geq 0$. Suppose that the transition in (5.4) requires $k+1$ applications of Transition Rule 4. Then it must be that $Proc(t) \xrightarrow{cond}_s P(t)$ and $(P(t), \Gamma, \hat{T}) \xrightarrow{a} (Proc'(t), \Gamma', \hat{T})$. The latter transition requires k applications of Transition Rule 4, so the inductive hypothesis implies that $(P(t), \Gamma, T) \xrightarrow{a} (Proc'(t), \Gamma', T)$. Hence, Translation Rule 4 implies that $(Proc(t), \Gamma, T) \xrightarrow{a} (Proc'(t), \Gamma', T)$, which completes our proof. ∎

Corollary 5.1.19. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Let T and $\hat{T}$ be instantiations of type T such that $\hat{T} \subseteq T$. Then, if*

$$(Proc(t), \Gamma, \hat{T}) \stackrel{tr}{\Longrightarrow} (Proc'(t), \Gamma', \hat{T}),$$

then

$$(Proc(t), \Gamma, T) \stackrel{tr}{\Longrightarrow} (Proc'(t), \Gamma', T).$$

Proof: Let s be a sequence of events such that $(Proc(t), \Gamma, \hat{T}) \stackrel{s}{\longmapsto} (Proc'(t), \Gamma', \hat{T})$ and $s \setminus \{\tau\} = tr$. Then the result follows from a simple induction on the length of s using Proposition 5.1.18. ∎

5.2 Threshold results

Recall that given an instantiation T of type t and a non-negative integer B, we defined (Definition 5.0.1) a *B-collapsing function* ϕ to be a function from T to $\{0\,..\,B\}$ such that

- $\phi(v) = v$ for all v in $\{0\,..\,B-1\}$,
- $\phi(v) = B$ for all v in $\{B\,..\,\#T-1\}$.

In other words, ϕ replaces all but a fixed finite number of members of t by a single value. Whenever B is clear from context, we call ϕ a *collapsing function*.

In this section we present the main results of our type reduction theory. Our aim is to develop techniques to show that

$$Spec(\hat{T}) \sqsubseteq \phi(Impl(T)) \text{ implies that } Spec(T) \sqsubseteq Impl(T), \quad \text{for all } T \text{ such that } T \supseteq \hat{T}, \text{ where } \phi : T \to \hat{T} \text{ is a collapsing function.} \tag{5.5}$$

The use of parameters in specifications and/or implementations leads to the problem of having to decide infinitely many refinements in order to deduce the answer to

Object	Application	Meaning
event	$\phi(c.v_1 \dots v_k)$	$c.\phi(v_1) \dots \phi(v_k)$
set/type	$\phi(S)$	$\{\phi(x) \mid x \in S\}$
trace	$\phi(tr)$	$\langle \phi(e) \mid e \leftarrow tr \rangle$
environment	$\phi(\Gamma)$	$\{x \mapsto \phi(v) \mid \Gamma(x) = v\}$
process	$\phi(P)$	$P[\![^{\phi(e)}/_e \mid e \in \Sigma]\!]$

Table 5.1: Lifting the definition of ϕ.

a verification problem. Our technique of using collapsing functions treats some values of type t as essentially identical.

Our two main results, Theorem 5.2.15 and Theorem 5.2.24, prove (5.5) in the traces and stable failures models, respectively. They require suitable assumptions on *Spec* (including **SeqNorm**) and *Impl* (that it is symmetric in t); they give a lower bound on the size of $\hat{T}$ based on the syntax of *Spec*.

The rest of this section is structured as follows. Below we lift ϕ to various objects. In Section 5.2.1 we present conditions, on both specifications and implementations, which we will use in subsequent theorems. We present and prove type reduction theorems for both the traces and the stable failures models in Section 5.2.2 and Section 5.2.3, respectively.

Given a boolean condition *cond*, we define $\phi(cond)$ to be like *cond*, except that every value or variable x of type t is replaced by $\phi(x)$. We adopt the notational convention that if x is a value or a variable of a type other than t or it is a type other than t, then $\phi(x) = x$.

We lift the application of ϕ to other common objects used in this thesis in the natural way (see Table 5.1).

Finally, given an instantiation T of type t and a B-collapsing function ϕ, we define

$$\phi^{-1}(v) = \begin{cases} \{v' \in T \mid \phi(v') = v\} & \text{if } v \in \{0 \,..\, B\} \\ \{v\} & \text{otherwise} \end{cases}.$$

Also, given T, we lift the definition of ϕ^{-1} to events:

$$\phi^{-1}(c.v_1 \dots v_k) = \{c.v'_1 \dots v'_k \mid \forall\, i \in \{1 \,..\, k\} \bullet v'_i \in \phi^{-1}(v_i)\},$$

as well as to sets of events:

$$\phi^{-1}(S) = \bigcup \{\phi^{-1}(e) \mid e \in S\}.$$

5.2.1 Conditions on processes

In this section we start by defining the notion of type symmetry in the distinguished type; our main theorems will require the implementation process to satisfy this property. We then define a property concerning the use of equality tests; our main theorems will require the specification process to satisfy this property.

The TypeSym condition

In Section 4.1.1 we defined the concept of data independence which, undoubtedly, is a very useful property for studying parameterised systems [CR98, RB99, CR99b, Low04, LNR04b]. However, in practice it turns out to be too strong for the implementations we consider, since we study parallel compositions of node processes indexed over the parameter. Such compositions are banned by data independence. This is why we define a weaker condition, which only requires all behaviours of a given process to be *symmetric* in the parameter. Informally, a process syntax satisfies the **TypeSym** condition if the behaviours of all its concretisations are invariant under permutations of values of parameter instantiations. The following expresses this formally.

Definition 5.2.1. A process syntax $Proc(t)$ satisfies **TypeSym** if for every T and every bijection $\pi : T \to T$, $Proc(T)$ and $Proc(T)[\![^{\pi(e)}/_e \mid e \in \Sigma]\!]$ are bisimilar.

Semantic definitions, like the one above, tend to be hard to check efficiently. So we note here sufficient syntactic conditions for **TypeSym**.

Proposition 5.2.2. *A process syntax $Proc(t)$ satisfies **TypeSym** if it uses no*

(i) constants of type t,

(ii) operations on type t, including polymorphic operations (e.g. tupling or lists),

(iii) functions whose domains or co-domains involve type t,

(iv) selections or indexing from sets involving t, unless the selection or indexing is over the whole of t, except this restriction does not apply to the alphabets of nodes in alphabetised parallel compositions index over t,

(v) conditional choices on t, except for equality and inequality tests.

Proof sketch (of Proposition 5.2.2): Let CSP_t be the set of CSP syntaxes parameterised by t, all of whose free variables (other that t itself) are of type t and which satisfy clauses (i)–(v) of the proposition. We use $[\![\pi]\!]$ to denote $[\![^{\pi(e)}/_e \mid e \in \Sigma]\!]$. Then

$$\mathcal{B} = \left\{ \left(P(T)[\![\Gamma]\!], (P(T)[\pi^{-1}(\Gamma)])[\![\pi]\!] \right) \mid P(t) \in CSP_t, \Gamma \in Env(T) \right\}$$

is the required strong bisimulation relation on the set of CSP processes whose syntaxes satisfy the conditions of Proposition 5.2.2. The proof is based on a structural induction on $P(t)$. ∎

The syntactic definition of data independence (Definition 4.1.1) comprises a superset of the requirements of Proposition 5.2.2, so we immediately have the following result.

Corollary 5.2.3. *Every data independent process satisfies **TypeSym**.*

Example 5.2.4. Consider a system built as the parallel composition of node processes $\mathcal{N}_i(t)$ for each $i \in t$:

$$Nodes(t) = \|\ i \in t \bullet [A(i,t)]\ \mathcal{N}_i(t).$$

This process syntax satisfies **TypeSym** provided the node process $\mathcal{N}_i(t)$ and the alphabet $A(i,t)$ satisfy the conditions of Proposition 5.2.2, so in particular they treat their "identity" parameter i polymorphically and any selection from t in $\mathcal{N}_i(t)$ is over the whole of t; informally, different nodes and alphabets need to be identical up to renaming of the identities.

Note, though, that $Nodes(t)$ does not satisfy data independence, since it contains a replicated operator (parallel composition) that is indexed over t.

Further, if we define the context $C_t[\cdot]$ that composes its argument with a controller process *Ctrl* and hides some events:

$$C_t[\cdot] = (\ \cdot \underset{X}{\|}\ Ctrl)\ \setminus\ Y,$$

then $C_t[Nodes(t)]$ satisfies **TypeSym** provided:

- the controller process *Ctrl* satisfies the conditions of Proposition 5.2.2; informally, it needs to treat different nodes in the same way,
- the sets X and Y satisfy the conditions of Proposition 5.2.2.

We will consider implementation processes of this form in Chapter 6.

End of example.

The syntactic requirements defined by Proposition 5.2.2 are sufficient, but not necessary for $Proc(t)$ to satisfy **TypeSym**, as the following example illustrates.

Example 5.2.5. The process $\Box\ y{:}t \bullet c?x{:}(t \setminus \{y\})!y \rightarrow STOP$ satisfies **TypeSym**. However, it does not satisfy the conditions of Proposition 5.2.2, in particular because x is selected from a proper subset of t.

End of example.

The following two remarks are direct consequences of the **TypeSym** condition.

Remark 5.2.6. Suppose that $Proc(t)$ satisfies **TypeSym**. Then, for all T:

(i) if $tr \in traces(Proc(T))$, then for all bijections $\pi : T \rightarrow T$, $\pi(tr) \in traces(Proc(T))$,

(ii) if $(tr, X) \in failures(Proc(T))$, then for all bijections $\pi : T \rightarrow T$, $(\pi(tr), \pi(X)) \in failures(Proc(T))$.

Equality tests

For a data independent process syntax, the syntactic condition $\mathbf{PosConjEqT}^t$, formulated by Lazić in [Laz99, Chapter 3] specifies that for every equality test on t, the positive branch is a prefix and the negative branch is simple the *STOP* process. In [Ros98, RB99], a weaker version of $\mathbf{PosConjEqT}^t$ is discussed, without the restrictions on the process in the positive branch. These definitions refer only to the traces model; we now extend them to other CSP models.

Definition 5.2.7. Given a CSP model $\mathcal{M}$ and a process syntax $Proc(t)$, we say that $Proc(t)$ satisfies the $\mathbf{PosConjEqT}^t_{\mathcal{M}}$ condition if for every conditional choice on t of the form

$$\text{if } cond \text{ then } P(x_1, \ldots, x_k) \text{ else } Q(x_1, \ldots, x_k)$$

within $Proc(t)$, we have that

- *cond* is a positive conjunction of equality tests on t (which gives rise to the name of the condition), and
- $P(v_1, \ldots, v_k) \sqsubseteq_{\mathcal{M}} Q(v_1, \ldots, v_k)$ for all values $v_1, \ldots, v_k$.

We decorate the name of the condition with t to emphasize the fact that the condition places restrictions only on equality tests on t; conditionals on types other than t can be arbitrary.

In Chapter 6 we will consider compositions of nodes with controller processes that satisfy $\mathbf{PosConjEqT}^t_{\mathrm{T}}$.

For technical reasons, it will be desirable in our work to assume that every positive branch of a conditional choice is a refinement of the negative branch. This can be viewed as a reversed version of $\mathbf{PosConjEqT}^t_{\mathcal{M}}$. Hence, we have the following definition.

Definition 5.2.8. Given a CSP model $\mathcal{M}$ and a process syntax $Proc(t)$, we say that $Proc(t)$ satisfies the $\mathbf{RevPosConjEqT}^t_{\mathcal{M}}$ condition if for every conditional choice on t of the form

$$\text{if } cond \text{ then } P(x_1, \ldots, x_k) \text{ else } Q(x_1, \ldots, x_k)$$

within $Proc(t)$, we have that

- *cond* is a positive conjunction of equality tests on t, and
- $Q(v_1, \ldots, v_k) \sqsubseteq_{\mathcal{M}} P(v_1, \ldots, v_k)$ for all values $v_1, \ldots v_k$.

Whenever $\mathcal{M}$ is clear from the context, we will simply write $\mathbf{RevPosConjEqT}^t$.

Example 5.2.9. The process syntax

$$Proc(t) = in?x{:}t?y{:}t?z{:}t \rightarrow \text{if } x = y \text{ then } out.x \rightarrow out.y \rightarrow STOP \text{ else } out\$z \rightarrow (out.y \rightarrow STOP \sqcap STOP)$$

satisfies $\mathbf{RevPosConjEqT}^t_{\mathrm{F}}$. However, the process syntax

$$Proc(t) = in?x{:}t?y{:}t \rightarrow \mathrm{if}\, x = y \;\mathrm{then}\; out.x \rightarrow STOP \;\mathrm{else}\; out.y \rightarrow STOP$$

does not satisfy $\mathbf{RevPosConjEqT}^t_{\mathrm{T}}$, because if x and y are distinct values, then $out.y \rightarrow STOP \not\sqsubseteq_{\mathrm{T}} out.x \rightarrow STOP$.

End of example.

Most specification processes that one tends to use in practice do not contain conditionals, so vacuously satisfy both $\mathbf{PosConjEqT}^t_{\mathcal{M}}$ and $\mathbf{RevPosConjEqT}^t_{\mathcal{M}}$ for all models $\mathcal{M}$. Further, our experience is that many specifications that do contain conditionals satisfy $\mathbf{RevPosConjEqT}^t_{\mathcal{M}}$.

In Section 5.2.1 we presented two similar conditions (**PosConjEqT** and **RevPosConjEqT**) that reduce the control variability introduced by conditionals. In a similar fashion, we now introduce a condition that bans conditionals on t altogether.

Definition 5.2.10. A process syntax $Proc(t)$ satisfies condition $\mathbf{NoEqT}^t$ if it does not possess any equality or inequality tests on t, whether implicit or explicit, which require maps from variables to values of type t in order to have the truth of their conditions fully evaluated.

In the above definition, by an explicit conditional we mean any construct of the form 'if *cond* then $P(t)$ else $Q(t)$'. By an implicit conditional we mean a parallel composition with arguments using the same channel name in one of their initial constructs and at least two of these constructs are outputs.

Example 5.2.11. The process syntax

$$\mathcal{N}_{myId}(t) = in?i{:}t \rightarrow \mathrm{if}\, myId = i \;\mathrm{then}\; a \rightarrow STOP \;\mathrm{else}\; b \rightarrow STOP$$

does not satisfy $\mathbf{NoEqT}^t$, since it contains an explicit equality test on t that requires a map from $\{myId, i\}$ to some values of type t in order to evaluate the truth of condition $myId = i$.

Similarly, the process syntax

$$\mathcal{N}_{myId}(t) = in?x{:}t \rightarrow out.x \rightarrow STOP \;\underset{\{|out|\}}{\parallel}\; in?y{:}t \rightarrow out.y \rightarrow STOP$$

does not satisfy $\mathbf{NoEqT}^t$, since an event on the *out* channel can be performed if and only if $x = y$, so there is an implicit equality test on t that requires a map from $\{x, y\}$ to some values of type t in order to evaluate the truth of the equality test.

Finally, the process syntax

$$\mathcal{N}_{myId}(t) = in?x{:}t \rightarrow \mathrm{if}\, x = x \;\mathrm{then}\; a \rightarrow STOP \;\mathrm{else}\; b \rightarrow STOP$$

satisfies $\mathbf{NoEqT}^t$, since $x = x$ evaluates to *True* without the need for any map from $\{x\}$ to a value of type t.

End of example.

5.2.2 Threshold results for the traces model

In this section we present the main results of our type reduction theory for use within the traces model.

We begin with a proposition that establishes that, provided $Proc(t)$ satisfies **SeqNorm** and $\mathbf{RevPosConjEqT}^t_{\mathrm{T}}$ and given a collapsing function ϕ, if

- tr is a trace of $(Proc(t), \Gamma_{init}, T)$ (for some sufficiently large T),
- $\phi(tr)^\frown\langle e\rangle$ is a trace of $(Proc(t), \phi(\Gamma_{init}), T)$, and
- e does not have outputs of type t from outside of $\{0\,..\,B-1\}$,

then every event that is like e, except it has arbitrary values of inputs of type t, is in the initials of $(Proc(t), \Gamma_{init}, T)/tr$. In both the statement and the proof of this proposition we take the underlying type of all configurations to be the fixed type T.

Proposition 5.2.12. *Let B be some natural number. Suppose that*

- *$Proc(t)$ satisfies **SeqNorm** and $\mathbf{RevPosConjEqT}^t_{\mathrm{T}}$,*
- *ϕ is a B-collapsing function, and*
- *T is an instantiation of type t of size at least $B+1$.*

Suppose further that

(i) $tr \in traces(Proc(t), \Gamma_{init})$,

(ii) $\phi(tr)^\frown\langle e\rangle \in traces(Proc(t), \phi(\Gamma_{init}))$ *with* $e = c.v_1 \ldots v_k$,

(iii) $\sigma^\frown\langle\epsilon\rangle \in SymbolicTraces(Proc(t))$ *and* $\sigma^\frown\langle\epsilon\rangle$ $generates_{\phi(\Gamma_{init})}$ $\phi(tr)^\frown\langle e\rangle$, *where ϵ is a visible symbolic event of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$, and*

(iv) $\forall\, i \in {!^t}(\epsilon) \bullet v_i \in \{0\,..\,B-1\}$.

Then[2]

$$\forall\, v' \in \{1\,..\,k\} \to Value \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i) \bullet tr^\frown\langle c.v'_1 \ldots v'_k\rangle \in traces(Proc(t), \Gamma_{init}).$$

Proof: We prove the result using a structural induction on $Proc(t)$. The most interesting cases are those for prefix, external choice and conditional choice.

STOP
The result holds vacuously in this case.

Prefix
Suppose that $Proc(t) = \alpha \to P(t)$ for some construct $\alpha = c'\S'_1 x'_1{:}X'_1 \ldots \S'_k x'_k{:}X'_k$ and

[2]The notation $\forall\, x \in X \mid P(x) \bullet Q(x)$ is equivalent to $\forall\, x \in X \bullet P(x) \Rightarrow Q(x)$.

some process syntax $P(t)$. We consider two cases now.

Subcase 1. Suppose that $tr = \langle\rangle$. Then, by Symbolic Prefix Rule 1 (p. 78) and Symbolic Prefix Rule 2 (p. 78), $\sigma \in \{\tau\}^*$. Hence $Proc(t) \xrightarrow{\tau}{}_s^* \xrightarrow{\epsilon}{}_s$. Using Translation Rule 3 (p. 86) and Remark 4.4.6 we get that

$$\begin{aligned} &\forall\, v' \in \{1\,..\,k\} \to Value \mid \\ &\quad (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = \Gamma_{init}(x_i)) \bullet \\ &\qquad \langle c.v'_1 \ldots v'_k \rangle \in traces(Proc(t), \Gamma_{init}) \end{aligned} \tag{5.6}$$

Since $tr = \langle\rangle$, assumption (iii) of the proposition and the definition of the *generates* relation (p. 87) imply together that $\langle\epsilon\rangle$ $generates_{\phi(\Gamma_{init})}$ $\langle e\rangle$. Therefore, again by the definition of *generates* and by assumption (iv),

$$\forall\, i \in !^t(\epsilon) \bullet v_i = (\phi(\Gamma_{init}))(x_i) \wedge v_i \in \{0\,..\,B-1\}. \tag{5.7}$$

Suppose that

$$\begin{aligned} &v' \in \{1\,..\,k\} \to Value \text{ is such that} \\ &\quad (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i). \end{aligned}$$

Then, from (5.7) we can infer that $\forall\, i \in !^t(\epsilon) \bullet v'_i = (\phi(\Gamma_{init}))(x_i) \wedge v'_i \in \{0\,..\,B-1\}$. However, the properties of ϕ imply that for all variables *var* and all values *val* we have that

$$(\phi(\Gamma_{init}))(var) = val \wedge val \in \{0\,..\,B-1\} \Rightarrow \Gamma_{init}(var) = val,$$

so

$$\forall\, i \in !^t(\epsilon) \bullet v'_i = \Gamma_{init}(x_i). \tag{5.8}$$

In addition, from the definition of *generates*, $\forall\, i \in !^{non\text{-}t}(\epsilon) \bullet v_i = \phi(\Gamma_{init})(x_i)$. But we know that $\forall\, i \in !^{non\text{-}t}(\epsilon) \bullet v'_i = v_i$, so

$$\forall\, i \in !^{non\text{-}t}(\epsilon) \bullet v'_i = (\phi(\Gamma_{init}))(x_i) = \Gamma_{init}(x_i)$$

with the last equality following from the fact that for all i in $!^{non\text{-}t}(\epsilon)$, x_i must be of a non-t type. Hence and from (5.8),

$$\forall\, i \in !(\epsilon) \bullet v'_i = \Gamma_{init}(x_i).$$

Therefore, (5.6) implies that $\langle c.v'_1 \ldots v'_k \rangle \in traces(Proc(t), \Gamma_{init})$. We have shown that

$$\begin{aligned} &\forall\, v' \in \{1\,..\,k\} \to Value \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v_i = v'_i) \bullet \\ &\quad \langle c.v'_1 \ldots v'_k \rangle \in traces(Proc(t), \Gamma_{init}), \end{aligned}$$

which is what we wanted to show.

Subcase 2. Suppose that $tr = \langle e'\rangle\hat{}\,tr'$ for some visible event a and some trace tr'. Then $\phi(tr) = \langle\phi(e')\rangle\hat{}\,\phi(tr')$. So clearly $\phi(tr)$ is non-empty and we know that

σ $generates_{\phi(\Gamma_{init})}$ $\phi(tr)$. By the definition of *generates*, it must be that there is at least one visible symbolic event within σ. Hence, $\sigma = \sigma_1 \hat{} \langle \epsilon' \rangle \hat{} \sigma_2$ for some visible symbolic event ϵ' and some symbolic traces σ_1 and σ_2 such that σ_1 is in $\{\tau\}^*$ (σ_1 cannot contain any conditional symbolic events because $Proc(t)$ is a prefix). Then,

$$Proc(t) \xrightarrow{\tau}{}^* \xrightarrow{\epsilon'} P'(t)$$

and

$$\sigma_2 \hat{} \langle \epsilon \rangle \in SymbolicTraces(P'(t)), \tag{5.9}$$

where $P'(t)$ is like $P(t)$, but with some substitutions of concrete values for the non-t type input variables of α, as dictated by the SSOS firing rules for prefix (Section 4.3.2). We aim to apply the inductive hypothesis to $P'(t)$.

We can infer using Translation Rule 3 (p. 86) and Remark 4.4.6 that

$$(Proc(t), \phi(\Gamma_{init})) \xrightarrow{\tau}{}^* \xrightarrow{\phi(e')} (P'(t), \phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e')),$$

where, recall from Section 4.6, $Match(\epsilon', \phi(e'))$ is a map from type t input variables of ϵ' to the corresponding concrete values of $\phi(e')$. We have that configuration $(P'(t), \phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e'))$ is unique (thanks to Lemma 5.1.12). We know from assumption (ii) that $\langle \phi(e') \rangle \hat{} \phi(tr') \hat{} \langle e \rangle \in traces(Proc(t), \Gamma_{init})$. Hence,

$$\phi(tr') \hat{} \langle e \rangle \in traces(P'(t), \phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e'))). \tag{5.10}$$

Similarly, since $Proc(t) \xrightarrow{\tau}{}^* \xrightarrow{\epsilon'} P'(t)$ and $tr = \langle e' \rangle \hat{} tr' \in traces(Proc(t), \Gamma_{init})$ (from assumption (i)), using Translation Rule 3 (p. 86), Remark 4.4.6 and Lemma 5.1.12 we can infer that

$$(Proc(t), \Gamma_{init}) \xrightarrow{\tau}{}^* \xrightarrow{e'} (P'(t), \Gamma_{init} \oplus Match(\epsilon', e')) \xRightarrow{tr'} . \tag{5.11}$$

Hence,

$$tr' \in traces(P'(t), \Gamma_{init} \oplus Match(\epsilon', e')). \tag{5.12}$$

Finally, assumption (iii) implies that

$$\sigma \hat{} \langle \epsilon \rangle = \sigma_1 \hat{} \langle \epsilon' \rangle \hat{} \sigma_2 \hat{} \langle \epsilon \rangle \; generates_{\phi(\Gamma_{init})} \; \phi(tr) \hat{} \langle e \rangle = \langle \phi(e') \rangle \hat{} \phi(tr') \hat{} \langle e \rangle.$$

So, by the definition of *generates* (p. 87),

$$\sigma_2 \hat{} \langle \epsilon \rangle \; generates_{\phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e'))} \; \phi(tr') \hat{} \langle e \rangle. \tag{5.13}$$

We can now deduce the inductive hypothesis for $P'(t)$, with tr' in place of tr, σ_2 in place of σ, and $\Gamma_{init} \oplus Match(\epsilon', e')$ in place of Γ_{init}: (5.12) gives us condition (i); (5.10) gives us condition (ii), observing that $\phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e')) = \phi(\Gamma_{init} \oplus Match(\epsilon', e'))$; and (5.9) and (5.13) give us condition (iii). Hence

$$\forall\, v' \in \{1 \,..\, k\} \to Value \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v_i = v'_i) \bullet \\ tr' \hat{} \langle c.v'_1 \ldots v'_k \rangle \in traces(P'(t), \Gamma_{init} \oplus Match(\epsilon', e')).$$

This, combined with (5.11), gives us that

$$\forall\, v' \in \{1\,..\,k\} \to \mathit{Value} \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v_i = v'_i) \bullet \langle e' \rangle \mathbin{^\frown} tr' \mathbin{^\frown} \langle c.v'_1 \ldots v'_k \rangle \in \mathit{traces}(\mathit{Proc}(t), \Gamma_{\mathit{init}}).$$

However, $tr = \langle e' \rangle \mathbin{^\frown} tr'$, so the result holds.

External choice
Suppose that $\mathit{Proc}(t) = P(t) \;\Box\; Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. **SeqNorm** implies that the channels of the initial symbolic events of $P(t)$ and $Q(t)$ are disjoint, so we have that either

$$\phi(tr) \mathbin{^\frown} \langle e \rangle \in \mathit{traces}(P(t), \phi(\Gamma_{\mathit{init}})) \quad \text{and} \quad tr \in \mathit{traces}(P(t), \Gamma_{\mathit{init}})$$

or

$$\phi(tr) \mathbin{^\frown} \langle e \rangle \in \mathit{traces}(Q(t), \phi(\Gamma_{\mathit{init}})) \quad \text{and} \quad tr \in \mathit{traces}(Q(t), \Gamma_{\mathit{init}}).$$

Without loss of generality we assume the former. Let $\rho \in \mathit{SymbolicTraces}(P(t))$ be such that $\rho\ \mathit{generates}_{\phi(\Gamma_{\mathit{init}})}\ \phi(tr)$. Then σ is like ρ, except that it may have some additional τ's (promoted from $Q(t)$) before the first visible symbolic event (recall that **SeqNorm** prevents conditional symbolic events before a visible symbolic event). Hence, $\sigma = \tau^a \mathbin{^\frown} \rho$ for some a. Therefore, since $\sigma \mathbin{^\frown} \langle \epsilon \rangle\ \mathit{generates}_{\phi(\Gamma_{\mathit{init}})}\ \phi(tr) \mathbin{^\frown} \langle e \rangle$,the definition of the *generates* relation (p. 87) implies

$$\rho \mathbin{^\frown} \langle \epsilon \rangle\ \mathit{generates}_{\phi(\Gamma_{\mathit{init}})}\ \phi(tr) \mathbin{^\frown} \langle e \rangle.$$

The result is now implied by the inductive hypothesis for $P(t)$ and the semantics of external choice.

Internal choice and timeout
Similar to the case for external choice, above.

Conditional choice
Suppose that $\mathit{Proc}(t) =$ if *cond* then $P(t)$ else $Q(t)$ for some process syntax $P(t)$ and $Q(t)$. If *cond* is not in *Cond*, then *cond* immediately evaluates to *True* or *False* and the result is immediately implied by the inductive hypothesis for $P(t)$ or $Q(t)$, respectively. So suppose *cond* is in *Cond*. Then, by the SSOS firing rules for conditional choice (see Section 4.3.2) it must be that

$$\sigma = \langle \mathit{cond} \rangle \mathbin{^\frown} \rho \quad \text{or} \quad \sigma = \langle \neg \mathit{cond} \rangle \mathbin{^\frown} \rho$$

for some symbolic trace ρ. We now perform a case analysis on the truth value of the evaluation of *cond* within the environments Γ_{init} and $\phi(\Gamma_{\mathit{init}})$.

Case 1. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = [\![cond]\!]_{\Gamma_{init}} = \mathit{True}$.
Then it must be that $\rho \in \mathit{SymbolicTraces}(P(t))$. From assumption (iii) we have that

$$\rho^\frown\langle\epsilon\rangle \ \mathit{generates}_{\phi(\Gamma_{init})} \ \phi(tr)^\frown\langle e\rangle.$$

In addition, from assumptions (i) and (ii) we have that

$$tr \in \mathit{traces}(P(t), \Gamma_{init}) \quad \text{and} \quad \phi(tr)^\frown\langle e\rangle \in \mathit{traces}(P(t), \phi(\Gamma_{init})).$$

The result is now implied in this case by the inductive hypothesis for $P(t)$ and the fact that $(\mathit{Proc}(t), \Gamma_{init}) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \Gamma_{init})$.

Case 2. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = [\![cond]\!]_{\Gamma_{init}} = \mathit{False}$.
This case is like Case 1, above, with $Q(t)$ in place of $P(t)$.

Case 3. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = \mathit{True} \wedge [\![cond]\!]_{\Gamma_{init}} = \mathit{False}$.
Then, by assumption (i) and (ii),

$$tr \in \mathit{traces}(Q(t), \Gamma_{init}) \quad \text{and} \quad \phi(tr)^\frown\langle e\rangle \in \mathit{traces}(P(t), \phi(\Gamma_{init})).$$

Since $\mathit{Proc}(t)$ satisfies $\mathbf{RevPosConjEqT}^t_{\mathrm{T}}$, we have that $(Q(t), \phi(\Gamma_{init})) \sqsubseteq_{\mathrm{T}} (P(t), \phi(\Gamma_{init}))$, so

$$\phi(tr)^\frown\langle e\rangle \in \mathit{traces}(Q(t), \phi(\Gamma_{init})).$$

Let $\rho'^\frown\langle\epsilon'\rangle \in \mathit{SymbolicTraces}(Q(t))$ be such that $\rho'^\frown\langle\epsilon'\rangle \ \mathit{generates}_{\phi(\Gamma_{init})} \ \phi(tr)^\frown\langle e\rangle$. Then, by Lemma 5.1.17, $!(\epsilon') \subseteq !(\epsilon)$. Hence, $!^t(\epsilon') \subseteq !^t(\epsilon)$, so assumption (iv) implies that

$$\forall\, i \in !^t(\epsilon') \bullet v_i \in \{0\,..\,B-1\}.$$

Therefore, by the inductive hypothesis for $Q(t)$,

$$\begin{array}{l} \forall\, v' \in \{1\,..\,k\} \to \mathit{Value} \mid \\ \quad (\forall\, i \in \$^t(\epsilon') \cup ?^t(\epsilon') \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon') \bullet v'_i = v_i) \bullet \\ \qquad tr^\frown\langle c.v'_1 \ldots v'_k\rangle \in \mathit{traces}(Q(t), \Gamma_{init}). \end{array} \tag{5.14}$$

Suppose $v' \in \{1\,..\,k\} \to \mathit{Value}$ is such that

$$(\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge \forall\, i \in !(\epsilon) \bullet v'_i = v_i.$$

The fact that $!(\epsilon') \subseteq !(\epsilon)$ implies that

$$\forall\, i \in !(\epsilon') \bullet v'_i = v_i. \tag{5.15}$$

In addition, we know that both ϵ and ϵ' give rise to $c.v_1 \ldots v_k$, so

$$\$^t(\epsilon') \cup ?^t(\epsilon') \cup !^t(\epsilon') = \$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon),$$

which means that

$$\$^t(\epsilon') \cup ?^t(\epsilon') \subseteq \$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon).$$

Therefore, since $\forall\, i \in !^t(\epsilon) \bullet v'_i \in T$ (as $\forall\, i \in !^t(\epsilon) \bullet v'_i = v_i$ and, by assumption (iv), $\forall\, i \in !^t(\epsilon) \bullet v_i \in T$) and $\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T$ (by our assumption about v'),

$$\forall\, i \in \$^t(\epsilon') \cup ?^t(\epsilon') \bullet v'_i \in T.$$

Combining this with (5.15) and (5.14) we get that

$$\forall\, v' \in \{1\,..\,k\} \to \mathit{Value} \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i) \bullet \\ tr\hat{}\langle c.v'_1 \ldots v'_k\rangle \in \mathit{traces}(Q(t), \Gamma_{\mathit{init}}).$$

The result now follows, because $(\mathit{Proc}(t), \Gamma_{\mathit{init}}) \stackrel{\langle\rangle}{\Longrightarrow} (Q(t), \Gamma_{\mathit{init}})$.

Case 4. Suppose that $[\![\mathit{cond}]\!]_{\phi(\Gamma_{\mathit{init}})} = \mathit{False} \wedge [\![\mathit{cond}]\!]_{\Gamma_{\mathit{init}}} = \mathit{True}$.
This case is not possible since *cond* is a conjunction of equality tests and for no function ϕ we can ever have $x = y$ and $\phi(x) \neq \phi(y)$.

Binding
Suppose that $\mathit{Proc}(t) = X(t)$ for some process identifier X and suppose that a global environment E maps X to some process definition P (i.e. $E(X) = P$). Then the SSOS firing rule for binding (p. 80) implies that

$$\sigma = \langle\tau\rangle\hat{}\,\rho$$

for some ρ in $\mathit{SymbolicTraces}(P(t))$. Then, assumption (iii) implies that $\rho\hat{}\langle\epsilon\rangle$ $\mathit{generates}_{\phi(\Gamma_{\mathit{init}})}$ $\phi(tr)\hat{}\langle e\rangle$. In addition, by the COSE translation rules (see Section 4.4.1), it must be that

- $tr \in \mathit{traces}(P(t), \Gamma_{\mathit{init}})$,
- $\phi(tr)\hat{}\langle e\rangle \in \mathit{traces}(P(t), \phi(\Gamma_{\mathit{init}}))$,

so the result is now implied by the inductive hypothesis for $P(t)$ and the fact that $(\mathit{Proc}(t), \Gamma_{\mathit{init}}) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \Gamma_{\mathit{init}})$. ∎

Recall that two symbolic traces σ and σ' are non-t equivalent, written $\sigma \equiv_{\text{non-}t} \sigma'$, if the channels and non-t parts of all visible symbolic events are identical between σ and σ' (see Definition 5.1.7). We will need the following definition in order to define our threshold.

Definition 5.2.13. Let $\mathit{Proc}(t)$ be a process syntax, σ a symbolic trace and ϵ a visible symbolic event. Then $!^t(\sigma, \epsilon)(\mathit{Proc}(t))$ is the set of indices of all output variables of type t in all constructs of $\mathit{Proc}(t)$ that end symbolic traces that are non-t equivalent to $\sigma\hat{}\langle\epsilon\rangle$. Formally,

$$!^t(\sigma, \epsilon)(\mathit{Proc}(t)) = \bigcup\{!^t(\epsilon') \mid \sigma'\hat{}\langle\epsilon'\rangle \in \mathit{SymbolicTraces}(\mathit{Proc}(t)) \\ \wedge\, \sigma'\hat{}\langle\epsilon'\rangle \equiv_{\text{non-}t} \sigma\hat{}\langle\epsilon\rangle\}.$$

Example 5.2.14. Let

$$\begin{aligned} \mathit{Proc}(t) = \mathit{in}!a?x{:}t?y{:}t \rightarrow\ & \text{then } \mathit{out}!x\$y{:}t \rightarrow \mathit{STOP} \\ & \text{else } \mathit{out}\$x{:}t!y{:}t \rightarrow \mathit{STOP}. \end{aligned}$$

where a is of a non-t type. Then

$$!^t(\langle \mathit{in}!a?x{:}t?y{:}t\rangle, \mathit{out}\$x{:}t\$y{:}t)(\mathit{Proc}(t)) = \{1, 2\},$$

since both of the constructs on channel *out* end symbolic traces of $\mathit{Proc}(t)$ that are non-t equivalent to the symbolic trace $\langle \mathit{in}!a?x{:}t?y{:}t, \mathit{out}\$x{:}t\$y{:}t\rangle$, with the construct from the positive branch contributing index 1 (the index of variable x) and the construct from the negative branch contributing index 2 (the index of variable y). Similarly

$$!^t(\langle \mathit{in}!b?x{:}t?y{:}t\rangle, \mathit{out}\$x{:}t\$y{:}t)(\mathit{Proc}(t)) = \{\},$$

since $\mathit{Proc}(t)$ cannot perform any symbolic trace that is non-t equivalent to $\langle \mathit{in}!b?x{:}t?y{:}t, \mathit{out}\$x{:}t\$y{:}t\rangle$.

End of example.

We now present the first of our two main results of this chapter. The following theorem establishes a threshold $\mathit{Thresh}_{\mathrm{T}}$ such that, provided $\mathit{Spec}(t)$ and $\mathit{Impl}(t)$ fulfil certain requirements, then for all $B \geq \mathit{Thresh}_{\mathrm{T}}$, if ϕ is a B-precollapsing function, then for all $n \geq B$, if

$$\mathit{Spec}(\{0\,..\,B\}) \sqsubseteq_{\mathrm{T}} \phi(\mathit{Impl}(\{0\,..\,n\})), \tag{5.16}$$

then

$$\mathit{Spec}(\{0\,..\,n\}) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(\{0\,..\,n\})$$

Observe that the specification side of (5.16) is independent of n and therefore remains constant as n varies. The implementation side of (5.16), however, is dependent on n (even though it only communicates values of type t in the range $\{0\,..\,B\}$, thanks to the external renaming by ϕ). So, theoretically, we still need to perform an unbounded number of checks (one for each value of n) in order to verify that $\mathit{Spec}(T) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T)$ for all instantiations T of type t. However, the usefulness of Theorem 5.2.15 comes from the fact that, if we can produce a fixed process *Abstr* such that $\mathit{Abstr} \sqsubseteq_{\mathrm{T}} \phi(\mathit{Impl}(\{0\,..\,n\})$ for all $n \geq B$ for some $B \geq \mathit{Thresh}_{\mathrm{T}}$, then it is enough to perform a single refinement check, namely $\mathit{Spec}(\{0\,..\,\max\{\mathit{Thresh}_{\mathrm{T}}, B\}\}) \sqsubseteq_{\mathrm{T}} \mathit{Abstr}$, in order to conclude that $\mathit{Spec}(\{0\,..\,n\}) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(\{0\,..\,n\})$ for all $n \geq \max\{\mathit{Thresh}_{\mathrm{T}}, B\}$. Combining this with manual checks for all $n \in \{0\,..\max\{\mathit{Thresh}_{\mathrm{T}}, B\} - 1\}$, we obtain the answer to whether $\mathit{Spec}(T) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T)$ for all instantiations T of type t. In Section 5.2.3 we will present an analogous result for the stable failures model. In addition, we will develop techniques for constructing suitable *Abstr* processes to fit the above description in Chapter 6.

Theorem 5.2.15. (Extendibility of traces refinement of systems with replicated components)
Suppose that

(i) $Spec(t)$ *satisfies* ***SeqNorm*** *and* $\boldsymbol{RevPosConjEqT}^t_{\mathrm{T}}$,

(ii) $Impl(t)$ *satisfies* ***TypeSym***,

(iii) $Thresh_{\mathrm{T}}$ *is the maximum number of unique indices of output variables reachable on non-*t *equivalent symbolic traces of* $Spec(t)$*:*

$$Thresh_{\mathrm{T}} = \max\{\#!^t(\sigma, \epsilon)(Spec(t)) \mid \sigma \,\hat{}\, \langle \epsilon \rangle \in SymbolicTraces(Spec(t))\},$$

(iv) $B \geq Thresh_{\mathrm{T}}$,

(v) T *is an instantiation of type* t *of size at least* $B + 1$, *and*

(vi) ϕ *is a* B*-collapsing function.*

Then if $Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(Impl(T))$, *then* $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$.

Proof: Suppose that $Spec(\phi(T)) \sqsubseteq_T \phi(Impl(T))$ and assume for a contradiction that $Spec(T) \not\sqsubseteq_T Impl(T)$. Consider a shortest trace that demonstrates this non-refinement; this trace is necessarily non-empty, so of the form $tr \,\hat{}\, \langle e \rangle$ such that

$$tr \,\hat{}\, \langle e \rangle \in traces(Impl(T)),$$
$$tr \in traces(Spec(T)),$$
$$tr \,\hat{}\, \langle e \rangle \notin traces(Spec(T)).$$

Suppose that $e = c.v_1 \ldots v_k$, and suppose that $tr \,\hat{}\, \langle \epsilon \rangle$ is generated by a symbolic trace $\sigma_1 \,\hat{}\, \langle \epsilon_1 \rangle$, where ϵ_1 is visible.

By assumptions (iii) and (iv), we have that $\#!^t(\sigma_1, \epsilon_1)(Spec(t)) \leq B$, so let $\pi : T \to T$ be a bijection that maps $\{v_i \mid i \in !^t(\sigma_1, \epsilon_1)(Spec(t))\}$ into $\{0 \,..\, B-1\}$. By Corollary 5.2.3, $Spec(t)$ satisfies **TypeSym**, so by Remark 5.2.6 we have that

$$\begin{aligned} &\pi(tr) \,\hat{}\, \langle \pi(e) \rangle \in traces(Impl(T)), \\ &\pi(tr) \in traces(Spec(T)), \\ &\pi(tr) \,\hat{}\, \langle \pi(e) \rangle \notin traces(Spec(T)). \end{aligned} \tag{5.17}$$

Hence,

$$\phi(\pi(tr)) \,\hat{}\, \langle \phi(\pi(e)) \rangle \in traces(\phi(Impl(T))).$$

But $Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(Impl(T))$, so

$$\phi(\pi(tr)) \,\hat{}\, \langle \phi(\pi(e)) \rangle \in traces(Spec(\phi(T))).$$

However, by Corollary 5.1.19, $\mathit{Spec}(T) \sqsubseteq_{\mathrm{T}} \mathit{Spec}(\phi(T))$, so

$$\phi(\pi(tr))\hat{}\langle\phi(\pi(e))\rangle \in \mathit{traces}(\mathit{Spec}(T)). \tag{5.18}$$

We can now apply Proposition 5.2.12 to $\mathit{Spec}(T)$, with $\pi(tr)$ in place of tr, and $\phi(\pi(e))$ in place of e: condition (i) is satisfied by (5.17); condition (ii) is satisfied by (5.18); condition (iii) is satisfied by taking a suitable choice of σ, and taking ϵ to be the symbolic event that generates $\phi(\pi(e))$; condition (iv) is satisfied since $!^t(\epsilon) \subseteq !^t(\sigma_1, \epsilon_1)(\mathit{Spec}(t))$ (since $\sigma\hat{}\langle\epsilon\rangle \equiv_{\mathit{non\text{-}t}} \sigma_1\hat{}\langle\epsilon_1\rangle$), and by construction, all the corresponding fields of $\phi(\pi(tr))$ are in $\{0\,..\,B-1\}$. Considering the valuation v' such that $c.v'_1 \ldots v'_k = \pi(e)$ then allows us to deduce that

$$\pi(tr)\hat{}\langle\pi(e)\rangle \in \mathit{traces}(\mathit{Spec}(T)).$$

This is a contradiction, which completes our proof. ∎

The following corollary shows how we can strengthen the threshold used in Theorem 5.2.15 if we ban conditionals on t.

Corollary 5.2.16. *Suppose that*

(i) $\mathit{Spec}(t)$ *satisfies* ***SeqNorm*** *and* $\boldsymbol{NoEqT}^t$,

(ii) $\mathit{Impl}(t)$ *satisfies* ***TypeSym***,

(iii) $\mathit{Thresh}_{\mathrm{T}}$ *is the maximum number of outputs of type* t *in constructs of* $\mathit{Spec}(t)$*:*

$$\mathit{Thresh}_{\mathrm{T}} = \max\{\#!^t(\alpha) \mid \alpha \textit{ is a construct of } \mathit{Spec}(t)\},$$

(iv) $B \geq \mathit{Thresh}_{\mathrm{T}}$,

(v) T *is an instantiation of type* t *of size at least* $B+1$, *and*

(vi) ϕ *is a* B*-collapsing function.*

Then if $\mathit{Spec}(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(\mathit{Impl}(T))$, *then* $\mathit{Spec}(T) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T)$.

Proof: Let $\sigma\hat{}\langle\epsilon\rangle$ and $\sigma'\hat{}\langle\epsilon'\rangle$ be symbolic traces of $\mathit{Spec}(t)$, with ϵ and ϵ' visible. Suppose that $\sigma\hat{}\langle\epsilon\rangle \equiv_{\mathit{non\text{-}t}} \sigma'\hat{}\langle\epsilon'\rangle$. Since $\mathit{Spec}(t)$ contains no equality tests on t, Lemma 5.1.8 implies that $\epsilon = \epsilon'$. Hence, $\#!^t(\sigma,\epsilon)(\mathit{Spec}(t)) = \#!^t(\epsilon)$. Therefore,

$$\begin{array}{rcl} \mathit{Thresh}_{\mathrm{T}} & = & \max\{\#!^t(\alpha) \mid \alpha \text{ is a construct of } \mathit{Spec}(t)\} \\ & \geq & \max\{\#!^t(\sigma,\epsilon)(\mathit{Spec}(t)) \mid \sigma\hat{}\langle\epsilon\rangle \in \mathit{SymbolicTraces}(\mathit{Spec}(t))\}, \end{array}$$

with equality in the usual case where every construct is reachable. In addition, observe that every process syntax that satisfies $\mathbf{NoEqT}^t$ also satisfies $\mathbf{RevPosConjEqT}^t_{\mathrm{T}}$. The result then follows from Theorem 5.2.15. ∎

Remark 5.2.17. Even if $\mathit{Spec}(t)$ uses a conditional on t, thanks to Lemma 5.1.17, $!^t(\alpha_{\mathit{then}}) \supseteq !^t(\alpha_{\mathit{else}})$, where α_{then} and α_{else} are the constructs in the "then" and "else" branches, respectively; hence the threshold from Corollary 5.2.16 still applies. It's only when $\mathit{Spec}(t)$ contains nested conditionals, like for example

$$\begin{aligned}&\text{if } x = y \text{ then } (\text{if } y = z \text{ then } \mathit{STOP}\\ &\qquad\qquad\qquad\qquad \text{else } c!x\$y{:}t \rightarrow \mathit{STOP})\\ &\qquad \text{else } (\text{if } y = z \text{ then } c\$x{:}t!y \rightarrow \mathit{STOP}\\ &\qquad\qquad\qquad\qquad \text{else } c\$x{:}t\$y{:}t),\end{aligned}$$

that one needs to consider multiple constructs together.

Remark 5.2.18. For every verification problem, the value of $\mathit{Thresh}_{\mathrm{T}}$ in Theorem 5.2.15 and Corollary 5.2.16 depends only on the specification.

Remark 5.2.19. For all specifications with a finite SSLTS, the value of $\mathit{Thresh}_{\mathrm{T}}$ in Theorem 5.2.15 and Corollary 5.2.16 can be calculated in a finite amount of time. All states that can be reached by traces over the same channels need to be considered together; this can be performed by a process similar to normalisation [Ros97, Appendix C].

Example 5.2.20. Recall process syntax $\mathit{Proc}(t)$ from Example 5.2.14, for which $!^t(\langle \mathit{in}!a?x{:}t?y{:}t\rangle, \mathit{out}\$x{:}t\$x{:}t)(\mathit{Proc}(t)) = \{1, 2\}$. It is clear that $!^t(\sigma, \epsilon)(\mathit{Proc}(t))$ has fewer elements for other values of σ and ϵ. Hence, $\mathit{Thresh}_{\mathrm{T}} = 2$ in this case.

End of example.

Example 5.2.21. Let

$$\begin{aligned}\mathit{Spec}(t) = c_1?i{:}t?j{:}t \rightarrow (&c_2!i?j{:}t!a!b \rightarrow c_3.i.i.i \rightarrow \mathit{STOP}\\ &\Box\\ &c_3!i!j\$l{:}t \rightarrow \mathit{STOP}),\end{aligned}$$

where a and b are some values of a non-t type. Then

$$\begin{aligned}&\#!^t(c_1?i{:}t?j{:}t) = 0,\\ &\#!^t(c_2!i?j{:}t!a!b) = 0,\\ &\#!^t(c_3.i.i.i) = 3,\\ &\#!^t(c_3!i!j\$l{:}t) = 2,\end{aligned}$$

Hence, the value of $\mathit{Thresh}_{\mathrm{T}}$ in Corollary 5.2.16 is $\max\{\#!^t(\alpha) \mid \alpha \text{ is a construct of } \mathit{Spec}(t)\} = 3$.

End of example.

The following example demonstrates how Theorem 5.2.15 can be applied in practice.

Example 5.2.22. Suppose that we model a simple client/server architecture, where two clients can establish a direct connection between themselves by using a central server. When idle, the server can accept a connection request (using channel

request_peer) from any process and reach a state, where it ignores any subsequent requests from the same process, but accepts a connection request from any other process and establishes a session between the two. To establish a connection, the server communicates to each of the two processes the identity of the other process (using channel *peer_id*). When a connection is established between, say, processes i and j an event *session_start.i.j* or *session_start.j.i* is communicated. The order of i and j does not play a major rule, except for establishing the order of identities used in later communication. The clients can terminate a connection (using channel *session_end*) before any communication happens and raise an unsuccessful flag (flag f). Alternatively, they may communicate some data (which we do not model) using channel *communicate* and then end the session with a successful flag (flag s). There can be at most one established connection within the entire system at any given time. We let t be the type of the clients' identities.

$$SERVER(t) = request\_peer?i{:}t \rightarrow SERVER'(i,t)$$

$$\begin{aligned}
&SERVER'(i,t) = \\
&\quad request_peer?j{:}t \rightarrow \\
&\qquad \text{if } i = j \text{ then } SERVER'(i,t) \\
&\qquad\quad \text{else } peer_id.i.j \rightarrow peer_id.j.i \rightarrow session_start.i.j \\
&\qquad\qquad \rightarrow session_end!i!j?flag{:}\{s,f\} \rightarrow SERVER(t)
\end{aligned}$$

$$\begin{aligned}
&CLIENT(i,t) = \\
&\quad request_peer.i \rightarrow peer_id!i?j{:}t \\
&\qquad \rightarrow (session_start.i.j \rightarrow CLIENT'(i,j,yes,t) \\
&\qquad\quad \Box \\
&\qquad\quad session_start.j.i \rightarrow CLIENT'(i,j,no,t))
\end{aligned}$$

$$\begin{aligned}
&CLIENT'(i,j,principal,t) = \\
&\quad \text{if } principal = yes \\
&\qquad \text{then } session_end.i.j.f \rightarrow CLIENT(i,t) \\
&\qquad\quad \Box \\
&\qquad\quad communicate.i.j \rightarrow session_end.i.j.s \rightarrow CLIENT(i,t) \\
&\qquad \text{else } session_end.j.i.f \rightarrow CLIENT(i,t) \\
&\qquad\quad \Box \\
&\qquad\quad communicate.j.i \rightarrow session_end.j.i.s \rightarrow CLIENT(i,t)
\end{aligned}$$

Since clients may communicate with each other, we let $CLIENTS(t)$ model a parallel composition of all client processes, each of which synchronises on all the events it can participate in (which we denote by $Alpha(i,t)$).

$$CLIENTS(t) = \Big\| \; i \in t \bullet [Alpha(i,t)] \; CLIENT(i,t)$$

$$\begin{aligned}
Alpha(i,t) = \{&request_peer.i, peer_id.i.j, session_start.i.j, session_start.j.i, \\
&session_end.i.j.flag, session_end.j.i.flag, communicate.i.j, \\
&communicate.j.i \mid j \in t, flag \in \{s,f\}\}
\end{aligned}$$

The complete implementation consists of the server running in parallel with all the clients (synchronising on the appropriate events). All the communication not related to establishing and terminating sessions is hidden.

$$Impl(t) = (SERVER(t) \underset{\{|request\_peer, peer\_id, session\_start, session\_end|\}}{\|} CLIENTS(t)) \setminus \{|\ request\_peer, peer\_id, communicate\ |\}$$

Suppose that we want to verify that there is never more than a single connection established within the system (this is a safety check, so we use the traces model). We let the specification process be as follows.

$$Spec(t) = session\_start\$i{:}t\$j{:}t \rightarrow session\_end!i!j\$status{:}\{s, f\} \rightarrow Spec(t).$$

The process syntaxes $Spec(t)$ and $Impl(t)$ meet the assumptions of Theorem 5.2.15. We have that

$$\#!^t(\langle\rangle, session\_start\$i{:}t\$j{:}t) = 0$$
$$\#!^t(\langle session\_start\$i{:}t\$j{:}t\rangle, session\_end.i.j.s) = 2$$
$$\#!^t(\langle session\_start\$i{:}t\$j{:}t\rangle, session\_end.i.j.f) = 2$$

and $\#!^t(\sigma, \epsilon) \geq 2$ for all other $\sigma\,\hat{}\,\langle\epsilon\rangle \in SymbolicTraces(Spec(t))$. Hence, the threshold, calculated as in Theorem 5.2.15, is

$$Thresh_{\mathrm{T}} = \max\{\#!^t(\sigma, \epsilon)(Spec(t)) \mid \sigma\,\hat{}\,\langle\epsilon\rangle \in SymbolicTraces(Spec(t))\} = 2.$$

(Since $Spec(t)$ satisfies $\mathbf{NoEqT}^t$ we could use the threshold from Corollary 5.2.16, which, in this case, gives the same value.)

Now, if we can find a process $Abstr$ such that $Spec(\{0, 1, 2\}) \sqsubseteq_{\mathrm{T}} Abstr$ and $Abstr \sqsubseteq_{\mathrm{T}} \phi(Impl(\{0\,..\,n\}))$ for all $n \geq 2$ (see Chapter 6), then, by Theorem 5.2.15, $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$ for all T of size 3 or more. The checks for $T = \{0\}$ and $T = \{0, 1\}$ can then be performed directly to complete the answer as to whether the implementation satisfies the specification, regardless of the instantiation of their parameters.

End of example.

5.2.3 Threshold results for the stable failures model

In this section we present type reduction theory results analogous to those in Section 5.2.2, but extended to the stable failures model.

We begin with a proposition that establishes that, provided $Proc(t)$ satisfies **SeqNorm** and $\mathbf{RevPosConjEqT}^t_{\mathrm{F}}$ and given a collapsing function ϕ, if

- tr is a trace of $(Proc(t), \Gamma_{init}, T)$ (for some sufficiently large T),
- $(\phi(tr), X)$ is a failure of $(Proc(t), \phi(\Gamma_{init}), T)$, and
- events in $initials(Proc(t), \Gamma_{init}, T)/tr$ do not have outputs of type t from outside of $\{0\,..\,B-1\}$,

then (tr, X) is a failure of $(Proc(t), \Gamma_{init}, T)/tr$.

In both the statement and the proof of this proposition we take the underlying type of all configurations to be the fixed type T.

Proposition 5.2.23. *Let B be some natural number. Suppose that*

- *$Proc(t)$ satisfies **SeqNorm** and **RevPosConjEqT**$^t_{\mathrm{F}}$,*
- *ϕ is a B-collapsing function, and*
- *T is an instantiation of type t of size at least $B+1$.*

Suppose further that

(i) $tr \in traces(Proc(t), \Gamma_{init})$,

(ii) $(\phi(tr), X) \in failures(Proc(t), \phi(\Gamma_{init}))$, *and*

(iii) if P is a configuration such that $(Proc(t), \Gamma_{init}) \stackrel{tr}{\Longrightarrow} P$, then every output value of type t of every event in $initials(P)$ is in $\{0 \,..\, B-1\}$.

Then $(tr, X) \in failures(Proc(t), \Gamma_{init})$.

Proof: We prove the result using a structural induction on $Proc(t)$.

STOP
If $Proc(t) = STOP$, then $tr = \langle\rangle$ and $(\langle\rangle, X) \in failures(STOP, \Gamma_{init})$ for all X.

Prefix
Suppose that $Proc(t) = \alpha \rightarrow Proc'(t)$ for some construct $\alpha = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and some process syntax $Proc'(t)$. We now consider two cases.

Subcase 1. Suppose that $tr = \langle\rangle$.
The fact that $(\phi(tr), X) \in failures(Proc(t), \phi(\Gamma_{init}))$ implies that there exists an environment Γ such that $\mathrm{dom}(\Gamma) = \$^t(\alpha)$ and

$$(Proc(t), \phi(\Gamma_{init})) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \phi(\Gamma_{init}) \oplus \Gamma) \ \ ref \ \ X,$$

where $P(t)$ is like $Proc(t)$, but with some substitutions of concrete values for the nondeterministic input variables of non-t types of α and with the effects of the application of $Replace_{\$\mapsto !}$, as dictated by the SSOS firing rules for prefix (see Section 4.3.2). Then,

$$(Proc(t), \Gamma_{init}) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \Gamma_{init} \oplus \Gamma)$$

by resolving the nondeterministic selections of α (if any) in the same way. We now show that $(P(t), \Gamma_{init} \oplus \Gamma)$ *ref* X. Observe that the only difference between the initial

events of two configurations (S, Γ_1) and (S, Γ_2) are the output values of type t that come from the environments Γ_1 and Γ_2. Therefore,

$$\begin{array}{l} \mathit{initials}(P(t), \Gamma_{\mathit{init}} \oplus \Gamma) = \\ \quad \{c.v'_1 \ldots v'_k \mid (\forall\, i \in !^t(\alpha) \bullet v'_i = (\Gamma_{\mathit{init}} \oplus \Gamma)\,(x_i)) \\ \qquad\qquad \wedge\, \exists\, c.v_1 \ldots v_k \in \mathit{initials}(P(t), \phi(\Gamma_{\mathit{init}}) \oplus \Gamma) \bullet \\ \qquad\qquad\qquad \forall\, i \in \{1\,..\,k\} \setminus !^t(\alpha) \bullet v'_i = v_i)\}. \end{array}$$

Let v be such that $c.v_1 \ldots v_k$ is in $\mathit{initials}(P(t), \phi(\Gamma_{\mathit{init}}) \oplus \Gamma)$ and let v' be such that $c.v'_1 \ldots v'_k$ is in $\mathit{initials}(P(t), \Gamma_{\mathit{init}} \oplus \Gamma)$. Also, let $i \in !^t(\alpha)$. Hence, $v_i = (\phi(\Gamma_{\mathit{init}}) \oplus \Gamma)(x_i)$ and $v'_i = (\Gamma_{\mathit{init}} \oplus \Gamma)(x_i)$. Therefore, by assumption (iii) of the proposition, $v'_i \in \{0\,..\,B-1\}$. So, thanks to the properties of ϕ, $\phi(v'_i) = v'_i$. Hence $(\phi(\Gamma_{\mathit{init}}) \oplus \Gamma)(x_i) = v'_i$, since $x_i \notin \mathrm{dom}(\Gamma)$. Therefore,

$$\forall\, i \in !^t(\alpha) \bullet v'_i = (\Gamma_{\mathit{init}} \oplus \Gamma)(x_i) = (\phi(\Gamma_{\mathit{init}}) \oplus \Gamma)(x_i) = v_i.$$

Hence,

$$\mathit{initials}(P(t), \Gamma_{\mathit{init}} \oplus \Gamma) = \mathit{initials}(P(t), \phi(\Gamma_{\mathit{init}}) \oplus \Gamma). \tag{5.19}$$

Now, since $(P(t), \Gamma_{\mathit{init}} \oplus \Gamma)$ is stable, $(P(t), \Gamma_{\mathit{init}} \oplus \Gamma)$ *ref* Y for every $Y \subseteq \Sigma \setminus \mathit{initials}(P(t), \Gamma_{\mathit{init}} \oplus \Gamma)$. However, the fact that $(P(t), \phi(\Gamma_{\mathit{init}}) \oplus \Gamma)$ *ref* X implies that $X \subseteq \Sigma \setminus \mathit{initials}(P(t), \phi(\Gamma_{\mathit{init}}) \oplus \Gamma)$, so by (5.19) $(P(t), \Gamma_{\mathit{init}} \oplus \Gamma)$ *ref* X. This implies that $(\langle\rangle, X) \in \mathit{failures}(\mathit{Proc}(t), \Gamma_{\mathit{init}})$, as required.

Subcase 2. Suppose that $tr \neq \langle\rangle$.
Then $tr = \langle e \rangle \hat{}\, tr'$ for some visible event e that matches α and some trace tr'. Let $\Gamma = \phi(\Gamma_{\mathit{init}}) \oplus \mathit{Match}(\alpha, \phi(e))$ and $\Gamma' = \Gamma_{\mathit{init}} \oplus \mathit{Match}(\alpha, e)$. From the assumptions of the proposition we can infer that

$$tr' \in \mathit{traces}(P(t), \Gamma'),$$

and

$$(\phi(tr'), X) \in \mathit{failures}(P(t), \Gamma),$$

where $P(t)$ is like $\mathit{Proc}'(t)$, but with some substitutions of concrete values for the non-t type input variables of α, as dictated by the SSOS firing rules for prefix (see Section 4.3.2). Assumption (iii), combined with the fact that $(\mathit{Proc}(t), \Gamma_{\mathit{init}}) \stackrel{\langle e \rangle}{\Longrightarrow} (P(t), \Gamma')$, implies that if P is a configuration such that $(P(t), \Gamma') \stackrel{tr'}{\Longrightarrow} P$, then every output of type t of every event in $\mathit{initials}(P)$ is in $\{0\,..\,B-1\}$. Observe that $\Gamma = \phi(\Gamma')$. So, by the inductive hypothesis for $P(t)$,

$$(tr', X) \in \mathit{failures}(P(t), \Gamma'),$$

which implies that

$$(tr, X) \in \mathit{failures}(\mathit{Proc}(t), \Gamma_{\mathit{init}}).$$

External choice
Suppose that $Proc(t) = P(t) \,\Box\, Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. We consider two cases.

Subcase 1. Suppose that $tr = \langle\rangle$.
Then, by the assumptions of the proposition and from the denotational semantics of $\Box$ (see e.g. [Ros97, Chapter 8]),

$$(\langle\rangle, X) \in \mathit{failures}(P(t), \phi(\Gamma_{\mathit{init}}))$$

and

$$(\langle\rangle, X) \in \mathit{failures}(Q(t), \phi(\Gamma_{\mathit{init}})).$$

We have that $\langle\rangle \in \mathit{traces}(P(t), \Gamma_{\mathit{init}})$ and $\langle\rangle \in \mathit{traces}(Q(t), \Gamma_{\mathit{init}})$. Assumption (iii), together with the semantics of external choice, implies that if P is a configuration such that $(P(t), \Gamma_{\mathit{init}}) \stackrel{tr}{\Longrightarrow} P$ or $(Q(t), \Gamma_{\mathit{init}}) \stackrel{tr}{\Longrightarrow} P$, then every output of type t of every event in $\mathit{initials}(P)$ is in $\{0 \,..\, B-1\}$. So, using the inductive hypothesis for $P(t)$ and $Q(t)$, we get that

$$(\langle\rangle, X) \in \mathit{failures}(P(t), \Gamma_{\mathit{init}}) \quad \text{and} \quad (\langle\rangle, X) \in \mathit{failures}(Q(t), \Gamma_{\mathit{init}}).$$

Hence, by the denotational semantics of external choice,

$$(\langle\rangle, X) \in \mathit{failures}(\mathit{Proc}(t), \Gamma_{\mathit{init}}).$$

Subcase 2. Suppose that $tr \neq \langle\rangle$.
Then **SeqNorm** implies that the channels of the initial events of $P(t)$ and $Q(t)$ are disjoint, so either

$$(\phi(tr), X) \in \mathit{failures}(P(t), \phi(\Gamma_{\mathit{init}})) \quad \text{and} \quad tr \in \mathit{traces}(P(t), \Gamma_{\mathit{init}})$$

or

$$(\phi(tr), X) \in \mathit{failures}(Q(t), \phi(\Gamma_{\mathit{init}})) \quad \text{and} \quad tr \in \mathit{traces}(Q(t), \Gamma_{\mathit{init}}).$$

Without loss of generality we assume the former. Assumption (iii), together with the semantics of external choice, implies that if P is a configuration such that $(P(t), \Gamma_{\mathit{init}}) \stackrel{tr}{\Longrightarrow} P$, then every output of type t of every event in $\mathit{initials}(P)$ is in $\{0 \,..\, B-1\}$. Then, the inductive hypothesis for $P(t)$ implies that

$$(tr, X) \in \mathit{failures}(P(t), \Gamma_{\mathit{init}}).$$

Hence, by the denotational semantics of external choice,

$$(tr, X) \in \mathit{failures}(\mathit{Proc}(t), \Gamma_{\mathit{init}}).$$

Internal choice
Similar to Subcase 2 of the external choice case, above, using the denotational semantics of $\sqcap$ (see e.g. [Ros97, Chapter 8]).

Sliding choice (timeout)
Suppose that $Proc(t) = P(t) \triangleright Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. We consider two cases.

Subcase 1. Suppose that $tr = \langle\rangle$.
Then, by the assumptions of the proposition and from the denotational semantics of $\triangleright$ (see e.g. [Ros97, Chapter 8]),

$$(\langle\rangle, X) \in failures(Q(t), \phi(\Gamma_{init})).$$

Clearly, $\langle\rangle \in traces(Q(t), \Gamma_{init})$. In addition, assumption (iii), combined with the fact that $(Proc(t), \Gamma_{init}) \stackrel{\langle\rangle}{\Longrightarrow} (Q(t), \Gamma_{init})$, implies that if P is a configuration such that $(Q(t), \Gamma_{init}) \stackrel{tr}{\Longrightarrow} P$, then every output of type t of every event in $initials(P)$ is in $\{0\,..\,B-1\}$. So, the inductive hypothesis for $Q(t)$ implies that

$$(\langle\rangle, X) \in failures(Q(t), \Gamma_{init}).$$

Hence, using the denotational semantics of sliding choice,

$$(\langle\rangle, X) \in failures(Proc(t), \Gamma_{init}).$$

Subcase 2. Suppose that $tr \neq \langle\rangle$.
This case is identical to Subcase 2 of the external choice case, above.

Conditional choice
Suppose that $Proc(t) =$ if $cond$ then $P(t)$ else $Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. If $cond$ is not in $Cond$, then it immediately evaluates to $True$ or $False$, in which case the result is implied by the inductive hypothesis for $P(t)$ or $Q(t)$, respectively. For a condition $cond$ in $Cond$ we perform a case analysis on the truth value of the evaluation of $cond$ within environments Γ_{init} and $\phi(\Gamma_{init})$.

Case 1. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = [\![cond]\!]_{\Gamma_{init}} = True$.
Then

$$(\phi(tr), X) \in failures(P(t), \phi(\Gamma_{init})) \quad \text{and} \quad tr \in traces(P(t), \Gamma_{init}).$$

In addition, assumption (iii), combined with the fact that $(Proc(t), \Gamma_{init}) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \Gamma_{init})$, implies that if P is a configuration such that $(P(t), \Gamma_{init}) \stackrel{tr}{\Longrightarrow} P$, then every output of type t of every event in $initials(P)$ is in $\{0\,..\,B-1\}$. Then, the inductive hypothesis for $P(t)$ implies that

$$(tr, X) \in failures(P(t), \Gamma_{init}).$$

Therefore,

$$(tr, X) \in \mathit{failures}(\mathit{Proc}(t), \Gamma_{\mathit{init}}).$$

Case 2. Suppose that $[\![\mathit{cond}]\!]_{\phi(\Gamma_{\mathit{init}})} = [\![\mathit{cond}]\!]_{\Gamma_{\mathit{init}}} = \mathit{False}$.
This case is like Case 1, above, with $Q(t)$ in place of $P(t)$.

Case 3. Suppose that $[\![\mathit{cond}]\!]_{\phi(\Gamma_{\mathit{init}})} = \mathit{True} \wedge [\![\mathit{cond}]\!]_{\Gamma_{\mathit{init}}} = \mathit{False}$.
Then

$$(\phi(tr), X) \in \mathit{failures}(P(t), \phi(\Gamma_{\mathit{init}})) \quad \text{and} \quad tr \in \mathit{traces}(Q(t), \Gamma_{\mathit{init}}).$$

However, $\mathit{Proc}(t)$ satisfies $\mathbf{RevPosConjEqT}^t_{\mathrm{F}}$, so $(Q(t), \phi(\Gamma_{\mathit{init}})) \sqsubseteq_{\mathrm{F}} (P(t), \phi(\Gamma_{\mathit{init}})$. Hence,

$$(\phi(tr), X) \in \mathit{failures}(Q(t), \phi(\Gamma_{\mathit{init}})).$$

In addition, assumption (iii), combined with the fact that $(\mathit{Proc}(t), \Gamma_{\mathit{init}}) \stackrel{\langle\rangle}{\Longrightarrow} (Q(t), \Gamma_{\mathit{init}})$, implies that if P is a configuration such that $(Q(t), \Gamma_{\mathit{init}}) \stackrel{tr}{\Longrightarrow} P$, then every output of type t of every event in $\mathit{initials}(P)$ is in $\{0 \,..\, B-1\}$. The inductive hypothesis for $Q(t)$ implies now that

$$(tr, X) \in \mathit{failures}(Q(t), \Gamma_{\mathit{init}}),$$

which implies that

$$(tr, X) \in \mathit{failures}(\mathit{Proc}(t), \Gamma_{\mathit{init}}).$$

Case 4. Suppose that $[\![\mathit{cond}]\!]_{\phi(\Gamma_{\mathit{init}})} = \mathit{False} \wedge [\![\mathit{cond}]\!]_{\Gamma_{\mathit{init}}} = \mathit{True}$.
This case is not possible, since *cond* is a conjunction of equality tests and for no function ϕ we can ever have $x = y$ and $\phi(x) \neq \phi(y)$.

Binding
Suppose that $\mathit{Proc}(t) = X(t)$ for some process identifier X and suppose that the global environment E maps X to some process definition P (i.e. $E(X) = P$). The result is implied by the inductive hypothesis for $P(t)$. ∎

The following theorem is our second key result of this chapter. It extends Theorem 5.2.15 to the stable failures model by establishing a threshold $\mathit{Thresh}_{\mathrm{F}}$ such that if $\mathit{Spec}(t)$ and $\mathit{Impl}(t)$ fulfil certain requirements, then, for all $B \geq \mathit{Thresh}_{\mathrm{F}}$, if $\mathit{Spec}(\{0 \,..\, B\}) \sqsubseteq_{\mathrm{F}} \phi(\mathit{Impl}(\{0 \,..\, n\})$ implies that $\mathit{Spec}(\{0 \,..\, n\}) \sqsubseteq_{\mathrm{F}} \mathit{Impl}(\{0 \,..\, n\})$ for all $n \geq B$.

Similarly as before, the specification side of this implication is independent of n and therefore remains constant as n varies. The implementation side, however, is dependent on n (even though it only communicates values of type t in the range

$\{0 \,..\, B\}$, thanks to the external renaming by ϕ). However, if we can produce a fixed process *Abstr* such that $Abstr \sqsubseteq_{\mathrm{F}} \phi(Impl(\{0 \,..\, n\})$ for all $n \geq B$ for some $B \geq Thresh_{\mathrm{F}}$ by construction, then it is enough to perform a single refinement check, namely $Spec(\{0 \,..\, \max\{Thresh_{\mathrm{F}}, B\}\}) \sqsubseteq_{\mathrm{F}} Abstr$, in order for Theorem 5.2.24 to imply that $Spec(\{0 \,..\, n\}) \sqsubseteq_{\mathrm{F}} Impl(\{0 \,..\, n\})$ for all $n \geq \max\{Thresh_{\mathrm{F}}, B\}$. Combining this with manual checks for all $n \in \{0 \,..\, \max\{Thresh_{\mathrm{F}}, B\} - 1\}$, we obtain the answer to whether $Spec(T) \sqsubseteq_{\mathrm{F}} Impl(T)$ for all instantiations T of type t.

Recall that, given a symbolic conditional event *cond* and an environment Γ, $[\![cond]\!]_\Gamma$ denotes the truth value of the proposition obtained from *cond* by substituting all free variables of type t with their corresponding values contained within Γ. We lift the definition of $[\![\cdot]\!]_\Gamma$ to symbolic traces without visible symbolic events in the following way. Given a symbolic trace σ in $(Cond \cup \{\tau\})^*$ we let $[\![\sigma]\!]_\Gamma$ be equal to $\bigwedge\{[\![cond]\!]_\Gamma \mid cond \text{ in } \sigma, cond \in Cond\}$. In addition, given a non-empty sequence *seq* we define *last*(*seq*) to be the last element of *seq*.

Theorem 5.2.24. (Extendibility of stable failures refinement of systems with replicated components)
Suppose that

(i) $Spec(t)$ *satisfies* ***SeqNorm*** *and* $\boldsymbol{RevPosConjEqT}^t_{\mathrm{F}}$*, and is divergence-free and has a finite alphabet for every finite instantiation of type* t*,*

(ii) $Impl(t)$ *satisfies* ***TypeSym****,*

(iii) no construct α *in* $Spec(t)$ *combines nondeterministic inputs of type* t *and deterministic input of any type, i.e. if* $\#\$^t(\alpha) > 0$*, then* $\#?(\epsilon) = 0$*,*

(iv) T *is an instantiation of type* t *such that* $\#T \geq B + 1$*, where* B *is as below,*

(v)
$$Thresh_{\mathrm{F}} = \max\{Thresh_{\mathrm{T}}, \max\{Thresh_{!^t}(\sigma, \Gamma) + Thresh_{?^t}(\sigma, \Gamma) \mid \sigma \in SymbolicTraces(Spec(t)) \wedge (\sigma = \langle\rangle \vee last(\sigma) \in Visible) \wedge \Gamma \in Env(T)\}\},$$
where

- $Thresh_{!^t}(\sigma, \Gamma)$ *counts the number of unique output variables of type* t *in all the visible symbolic events* ϵ *available in* $Spec(t)$ *immediately after* σ *such that all conditionals between the last symbolic event of* σ *and* ϵ *evaluate to True, i.e.*

  $$Thresh_{!^t}(\sigma, \Gamma) = \#\{x_i \mid Spec(t) \xrightarrow{\sigma}_s \xrightarrow{\rho}_s \xrightarrow{\epsilon}_s \wedge \rho \in (Cond \cup \{\tau\})^* \wedge [\![\rho]\!]_\Gamma = True \wedge \epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \in Visible \wedge i \in !^t(\epsilon)\},$$

- $Thresh_{?^t}(\sigma, \Gamma)$ *counts the number of (not necessarily unique) input variables of type* t *in all the visible symbolic events* ϵ *available in* $Spec(t)$ *immediately after* σ *such that all conditionals between the last symbolic event of* σ *and* ϵ *evaluate to True, i.e.*

$$Thresh_{?^t}(\sigma,\Gamma) = \Sigma\{\#?^t(\epsilon) \mid Spec(t) \xrightarrow{\sigma}_s \xrightarrow{\rho}_s \xrightarrow{\epsilon}_s \wedge \rho \in (Cond \cup \{\tau\})^* \\ \wedge \epsilon \in Visible \wedge [\![\rho]\!]_\Gamma = True\},$$

- *$Thresh_{\mathrm{T}}$ is as in Theorem 5.2.15,*

(vi) $B \geq Thresh_{\mathrm{F}}$, and

(vii) ϕ is a B-collapsing function.

Then, if $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$, then $Spec(T) \sqsubseteq_{\mathrm{F}} Impl(T)$.

Proof: Suppose that the refinement $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$ holds and assume for contradiction that $Spec(T) \not\sqsubseteq_{\mathrm{F}} Impl(T)$. Refinement in the stable failures model implies refinement in the traces model, so $Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(Impl(T))$. Then, by Theorem 5.2.15 (which is applicable since its assumptions are weaker than those of this theorem), $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$.

Consider a minimal counterexample (tr, X) to the refinement $Spec(T) \sqsubseteq_{\mathrm{F}} Impl(T)$, i.e.

$$(tr, X) \in failures(Impl(T)), \tag{5.20}$$

$$(tr, X) \notin failures(Spec(T)), \tag{5.21}$$

$$\forall\, e \in X \bullet (tr, X \setminus \{e\}) \in failures(Spec(T)). \tag{5.22}$$

Observe that there is such a minimal counterexample since we have assumed that specifications are divergence-freedom and have finite alphabets.

Combining (5.21) and (5.22) we obtain that for all events e in X there exists a state $P_e(T)$ such that

$$Spec(T) \stackrel{tr}{\Longrightarrow} P_e(T) \wedge P_e(T)\ ref\ (X) \setminus \{e\} \wedge P_e(T) \xrightarrow{e} . \tag{5.23}$$

This also means that every event in X is accepted in some stable state of $Spec(T)/tr$. Hence,

$$X \subseteq initials(Spec(T)/tr). \tag{5.24}$$

We now aim to show that X is dependent upon at most $Thresh_{\mathrm{F}}$ values from T, in a sense that we make precise below. We begin with two properties of X.

1. Firstly, we prove that X is closed under type t nondeterministic inputs of the specification, i.e. we suppose that $e = c.v_1 \ldots v_k \in X$ matches a unique construct α of $Spec(t)$ (uniqueness follows from Corollary 5.1.15) with $\#\$^t(\alpha) > 0$ (which, by assumption (iii), implies that $\#?(\alpha) = 0$) and show that

$$\forall\, v' : \{1\,..\,k\} \to Value \mid \\ (\forall\, i \in \$^t(\alpha) \bullet v'_i \in T) \wedge (\forall\, i \in \{1\,..\,k\} \setminus \$^t(\alpha) \bullet v'_i = v_i) \bullet \\ c.v'_1 \ldots v'_k \in X. \tag{5.25}$$

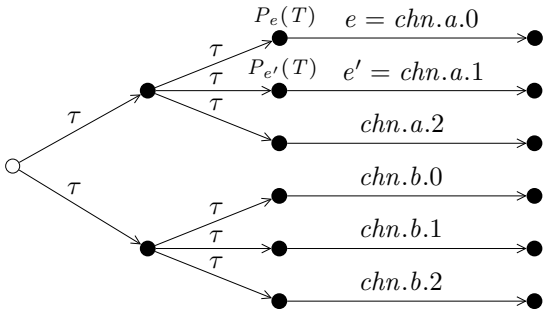


Figure 5.2: The LTS of $Proc(\{0,1,2\})$, where $Proc(t) = chn\$x{:}\{a,b\}\$y{:}t$.

Let v' be as in (5.25) and let $e' = c.v'_1 \ldots v'_k$. Assume for a contradiction that e' is not an event in X. Consider the same behaviour that leads to the stable state $P_e(T)$ of (5.23) where $X \setminus \{e\}$ is refused and e is accepted, except that the nondeterministic selections of α are resolved in a way such that the values v'_i are chosen instead of v_i for all $i \in \$^t(\alpha)$; call this stable state $P_{e'}(T)$ (see Figure 5.2 for an example). The initials of $P_{e'}(T)$ are the same[3] as those of $P_e(T)$, except they contain e' instead of e. Therefore, since $X \setminus \{e\}$ is refused in $P_e(T)$, $X \setminus \{e'\}$ must be refused in $P_{e'}(T)$. However, $e' \notin X$ by assumption, so X is refused in $P_{e'}(T)$, which contradicts (5.21).

2. Secondly, we show that X contains no pairs of events that differ only in values of deterministic inputs of any type, i.e. we assume that $e = c.v_1 \ldots v_k \in X$ is an event matching a unique construct α of $Spec(t)$ (uniqueness guaranteed by Corollary 5.1.15) with $\#?(\alpha) > 0$ (which, by assumption (iii), implies that $\#\$^t(\alpha) = 0$) and show that

$$\begin{aligned} &\forall\, v' : \{1\,..\,k\} \to Value \mid \\ &\quad (\forall\, i \in \{1\,..\,k\} \setminus ?(\alpha) \bullet v'_i = v_i) \wedge (\exists\, i \in ?(\alpha) \bullet v'_i \neq v_i) \bullet \\ &\qquad c.v'_1 \ldots v'_k \notin X. \end{aligned} \tag{5.26}$$

Let v' be as in (5.26) and assume for a contradiction that $e' = c.v'_1 \ldots v'_k$ is an event in X. Consider the state $P_e(T)$ of (5.23) where $X \setminus \{e\}$ is refused and e is available. Clearly e' is refused in this state, since we assumed it to be in X and hence in $X \setminus \{e\}$ (since $e \neq e'$). This is a contradiction since e and e' differ only in the values of deterministic inputs and hence e is available if and only if e' is.

Recall that tr is a trace of $Spec(T) = (Spec(t), \{\}, T)$; let $(Spec'(t), \Gamma, T)$ be the unique resulting configuration (with uniqueness following from Lemma 5.1.12), i.e.

$$(Spec(t), \{\}, T) \stackrel{s}{\longmapsto} (Spec'(t), \Gamma, T) \tag{5.27}$$

[3]This would not be true if specifications could contain constructs that combine nondeterministic selections over type t and deterministic inputs over any type. See Example 5.2.28 and Example 5.2.29 below.

for some sequence of concrete events s such that s does not end with a τ and $s \setminus \{\tau\} = tr$. Observe that

$$(Spec(t), \{\}, T) \stackrel{tr}{\Longrightarrow} (Spec'(t), \Gamma, T).$$

So, thanks to the uniqueness of $Spec'(t)$ and Γ, and since s does not end with a τ,

$$initials(Spec(T)/tr) = initials(Spec'(t), \Gamma, T).$$

Let $S = S_1 \cup S_2$, where

$$S_1 = \{v_i \mid e = c.v_1 \ldots v_k \in initials(Spec(T)/tr) \\ \wedge\ e \text{ matches a unique construct } \alpha \text{ of } Spec(t) \wedge i \in !^t(\alpha)\} \tag{5.28}$$

and

$$S_2 = \{v_i \mid c.v_1 \ldots v_k \in X \wedge i \in \{1 \,..\, k\} \wedge v_i \in T \\ \wedge\ \exists\, v' \in T \bullet c.v_1 \ldots v_{i-1}.v'.v_{i+1} \ldots v_k \notin X\}. \tag{5.29}$$

We now show that B is an upper bound on $\#S$.

Firstly, we deduce from the closure of X under type t nondeterministic inputs of the specification (clause 1, above) that S_2 contains no values of type t that come from nondeterministic selections of constructs of $Spec(t)$. Hence and by (5.24), S_2 is a subset of the set of values of type t in the events in $initials(Spec(T)/tr) = initials(Spec'(t), \Gamma, T)$ that come from deterministic inputs and outputs only. Observe that if a concrete event of the specification is obtained from a symbolic event using the translation rules of COSE, then all the preceding conditional symbolic events have to evaluate to *True* in some appropriate environments. Also, all conditional symbolic events occurring between any two visible symbolic events are always evaluated within the same environment. Therefore, when working out $initials(Spec'(t), \Gamma, T)$, we can ignore those initial visible symbolic events of $Spec'(t)$ that are preceded by a conditional symbolic event that evaluates to *False* in Γ. Finally, from (5.27) and the translation rules of COSE (see Section 4.4.1) we have that there exists $\sigma \in SymbolicTraces(Spec(t))$ such that either $\sigma = \langle\rangle$ or $last(\sigma) \in Visible$ and $Spec(t) \stackrel{\sigma}{\longmapsto}_s Spec'(t)$ (so $\sigma\ generates_{\{\}}\ s$). Hence, the number of type t values matched by deterministic inputs in the events in S_2 is at most

$$Thresh_{?^t}(\sigma, \Gamma) = \sum\{\#?^t(\epsilon) \mid Spec(t) \stackrel{\sigma}{\longmapsto}_s \stackrel{\rho}{\longmapsto}_s \stackrel{\epsilon}{\longrightarrow}_s \wedge\ \rho \in (Cond \cup \{\tau\})^* \\ \wedge\ \epsilon \in Visible \wedge [\![\rho]\!]_\Gamma = True\}.$$

Now, since we assumed no constants of type t in the definition of $Spec(t)$, any type t output value in S must come from some environment. Therefore, the total number of output values of type t in S can never be greater than the total number of different output variable names used in the constructs of $Spec(t)$ that are matched by the members of $initials(Spec(T)/tr)$, i.e. the total number of output values of type t in S is at most

$$Thresh_{!^t}(\sigma, \Gamma) = \\ \#\{x_i \mid Spec(t) \stackrel{\sigma}{\longmapsto}_s \stackrel{\rho}{\longmapsto}_s \stackrel{\epsilon}{\longrightarrow}_s \wedge\ \rho \in (Cond \cup \{\tau\})^* \wedge [\![\rho]\!]_\Gamma = True \\ \wedge\ \epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \in Visible \wedge i \in !^t(\epsilon)\}.$$

Summarising the last two paragraphs: all elements of S either match deterministic inputs in the events in S_2 (at most $Thresh_{?^t}(\sigma, \Gamma)$ such values), or match outputs in either S_1 or S_2 (at most $Thresh_{!^t}(\sigma, \Gamma)$ such values). Therefore,

$$\begin{array}{ll} & \#S \\ \leq & Thresh_{!^t}(\sigma, \Gamma) + Thresh_{?^t}(\sigma, \Gamma) \\ \leq & \max\{\, Thresh_{!^t}(\sigma', \Gamma') + Thresh_{?^t}(\sigma', \Gamma') \mid \sigma' \in SymbolicTraces(Proc(t)) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, (\sigma' = \langle\rangle \vee last(\sigma') \in Visible) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\, \Gamma' \in Env(T)\} \\ \leq & Thresh_{\mathrm{F}} \\ \leq & B. \end{array}$$

Let $\pi : T \to T$ be a bijection that maps S into $\{0 \,..\, B-1\}$. Then, using Remark 5.2.6, we can infer from (5.20) and (5.21) that

$$(\pi(tr), \pi(X)) \in failures(Impl(T)) \tag{5.30}$$

and

$$(\pi(tr), \pi(X)) \notin failures(Spec(T)). \tag{5.31}$$

Now, by the denotational semantics of renaming [Ros97, Chapter 8],

$$\begin{array}{l} failures(\phi(Impl(T))) \\ \quad = \{(\phi(tr'),\, Y) \mid (tr', \phi^{-1}(Y)) \in failures(Impl(T))\}. \end{array} \tag{5.32}$$

Let $c.v_1 \ldots v_k$ be an event in X. Let i be in $\{1 \,..\, k\}$ such that v_i is of type t. Our construction of S implies that either (1) v_i is in S, in which case $\pi(v_i)$ is in $\{0 \,..\, B-1\}$, or (2) v_i matches a nondeterministic input of type t, in which case the closure of X under nondeterministic inputs of the specification (clause 1 on p. 131) implies that for all values v' in T, $c.v_1 \ldots v_{i-1}.v'.v_{i+1} \ldots v_k$ is in X. Therefore, if $c.w_1 \ldots w_k$ is an event in $\pi(X)$, then for every i in $\{1 \,..\, k\}$ such that w_i is of type t, we have that either w_i is in $\{0 \,..\, B-1\}$ or $c.w_1 \ldots w_{i-1}.w'.w_{i+1} \ldots w_k$ is in $\pi(X)$ for all values w' in T. This, thanks to the definition of ϕ, means that $\forall\, e \in \phi(\pi(X)) \bullet \phi^{-1}(e) \subseteq \pi(X)$. This implies that $\phi^{-1}(\phi(\pi(X))) \subseteq \pi(X)$, which trivially implies that

$$\phi^{-1}(\phi(\pi(X))) = \pi(X).$$

Combining with (5.30) and (5.32), we get that

$$(\phi(\pi(tr)), \phi(\pi(X))) \in failures(\phi(Impl(T))).$$

However, $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$, so

$$(\phi(\pi(tr)), \phi(\pi(X))) \in failures(Spec(\phi(T))). \tag{5.33}$$

We now show that

$$(\phi(\pi(tr)), \pi(X)) \in failures(Spec(T)). \tag{5.34}$$

Firstly, (5.33) implies that there exists a configuration $(P(t), \Gamma, \phi(T))$ such that

$$\mathit{Spec}(\phi(T)) = (\mathit{Spec}(t), \{\}, \phi(T)) \xrightarrow{\phi(\pi(tr))} (P(t), \Gamma, \phi(T)) \ \mathit{ref}\ \phi(\pi(X)). \tag{5.35}$$

Let $e = c.v_1 \ldots v_k$ be in $\phi(\mathit{initials}(P(t), \Gamma, T))$ and let α be the unique construct of $\mathit{Spec}(t)$ that e matches (with uniqueness following from Corollary 5.1.15). Then there exists a function $v' : \{1 \,..\, k\} \to \mathit{Value}$ such that

$$e' = c.v'_1 \ldots v'_k \in \mathit{initials}(P(t), \Gamma, T) \tag{5.36}$$

and $\phi(e') = e$, i.e.

$$\forall\, i \in \{1 \,..\, k\} \bullet \phi(v'_i) = v_i. \tag{5.37}$$

We know that $(P(t), \Gamma, \phi(T))$ must be a stable state, as otherwise it would not be able to refuse $\phi(\pi(X))$. This means that all values of nondeterministic selections of constructs that generate the events in $\mathit{initials}(P(t), \Gamma, \phi(T))$ had been chosen before this state was reached (and are necessarily in $\phi(T)$). Hence and since $\Gamma \in \mathit{Env}(\phi(T))$ implies $\Gamma \in \mathit{Env}(T)$, we have that $\mathit{initials}(P(t), \Gamma, \phi(T))$ and $\mathit{initials}(P(t), \Gamma, T)$ are the same, except for values of deterministic inputs of type t. Formally,

$$\begin{aligned} &\mathit{initials}(P(t), \Gamma, \phi(T)) = \\ &\quad \{c.w_1 \ldots w_k \mid c.w'_1 \ldots w'_k \in \mathit{initials}(P(t), \Gamma, T) \text{ matches construct } \alpha' \\ &\qquad\qquad \wedge\ w \in \{1 \,..\, k\} \to \mathit{Value} \wedge (\forall\, i \in \{1 \,..\, k\} \setminus ?^t(\alpha') \bullet w_i = w'_i) \\ &\qquad\qquad \wedge\ \forall\, i \in ?^t(\alpha') \bullet w_i \in \phi(T)\}. \end{aligned} \tag{5.38}$$

Also, since $\Gamma \in \mathit{Env}(\phi(T))$, all values of type t used in the events of $\mathit{initials}(P(t), \Gamma, T)$ and that match nondeterministic selections or outputs, are in $\phi(T) = \{0 \,..\, B\}$:

$$\forall\, i \in \$^t(\alpha) \cup !^t(\alpha) \bullet v'_i \in \{0 \,..\, B\},$$

which, thanks to the properties of ϕ, and combined with the fact that for all non-t values val, $\phi(\mathit{val}) = \mathit{val}$, gives us that

$$\forall\, i \in \{1 \,..\, k\} \setminus ?^t(\alpha) \bullet \phi(v'_i) = v'_i.$$

Hence and by the definition of v' (5.37),

$$\forall\, i \in \{1 \,..\, k\} \setminus ?^t(\alpha) \bullet v_i = v'_i. \tag{5.39}$$

In addition, since $c.v_1 \ldots v_k$ is in $\phi(\mathit{initials}(P(t), \Gamma, T))$, it must be that

$$\forall\, i \in ?^t(\alpha) \bullet v_i \in \phi(T). \tag{5.40}$$

Combining (5.36), (5.38), (5.39) and (5.40) we get that e is in $\mathit{initials}(P(t), \Gamma, \phi(T))$. Hence

$$\phi(\mathit{initials}(P(t), \Gamma, T)) \subseteq \mathit{initials}(P(t), \Gamma, \phi(T)). \tag{5.41}$$

Conversely, let $e = c.v_1 \ldots v_k$ be in $\mathit{initials}(P(t), \Gamma, \phi(T))$. From Proposition 5.1.18 we can infer that

$$\mathit{initials}(P(t), \Gamma, \phi(T)) \subseteq \mathit{initials}(P(t), \Gamma, T),$$

so e is in $\mathit{initials}(P(t), \Gamma, T)$. Hence, $\phi(e)$ is in $\phi(\mathit{initials}(P(t), \Gamma, T))$. However, for all i in $\{1 \,..\, k\}$, v_i is either a value of a non-t type or it is a value in $\phi(T)$. Hence,

$$\forall\, i \in \{1 \,..\, k\} \bullet \phi(v_i) = v_i,$$

so $\phi(e) = e$ and therefore $e \in \phi(\mathit{initials}(P(t), \Gamma), T)$. Hence,

$$\mathit{initials}(P(t), \Gamma, \phi(T)) \subseteq \phi(\mathit{initials}(P(t), \Gamma, T)).$$

Combining the above with (5.41) we have that

$$\phi(\mathit{initials}(P(t), \Gamma, T)) = \mathit{initials}(P(t), \Gamma, \phi(T)). \tag{5.42}$$

We now aim to show that

$$(P(t), \Gamma, T) \;\mathit{ref}\; \pi(X). \tag{5.43}$$

Suppose for a contradiction that there exists an event x in $\pi(X) \cap \mathit{initials}(P(t), \Gamma, T)$. Then (5.42) implies that $\phi(x) \in \phi(\pi(X)) \cap \mathit{initials}(P(t), \Gamma, \phi(T))$. This means that $\phi(\pi(X)) \cap \mathit{initials}(P(t), \Gamma, \phi(T))$ is non-empty, which contradicts (5.35). Hence, (5.43) holds. Finally, applying Corollary 5.1.19 to (5.35) we have that

$$(\mathit{Spec}(t), \{\}, T) \xrightarrow{\phi(\pi(tr))} (P(t), \Gamma, T).$$

This, combined with (5.43) implies that (5.34) holds.

We now seek to apply Proposition 5.2.23 to $\pi(tr)$ and $\pi(X)$. From (5.30), $\pi(tr) \in \mathit{traces}(\mathit{Impl}(T))$. However, we have already shown that $\mathit{Spec}(T) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T)$, so

$$\pi(tr) \in \mathit{traces}(\mathit{Spec}(T)) = \mathit{traces}(\mathit{Spec}(t), \{\}, T).$$

This gives us condition (i) of Proposition 5.2.23. Equation (5.34) (and the fact that $\phi(\{\}) = \{\}$) gives us condition (ii). In addition, our definition of S_1 (5.28), combined with the definition of π, implies that every output value of type t of every event in $\mathit{initials}(\mathit{Spec}(T)/\pi(tr))$ is in $\{0 \,..\, B-1\}$, which gives us condition (iii). Hence, we can infer from it that

$$(\pi(tr), \pi(X)) \in \mathit{failures}(\mathit{Spec}(t), \{\}, T) = \mathit{failures}(\mathit{Spec}(T)).$$

This is a contradiction to (5.31), which completes our proof. ∎

Some observations related to Theorem 5.2.24 are now in order.

Remark 5.2.25. For every verification problem, the value of $\mathit{Thresh}_{\mathrm{F}}$ in Theorem 5.2.15 depends only on the specification.

Remark 5.2.26. For all specifications with a finite SSLTS, the value of $\mathit{Thresh}_{\mathrm{F}}$ in Theorem 5.2.15 can be calculated in a finite amount of time. The term $\mathit{Thresh}_{\mathrm{T}}$ can be calculated as in Remark 5.2.19. The other term can be obtained by calculating $\mathit{Thresh}_{!^t}(\sigma, \Gamma) + \mathit{Thresh}_{?^t}(\sigma, \Gamma)$ for each symbolic state that is either the initial state or that has an incoming visible transition.

Remark 5.2.27. The values of $Thresh_{!t}(\sigma, \Gamma)$ and $Thresh_{?t}(\sigma, \Gamma)$, used in Theorem 5.2.24, have simple upper bounds: $W^{\mathtt{Spec}}$ and $I_?^{\mathtt{Spec}}$, respectively, defined as in [Ros97, Chapter 15]:

- $W^{\mathtt{Spec}}$ is the maximum number of type t values that the specification needs to store for use within subsequent events,
- $I_?^{\mathtt{Spec}}$ is the maximum number of type t values that any state of the specification can deterministically input through *all* of its initially available symbolic events.

Then, $\max\{Thresh_{\mathrm{T}}, W^{\mathtt{Spec}} + I_?^{\mathtt{Spec}}\}$ is an upper bound for $Thresh_{\mathrm{F}}$. In practice this approximation may be improved by counting the number of values that were double-counted in obtaining $W^{\mathtt{Spec}} + I_?^{\mathtt{Spec}}$. This can result in up to halving the calculated upper bound for $Thresh_{\mathrm{F}}$ and often gives precisely the value of $Thresh_{\mathrm{F}}$.

At first, condition (iii) of Theorem 5.2.24 — that no construct of the specification combines nondeterministic inputs of type t and deterministic inputs of any type — may seem somewhat arbitrary. However, without it, there are specifications $Spec(t)$ and implementations $Impl(t)$ such that no threshold exists: for all values of B, there exists an instantiation T of type t such that $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$ and yet $Spec(T) \not\sqsubseteq_{\mathrm{F}} Impl(T)$. The following examples illustrate such pairs of processes, where a nondeterministic input of type t is combined with a deterministic input of type t (Example 5.2.28) and with a deterministic input of a non-t type (Example 5.2.29).

Example 5.2.28. Let

$$Spec(t) = c\$x{:}t?y{:}t \rightarrow STOP$$

and

$$Impl(t) = \Box\, y{:}t \bullet c?x{:}(t \setminus \{y\})!y \rightarrow STOP.$$

Note, in particular that $Impl(t)$ satisfies **TypeSym**.

Let B be an arbitrary positive number, and let ϕ be as in the statement of Theorem 5.2.24. Let $T = \{0\,..\,N\}$, where $N \geq B + 1$. It is easy to see that $traces(\phi(Impli(T))) \subseteq traces(Spec(\phi(T)))$. Further, whatever value $Impl(T)$ chooses for y, $(T \setminus \{y\}) \cap \{B\,..\,N\} \neq \{\}$; hence $(\langle\rangle, \{c.B.B\}) \notin failures(\phi(Impl(T)))$. This helps to see that

$$\begin{array}{ll} & failures(\phi(Impl(T))) \\ = & \{(\langle\rangle, X) \mid X \subseteq \{c.x.x \mid x \in \{0\,..\,B-1\}\}\} \\ & \cup\{(\langle c.x.y\rangle, X) \mid y \in \{0\,..\,B\} \wedge x \in \{0\,..\,B\} \setminus \{y\} \wedge X \subseteq \Sigma\} \\ \subseteq & \{(\langle\rangle, X) \mid X \subseteq \{c.x.y \mid x \in \{0\,..\,B\} \setminus \{p\} \wedge y \in \{0\,..\,B\}\} \wedge p \in \{0\,..\,B\}\} \\ & \cup\{(\langle c.x.y\rangle, X) \mid x, y \in \{0\,..\,B\} \wedge X \subseteq \Sigma\} \\ = & failures(Spec(\phi(T))). \end{array}$$

Hence,

$$Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T)).$$

However,

$$(\langle\rangle, \{c.x.x \mid x \in T\}) \in \mathit{failures}(\mathit{Impl}(T)) \setminus \mathit{failures}(\mathit{Spec}(T)),$$

so $\mathit{Spec}(T) \not\sqsubseteq_{\mathrm{F}} \mathit{Impl}(T)$.

End of example.

Example 5.2.29. Let B be an arbitrary positive integer, and $Y = \{y_1, y_2\}$ a type other than t of size 2. Let

$$\mathit{Spec}(t) = c\$x{:}t?y{:}Y \rightarrow \mathit{STOP}$$

and

$$\mathit{Impl}_B(t) = \Box\, y{:}Y \bullet \left(\sqcap X \subseteq t \wedge \#X = B + 1 \bullet c?x{:}X!y \rightarrow \mathit{STOP}\right).$$

Note, in particular, that $\mathit{Impl}(t)$ satisfies **TypeSym**.

Let ϕ be as in the statement of Theorem 5.2.24 and let $T = \{0 \,..\, N\}$, where $N \geq 2B+1$. It is easy to see that $\mathit{traces}(\phi(\mathit{Impl}(T))) \subseteq \mathit{traces}(\mathit{Spec}(\phi(T)))$. Further, whatever value $\mathit{Impl}_B(T)$ chooses for X, $X \cap \{B \,..\, N\} \neq \{\}$; hence $(\langle\rangle, \{c.B.y\}) \notin \mathit{failures}(\phi(\mathit{Impl}_B(T)))$ for every $y \in Y$. This helps to see that

$$\begin{array}{ll} & \mathit{failures}(\phi(\mathit{Impl}_B(T))) \\ \subseteq & \{(\langle\rangle, R) \mid R \subseteq \{c.x.y \mid x \in \{0 \,..\, B-1\} \wedge y \in Y\}\} \\ & \cup\{(\langle c.x.y\rangle, R) \mid x \in \{0 \,..\, B\} \wedge y \in Y \wedge R \subseteq \Sigma\} \\ \subseteq & \{(\langle\rangle, R) \mid R \subseteq \{c.x.y \mid x \in \{0 \,..\, B\} \setminus \{p\} \wedge y \in Y\} \wedge p \in \{0 \,..\, B\}\} \\ & \cup\{(\langle c.x.y\rangle, R) \mid x \in \{0 \,..\, B\} \wedge y \in Y \wedge R \subseteq \Sigma\} \\ = & \mathit{failures}(\mathit{Spec}(\phi(T))). \end{array}$$

Hence,

$$\mathit{Spec}(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(\mathit{Impl}_B(T)).$$

However, suppose the two values chosen for X when $y = y_1$ and $y = y_2$ are disjoint (this is possible since $\#T \geq 2B + 2$). Then, $\mathit{Impl}_B(T)$ has an initial failure $(\langle\rangle, R)$ such that for each $x \in T$, there is some z such that $c.x.z \in R$; this is not a failure allowed by $\mathit{Spec}(T)$, so $\mathit{Spec}(T) \not\sqsubseteq_{\mathrm{F}} \mathit{Impl}_B(T)$.

End of example.

Example 5.2.30. Recall the server/client system from Example 5.2.22. Suppose we now want to model slightly different clients, which, in addition to being able to communicate some data between themselves, can also use channel *agree* to agree on the identity of some arbitrary client, i.e. we modify the definition of CLIENT' to be the following (and we add suitable events using the *agree* channel to the alphabets of clients).

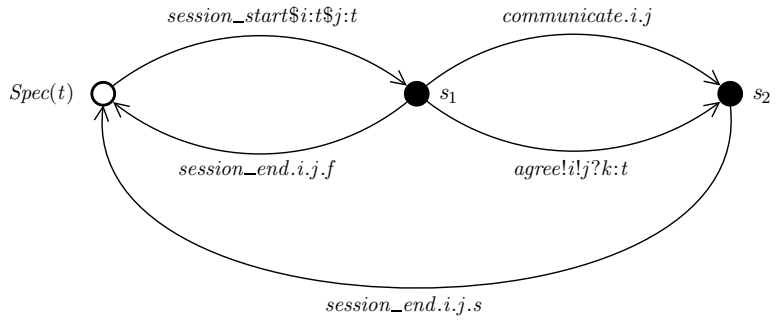


Figure 5.3: SSLTS of the *Spec*(t) process definition from Example 5.2.22.

$$
\begin{aligned}
&CLIENT'(i, j, principal, t) = \\
&\quad \text{if } principal = yes \\
&\qquad \text{then } session\_end.i.j.f \rightarrow CLIENT(i, t) \\
&\qquad\quad \Box \\
&\qquad\quad communicate.i.j \rightarrow session\_end.i.j.s \rightarrow CLIENT(i, t) \\
&\qquad\quad \Box \\
&\qquad\quad agree!i!j?k{:}t \rightarrow session\_end.i.j.s \rightarrow CLIENT(i, t) \\
&\qquad \text{else } session\_end.j.i.f \rightarrow CLIENT(i, t) \\
&\qquad\quad \Box \\
&\qquad\quad communicate.j.i \rightarrow session\_end.j.i.s \rightarrow CLIENT(i, t) \\
&\qquad\quad \Box \\
&\qquad\quad agree!j!i?k{:}t \rightarrow session\_end.j.i.s \rightarrow CLIENT(i, t)
\end{aligned}
$$

In addition to verifying that there is never more than one established connection within the system (as in Example 5.2.22), we now also want to check all the communication-related events (i.e. those using channels *session_start*, *communicate*, *agree* and *session_end*) are available in the right states. Therefore we do not hide these events in our implementation, i.e. we now let

$$
\begin{aligned}
Impl(t) = (SERVER(t) &\underset{\{|request\_peer, peer\_id, session\_start, session\_end|\}}{\parallel} CLIENTS(t)) \\
&\setminus \{|\ request\_peer, peer\_id\ |\}
\end{aligned}
$$

and

$$
\begin{aligned}
Spec(t) = session\_start\$i{:}t\$j{:}t \\
&\rightarrow (session\_end.i.j.f \rightarrow Spec(t) \\
&\quad \Box \\
&\quad communicate.i.j \rightarrow session\_end.i.j.s. \rightarrow Spec(t) \\
&\quad \Box \\
&\quad agree!i!j?k{:}t \rightarrow session\_end.i.j.s \rightarrow Spec(t)).
\end{aligned}
$$

Firstly, we have that $Thresh_{\mathrm{T}} = 2$. In order to establish $Thresh_{\mathrm{F}}$, we use the SSLTS of *Spec*(t) (Figure 5.3) and count the number of deterministic inputs of type t and outputs of type t every state with a visible symbolic event available.

- state $Spec(t)$, $\sigma = \langle\rangle$, any Γ:
  $Thresh_{!^t}(\sigma, \Gamma) = 0$, $Thresh_{?^t}(\sigma, \Gamma) = 0$,

- state s_1, $\sigma = \langle session\_start\$i{:}t\$j{:}t\rangle$, any Γ:
  $Thresh_{!^t}(\sigma, \Gamma) = 2$, $Thresh_{?^t}(\sigma, \Gamma) = 1$,

- state s_2, $\sigma = \langle session\_start\$i{:}t\$j{:}t, communicate.i.j\rangle$, any Γ:
  $Thresh_{!^t}(\sigma, \Gamma) = 2$, $Thresh_{?^t}(\sigma, \Gamma) = 0$

- state s_2, $\sigma = \langle session\_start\$i{:}t\$j{:}t, agree!i!j?k{:}t\rangle$, any Γ:
  $Thresh_{!^t}(\sigma, \Gamma) = 2$, $Thresh_{?^t}(\sigma, \Gamma) = 0$

Any other symbolic trace σ ends in one of the symbolic states listed above, so there cannot be any other values of $Thresh_{!^t}(\sigma, \Gamma)$, $Thresh_{?^t}(\sigma, \Gamma)$. Hence, $Thresh_{\mathrm{F}} = \max\{Thresh_{\mathrm{T}}, 3\} = 3$ in this case. Then, if we can find a process $Abstr$ such that $Spec(\{0\,..\,3\}) \sqsubseteq_{\mathrm{T}} Abstr$ and $Abstr \sqsubseteq_{\mathrm{T}} \phi(Impl(0\,..\,n))$ for all $n \geq 3$, then $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$ for all T of size 4 or more. The checks for $T = \{0\}$, $T = \{0, 1\}$ and $T = \{0, 1, 2\}$ can be then performed directly to complete the answer as to whether the implementation satisfies the specification, irrespectively of the instantiation of their parameters.

End of example.

5.3 Conclusions

In this paper we presented our type reduction theory, which, in many respects, is similar to Lazić's data independence theory for CSP [LR98]. Both of these theories aim to prove that

$$Spec(T) \sqsubseteq Impl(T)$$

for all sufficiently large T. However, the main difference is that while data independence allows one to deduce the above given that

$$Spec(\phi(T)) \sqsubseteq Impl(\phi(T)), \tag{5.44}$$

where ϕ is a B-collapsing function (see Definition 5.0.1) for some sufficiently large B, our theory requires that

$$Spec(\phi(T)) \sqsubseteq \phi(Impl(T)). \tag{5.45}$$

Observe that the refinement check in (5.44) is independent of T for all T of size at least $B + 1$, while the one in (5.45) is not. Hence, our theory does not, on its own, solve the problem of an infinite number of refinement checks needed to solve a given parameterised verification problem. In practice, the results presented in this chapter need to be combined with a suitable abstraction method that produces processes $Abstr$ that are independent of the instantiation of type t and such that $Abstr \sqsubseteq \phi(Impl(T))$ for all sufficiently large T by construction. Then, thanks to transitivity of refinement, verifying that $Abstr$ refines $Spec(t)$ implies that (5.45) holds for all types T large

enough. In Chapter 6 we will present techniques, based on counter abstraction, that produce such abstract models for use within the traces model.

Our type reduction theory heavily relies on the **SeqNorm** condition (defined in Section 4.1.2). In Section 5.1 we presented a number of consequences of this condition. The results imply a great deal of regularity for all specifications that we consider (as they are all required to satisfy **SeqNorm**) and allow us to deduce certain behaviours of $Spec(T)$, given some corresponding behaviours of $Spec(\phi(T))$.

In Section 5.2.1 we defined the following conditions:

- **PosConjEqT**^{t} (to be satisfied in Chapter 6 by controller processes) and **RevPosConjEqT**^{t} (to be satisfied by all specifications under our consideration), both of which limit the influence of conditionals on t on the behaviour on a given process,
- **NoEqT**^{t}, which bans conditionals on t, and
- **TypeSym** (to be satisfied by all implementations under our consideration), which can be seen as a weaker version of data independence and ensures invariance of process behaviours under permutations of the distinguished type.

Since **TypeSym** was defined semantically, we also provided a set of sufficient syntactic conditions that imply **TypeSym**. Finally, in Sections 5.2.2 and 5.2.3 we presented the main results of our theory for the traces and stable failures models, respectively. In both cases we provided a formula, dependant only on the specification, for calculating the minimum size of the collapsed type.

5.3.1 Automation

We can automatically check process syntaxes for the syntactic requirements of data independence (Definition 4.1.1) and **SeqNorm** (Definition 4.1.4). Each such check is guaranteed to terminate. Checking for the semantic requirements of the **TypeSym** condition (Definition 5.2.1) is difficult in practice due to the universal quantification over all instantiations of the type parameter t. However, we can automatically verify implementation definitions as to whether they satisfy the five simple semantic conditions of Proposition 5.2.2 and infer **TypeSym**. When doing so, one needs to remember the possibility of false negatives. Such syntactic checks always terminate.

Checking for **RevPosConjEqT**^{t} satisfiability (Definition 5.2.8) is the most problematic when it comes to automation. The problem lies in the universal quantification over all instantiations of the parameter variables of the arguments of conditional choices. Currently, it is left to the user to provide a proof that for every conditional choice of the form "if $cond$ then $P(x_1, \ldots, x_k)$ else $Q(x_1, \ldots, x_k)$" in a given specification, where $cond \in Cond$, $cond$ is a positive conjunction of equality tests on t and $Q(v_1, \ldots, v_k) \sqsubseteq P(v_1, \ldots, v_k)$ for all values $v_1, \ldots, v_k$. In general, the problem of **RevPosConjEqT**^{t} satisfiability is undecidable, since a general (undecidable) PMCP problem of the form $Spec(x) \stackrel{?}{\sqsubseteq} Impl(x)$, where x is a parameter, can be reduced to checking whether

$in?i{:}x?j{:}x \rightarrow \text{if } i = j \text{ then } Impl(x) \text{ else } Spec(x)$

satisfies **RevPosConjEqT**^{t}. However, in most practical situations it is not too difficult to provide a convincing proof that, regardless of parameters, the "then" branch of every conditional choice on t forms a refinement of its "else" branch, as often the branches are similar, except for the use of operators that introduce different levels of nondeterminism (e.g. using $\sqcap$ in the positive branch versus $\Box$ in the negative one).

As noted in Remark 5.2.19 and Remark 5.2.26, the calculation of thresholds in Theorem 5.2.15 and Theorem 5.2.24 can be fully automated. Termination is guaranteed for all specifications with finite-state SSLTSs.

5.3.2 Multiple distinguished types

Throughout this chapter we assumed the presence of a single distinguished type t. It is not overly difficult to extend our techniques to any finite number of distinguished types, say $t_1, t_2, \ldots, t_n$, provided all of them are pairwise independent. All requirements are extended in the natural way, e.g. each specification $Spec(t_1, t_2, \ldots, t_n)$ must now be data independent in all of the n types and each implementation $Impl(t_1, \ldots t_n)$ must satisfy **TypeSym** with respect to each of t_1, $t_2, \ldots, t_n$. The threshold in each of Theorem 5.2.15 and Theorem 5.2.24 is then replaced by a tuple of thresholds $(Thresh_1, Thresh_2, \ldots, Thresh_n)$, where each $Thresh_i$ is a threshold for the collapsing of the values of type t_i.

Chapter 6

Counter abstraction of systems with full identity awareness

In this chapter we return to the following problem, which we attempted to solve in Chapter 3.

> *Given a concurrent system Impl(T), consisting of $\#T$ similar processes, and a specification Spec(T), is it true that Impl(T) satisfies Spec(T) for all sizes of T?*

In Section 3.2 and Section 3.3 we showed how to build counter abstraction models of systems consisting of an unbounded number of node processes, none of which contains any node identifiers within its definition. However, in the majority of real world applications node processes do, in fact, use their own identity and/or identities of other nodes. The biggest problem this creates is the fact that the alphabets of processes can be unboundedly large. In this section we address this issue and show how to model check such systems by extending the standard counter abstraction methods to implementations with node processes that use node identifiers in their definitions. The procedure consists of two steps. Firstly, we reduce a given system with an unbounded number of node processes, each with an unbounded alphabet, to a counter-based system of an unbounded size. The counter-based system has a fixed-size alphabet, but is still dependent on an instantiation T of the distinguished type t, since each counter is allowed to take values that depend on the size of T. The second step is an application of counter abstraction techniques with thresholds, similar to those described in Section 3.3.

The main changes from Chapter 3 include the following.

- Nodes run in alphabetised parallel, with alphabets parameterised by node identities and their type.
- Nodes can use their own identities to model private communication.
- Nodes can use their own identities, together with identities of other nodes to model two-way synchronisation.

- Nodes can pass identities of other nodes between themselves or to/from a controller process and/or the environment.
- A collapsing function, ϕ, is used to collapse all node identities to a finite (and usually small) set of values.
- Only the traces model is considered.

The rest of this chapter is structured as follows. In Section 6.1 we state our assumptions and provide some useful general observations. Section 6.2 describes the idea of counter abstraction with extensions to node identifiers within definitions of processes. Counters are allowed to take non-negative integer values, bounded by the size of T. Therefore, the obtained models are still dependent upon T. We provide separate abstract models for systems with nodes that do not contain equality or inequality tests on t (Section 6.3) and for those that might contain such tests (Section 6.4). In both cases we prove that the abstract model forms traces anti-refinement of the parallel composition of all node processes, renamed under ϕ. In Section 6.5 we present a method for obtaining abstract models that are independent of T by using threshold functions that limit the range of values that counters can attain. In Section 6.5.2 we establish that threshold-based counter abstraction models form traces anti-refinements of ϕ-renamed parallel compositions of all nodes. Finally, Section 6.6 extends the refinement results to implementations in which node processes run in parallel with a controller process, possibly with some communication hidden. We also link these result with the type reduction theory (Chapter 5) to allow us to perform a single refinement check in order to deduce refinement of infinitely many specification/implementation pairs.

6.1 Preliminaries

For the rest of this chapter we let B be a fixed, non-negative integer. B will be a parameter used in the construction of abstract models, where it will describe the number of node processes that are modelled explicitly; we will explain the reasons for this later in this section. The value of B for a given implementation is determined by the type reduction theory we described in Chapter 5 (Theorem 6.6.8 will choose B to be as in Theorem 5.2.15).

Throughout this chapter, we will require synchronisation and hiding sets used in nodes' contexts to be constructed in a "regular" manner (with respect to the distinguished type), in the sense defined as follows.

Definition 6.1.1. We define the set definition $\{c.x_1 \ldots x_n \mid x_1 \in X_1(t), \ldots, x_n \in X_n(t)\}$, where c is some channel name, to be a *simply polymorphic in t* if $x_1, \ldots, x_n$ are all distinct, and for each i in $\{1 \,..\, n\}$, either $X_i(t) = t$ or $X_i(t)$ is not related to t in any way. We define the union of any number of set definitions that are simply polymorphic in T and that use distinct channel names to be *polymorphic in t*.

The implementations we consider in this chapter are of the form

$$Impl(t) = \Big((Nodes(t) \setminus X(t)) \underset{Y(t)}{\parallel} Ctrl(t) \Big) \setminus Z(t), \tag{6.1}$$

where

- $Nodes(t)$ is of the form $\|\ myId \in t \bullet [A(myId, t)]\ \mathcal{N}_{myId}(t)$,
- $\mathcal{N}_{myId}(t)$ models a single, finite-state node with identity $myId$ and with awareness of all node identities in t; we assume that $\mathcal{N}_{myId}(t)$ satisfies **SeqNorm** (see Section 4.1.2) and never changes the value of $myId$ (i.e. it cannot use constructs of the form $?myId$ or $\mathcal{N}_{otherId}(t)$, where $myId \neq otherId$); we assume that all node processes are generated from a single template and that no constants of type t are used within $Impl(t)$,
- $Ctrl(t)$ is a data independent controller process that satisfies $\mathbf{PosConjEqT}^t_{\mathrm{T}}$ (see Section 5.2.1),
- $X(t)$, $Y(t)$ and $Z(t)$ are definitions of sets of events that are polymorphic in t, and
- $A(myId, t)$ is a set of events that satisfies the conditions of Proposition 5.2.2.

Most parameterised systems one encounters in practice can be written in the above form.

Remark 6.1.2. Given $\mathcal{N}_{myId}(t)$ defined as above, Proposition 5.2.2 implies that $Impl(t)$ satisfies **TypeSym** (see Example 5.2.4).

The problem of uniform verification of implementations defined as in (6.1) with specifications satisfying **SeqNorm** (Definition 4.1.4) and $\mathbf{RevPosConjEqT}^t_{\mathrm{T}}$ (Definition 5.2.8) is undecidable. This follows from the following, stronger result.

Theorem 6.1.3. *The problem of uniform verification of* $|||\ i \in t \bullet \mathcal{N}_{myId}(t)$ *with a specification satisfying* $\boldsymbol{SeqNorm}$ *and* $\boldsymbol{RevPosConjEqT}^t_{\mathrm{T}}$*, where* $\mathcal{N}_{myId}(t)$ *is defined as above, is undecidable.*

Proof sketch: We can reduce the Halting Problem [Sip96] to the problem of uniform verification of $|||\ i \in t \bullet \mathcal{N}_{myId}(t)$ as follows. We use $|||\ i \in t \bullet \mathcal{N}_{myId}(t)$ to simulate a Turing machine that uses at most $\#t$ cells of its tape. This construction[1] can be outlined as follows. Each node process simulates one cell of the tape and remembers its contents. Once the tape head has visited a cell, the corresponding node remembers its left-hand neighbour (except for the node simulating the left-most cell). New cells are added to this linked-list on the fly whenever the simulated Turing machine needs to use an additional cell of the tape. When a node simulates the cell that is the current position of the head, it checks if the current state is an accepting state; if so then performs the event $halt.myId$ and halts; otherwise it communicates the new control state to an appropriate node, as determined by the direction of the head movement.

Given the above construction, the property "the Turing machine halts after using at most $\#t$ cells" is equivalent to the following refinement, which ensures that no *halt* event can be performed.

$$STOP \sqsubseteq_{\mathrm{T}} Nodes(t) \setminus (\Sigma \setminus \{|\ halt\ |\})$$

[1] Gavin Lowe, personal communication, 2010.

Hence, the property "the Turing machine halts" is equivalent to the uniform verification of the above refinement. The former property is well known to be undecidable; hence our uniform verification property is undecidable in general. ∎

Remark 6.1.4. The fact that all node processes are generated from a single template, $\mathcal{N}_{myId}(t)$, combined with data independence of nodes, implies that for all instantiations T of type t and all identities me in T the process $\mathcal{N}_{me}(T)$ can be represented using the configuration $(\mathcal{N}_{myId}(t), \{myId \mapsto me\}, T)$ (see Section 4.4). Although the proof of congruence of SSOS and COSE to a standard operational semantics (see Theorem 4.5.1) assumed initial environments to be the empty ones, it is easy to see that the congruence still holds if initial environments hold nodes' own identities by observing that $(\mathcal{N}_{myId}(t), \{myId \mapsto me\}, T)$ and $\mathcal{N}_{myId}(T)[\{myId \mapsto me\}] = \mathcal{N}_{me}(T)$ are related under the bisimulation relation $\mathcal{B}$ used in the proof of Theorem 4.5.1.

Example 6.1.5. Let

$$\mathcal{N}_{myId}(t) = in?i{:}t \rightarrow out.myId.i \rightarrow STOP$$

be a template for a node process with identity *myId* and awareness of identities of type t. Suppose further that T is an instantiation of type t and me is in T. Then

$$\mathcal{N}_{me}(T) = (in?i{:}t \rightarrow out.myId.i \rightarrow STOP, \{myId \mapsto me\}, T).$$

End of example.

We now define a useful quantity that we will repeatedly use within this chapter.

Definition 6.1.6. We let $W_{\mathcal{N}}$ denote the maximum number of node identities, other that *myId*, that $\mathcal{N}_{myId}(t)$ can store for future use.

$W_{\mathcal{N}}$ counts not only identities that are stored for use within subsequent events, but also identities that are stored for comparison within guards of conditional choices. Also, recall that every nondeterministic selection (\$) is first resolved by performing a τ and then is replaced by an output using some output variable (see Section 4.2.2). Hence, all values that are input using nondeterministic selections should be counted as stored, even though they may be only stored for an immediate output.

Note that $W_{\mathcal{N}}$ is well-defined, since the assumption that nodes are finite-state implies that the maximum number of node identities that every node can store for future use is bounded.

Example 6.1.7. $W_{\mathcal{N}}$ is 2 for both of the following two processes.

$$\begin{aligned}\mathcal{N}^1_{myId}(t) = in?i{:}t \rightarrow (&out_1.myId.i \rightarrow STOP\\ &\Box\\ &in?j{:}t \rightarrow out_2.myId.i.j \rightarrow STOP)\end{aligned}$$

$$\mathcal{N}^2_{myId}(t) = in_1\$i{:}t \rightarrow STOP \;\Box\; in_2\$j{:}t \rightarrow STOP$$

.

End of example.

In addition to storing up to $W_{\mathcal{N}}$ identities of other nodes at any given time, every node process stores its own identity at all times. Therefore, each state of $\mathcal{N}_{me}(T)$ generated using COSE (see Section 4.4) is a configuration $(cst, \{myId \mapsto me, x_1 \mapsto id_1, \ldots, x_n \mapsto id_n\}, T)$ for $0 \leq n \leq W_{\mathcal{N}}$, where cst is a symbolic state that records the control state and $\{myId \mapsto me, x_0 \mapsto id_0, \ldots, x_n \mapsto id_n\}$ is an environment that stores the identities $me, id_0, \ldots, id_n$ in the variables $myId, x_0, \ldots, x_n$, respectively. We assume that each x_i is a free variable within cst, so that environments are minimal, i.e. values not needed in the future are immediately forgotten, except for the value of $myId$, which is never forgotten. Note that control states can include values of types other than t.

Example 6.1.8. Let $\mathcal{N}^1_{myId}(T)$ and $\mathcal{N}^2_{myId}(T)$ be as in Example 6.1.7. Then the states of $\mathcal{N}^1_0(\{0,1\})$ are

$(\mathcal{N}^1_{myId}(t), \{myId \mapsto 0\}, \{0,1\}),$
$(cst, \{myId \mapsto 0, i \mapsto 0\}, \{0,1\}),$
$(cst, \{myId \mapsto 0, i \mapsto 1\}, \{0,1\}),$
$(STOP, \{myId \mapsto 0\}, \{0,1\}),$
$(out_2.mydId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 0, j \mapsto 0\}, \{0,1\}),$
$(out_2.mydId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 0, j \mapsto 1\}, \{0,1\}),$
$(out_2.mydId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 1, j \mapsto 0\}, \{0,1\}),$
$(out_2.mydId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 1, j \mapsto 1\}, \{0,1\}),$

where

$$cst = out_1.myId.i \rightarrow STOP \;\Box\; in?j{:}t \rightarrow out_2.myId.i.j \rightarrow STOP$$

and the states of $\mathcal{N}^2_1(\{0,1\})$ are

$(\mathcal{N}^2_{myId}(t), \{myId \mapsto 1\}, \{0,1\}),$
$(in_1.i \rightarrow STOP \;\Box\; in_2\$j{:}t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 0\}, \{0,1\}),$
$(in_1.i \rightarrow STOP \;\Box\; in_2\$j{:}t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 1\}, \{0,1\}),$
$(in_1\$i{:}t \rightarrow STOP \;\Box\; in_2.j \rightarrow STOP, \{myId \mapsto 1, j \mapsto 0\}, \{0,1\}),$
$(in_1\$i{:}t \rightarrow STOP \;\Box\; in_2.j \rightarrow STOP, \{myId \mapsto 1, j \mapsto 1\}, \{0,1\}),$
$(in_1.i \rightarrow STOP \;\Box\; in_2.j \rightarrow STOP, \{myId \mapsto 1, i \mapsto 0, j \mapsto 0\}, \{0,1\}),$
$(in_1.i \rightarrow STOP \;\Box\; in_2.j \rightarrow STOP, \{myId \mapsto 1, i \mapsto 0, j \mapsto 1\}, \{0,1\}),$
$(in_1.i \rightarrow STOP \;\Box\; in_2.j \rightarrow STOP, \{myId \mapsto 1, i \mapsto 1, j \mapsto 0\}, \{0,1\}),$
$(in_1.i \rightarrow STOP \;\Box\; in_2.j \rightarrow STOP, \{myId \mapsto 1, i \mapsto 1, j \mapsto 1\}, \{0,1\}),$
$(STOP, \{myId \mapsto 1\}, \{0,1\}).$

End of example.

We make the notational convention that if (cst, Γ, T) and (cst', Γ', T) are two states such that $(cst, \Gamma, T) \xrightarrow{a} (cst', \Gamma', T)$ for some event a, and $\Gamma(myId) = me$ (i.e. we deal with a node process with identity me), then we decorate the transition relation symbol with me to indicate the identity of the process and write $(cst, \Gamma, T) \xrightarrow{a}_{me} (cst', \Gamma', T)$.

Throughout this chapter we assume that all synchronisations between node processes involve either exactly two processes or all $\#T$ processes; the vast majority of processes used in practice do not require synchronisation with any other arity. We allow events to contain node identities as payloads, i.e. identities that nodes can use in ways not related to synchronisation. Thanks to such payloads, events can be used, for example, to transmit an identity from an external controller to a single node, pass an identity from one node to another, or input the same identity to all nodes.

The above assumptions mean that every event performed by a node can be classified as of one of the following types.

α-type: private events of nodes

These events are of the form $c.f_\alpha(me, payloadIDs)$ for some identity me, where c is some channel name and $f_\alpha(me, payloadIDs)$ is a construct of the form $v_1 \ldots v_k$ such that there is a non-empty indexing set $\mathcal{I} \subseteq \{1 \,..\, k\}$ such that for all i in $\mathcal{I}$, $v_i = me$; and $payloadIDs = \langle v_j \mid j \in \{1 \,..\, k\} \setminus \mathcal{I} \wedge v_j \in T \rangle$. For all types T, an α-type event $c.f_\alpha(me, payloadIDs)$ is only in the alphabet of the node with identity me. We also refer to α-type events as α-events.

β-type: events synchronised between two nodes

These event are of the form $c.f_\beta(me, other, payloadIDs)$ for some distinct identities me and $other$, where c is some channel name and $f_\beta(me, other, payloadIDs)$ is a construct of the form $v_1 \ldots v_k$ such that there are non-empty indexing sets $\mathcal{I}_{me} \subseteq \{1 \,..\, k\}$ and $\mathcal{I}_{other} \subseteq \{1 \,..\, k\}$ such that for all i in $\mathcal{I}_{me}$, $v_i = me$; for all i in $\mathcal{I}_{other}$, $v_i = other$; and $payloadIDs = \langle v_j \mid j \in \{1 \,..\, k\} \setminus \{i, j\} \wedge v_j \in T \rangle$. Clearly, for types T of size 1, there cannot be any β-type events. For all types T such that $\#T \geq 2$, a β-type event $c.f_\beta(me, other, payloadIDs)$ requires the synchronisation of the nodes with identities me and $other$, i.e. each such event is in the alphabets of only those two nodes. We also refer to β-type events as β-events.

γ-type: events synchronised between all nodes

These events are of the form $c.f_\gamma(payloadIDs)$, where c is some channel name and $f_\gamma(payloadIDs)$ is a construct of the form $v_1 \ldots v_k$, where $payloadIDs = \langle v_i \mid i \in \{1 \,..\, k\} \wedge v_j \in T \rangle$. Every γ-type event is in the alphabet of all node processes, regardless of the type T. Observe that every event that does not contain any values of type t is a γ-event with $payloadIDs = \{\}$. We also refer to γ-type events as γ-events.

We assume that the nodes' alphabets are generated from such events in a uniform way. More formally, we assume that there exist disjoint indexing sets $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ such

that for each me and T,

$$\begin{aligned} A(me, T) = &\{c_\alpha.f_\alpha(me, payloadIDs) \mid \alpha \in \mathcal{A} \wedge payloadIDs \in T^{n_\alpha}\} \cup \\ &\{c_\beta.f_\beta(me, other, payloadIDs), c_\beta.f_\beta(other, me, payloadIDs) \mid \\ &\qquad \beta \in \mathcal{B}, other \in T \setminus \{me\} \wedge payloadIDs \in T^{n_\beta}\} \cup \\ &\{c_\gamma.f_\gamma(payloadIDs) \mid \gamma \in \mathcal{C} \wedge payloadIDs \in T^{n_\gamma}\}, \end{aligned} \tag{6.2}$$

where $c_\alpha, c_\beta, c_\gamma$ are channel names, f_α, f_β, f_γ are function, $n_\alpha, n_\beta, n_\gamma$ are natural number, and the events in the three sets are, respectively, α-, β- and γ-events.

Remark 6.1.9. We note that:

(i) our construction allows values of types different from t to be present in α-, β- and γ-events,

(ii) given T and an identity me in T, the alphabet $A(me, T)$ is partitioned by the sets of α-, β- and γ-events, and

(iii) the type of an event e can be decided by choosing T of size at least 3 and counting the number of identities i for which e is in $A(i, T)$ (this will be of practical importance for the implementation of a tool presented in Chapter 7).

We further assume that the functions are disjoint in the following sense:

(iv) if two events agree on all non-T parts, then they are generated using the same function.

Example 6.1.10. Let

$$\begin{aligned} \mathcal{N}_{myId}(t) = &\ a?x{:}X \rightarrow b?i{:}t \rightarrow c!myId?i{:}t \rightarrow STOP \\ &\ \Box \\ &\ c?i{:}t!myId \rightarrow d?i{:}t?j{:}t \rightarrow STOP \\ &\ \Box \\ &\ e!myId?i{:}t \rightarrow STOP \end{aligned}$$

for some type X not related to t. Also, let

$$\begin{aligned} &f_{(a,x)}(\langle\rangle) = x, \qquad \text{for } x \in X \\ &f_b(\langle i \rangle) = i \\ &f_c(me, other, \langle\rangle) = me.other \\ &f_d(\langle i, j \rangle) = i.j \\ &f_e(me, \langle i \rangle) = me.i. \end{aligned}$$

Then

$$\begin{aligned} A(me, T) = &\{e.f_e(me, \langle i \rangle) \mid i \in T\} \cup \\ &\{c.f_c(me, other, \langle\rangle), c.f_c(other, me, \langle\rangle) \mid other \in T \setminus \{me\}\} \cup \\ &\{a.f_{(a,x)}(\langle\rangle), b.f_b(\langle i \rangle), d.f_d(\langle i, j \rangle) \mid x \in X \wedge i, j \in T\} \end{aligned}$$

is the alphabet of $\mathcal{N}_{me}(T)$, where the three sets give the α-, β- and γ-events, respectively. Note that we block the event $c.me.me$ by omitting it from the alphabet.

End of example.

For the rest of this chapter let ϕ be a B-collapsing function (see Definition 5.0.1). In addition, let $\sim$ be an equivalence relation defined over the control states of node processes. This relation adds an additional layer of abstraction (orthogonal to counter abstraction) by allowing us to merge several concrete states into a single state of the counter state machine. For example, if we are interested in verifying mutual exclusion properties and there are multiple operations that a node performs outside and/or inside its critical section, we may join states into Non-critical/Trying/Critical equivalence classes using $\sim$.

Definition 6.1.11. We define $\approx_\phi$ to be an equivalence relation on the states of node processes by saying that two states (cst, Γ, T) and (cst', Γ', T') are equivalent if

- their control parts are related under $\sim$ ($cst \sim cst'$),
- their environments are equal under ϕ ($\phi(\Gamma) = \phi(\Gamma')$), and
- their underlying types are equal under ϕ ($\phi(T) = \phi(T')$).

Whenever ϕ is clear from the context, we simply write $\approx$. The equivalence class of state (cst, Γ, T) under $\approx$ is

$$[(cst, \Gamma, T)] = \{(cst', \Gamma', T') \mid cst \sim cst' \wedge \phi(\Gamma) = \phi(\Gamma') \wedge \phi(T) = \phi(T')\}.$$

The counter abstraction techniques described in this chapter work by counting the number of node processes in $\approx_\phi$-equivalent states. One approach would be to count all nodes. However, if the identity of a node is less than B, then this node's environment can never equal that of another node under ϕ, since their *myId* variables will always be mapped to values that are different under ϕ. Thus, if we were to counter abstract those nodes, we would be modelling their behaviour explicitly. This is of no benefit over a simpler approach, where the nodes with identities in $\{0..B-1\}$ are modelled explicitly (with a reduction of their awareness of node identities to a fixed set, independent of t) and we counter abstract all other nodes, using the state machines of nodes with identities in $\{B\,..\,B+m\}$, where m is a positive integer. When dealing with processes that contain no equality tests on t, we will take $m = 1$ (Section 6.3). Otherwise, we will take $m = W_{\mathcal{N}}$ (Section 6.4).

6.2 Defining counter abstraction models with unbounded counters

In this section, given a type T and non-negative values of parameters B and n (the latter being either 0 or equal to parameter m, defined in the previous section), we create an abstract model $ABS_\infty^{T,B,n}$ such that for sufficiently large T and n,

$$ABS_\infty^{T,B,n} \sqsubseteq_{\mathrm{T}} \phi(Nodes(T)). \tag{6.3}$$

As mentioned before, the abstract model consists of two parts. The first is the parallel composition of the nodes $\mathcal{N}_0(\{0..B+n\}), \ldots, \mathcal{N}_{B-1}(\{0..B+n\})$. The second part is

a counter state machine modelling all the nodes with identities in $\{B \,..\, \#T-1\}$; we assume that $\#T \geq B+1$, so there is at least one such node. In effect, this counter state machine collapses the identities in $\{B \,..\, \#T-1\}$ to the single identity B. However, we perform this collapse in two stages. Let $m = \max\{n, 1\}$.

- We first build a counter state machine $\zeta^T_\infty(\mathcal{N}_B(\{0 \,..\, B+m\}))$ that abstracts the nodes $\mathcal{N}_B(\{0 \,..\, B+m\}), \ldots, \mathcal{N}_{\#T-1}(\{0 \,..\, B+m\})$, i.e. where those nodes have awareness of the node identities in $\{0 \,..\, B+m\}$. We present this construction in Section 6.2.1. The counter state machine is constructed from the the state machines of nodes $\mathcal{N}_B(\{0 \,..\, B+m\}), \ldots, \mathcal{N}_{B+m}(\{0 \,..\, B+m\})$; however $\forall\, i, j \in \{B \,..\, B+m\} \bullet \mathcal{N}_i(\phi(T)) = \mathcal{N}_j(\phi(T))[\![{}^{i,j}/_{j,i}]\!]$, so we write the counter state machine as a function of just $\mathcal{N}_B(\{0 \,..\, B+m\})$.

- Then:

  - if $n = 0$, we apply a renaming $\mathcal{R}$ to replace every instance of $B+1$ by B: let $\mathcal{R}(B+1) = B$ and $\mathcal{R}(i) = i$ for all $i \neq B+1$ and lift $\mathcal{R}$ to events in the natural way;
  - if $n > 0$, we compose the counter state machine in parallel with the parallel composition of the nodes $\mathcal{N}_0(\{0 \,..\, B+n\}), \ldots, \mathcal{N}_{B-1}(\{0 \,..\, B+n\})$, and then apply ϕ renaming.

This two-stage process is necessary to ensure synchronisations on β-events between two counter-abstracted nodes are captured correctly (see Case 3 of the definition of the transition relation on page 153 and Case 2 of Lemma 6.3.6, below).

Hence, we define our abstractions by:

$$ABS^{T,B,0}_\infty \mathrel{\widehat{=}} \left(\begin{array}{c} \|\ i \in \{0 \,..\, B-1\} \bullet [A(i, \{0 \,..\, B\})]\ \mathcal{N}_i(\{0 \,..\, B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})} \|_{A^B_\zeta} \\ \left(\zeta^T_\infty(\mathcal{N}_B(\{0 \,..\, B+1\}))\right)[\![\mathcal{R}]\!] \end{array} \right),$$

where

$$A^B_\zeta \mathrel{\widehat{=}} \mathcal{R}(A(B, \{0 \,..\, B+1\}) \cup A(B+1, \{0 \,..\, B+1\})),$$

and

$$ABS^{T,B,m}_\infty \mathrel{\widehat{=}} \phi \left(\begin{array}{c} \|\ i \in \{0 \,..\, B-1\} \bullet [A(i, \{0 \,..\, B+m\})]\ \mathcal{N}_i(\{0 \,..\, B+m\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B+m\})} \|_{\bigcup_{i=B}^{B+m} A(i,\{0..B+m\})} \\ \zeta^T_\infty(\mathcal{N}_B(\{0 \,..\, B+m\})) \end{array} \right).$$

The main characteristic of $ABS^{T,B,0}_\infty$ and $ABS^{T,B,m}_\infty$ is their fixed, finite size alphabet. However, both of these models are still dependent on the size of T, since each counter within the counter state machines can take values between 0 and $\#T - B$. In Section 6.5 we will remove this dependence by placing thresholds on these counters.

In Section 6.3 we show that (6.3) holds with $n = 0$ for T large enough, provided node processes contain no equality and inequality tests on t. An analogous result is

proved about $ABS_\infty^{T,B,W_\mathcal{N}}$ for node processes that might contain equality or inequality tests on t in Section 6.4.

For the rest of this chapter we let

- $(S(i), s_0(i), \Sigma(i, \{0\,..\,B+m\}) \cup \{\tau\}, \longrightarrow_i)$ be the state machine of node $\mathcal{N}_i(\{0\,..\,B+m\})$, for i in $\{B\,..\,B+m\}$,
- $S = \bigcup\{S(i) \mid i \in \{B\,..\,B+m\}\}$ (i.e. S is the set of state of nodes $\mathcal{N}_B(\{0\,..\,B+m\}),\ldots,\mathcal{N}_{B+m}(\{0\,..\,B+m\})$, and
- $E = \{[st_1], \ldots, [st_k]\} = S/\approx$ (note that k is the number of equivalence classes of S under $\approx$).

Since we assumed that all processes have only finitely many states, S is finite; hence E is also finite. From Remark 6.1.4 we have that

$$\forall\, i \in \{B\,..\,B+m\} \bullet s_0(i) = (\mathcal{N}_{myId}(t), \{myId \mapsto i\}, \{0\,..\,B+m\}).$$

Hence,

$$\forall\, i, j \in \{B\,..\,B+m\} \bullet s_0(i) \approx s_0(j).$$

since $\phi(B) = \phi(B+1) = \ldots \phi(B+m) = B$. Without loss of generality, we assume that $[st_1]$ is the equivalence class within E that contains all initial states $s_0(i)$ for i in $\{B\,..\,B+m\}$.

6.2.1 Constructing the counter state machine

The techniques described in this section extend those presented in Section 3.2.1. We now present a method that, for every $m > 0$, generates a counter state machine

$$\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+m\})) = (A_\infty(T), a_\infty(T), \Sigma_\infty^\tau(T), \longrightarrow_\infty(T))$$

that abstracts the nodes $\mathcal{N}_B(\{0\,..\,B+m\}),\ldots,\mathcal{N}_{\#T-1}(\{0\,..\,B+m\})$. Whenever T is clear from context, we omit it from the right-hand side and simply write $(A_\infty, a_\infty, \Sigma_\infty^\tau, \longrightarrow_\infty)$. Let T be of size at least $B+1$. Recall that k is the number of equivalence classes of node states.

The states of A_∞ are k-tuples of non-negative integer counters

$$(c_1, \ldots, c_k).$$

such that $\sum_{i=1}^k c_i = \#T - B$. Informally, c_j counts the number of node processes in a state in equivalence class $[st_j]$. More precisely, consider a concrete state $(s_B, \ldots, s_{\#T-1})$, where for each i in $\{B\,..\,\#T-1\}$, s_i is a state of $\mathcal{N}_i(\{0\,..\,B+m\})$, i.e. it records the local state of node i. Then the corresponding abstract state (in A_∞) is the tuple $(c_1, \ldots, c_k)$ such that c_j counts the number of nodes in states within the equivalence class of st_j, i.e.

$$\forall\, j \in \{1\,..\,k\} \bullet c_j = \#\{i \in \{B\,..\,\#T-1\} \mid s_i \approx st_j\}.$$

Example 6.2.1. Let

$$\begin{aligned}\mathcal{N}_{myId}(t) = {} & init!myId?other{:}t \rightarrow send1.myId.other \\ & \rightarrow send2.myId.other \rightarrow \mathcal{N}_{myId}(t)\end{aligned}$$

and let

$$\begin{aligned}cst_0 &= init!myId?other{:}t \rightarrow send1.myId.other \rightarrow send2.myId.other \rightarrow \mathcal{N}_{myId}(t) \\ cst_1 &= send1.myId.other \rightarrow send2.myId.other \rightarrow \mathcal{N}_{myId}(t) \\ cst_{1'} &= send2.mydId.other \rightarrow \mathcal{N}_{myId}(t)\end{aligned}$$

Let $\sim$ be an equivalence relation on $\{cst_0, cst_1, cst_{1'}\}$ such that $cst_1 \sim cst_{1'}$. Let $B = 1$, $m = 1$ and let ϕ be a B-collapsing function (i.e. ϕ maps $\{B\,..\,B+m\} = \{1,2\}$ onto $\{B\} = \{1\}$). Then there are three equivalence classes of node states under $\approx$ (each such equivalence class is infinitely large, so for illustration purposes we only list the states of the nodes used in building a counter state machine, namely $\mathcal{N}_B(\hat{T}), \ldots, \mathcal{N}_{B+m}(\hat{T})$), where $\hat{T} = \{0\,..\,B+m\} = \{0,1,2\}$:

$$[(cst_0, \{myId \mapsto 1\}, \hat{T})] = \{(cst_0, \{myId \mapsto 1\}, \hat{T}), (cst_0, \{myId \mapsto 2\}, \hat{T}), \ldots\}$$

$$\begin{aligned}&[(cst_1, \{myId \mapsto 1, other \mapsto 0\}, \hat{T})] = \\ &\quad \{(cst_1, \{myId \mapsto 1, other \mapsto 0\}, \hat{T}), (cst_{1'}, \{myId \mapsto 1, other \mapsto 0\}, \hat{T}), \\ &\quad\ (cst_1, \{myId \mapsto 2, other \mapsto 0\}, \hat{T}), (cst_{1'}, \{myId \mapsto 2, other \mapsto 0\}, \hat{T}), \ldots\}\end{aligned}$$

$$\begin{aligned}&[(cst_1, \{myId \mapsto 1, other \mapsto 1\}, \hat{T})] = \\ &\quad \{(cst_1, \{myId \mapsto 1, other \mapsto 1\}, \hat{T}), (cst_1, \{myId \mapsto 1, other \mapsto 2\}, \hat{T}), \\ &\quad\ (cst_{1'}, \{myId \mapsto 1, other \mapsto 1\}, \hat{T}), (cst_{1'}, \{myId \mapsto 1, other \mapsto 2\}, \hat{T}), \\ &\quad\ (cst_1, \{myId \mapsto 2, other \mapsto 1\}, \hat{T}), (cst_1, \{myId \mapsto 2, other \mapsto 2\}, \hat{T}), \\ &\quad\ (cst_{1'}, \{myId \mapsto 2, other \mapsto 1\}, \hat{T}), (cst_{1'}, \{myId \mapsto 2, other \mapsto 2\}, \hat{T}), \ldots\}.\end{aligned}$$

Let T be some type. Since there are 3 equivalence classes in E, each state of $\zeta_\infty^T(\mathcal{N}_B(\hat{T}))$ is a triple of integer counter (c_1, c_2, c_3), where for every i, c_i counts how many of the node processes $\mathcal{N}_B(\hat{T}), \ldots, \mathcal{N}_{\#T-1}(\hat{T})$ are in a state within the i-th equivalence class. Each counter can take a value between 0 and $\#T - B$.

End of example.

Let *count* be a state in A_∞. Given a state of a node process, say st, we let $count[st]$ denote the coordinate of *count* corresponding to equivalence class $[st]$. Then, we let $count[st \leftarrow x]$ denote a tuple of counters like *count*, but with x substituted for the value of the counter corresponding to the equivalence class $[st]$. Also, we let $count[st{+}{+}]$ and $count[st{-}{-}]$ denote the tuple *count* with the counter corresponding to the equivalence class $[st]$ incremented and decremented, respectively, by 1, i.e. $count[st{+}{+}] = count[st \leftarrow count[st]{+}1]$ and $count[st{-}{-}] = count[st \leftarrow count[st]{-}1]$. As a shorthand notation we will also write, for example, $count[st{-}{-}, st'{+}{+}]$ to mean the tuple *count* with the counter corresponding to $[st']$ incremented and the counter corresponding to $[st]$ decremented, where both operations happen atomically; observe that this leaves *count* unchanged if $st \approx st'$.

Since we assumed that $[st_1]$ is the equivalence class of the initial states of nodes with identities in $\{B\,..\,B+m\}$, the initial state, a_∞, is the tuple $(\#T - B, 0, \ldots, 0)$.

This naturally corresponds to the situation where all the node processes with identities greater or equal to B are in their initial states.

To define the labels of the counter machine, Σ_∞^τ, it is enough to observe that the transformed system can only perform events that one of $\mathcal{N}_B(\{0\,..\,B+m\}), \ldots,$ $\mathcal{N}_{B+m}(\{0\,..\,B+m\})$ can perform. Hence, we define

$$\Sigma_\infty = \bigcup\{\Sigma(i, \{0\,..\,B+m\}) \mid i \in \{B\,..\,B+m\}\}$$

and

$$\Sigma_\infty^\tau = \Sigma_\infty \cup \{\tau\}.$$

Finally, we define the transition relation, $\longrightarrow_\infty$. Let *count* and *count*$'$ be two states in A_∞. We define the transitions from *count* to *count*$'$ by a case analysis on the type of the events.

(1) There is an τ-transition between *count* and *count*$'$ if and only if

- there is a concrete state *st* in S in which τ can be executed to reach some state *st*$'$,
- the counter of *count* corresponding to $[st]$ is at least 1, and
- the counters of *count*$'$ are updated correctly: 1 is subtracted from the counter corresponding to $[st]$ and added to the counter corresponding to $[st']$; if $[st] = [st']$, then the counters remain unmodified.

Formally,

$$\begin{array}{ll} & \mathit{count} \xrightarrow{\tau}_\infty \mathit{count}' \\ \Leftrightarrow & \exists\, \mathit{me} \in \{B\,..\,B+m\} \bullet \\ & \quad \exists\, \mathit{st}, \mathit{st}' \in S(\mathit{me}) \bullet \mathit{st} \xrightarrow{\tau}_{\mathit{me}} \mathit{st}' \wedge \mathit{count}[\mathit{st}] \geq 1 \\ & \qquad\qquad \wedge\ \mathit{count}' = \mathit{count}[\mathit{st}^{--}, \mathit{st}'^{++}]. \end{array} \tag{6.4}$$

(2) Suppose now that *me* is a fixed identity in $\{B\,..\,B+m\}$. Let e in Σ_∞ be either

- an α-event of the form $c.f_\alpha(\mathit{me}, \mathit{payloadIDs})$, or
- a β-event of the form $c.f_\beta(\mathit{me}, \mathit{other}, \mathit{payloadIDs})$, where $\mathit{other} \in \{0\,..\,B-1\}$.

This case is very similar to the case for τ, above. Formally, for every e as above,

$$\begin{array}{ll} & \mathit{count} \xrightarrow{e}_\infty \mathit{count}' \\ \Leftrightarrow & \exists\, \mathit{st}, \mathit{st}' \in S(\mathit{me}) \bullet \mathit{st} \xrightarrow{e}_{\mathit{me}} \mathit{st}' \wedge \mathit{count}[\mathit{st}] \geq 1 \\ & \qquad\qquad \wedge\ \mathit{count}' = \mathit{count}[\mathit{st}^{--}, \mathit{st}'^{++}]. \end{array} \tag{6.5}$$

(3) Now, let *me* and *other* be two distinct identities in $\{B\,..\,B+m\}$. Let e in Σ_∞ be a β-event of the form $c.f_\beta(\mathit{me}, \mathit{other}, \mathit{payloadIDs})$. Then there is an e-labelled transition between *count* and *count*$'$ if and only if

- there are concrete states st_{me} and st_{other} in $S(me)$ and $S(other)$, respectively, in both of which e can be executed to reach some states st'_{me} and st'_{other}, respectively,
- if $[st_{me}] = [st_{other}]$, then the counters of *count* corresponding to $[st_{me}]$ and $[st_{other}]$ are at least 2; if $[st_{me}] \neq [st_{other}]$, then each of $count[st_{me}]$ and $count[st_{other}]$ is at least 1,
- the counters of *count'* are updated correctly: the counters corresponding to $[st_{me}]$ and $[st_{other}]$ are decreased by 1 and the counters corresponding to $[st'_{me}]$ and $[st'_{other}]$ are increased by 1, with all the changes happening atomically.

Formally, for every e as above,

$$\begin{array}{ll} & count \xrightarrow{e}_{\infty} count' \\ \Leftrightarrow & \exists\, st_{me}, st'_{me} \in S(me), st_{other}, st'_{other} \in S(other) \bullet \\ & \quad st_{me} \xrightarrow{e}_{me} st'_{me} \wedge st_{other} \xrightarrow{e}_{other} st'_{other} \\ & \quad \wedge\ count[st_{me}] \geq p \wedge count[st_{other}] \geq p \\ & \quad \wedge\ count' = count[st_{me}{}^{--}, st_{other}{}^{--}, st'_{me}{}^{++}, st'_{other}{}^{++}], \\ & \text{where } p = 2 \text{ if } [st_{me}] = [st_{other}] \text{ and } p = 1 \text{ otherwise.} \end{array} \tag{6.6}$$

(4) Finally, let e in Σ_∞ be a γ-event. Then, there is a transition labelled with e between *count* and *count'* if and only if there are e-transitions that allow simultaneous movement of all (i.e. $\#T - B$) nodes with identities in $\{B\,..\,\#T - 1\}$ between states of S, such that for every i in $\{1\,..\,k\}$, the number of processes indicated by $count[st_i]$ can move from states in $[st_i]$ and the number of processes indicated by $count'[st_i]$ can move into states in $[st_i]$. This is possible if and only if

- for every i in $\{1..k\}$, there are $count[st_i]$ many (not necessarily all distinct) concrete states src^i_j in S for $1 \leq j \leq count[st_i]$, in all of which e can be executed to reach some state tgt^i_j, and
- every counter of *count'* is equal to the total number of the target states tgt^i_j that are in the equivalence class to which the counter corresponds.

Formally, for every e as above,

$$\begin{array}{l} count \xrightarrow{e}_{\infty} count' \\ \Leftrightarrow \\ \exists\ src, tgt : \mathbb{N} \times \mathbb{N} \to S \bullet \\ \quad \forall\, i \in \{1\,..\,k\}, j \in \{1\,..\,count[st_i]\} \bullet \\ \qquad src^i_j \approx st_i \wedge \exists\, me \in \{B\,..\,B+m\} \bullet src^i_j \xrightarrow{e}_{me} tgt^i_j \\ \quad \wedge \forall\, st \in S \bullet count'[st] = \#\{(i,j) \mid i \in \{1\,..\,k\} \wedge j \in \{1\,..\,count[st_i]\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ tgt^i_j \approx st\}. \end{array} \tag{6.7}$$

By clause (iii) of Remark 6.1.9, the four parts of the definition of $\longrightarrow_\infty$ are pairwise disjoint, so together they combine into a well-formed definition of the abstract transition relation.

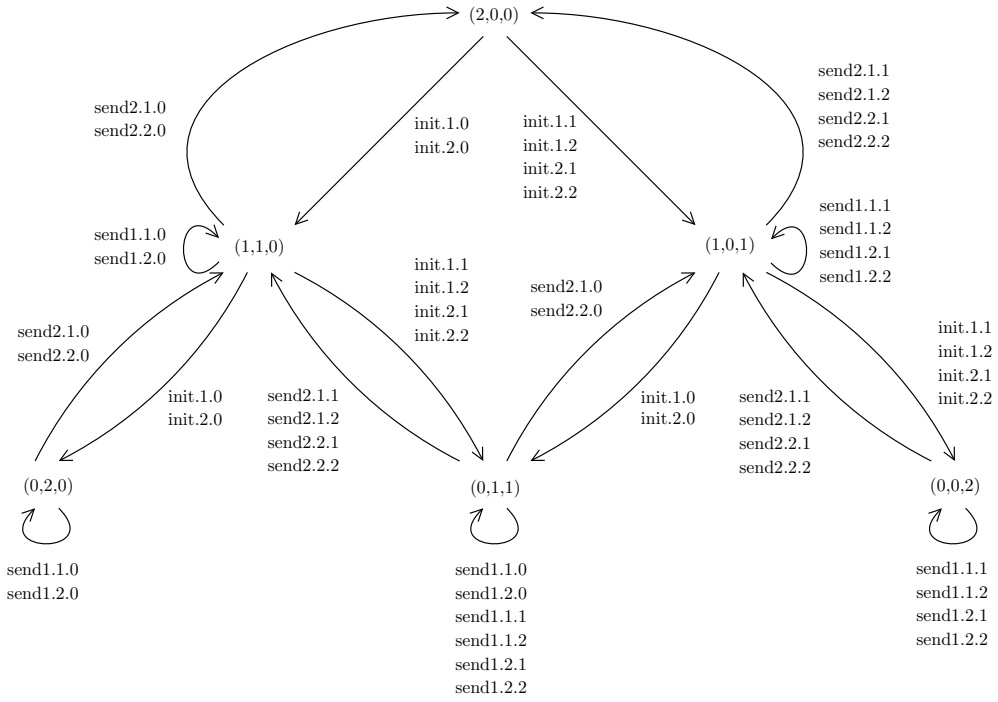


Figure 6.1: Counter state machine $\zeta_\infty^{\{0,1,2\}}(\mathcal{N}_1(\{0,1,2\}))$, for $\mathcal{N}_B(\{0\,..\,B+m\})$ defined as in Example 6.2.1.

Example 6.2.2. Recall the $\mathcal{N}_{myId}(t)$ process from Example 6.2.1. Let $B = m = 1$. For every T such that $\#T \geq B+1$, we can use the method described above to construct $\zeta_\infty^T(\mathcal{N}_1(\{0,1,2\}))$. Figure 6.1 shows the state machine of $\zeta_\infty^{\{0,1,2\}}(\mathcal{N}_1(\{0,1,2\}))$. For all values of T, the state machine's states are all tuples (c_1, c_2, c_3) of non-negative integers such that $c_1 + c_2 + c_3 = \#T - 1$; but the state machine always has the same alphabet.

End of example.

6.3 Refinement results for processes without equality tests

The main aim of this section is to prove that, provided node process contain no equality and inequality tests on t, for T of size at least $B+1$,

$$ABS_\infty^{T,B,0} \sqsubseteq_{\mathrm{T}} \phi(Nodes(T)).$$

Conditional choices play an important role in control flow of processes. However, when using collapsing functions, they complicate the analysis, since a conditional that evaluates to *False* in a standard model may evaluate to *True* in a model with a collapsed type (e.g. $x = y$ evaluates to *False* for distinct x and y, but $\phi(x) = \phi(y)$ evaluates to *True* if x and y are mapped to the same value by ϕ). We therefore ban

conditional choice (i.e. we assume that nodes satisfy $\mathbf{NoEqT}^t$; see Definition 5.2.10) for the moment. We will consider processes with conditional choices in Section 6.4.

We begin with a regularity result, relating transitions of concrete node processes for different instantiations of type t.

Lemma 6.3.1. *Suppose $\mathcal{N}_{myId}(t)$ satisfies $\boldsymbol{NoEqT}^t$. Let T and $\hat{T}$ be two instantiations of type t and let ψ be a function from T to $\hat{T}$. Then if*

$$(cst, \Gamma, T) \xrightarrow{a}_{me} (cst', \Gamma', T)$$

for some $me \in T$, then

$$(cst, \psi(\Gamma), \hat{T}) \xrightarrow{\psi(a)}_{\psi(me)} (cst', \psi(\Gamma'), \hat{T}).$$

Proof: If cst is a control state of $\mathcal{N}_{me}(T)$, then it must also be a control state of $\mathcal{N}_{\psi(me)}(\hat{T})$, since the choice of an instantiation of type t cannot influence the control flow of $\mathcal{N}_{myId}(t)$, by data independence.

Suppose that $(cst, \Gamma, T) \xrightarrow{a}_{me} (cst', \Gamma', T)$. Since $\mathcal{N}_{myId}(t)$ is data independent, the availability of events, ignoring the values of type t they input or output, is not influenced by the choice of an instantiation of t. In addition, no constants of type t are used, so all outputs of type t within a must be through variables in $\mathrm{dom}(\Gamma)$; hence, from the state $(cst, \psi(\Gamma), \hat{T})$, each type t output value v in a is replaced by $\psi(v)$. Also, Remark 4.1.3 implies that whenever a node identity v is input in some state, then every other identity could be input in the same state; particular, from the state $(cst, \psi(\Gamma), \hat{T})$ the value $\psi(v)$ could be input. This implies that $(cst, \psi(\Gamma), \hat{T})$ can perform the event $\psi(a)$. The control flow is unaffected, by data independence and lack of conditional choices on t. Finally, the environment is updated according to the values input, i.e. to $\psi(\Gamma')$. ∎

In our future considerations we will often find the following definition particularly useful.

Definition 6.3.2. Given a type T and $K \geq B$, a *K-precollapsing function* ψ is a function from T to $\{0 \,..\, K\}$ such that

- $\psi(v) = v$ for all v in $\{0 \,..\, B-1\}$, and
- $\psi(v) \in \{B \,..\, K\}$ for all v in $\{B \,..\, \#T - 1\}$.

Remark 6.3.3. Let T, B and $K \geq B$ be given. Then if ϕ is a B-collapsing function and ψ is a K-precollapsing function, then $\phi \circ \psi = \phi$.

We now aim for a result that says that if:

- the nodes satisfy $\mathbf{NoEqT}^t$,
- the size of T is at least $B + 1$, and
- a parallel composition of nodes in states st_i, for $i \in T$, can perform an event a to reach a parallel composition of nodes in states st'_i, for $i \in T$,

then a $\phi(a)$-labelled transition is available between the corresponding states of $ABS_{\infty}^{T,B,0}$. We split the result into two parts: one dealing with events performable by one or more nodes with identities in $\{0\,..\,B-1\}$ (Lemma 6.3.5), and the other dealing with events performable by one or more nodes with identities in $\{B\,..\,\#T-1\}$ (Lemma 6.3.6). We begin with a small lemma concerning the alphabets of the parallel components.

Lemma 6.3.4. *If $i \in \{0\,..\,B-1\}$, then $a \in A(i,T) \Leftrightarrow \phi(a) \in A(i,\{0\,..\,B\})$.*

Proof: A straightforward case analysis, based on (6.2). ∎

In the following lemma (and also in later results) we use parallel compositions of configurations. Even though our description of COSE (see Section 4.4) did not provide the semantics for such constructs, each configuration that we consider here is a state of some node process. Therefore, we can use the standard operational semantics to define the meaning of such parallel compositions: if for every i in some indexing set $\mathcal{I}(T)$, (cst_i, Γ_i, T) is equal to some state $\mathcal{N}'_i(T)$ of node $\mathcal{N}_i(T)$, then we define the semantics of

$$\Big\|\ i \in \mathcal{I}(T) \bullet [A(i,T)]\ (cst_i, \Gamma_i, T)$$

to be the same as the semantics of

$$\Big\|\ i \in \mathcal{I}(T) \bullet [A(i,T)]\ \mathcal{N}'_i(T).$$

Lemma 6.3.5. *Suppose that $\mathcal{N}_{myId}(t)$ satisfies $\boldsymbol{NoEqT}^t$. Let T be an instantiation of type t such that $\#T \geq B+1$. For all i in T, let (cst_i, Γ_i, T) be a state of $\mathcal{N}_i(T)$ and suppose that*

$$\Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,T)]\ (cst_i, \Gamma_i, T)$$

$$\xrightarrow{\ a\ } \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T)$$

for some control states cst'_i and environments Γ'_i. Then

$$\Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\})$$

$$\xrightarrow{\ \phi(a)\ } \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}).$$

Proof: We prove the result using a case analysis on the kind of event that a can be.

Case 1. Suppose that a is one of the following:

(i) an α-event of the form $c.f_\alpha(me, payloadIDs)$ (i.e. a private event of $\mathcal{N}_{me}(T)$) for some identity me in $\{0\,..\,B-1\}$ and some list of identities $payloadIDs$,

(ii) a β-event of the form $c.f_\beta(me, other, payloadIDs)$ (i.e. an event shared by $\mathcal{N}_{me}(T)$ and $\mathcal{N}_{other}(T)$) for some identities me in $\{0\,..\,B-1\}$ and $other$ in $\{B\,..\,\#T-1\}$ and some list of identities $payloadIDs$, or

(iii) the internal τ event.

Then

$$(cst_{me}, \Gamma_{me}, T) \xrightarrow{a}_{me} (cst'_{me}, \Gamma'_{me}, T) \tag{6.8}$$

and

$$\forall\, i \in \{0\,..\,B-1\} \setminus \{me\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T), \tag{6.9}$$

where me is as in (i) or (ii) if a is an α- or β-event, respectively, and is some identity in $\{0\,..\,B-1\}$ if $a = \tau$. Hence,

$$\forall\, i \in \{0\,..\,B-1\} \setminus \{me\} \bullet (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) = (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}). \tag{6.10}$$

Using (6.8), Lemma 6.3.1 implies that

$$(cst_{me}, \phi(\Gamma_{me}), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{\phi(me)} (cst'_{me}, \phi(\Gamma'_{me}), \{0\,..\,B\}).$$

However, $\phi(me) = me$ (since $me \in \{0\,..\,B-1\}$), so

$$(cst_{me}, \phi(\Gamma_{me}), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{me} (cst'_{me}, \phi(\Gamma'_{me}), \{0\,..\,B\}). \tag{6.11}$$

In addition, if a is as in either (i) or (ii), then

$$\phi(a) \in A(me, \{0\,..\,B\}) \setminus \textstyle\bigcup_{i \in \{0..B\} \setminus \{me\}} A(i, \{0\,..\,B\})$$

using Lemma 6.3.4. Also, if $a = \tau$, then $\phi(a) = \tau$. Hence, combining (6.10) and (6.11), we get that

$$\begin{array}{c} \\ \xrightarrow{\phi(a)} \end{array} \begin{array}{l} \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i, \{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i, \{0\,..\,B\})]\ (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}), \end{array}$$

as required.

Case 2. Suppose that a is a β-event of the form $c.f_\beta(x, y, \mathit{payloadIDs})$ (i.e. an event shared by $\mathcal{N}_x(T)$ and $\mathcal{N}_y(T)$) for some distinct identities x and y in $\{0\,..\,B-1\}$ and some list of identities *payloadIDs*.
Then

$$(cst_x, \Gamma_x, T) \xrightarrow{a}_x (cst'_x, \Gamma'_x, T), \tag{6.12}$$

$$(cst_y, \Gamma_y, T) \xrightarrow{a}_y (cst'_y, \Gamma'_y, T), \tag{6.13}$$

and

$$\forall\, i \in \{0\,..\,B-1\} \setminus \{x, y\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma, T). \tag{6.14}$$

From (6.12) and (6.13) we can infer using Lemma 6.3.1 that

$$(cst_x, \phi(\Gamma_x), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{\phi(x)} (cst'_x, \phi(\Gamma'_x), \{0\,..\,B\})$$

and

$$(cst_y, \phi(\Gamma_y), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{\phi(y)} (cst'_y, \phi(\Gamma'_y), \{0\,..\,B\}).$$

Since x and y are both in $\{0\,..\,B-1\}$, we have that $\phi(x) = x$ and $\phi(y) = y$. Therefore,

$$(cst_x, \phi(\Gamma_x), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{x} (cst'_x, \phi(\Gamma'_x), \{0\,..\,B\}) \tag{6.15}$$

and

$$(cst_y, \phi(\Gamma_y), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{y} (cst'_y, \phi(\Gamma'_y), \{0\,..\,B\}). \tag{6.16}$$

In addition, from (6.14) we have that

$$\begin{array}{l}\forall\, i \in \{0\,..\,B-1\} \setminus \{x, y\} \bullet \\ \quad (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}) = (cst_i, \phi(\Gamma_i), \{0\,..\,B\}).\end{array} \tag{6.17}$$

Finally,

$$\phi(a) \in \Big(A(x, \{0\,..\,B\}) \cap A(y, \{0\,..\,B\})\Big) \setminus \textstyle\bigcup_{i \in \{0..B\} \setminus \{x,y\}} A(i, \{0\,..\,B\})$$

by Lemma 6.3.4. Hence, and from (6.15), (6.16) and (6.17),

$$\begin{array}{l}\quad \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i, \{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ \xrightarrow{\phi(a)} \quad \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i, \{0\,..\,B\})]\ (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}),\end{array}$$

as required.

Case 3. Suppose that a is a γ-event of the form $c.f_\gamma(payloadIDs)$ (i.e. an event shared by all nodes) for some list of identities *payloadIDs*.
Then

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \Gamma_i, T) \xrightarrow{a}_{i} (cst'_i, \Gamma'_i, T). \tag{6.18}$$

Hence, using Lemma 6.3.1 we can infer that

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{\phi(i)} (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}).$$

However, for all i in $\{0\,..\,B-1\}$, $\phi(i) = i$, so

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \xrightarrow{\phi(a)}_{i} (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}). \tag{6.19}$$

Further,

$$\forall\, i \in \{0\,..\,B-1\} \bullet \phi(a) \in A(i, \{0\,..\,B\})$$

by Lemma 6.3.4. Hence, and from (6.19), we have that

$$\begin{array}{l}\quad \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i, \{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ \xrightarrow{\phi(a)} \quad \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i, \{0\,..\,B\})]\ (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}),\end{array}$$

as required. This completes our proof. ∎

Lemma 6.3.6. *Suppose that $\mathcal{N}_{myId}(t)$ satisfies **NoEqT**^{t}. Let T be an instantiation of type t such that $\#T \geq B+1$. For all i in T, let (cst_i, Γ_i, T) be a state of $\mathcal{N}_i(T)$ and suppose that*

$$\begin{array}{rl} & \Big\| \; i \in \{B \,..\, \#T-1\} \bullet [A(i,T)] \; (cst_i, \Gamma_i, T) \\ \xrightarrow{a} & \Big\| \; i \in \{B \,..\, \#T-1\} \bullet [A(i,T)] \; (cst'_i, \Gamma'_i, T) \end{array}$$

for some control states cst'_i and environments Γ'_i. Let count and count$'$ be states of $\zeta^T_\infty(\mathcal{N}_B(\{0\,..\,B+1\}))$ such that for every state st we have that

$$count[st] = \#\{i \in \{B \,..\, \#T-1\} \mid (cst_i, \Gamma_i, T) \approx st\}$$

and

$$count'[st] = \#\{i \in \{B \,..\, \#T-1\} \mid (cst'_i, \Gamma'_i, T) \approx st\}.$$

Then

$$count[\![\mathcal{R}]\!] \xrightarrow{\phi(a)} count'[\![\mathcal{R}]\!],$$

where $\mathcal{R}$ is the renaming that replaces every instance of $B+1$ by B.

Proof: We prove the result using a case analysis on the kind of event that a can be.

Case 1. Suppose that a is one of the following:

(i) an α-event of the form $c.f_\alpha(me, payloadIDs)$ (i.e. a private event of $\mathcal{N}_{me}(T)$) for some identity me in $\{B \,..\, \#T-1\}$ and some list of identities $payloadIDs$,

(ii) a β-event of the form $c.f_\beta(me, other, payloadIDs)$ (i.e. an event shared by $\mathcal{N}_{me}(T)$ and $\mathcal{N}_{other}(T)$) for some identities me in $\{B \,..\, \#T-1\}$ and $other$ in $\{0 \,..\, B-1\}$ and some list of identities $payloadIDs$, or

(iii) the internal τ event.

Then

$$(cst_{me}, \Gamma_{me}, T) \xrightarrow{a}_{me} (cst'_{me}, \Gamma'_{me}, T) \tag{6.20}$$

and

$$\forall\, i \in \{B \,..\, \#T-1\} \setminus \{me\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T), \tag{6.21}$$

where me is as in (i) or (ii) if a is an α- or β-event, respectively, and is some identity in $\{B \,..\, \#T-1\}$ if $a = \tau$. Hence, using the definitions of $count$ and $count'$,

$$count' = count[(cst_{me}, \Gamma_{me}, T)^{--}, (cst'_{me}, \Gamma'_{me}, T)^{++}].$$

The definition of the $\approx$ equivalence relation (Definition 6.1.11) implies that

$$[(cst_{me}, \Gamma_{me}, T)] = [(cst_{me}, \phi(\Gamma_{me}), \{0\,..\,B+1\})]$$
$$[(cst'_{me}, \Gamma'_{me}, T)] = [(cst'_{me}, \phi(\Gamma'_{me}), \{0\,..\,B+1\})].$$

Therefore,

$$count' = count[(cst_{me}, \phi(\Gamma_{me}), \{0\,..\,B+1\})^{--}, \quad (cst'_{me}, \phi(\Gamma'_{me}), \{0\,..\,B+1\})^{++}]. \tag{6.22}$$

In addition, using Lemma 6.3.1 we can infer from (6.20) that

$$(cst_{me}, \phi(\Gamma_{me}), \{0\,..\,B+1\}) \xrightarrow{\phi(a)}_B (cst'_{me}, \phi(\Gamma'_{me}), \{0\,..\,B+1\}).$$

Further, since $count[(cst_{me}, \Gamma_{me}, T)] \geq 1$, $count[(cst_{me}, \phi(\Gamma_{me}), \{0\,..\,B+1\})] \geq 1$. Hence, we can infer that

$$count \xrightarrow{\phi(a)}_\infty count'$$

using the definition of $\longrightarrow_\infty$ ((6.5) if a is an α- or β-event or (6.4) if $a = \tau$). Hence, and thanks to the fact that $\mathcal{R} \circ \phi = \phi$, we have that

$$count[\![\mathcal{R}]\!] \xrightarrow{\phi(a)}_\infty count'[\![\mathcal{R}]\!],$$

as required.

Case 2. Suppose that a is a β-event of the form $c.f_\beta(x, y, payloadIDs)$ (i.e. an event shared by $\mathcal{N}_x(T)$ and $\mathcal{N}_y(T)$) for some distinct identities x and y in $\{B\,..\,\#T-1\}$ and some list of identities *payloadIDs*.
Then

$$(cst_x, \Gamma_x, T) \xrightarrow{a}_x (cst'_x, \Gamma'_x, T), \tag{6.23}$$

$$(cst_y, \Gamma_y, T) \xrightarrow{a}_y (cst'_y, \Gamma'_y, T), \tag{6.24}$$

and

$$\forall\, i \in \{B\,..\,\#T-1\} \setminus \{x, y\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma, T). \tag{6.25}$$

Hence, using the definitions of *count* and *count'*,

$$count' = count[(cst_x, \Gamma_x, T)^{--}, (cst_y, \Gamma_y, T)^{--}, \quad (cst'_x, \Gamma'_x, T)^{++}, (cst'_y, \Gamma'_y, T)^{++}] \tag{6.26}$$

Let ψ be a $(B+1)$-precollapsing function such that $\psi(x) = B$ and $\psi(y) = B+1$ (there is at least one such function, since x and y are two distinct identities in $\{B\,..\,\#T-1\}$, implying $\#T \geq B+2$). We have that $\phi(T) = \phi(\{0\,..\,B+1\})$ and, by Remark 6.3.3,

$\phi(\psi(\Gamma)) = \phi(\Gamma)$ for every environment Γ. Hence, by the definition of the $\approx$ equivalence relation (Definition 6.1.11),

$$[(cst_x, \Gamma_x, T)] = [(cst_x, \psi(\Gamma_x), \{0\,..\,B+1\})]$$
$$[(cst'_x, \Gamma'_x, T)] = [(cst'_x, \psi(\Gamma'_x), \{0\,..\,B+1\})]$$

and

$$[(cst_y, \Gamma_y, T)] = [(cst_y, \psi(\Gamma_y), \{0\,..\,B+1\})]$$
$$[(cst'_y, \Gamma'_y, T)] = [(cst'_y, \psi(\Gamma'_y), \{0\,..\,B+1\})].$$

Therefore, (6.26) implies that

$$count' = count[(cst_x, \psi(\Gamma_x), \{0\,..\,B+1\})^{--}, (cst_y, \psi(\Gamma_y), \{0\,..\,B+1\})^{--}, \quad (6.27)$$
$$(cst'_x, \psi(\Gamma'_x), \{0\,..\,B+1\})^{++}, (cst'_y, \psi(\Gamma'_y), \{0\,..\,B+1\})^{++}]$$

In addition, using Lemma 6.3.1 we can infer from (6.23) and (6.24) that

$$(cst_x, \psi(\Gamma_x), \{0\,..\,B+1\}) \xrightarrow{\psi(a)}_B (cst'_x, \psi(\Gamma'_x), \{0\,..\,B+1\})$$

and

$$(cst_y, \psi(\Gamma_y), \{0\,..\,B+1\}) \xrightarrow{\psi(a)}_{B+1} (cst'_y, \psi(\Gamma'_y), \{0\,..\,B+1\}).$$

Let $p = 2$ if $(cst_x, \psi(\Gamma_x), T) \approx (cst_y, \psi(\Gamma_y), \{0\,..\,B+1\})$ and $p = 1$ otherwise; then, since $count[(cst_x, \Gamma_x, T)] \geq p$ and $count[(cst_y, \Gamma_y, T)] \geq p$,

$$count[(cst_x, \psi(\Gamma_x), \{0\,..\,B+1\})] \geq p$$
$$count[(cst_y, \psi(\Gamma_y), \{0\,..\,B+1\})] \geq p.$$

Hence, from (6.27) and the definition of $\longrightarrow_\infty$ (6.6),

$$count \xrightarrow{\psi(a)}_\infty count'.$$

Hence, and since $\mathcal{R} \circ \psi = \phi$ (as ψ collapses T to $\{0\,..\,B+1\}$ and $\mathcal{R}$ renames $B+1$ to B),

$$count[\![\mathcal{R}]\!] \xrightarrow{\phi(a)}_\infty count'[\![\mathcal{R}]\!],$$

as required.

Note that this is the case where it was necessary to counter abstract the nodes with identities in $\{B\,..\,\#T-1\}$ in a two-stage process: first mapping onto the nodes with identities B and $B+1$ (to ensure $\psi(x) \neq \psi(y)$); and then renaming $B+1$ to B.

Case 3. Suppose that a is a γ-event of the form $c.f_\gamma(payloadIDs)$ (i.e. an event shared by all nodes) for some list of identities *payloadIDs*.
Then

$$\forall\, i \in \{B\,..\,\#T-1\} \bullet (cst_i, \Gamma_i, T) \xrightarrow{a}_i (cst'_i, \Gamma'_i, T).$$

Hence, using Lemma 6.3.1 we can infer from definition of ϕ that

$$\forall\, i \in \{B..\#T{-}1\} \bullet (cst_B, \phi(\Gamma_B), \{0..B{+}1\}) \xrightarrow{\phi(a)}_B (cst'_i, \phi(\Gamma'_i), \{0..B{+}1\}). \quad (6.28)$$

The definition of the $\approx$ equivalence relation (Definition 6.1.11) implies that

$$\forall\, i \in \{B\,..\,\#T-1\} \bullet [(cst_i, \Gamma_i, T)] = [(cst_i, \phi(\Gamma_i), \{0\,..\,B+1\})] \quad (6.29)$$

and

$$\forall\, i \in \{B\,..\,\#T-1\} \bullet [(cst'_i, \Gamma'_i, T)] = [(cst'_i, \phi(\Gamma'_i), \{0\,..\,B+1\})]. \quad (6.30)$$

From (6.29) and by the definition of *count* we have that

$$\begin{array}{l}\forall\, i \in \{1\,..\,k\} \bullet \\ \quad count[st_i] = \#\{j \in \{B\,..\,\#T-1\} \mid (cst_j, \Gamma_j, T) \approx st_i\} \\ \qquad\qquad = \#\{j \in \{B\,..\,\#T-1\} \mid (cst_j, \phi(\Gamma_j), \{0\,..\,B+1\}) \approx st_i\}.\end{array}$$

Hence, for every i in $\{1\,..\,k\}$ there are $count[st_i]$ many states

$$\begin{array}{rcl} src^i_1 & = & (cst_{j^i_1}, \phi(\Gamma_{j^i_1}), \{0\,..\,B+1\}), \\ src^i_2 & = & (cst_{j^i_2}, \phi(\Gamma_{j^i_2}), \{0\,..\,B+1\}), \\ & \vdots & \\ src^i_{count[st_i]} & = & (cst_{j^i_{count[st_i]}}, \phi(\Gamma_{j^i_{count[st_i]}}), \{0\,..\,B+1\}), \end{array} \quad (6.31)$$

all related to st_i by $\approx$, where for all i in $\{1\,..\,k\}$ and for all l in $\{1\,..\,count[st_i]\}$, the indices j^i_l are in $\{B\,..\,\#T-1\}$ and are all distinct. From (6.28) we have that

$$\forall\, i \in \{1\,..\,k\} \bullet \forall\, l \in \{1\,..\,count[st_i]\} \bullet src^i_l \xrightarrow{\phi(a)}_B tgt^i_l, \quad (6.32)$$

where for all i in $\{1\,..\,k\}$,

$$\begin{array}{rcl} tgt^i_1 & = & (cst'_{j^i_1}, \phi(\Gamma'_{j^i_1}), \{0\,..\,B+1\}), \\ tgt^i_2 & = & (cst'_{j^i_2}, \phi(\Gamma'_{j^i_2}), \{0\,..\,B+1\}), \\ & \vdots & \\ tgt^i_{count[st_i]} & = & (cst'_{j^i_{count[st_i]}}, \phi(\Gamma'_{j^i_{count[st_i]}}), \{0\,..\,B+1\}), \end{array} \quad (6.33)$$

with each j^i_l as above. Further,

$$\begin{array}{rcl} & & \wr(cst'_B, \phi(\Gamma'_B), \{0\,..\,B+1\}), \ldots, (cst'_{\#T-1}, \phi(\Gamma'_{\#T-1}), \{0\,..\,B+1\})\wr \\ & = & \wr\, tgt^i_l \mid i \in \{1\,..\,k\}, l \in \{1\,..\,count[st_i]\}\wr \end{array}$$

and

$$\#\{B\,..\,\#T-1\} = \#\{(i,l) \mid i \in \{1\,..\,k\} \wedge l \in \{1\,..\,count[st_i]\}\}.$$

Hence, from (6.30), we have that

$$\begin{array}{l}\forall\, st \in S \bullet \\ \quad count'[st] = \#\{j \in \{B\,..\,\#T-1\} \mid (cst'_j, \Gamma'_j, T) \approx st\} \\ \qquad\qquad = \#\{j \in \{B\,..\,\#T-1\} \mid (cst'_j, \phi(\Gamma'_j), \{0\,..\,B+1\}) \approx st\} \\ \qquad\qquad = \#\{(i,l) \mid i \in \{1\,..\,k\} \wedge l \in \{1\,..\,count[st_i]\} \wedge tgt^i_l \approx st\}.\end{array} \quad (6.34)$$

Combining (6.31)–(6.34) and the assumption of the lemma that $\#T \geq B+1$, we use the definition of $\longrightarrow_\infty$ (6.7) to infer that

$$count \xrightarrow{\phi(a)}_\infty count'.$$

Hence and since $\mathcal{R} \circ \phi = \phi$,

$$count[\![\mathcal{R}]\!] \xrightarrow{\phi(a)}_\infty count'[\![\mathcal{R}]\!],$$

as required. This completes our proof. ∎

We will now combine Lemma 6.3.5 and Lemma 6.3.6 into a result for all events performable by the parallel composition of all nodes. We begin with a lemma concerning the alphabets.

Lemma 6.3.7. $a \in \bigcup_{i=B}^{\#T-1} A(i,T) \Leftrightarrow \phi(a) \in A_\zeta^B$.

Proof: The left-to-right direction is proved by a case analysis over the type of a, using (6.2)). We sketch the case for β-events for illustration.

So suppose $a = c.f_\beta(i, other, payloadIDs) \in A(i,T)$ is a β-event with $i \in \{B\,..\,\#T-1\}$, $other \in T \backslash \{i\}$ and $payloadIDs \in T^n$. (The case of $c.f_\beta(other, i, payloadIDs)$ is very similar.) If $other \in \{0\,..\,B-1\}$, then

$$\phi(a) = c.f_\beta(B, \phi(other), \phi(payloadIDs)) \in A(B, \{0\,..\,B\}) \subseteq A_\zeta^B,$$

using (6.2). Alternatively, if $other \in \{B\,..\,\#T-1\} \setminus \{i\}$ then

$$\phi(a) = c.f_\beta(\phi(i), \phi(other), \phi(payloadIDs)) = c.f_\beta(B, B, \phi(payloadIDs)).$$

Now, $c.f_\beta(B, B+1, \phi(payloadIDs)) \in A(B, \{0\,..\,B+1\})$, using (6.2). Hence

$$\phi(a) = \mathcal{R}(c.f_\beta(B, B+1, \phi(payloadIDs))) \in \mathcal{R}(A(B, \{0\,..\,B+1\})) \subseteq A_\zeta^B.$$

For the right-to-left direction, suppose $\phi(a) \in A_\zeta^B = \mathcal{R}(A(B, \{0\,..\,B+1\}) \cup A(B+1, \{0\,..\,B+1\}))$. Then suppose $\phi(a) = \mathcal{R}(a')$ for some $a' \in A(B, \{0\,..\,B+1\})$ (the case of $a' \in A(B+1, \{0\,..\,B+1\})$ is very similar). We can perform a case analysis over the type of a'. We sketch the case for β-events for illustration.

So suppose $a' = c.f_\beta(B, other, payloadIDs)$ is a β-event, with $other \in \{0\,..\,B+1\} \setminus \{B\}$ and $payloadIDs \in \{0\,..\,B+1\}^n$. (The case of $c.f_\beta(other, B, payloadIDs)$ is very similar.) Then

$$\phi(a) = \mathcal{R}(a') = c.f_\beta(B, \mathcal{R}(other), \mathcal{R}(payloadIDs)).$$

Now, a and a' agree on non-T fields, so they must be generated using the same function f_β (see clause (iv) of Remark 6.1.9). So $a = c.f_\beta(i, j, payloadIDs')$ for some $i \in T$, $j \in T \setminus \{i\}$ and $payloadIDs' \in T^n$, where $\phi(i) = B$, $\phi(j) = \mathcal{R}(other)$ and $\phi(payloadIDs') = \mathcal{R}(payloadIDs)$. Hence $i \geq B$, and $a \in A(i,T) \subseteq \bigcup_{k=B}^{\#T-1} A(k,T)$, using (6.2). ∎

Proposition 6.3.8. *Suppose that $\mathcal{N}_{myId}(t)$ satisfies **NoEqT**^{t}. Let T be an instantiation of type t such that $\#T \geq B+1$. For all i in T, let (cst_i, Γ_i, T) be a state of $\mathcal{N}_i(T)$ and suppose that*

$$\begin{array}{rl} & \Big\| \; i \in T \bullet [A(i,T)] \; (cst_i, \Gamma_i, T) \\ \xrightarrow{a} & \Big\| \; i \in T \bullet [A(i,T)] \; (cst'_i, \Gamma'_i, T) \end{array}$$

for some control states cst'_i and environments Γ'_i. Let count and count$'$ be states of $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+1\}))$ such that for every state st we have that

$$count[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst_i, \Gamma_i, T) \approx st\}$$

and

$$count'[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst'_i, \Gamma'_i, T) \approx st\}.$$

Then

$$\begin{array}{rl} & \left(\begin{array}{c} \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})] \; (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})}\big\|_{A_\zeta^B} \\ count[\![\mathcal{R}]\!] \end{array} \right) \\ \xrightarrow{\phi(a)} & \left(\begin{array}{c} \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})] \; (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})}\big\|_{A_\zeta^B} \\ count'[\![\mathcal{R}]\!] \end{array} \right), \end{array}$$

where $\mathcal{R}$ is a renaming that replaces every instance of $B+1$ by B.

Proof: The proof is by case analysis over the membership of a in the alphabet of the first B nodes, $\bigcup_{i=0}^{B-1} A(i,T)$, and/or the alphabet of the remaining nodes, $\bigcup_{i=B}^{\#T-1} A(i,T)$.

Case 1. Suppose that the transition is due to the first B nodes only: either a is in $\bigcup_{i=0}^{B-1} A(i,T)$ and not in $\bigcup_{i=B}^{\#T-1} A(i,T)$, or that $a = \tau$, and

$$\begin{array}{rl} & \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,T)] \; (cst_i, \Gamma_i, T) \\ \xrightarrow{a} & \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,T)] \; (cst'_i, \Gamma'_i, T), \end{array}$$

and

$$\forall\, i \in \{B\,..\,\#T-1\} \bullet (cst_i, \Gamma_i, T) = (cst'_i, \Gamma'_i, T). \tag{6.35}$$

Then, Lemma 6.3.5 implies that

$$\begin{array}{rl} & \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})] \; (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ \xrightarrow{\phi(a)} & \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})] \; (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}). \end{array} \tag{6.36}$$

In addition, $count' = count$ by (6.35), so

$$count'[\![\mathcal{R}]\!] = count[\![\mathcal{R}]\!]. \tag{6.37}$$

Finally, if a is a visible event, then

$$\phi(a) \in \left(\bigcup_{i=0}^{B-1} A(i, \{0\,..\,B\})\right) \setminus A_\zeta^B.$$

by Lemma 6.3.4 and Lemma 6.3.7. Hence, the result follows from (6.36) and (6.37).

Case 2. Suppose that the transition is due to nodes with identities in $\{B\,..\,\#T-1\}$: either a is in $\bigcup_{i=B}^{\#T-1} A(i,T)$ and not in $\bigcup_{i=0}^{B-1} A(i,T)$, or $a=\tau$, and

$$\begin{aligned} &\Big\|\ i \in \{B\,..\,\#T-1\} \bullet [A(i,T)]\ (cst_i, \Gamma_i, T) \\ \xrightarrow{a}\ &\Big\|\ i \in \{B\,..\,\#T-1\} \bullet [A(i,T)]\ (cst_i', \Gamma_i', T), \end{aligned}$$

and

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \Gamma_i, T) = (cst_i', \Gamma_i', T).$$

Then, Lemma 6.3.6 implies that

$$count[\![\mathcal{R}]\!] \xrightarrow{\phi(a)} count'[\![\mathcal{R}]\!]. \tag{6.38}$$

Further,

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) = (cst_i', \phi(\Gamma_i'), \{0\,..\,B\}). \tag{6.39}$$

Finally, if a is a visible event, then

$$\phi(a) \in A_\zeta^B \setminus \left(\bigcup_{i=0}^{B-1} A(i, \{0\,..\,B\})\right).$$

by Lemma 6.3.4 and Lemma 6.3.7. Hence, the result follows from (6.38) and (6.39).

Case 3. Suppose the transition involves both sets of nodes: a is in both $\bigcup_{i=0}^{B-1} A(i,T)$ and $\bigcup_{i=B}^{\#T-1} A(i,T)$. Then,

$$\begin{aligned} &\Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,T)]\ (cst_i, \Gamma_i, T) \\ \xrightarrow{a}\ &\Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,T)]\ (cst_i', \Gamma_i', T) \end{aligned}$$

and

$$\begin{aligned} &\Big\|\ i \in \{B\,..\,\#T-1\} \bullet [A(i,T)]\ (cst_i, \Gamma_i, T) \\ \xrightarrow{a}\ &\Big\|\ i \in \{B\,..\,\#T-1\} \bullet [A(i,T)]\ (cst_i', \Gamma_i', T). \end{aligned}$$

Then, Lemma 6.3.5 implies that

$$\begin{array}{l} \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ \xrightarrow{\phi(a)} \quad \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}) \end{array} \tag{6.40}$$

and Lemma 6.3.6 implies that

$$count[\![\mathcal{R}]\!] \xrightarrow{\phi(a)} count'[\![\mathcal{R}]\!]. \tag{6.41}$$

Finally,

$$\phi(a) \in \left(\textstyle\bigcup_{i=0}^{B-1} A(i,\{0\,..\,B\})\right) \cap A_\zeta^B.$$

by Lemma 6.3.4 and Lemma 6.3.7. Hence, the result follows from (6.40) and (6.41). This completes our proof. ∎

Recall that, given processes P and Q, and a sequence of visible or internal events $s = \langle a_1, \ldots, a_n \rangle$, we write

$$P \stackrel{s}{\longmapsto} Q$$

to mean that there exist processes P_i for i in $\{0\,..\,n\}$ such that $P_0 = P$, $P_n = Q$ and

$$\forall\, i \in \{0\,..\,n\} \bullet P_i \xrightarrow{a_{i+1}} P_{i+1}.$$

In addition, $P \stackrel{s}{\longmapsto}$ if there exists some process Q such that $P \stackrel{s}{\longmapsto} Q$.

We now present a result that says that, given nodes that contain no equality tests, if $\phi(Nodes(T))$ can perform a sequence of events s and reach some state P, then $ABS_\infty^{T,B,0}$ can also perform s and reach a state corresponding to P.

Lemma 6.3.9. *Suppose that $\mathcal{N}_{myId}(t)$ satisfies **NoEqT**^{t}. Let T be an instantiation of type t such that $\#T \geq B+1$. Then if*

$$\phi(Nodes(T)) \stackrel{s}{\longmapsto} \phi\left(\Big\| \; i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T)\right)$$

for some control states cst_i and environments Γ_i, then

$$ABS_\infty^{T,B,0} \stackrel{s}{\longmapsto} \left(\begin{array}{c} \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})} \big\|_{A_\zeta^B} \\ count[\![\mathcal{R}]\!] \end{array} \right),$$

where $count$ is a state of $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+1\}))$ such that for all states st,

$$count[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst_i, \Gamma_i, T) \approx st\}$$

and $\mathcal{R}$ is the renaming function that replaces every occurrence of $B+1$ by B.

Proof: We prove the result by an induction on the length of s.

Base Case. Suppose that $s = \langle\rangle$.

Then $(cst_i, \Gamma_i, T) = \mathcal{N}_i(T)$ and $\Gamma_i = \{myId \mapsto i\}$ for all i in T. Therefore, by Remark 6.1.4,

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) = \mathcal{N}_{\phi(i)}(\{0\,..\,B\}).$$

However, for all i in $\{0\,..\,B-1\}$, $\phi(i) = i$. Hence,

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) = \mathcal{N}_i(\{0\,..\,B\}).$$

Also, by the definition of *count*,

$$count = (\#T - B, 0, \ldots, 0).$$

Recall that

$$ABS^{T,B,0}_{\infty} = \left(\begin{array}{c} \|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ \mathcal{N}_i(\{0\,..\,B\}) \\ {\scriptstyle \|_{A^B_\zeta}} \\ (\#T - B, 0, \ldots, 0)[\![\mathcal{R}]\!] \end{array} \right).$$

Hence,

$$ABS^{T,B,0}_{\infty} \stackrel{\langle\rangle}{\longmapsto} \left(\begin{array}{c} \|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ {\scriptstyle \|_{A^B_\zeta}} \\ count[\![\mathcal{R}]\!] \end{array} \right),$$

establishing the result in the base case.

Inductive Case. Suppose the result holds for sequence s and let $sa = s\hat{}\langle a\rangle$ for some event a. Suppose that

$$\phi(Nodes(T)) \stackrel{sa}{\longmapsto} \phi\left(\|\ i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T)\right)$$

for some control states cst'_i and environments Γ'_i. Then, for every i in T there exists a control state cst_i and an environment Γ_i such that

$$\begin{aligned} \phi(Nodes(T)) &\stackrel{s}{\longmapsto} \phi\left(\|\ i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T)\right) \\ &\stackrel{a}{\longrightarrow} \phi\left(\|\ i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T)\right). \end{aligned} \tag{6.42}$$

Then, by the inductive hypothesis,

$$\begin{aligned} &ABS^{T,B,0}_{\infty} \\ &\stackrel{s}{\longmapsto} \left(\begin{array}{c} \|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ {\scriptstyle \|_{A^B_\zeta}} \\ count[\![\mathcal{R}]\!] \end{array} \right), \end{aligned} \tag{6.43}$$

where *count* is a state of $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+1\}))$ such that for all states st,

$$count[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst_i, \Gamma_i, T) \approx st\}.$$

From (6.42) we can infer that there exists an event a' such that $\phi(a') = a$ and

$$\Big\|\ i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T) \xrightarrow{a'} \Big\|\ i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T).$$

Hence, by Proposition 6.3.8 we have that

$$\left(\begin{array}{c} \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst_i, \phi(\Gamma_i), \{0\,..\,B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})}\big\|_{A_\zeta^B} \\ count[\![\mathcal{R}]\!] \end{array}\right)$$

$$\xrightarrow{\phi(a')} \left(\begin{array}{c} \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})}\big\|_{A_\zeta^B} \\ count'[\![\mathcal{R}]\!] \end{array}\right),$$

where *count*′ is a state of $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+1\}))$ such that for all states st,

$$count'[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst'_i, \Gamma'_i, T) \approx st\}.$$

By combining the above with (6.43) and by recalling that $\phi(a') = a$, we get that

$$ABS_\infty^{T,B,0} \xrightarrow{sa} \left(\begin{array}{c} \Big\|\ i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\ (cst'_i, \phi(\Gamma'_i), \{0\,..\,B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})}\big\|_{A_\zeta^B} \\ count'[\![\mathcal{R}]\!] \end{array}\right),$$

which completes our inductive proof. ∎

The following corollary is a simple, but important consequence of Lemma 6.3.9.

Corollary 6.3.10. *Let T be an instantiation of type t such that $\#T \geq B+1$. Then if $\mathcal{N}_{myId}(t)$ satisfies $\boldsymbol{NoEqT}^t$, then*

$$ABS_\infty^{T,B,0} \sqsubseteq_{\mathrm{T}} \phi(Nodes(T)).$$

Proof: Let tr be a trace of $\phi(Nodes(T))$ and let s be a sequence of events such that $s \setminus \{\tau\} = tr$ and $\phi(Nodes(T)) \xrightarrow{s}$. Then, Lemma 6.3.9 implies that $ABS_\infty^{T,B,0} \xrightarrow{s}$, which means that tr is a trace of $ABS_\infty^{T,B,0}$. ∎

6.4 Refinement results for processes with equality tests

This section is in many ways similar to the previous one. It establishes a result analogous to Corollary 6.3.10, but for node processes for which $W_{\mathcal{N}}$ (the maximum number of identities, other than its own, remembered by a node) is at least 1. The result uses abstract models that are larger than those in Section 6.3, so for processes

that satisfy $\mathbf{NoEqT}^t$ it is better to use Corollary 6.3.10. Since nodes do not contain constants of type t (as they satisfy data independence), if a node does not satisfy $\mathbf{NoEqT}^t$, then it must contain an equality or inequality test using at least one stored identity of type t, other than *myId*. Hence, for all such node processes, it must be that $W_{\mathcal{N}} \geq 1$.

Example 6.4.1, below, illustrates why Corollary 6.3.10 no longer holds if node process do not satisfy $\mathbf{NoEqT}^t$.

Example 6.4.1. Let

$$\mathcal{N}_{myId}(t) = in?x{:}t?y{:}t \rightarrow \mathcal{N}'_{myId}(t)$$
$$\mathcal{N}'_{myId}(t) = \text{if } x = myId \vee y = myId \vee x = y \text{ then } c_1.myId.x.y \rightarrow STOP \text{ else } c_2.myId.x.y \rightarrow STOP.$$

Let $B = 0$, so $\{0..B{+}1\} = \{0, 1\}$. In the nodes $\mathcal{N}_B(\{0..B{+}1\})$ and $\mathcal{N}_{B+1}(\{0..B{+}1\})$, the three variables x, y and *myId* cannot hold three different values in $\{0\,..\,B+1\} = \{0, 1\}$, so the test in the conditional always gives *True*. Hence, these nodes cannot perform any communication over channel c_2. Let $T = \{0, 1, 2\}$. Since $\zeta^T_\infty(\mathcal{N}_B(\{0\,..\,B+1\}))$ is built using nodes $\mathcal{N}_B(\{0\,..\,B+1\})$ and $\mathcal{N}_{B+1}(\{0\,..\,B+1\})$, this counter state machine also cannot perform any communication over channel c_2, and hence, no communication on channel c_2 is available within $ABS^{\{0,1,2\},0,0}_\infty$. However, the event $e = c_2.0.1.2$ is available within $Nodes(\{0, 1, 2\})$, which means that $\phi(e) = c_2.0.0.0$ is available within $\phi(Nodes(\{0, 1, 2\}))$. Therefore, $ABS^{\{0,1,2\},0,0}_\infty \not\sqsubseteq_{\mathrm{T}} \phi(Nodes(\{0, 1, 2\}))$, contrary to our requirements.

End of example.

The root of the problem exhibited by Example 6.4.1 is that in Lemma 6.3.1 ψ is no injective over the values stored by a node. This means that an equality or inequality test may use two identities that are remembered by a node and which are both mapped to the same value in $\{B, B + 1\}$ by ψ. Hence, there may be some transitions available after an equality test that evaluates to *False* (respectively, *True*) in $\mathcal{N}_i(T)$ for some i in $\{B\,..\,\#T-1\}$, but that evaluates to *True* (respectively, *False*) in $\mathcal{N}_i(\{0\,..\,B+1\})$ for i in $\{B, B+1\}$.

Recall that $W_{\mathcal{N}}$ (Definition 6.1.6) is the maximum number of node identities (other than its own) that any given node needs to store. In this section we prove that taking $m = W_{\mathcal{N}}$ ensures that all transitions of $\phi(Nodes(T))$ are available in $ABS^{T,B,m}_\infty$.

Remark 6.4.2. Throughout this section we assume that all β- and γ-events contain no payloads with node identities, i.e. every β-event is of the form $c.f_\beta(me, other)$ for some distinct identities *me* and *other* and every γ-event is of the form $c.f_\gamma$. Example 6.4.8 will illustrate that without this assumption Proposition 6.4.7, one of the main results within this section, fails to hold.

We begin by defining some useful notation and establishing several auxiliary results.

Definition 6.4.3. Let $e = c.v_1 \dots v_n$ be a visible event and let $\epsilon = c\S_1 x_1{:}X_1 \dots \S_n x_n{:}X_n$ be a (not necessarily unique) construct it matches after resolving all nondeterministic selections (i.e. $\S_i \in \{?, !\}$ for all $i \in \{1 \,..\, n\}$). Then

$$\mathit{inputs}^t_?(e, \epsilon) \mathrel{\widehat{=}} \{v_i \mid i \in \{1 \,..\, n\} \wedge \S_i = ? \wedge X_i = t\},$$

$$\mathit{outputs}^t(e, \epsilon) \mathrel{\widehat{=}} \{v_i \mid i \in \{1 \,..\, n\} \wedge \S_i = ! \wedge X_i = t\},$$

and, given an environment Γ,

$$\mathit{remembered}(v_i, e, \epsilon, \Gamma) \mathrel{\widehat{=}} x_i \in \mathrm{dom}(\Gamma)$$

for all i in $\{1 \,..\, n\}$.

We make the notational convention that τ matches every construct and

$$\mathit{inputs}^t_?(\tau) = \mathit{outputs}^t(\tau) = \{\}.$$

Observe that when working with a process whose definition satisfies **SeqNorm** (see Definition 4.1.4), Corollary 5.1.15 guarantees uniqueness of constructs (even after resolving nondeterministic selections) that a given event matches. Whenever the construct that an event e matches is obvious from context (e.g. by being unique) or indifferent, we omit it and simply write $\mathit{inputs}^t_?(e)$, $\mathit{outputs}^t(e)$ and $\mathit{remembered}(v_i, e, \Gamma)$.

Given instantiations T and $\hat{T}$ of type t, we need a regularity result, analogous to Lemma 6.3.1, relating transitions of concrete node processes for different instantiations of type t.

Lemma 6.4.4. *Let T and $\hat{T}$ be two instantiations of t. Suppose that $(cst, \Gamma, T) \xrightarrow{a}_{me} (cst', \Gamma', T)$ for some identity $me \in T$. Then, for all injective partial functions $\psi, \psi' : T \nrightarrow \hat{T}$ such that*

- $\mathrm{dom}(\psi) = \mathrm{Im}(\Gamma)$,
- $\mathrm{dom}(\psi') = \mathrm{Im}(\Gamma')$, *and*
- $\psi'(v) = \psi(v)$ *for* $v \in \mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$

we have that $(cst, \psi(\Gamma), \hat{T}) \xrightarrow{\hat{a}}_{\psi(me)} (cst', \psi'(\Gamma'), \hat{T})$ for every event $\hat{a}$ that is like a, except that

- *every value v in $\mathit{outputs}^t(a)$ is replaced by $\psi(v)$,*
- *every value v in $\mathit{inputs}^t_?(a)$ such that $\mathit{remembered}(v, a, \Gamma')$ is replaced by $\psi'(v)$, and*
- *every value v in $\mathit{inputs}^t_?(a)$ such that $\neg\mathit{remembered}(v, a, \Gamma')$ is replaced by an arbitrary value in $\hat{T}$.*

Proof: If cst is a control state of $\mathcal{N}_{me}(T)$, then it must also be a control state of $\mathcal{N}_{\psi(me)}(\hat{T})$, since the choice of an instantiation of type t cannot influence the control flow of $\mathcal{N}_{myId}(t)$, by data independence.

Suppose $(cst, \Gamma, T) \xrightarrow{a}_{me} (cst', \Gamma', T)$. Since $\mathcal{N}_{myId}(t)$ is data independent, the availability of events, ignoring the values of type t they input or output, is not influenced by the choice of an instantiation of t. In addition, no constants of type t are used, so all outputs of type t within a must be through variables that are in $\mathrm{dom}(\Gamma)$. Also, Remark 4.1.3 implies that whenever a node identity v is input in some state, then every other identity could be input in the same state. Hence, $(cst, \psi(\Gamma), \hat{T})$ can perform $\hat{a}$, an arbitrary event like a, but where:

- every output value v of type t in a, which must be the result of looking up a variable x in Γ, is replaced by the value assigned to x by $\psi(\Gamma)$, i.e. $\psi(v)$,
- every deterministic input value v of type t in a that is stored by the node for future use is replaced by $\psi'(v)$, and
- every deterministic input value v of type t in a that is not stored by the node for future use is replaced by some arbitrary value in $\hat{T}$.

We have that:

- the control flow, other than through conditional choices on t, is unaffected, by data independence,
- ψ' is an injective function from $\mathrm{Im}(\Gamma')$ to $\hat{T}$ (which means that any equality or inequality tests evaluates to *True* under Γ' if and only if it evaluates to *True* under $\psi'(\Gamma')$, i.e. no conditional choice immediately after $\hat{a}$ can influence the reachability of $(cst', \psi'(\Gamma'), \hat{T})$), and
- $\psi'(v) = \psi(v)$ for $v \in \mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$ (i.e. the remembered values remain unmodified),

so after performing $\hat{a}$, $(cst, \psi(\Gamma), \hat{T})$ behaves like $(cst', \psi'(\Gamma'), \hat{T})$, as required. ∎

The following corollary applies Lemma 6.4.4 to instantiations $\hat{T} = \{0 \,..\, B + W_{\mathcal{N}}\}$ of type t and functions ψ and ψ' that are $(B + W_{\mathcal{N}})$-precollapsing.

Corollary 6.4.5. *Let T be an instantiation of t and let $\hat{T} = \{0 \,..\, B + W_{\mathcal{N}}\}$. Then, for every injective partial $(B + W_{\mathcal{N}})$-precollapsing function $\psi : T \nrightarrow \hat{T}$ such that $\mathrm{dom}(\psi) = \mathrm{Im}(\Gamma)$ we have that if $(cst, \Gamma, T) \xrightarrow{a}_{me} (cst', \Gamma', T)$ for some $me \in T$, then there exists an injective partial $(B + W_{\mathcal{N}})$-precollapsing function $\psi' : T \nrightarrow \hat{T}$ such that*

- $\mathrm{dom}\, \psi' = \mathrm{Im}(\Gamma')$,
- $\psi'(v) = \psi(v)$ *for* $v \in \mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$,

and $(cst, \psi(\Gamma), \hat{T}) \xrightarrow{\hat{a}}_{\psi(me)} (cst', \psi'(\Gamma'), \hat{T})$ for every event $\hat{a}$ that is like a, except that

- *every value v in $outputs^t(a)$ is replaced by $\psi(v)$,*
- *every value v in $inputs^t_?(a)$ such that $remembered(v, a, \Gamma')$ is replaced by $\psi'(v)$, and*
- *every value v in $inputs^t_?(a) \cap \{B \,..\, \#T - 1\}$ such that $\neg remembered(v, a, \Gamma')$ is replaced by an arbitrary value in $\{B \,..\, B + W_{\mathcal{N}}\}$.*

Proof: Let ψ be defined as in the statement of the corollary. We show that there exists an injective partial $(B + W_{\mathcal{N}})$-precollapsing function $\psi' : T \nrightarrow \hat{T}$ such that

- $\mathrm{dom}(\psi') = \mathrm{Im}(\Gamma')$, and
- $\psi'(v) = \psi(v)$ for $v \in \mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$.

We construct ψ' in three stages. Firstly, let $\psi'_1 : \{0 \,..\, B-1\} \nrightarrow \{0 \,..\, B-1\}$ be such that

- $\mathrm{dom}(\psi'_1) = \mathrm{Im}(\Gamma') \cap \{0 \,..\, B-1\}$, and (6.44)
- $\psi'_1(v) = v$ for $v \in \{0 \,..\, B-1\} \cap \mathrm{dom}(\psi'_1)$, (6.45)

i.e. ψ'_1 is the identity function on all the values in $\mathrm{Im}(\Gamma')$ that are less than B. This means that ψ'_1 is injective. Observe that since $\psi(v) = v$ for $v \in \{0 \,..\, B-1\} \cap \mathrm{Im}(\Gamma)$, (6.44) and (6.45) imply that $\psi'_1(v) = \psi(v)$ for $v \in \mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma') \cap \{0 \,..\, B-1\}$.

Secondly, let ψ'_2 be such that $\psi'_2 = (\mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma') \cap \{B \,..\, \#T-1\}) \lhd \psi$. This means that ψ'_2 is injective, since ψ is. Now, by the definition of a $(B + W_{\mathcal{N}})$-precollapsing function (Definition 6.3.2),

$$\forall\, v \in \mathrm{Im}(\Gamma) \cap \{B \,..\, \#T - 1\} \bullet \psi(v) \in \{B \,..\, B + W_{\mathcal{N}}\},$$

so

$$\forall\, v \in \mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\} \bullet \psi'_2(v) \in \{B \,..\, B + W_{\mathcal{N}}\}.$$

Finally, we construct a function ψ'_3 from

$$X = (\mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\}) \setminus \mathrm{Im}(\Gamma)$$

to

$$Y = \{B \,..\, B + W_{\mathcal{N}}\} \setminus V,$$

where

$$V = \mathrm{Im}(\psi'_2).$$

If $X = \{\}$, then ψ'_3 is the empty function, which is vacuously injective. Otherwise we proceed as follows. Since $\mathcal{N}_{me}(T)$ can remember at most $W_{\mathcal{N}}$ node identities other than me, $\#\,\mathrm{Im}(\Gamma') \leq W_{\mathcal{N}} + 1$. In addition,

$$\begin{aligned} X &= (\mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\}) \setminus \mathrm{Im}(\Gamma) \\ &= (\mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\}) \setminus (\mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\}) \end{aligned}$$

and

$$\#V = \#(\mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\}).$$

Hence

$$\begin{aligned}\#X &= \#(\mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\}) - \#(\mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma') \cap \{B \,..\, \#T - 1\}) \\ &\leq W_{\mathcal{N}} + 1 - \#V \\ &= \#Y.\end{aligned}$$

This means that there is at least one injective function from X to Y. Let ψ'_3 be such an injective function.

We now let $\psi' = \psi'_1 \cup \psi'_2 \cup \psi'_3$. Then $\psi' : \mathrm{Im}(\Gamma') \to \{B \,..\, B + W_{\mathcal{N}}\}$ is well-defined, since $\mathrm{dom}(\psi'_1) \cup \mathrm{dom}(\psi'_2) \cup \mathrm{dom}(\psi'_3) = \mathrm{Im}(\Gamma')$ and the domains of ψ'_1, ψ'_2 and ψ'_3 are pairwise disjoint. In addition, ψ' is injective, since all of ψ'_1, ψ'_2 and ψ'_3 are, and the images of ψ'_1, ψ'_2 and ψ'_3 are pairwise disjoint. Finally, it is easy to check that the properties of ψ'_1, ψ'_2 and ψ'_3 imply that

- $\psi'(v) = v$ for $v \in \{0 \,..\, B - 1\} \cap \mathrm{dom}(\psi')$,
- $\psi'(v) \in \{B \,..\, B + W_{\mathcal{N}}\}$ for $v \in \{B \,..\, \#T - 1\} \cap \mathrm{dom}(\psi')$, and
- $\psi'(v) = \psi(v)$ for $v \in \mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$,

which proves the existence of an injective function ψ' as defined in the conclusion of the corollary.

Observe that the statement

> "every value v in $\mathit{inputs}^t_?(a) \cap \{B..\#T-1\}$ such that $\neg \mathit{remembered}(v, a, \Gamma')$ is replaced by an arbitrary value in $\{B \,..\, B + W_{\mathcal{N}}\}$"

is more specific than

> "every value v in $\mathit{inputs}^t_?(a)$ such that $\neg \mathit{remembered}(v, a, \Gamma')$ is replaced by an arbitrary value in $\{0 \,..\, B + W_{\mathcal{N}}\}$".

Hence, the result follows from Lemma 6.4.4. ∎

The following corollary applies Lemma 6.4.4 to instantiations $\hat{T} = \{0 \,..\, B + W_{\mathcal{N}}\}$ of type t and functions ψ and ψ' that are bijective on $\hat{T}$ and are the identity function on $\{0 \,..\, B - 1\}$.

Corollary 6.4.6. *Let $\hat{T} = \{0 \,..\, B + W_{\mathcal{N}}\}$. Then, for every bijection $\pi : \hat{T} \to \hat{T}$ such that $\pi(v) = v$ for all v in $\{0 \,..\, B - 1\}$, if $(cst, \Gamma, \hat{T}) \xrightarrow{a}_{me} (cst', \Gamma', \hat{T})$ for some $me \in T$, then for every bijection $\pi' : \hat{T} \to \hat{T}$ such that*

- *$\pi'(v) = v$ for all v in $\{0 \,..\, B - 1\}$,*
- *$\pi'(v) = \pi(v)$ for all v in $\mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$,*

we have that $(cst, \pi(\Gamma), \hat{T}) \xrightarrow{\hat{a}}_{\pi(me)} (cst', \pi'(\Gamma'), \hat{T})$ *for every event* $\hat{a}$ *that is like* a*, except that*

- *every value* v *in* $outputs^t(a)$ *is replaced by* $\pi(v)$*,*
- *every value* v *in* $inputs^t_?(a)$ *such that* $remembered(v, a, \Gamma')$ *is replaced by* $\pi'(v)$*, and*
- *every value* v *in* $inputs^t_?(a) \cap \{B \,..\, \#T - 1\}$ *such that* $\neg remembered(v, a, \Gamma')$ *is replaced by an arbitrary value in* $\{B \,..\, B + W_{\mathcal{N}}\}$*.*

Proof: Let π and π' be bijections from $\hat{T}$ to itself as in the statement of the corollary. Then, since π and π' are bijections that are the identity function on $\{0 \,..\, B-1\}$, it must be that for any v in $\{B \,..\, B + W_{\mathcal{N}}\}$, $\pi(v)$ and $\pi'(v)$ are in $\{B \,..\, B + W_{\mathcal{N}}\}$. Let $\psi = \mathrm{Im}(\Gamma) \lhd \pi$ and $\psi' = \mathrm{Im}(\Gamma') \lhd \pi'$. Since every bijection is injective, so must be every function obtained from a bijection using domain restriction. Therefore $\psi : \hat{T} \nrightarrow \hat{T}$ is injective and such that $\mathrm{dom}(\psi) = \mathrm{Im}(\Gamma)$, $\psi' : \hat{T} \nrightarrow \hat{T}$ is injective and such that $\mathrm{dom}(\psi') = \mathrm{Im}(\Gamma')$, and $\psi'(v) = \psi(v)$ for all v in $\mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$. Observe that

- $\pi(\Gamma) = \psi(\Gamma)$,
- $\pi(me) = \psi(me)$ (since $me \in \mathrm{Im}(\Gamma)$),
- $\pi'(\Gamma') = \psi'(\Gamma')$,
- $v \in outputs^t(a) \Rightarrow \pi(v) = \psi(v)$ (since $v \in outputs^t(a)$ implies $v \in \mathrm{Im}(\Gamma)$),
- $v \in inputs^t_?(a) \wedge remembered(v, a, \Gamma') \Rightarrow \pi'(v) = \psi'(v)$,

and that the statement

> "every value v in $inputs^t_?(a) \cap \{B \,..\, \#T - 1\}$ such that $\neg remembered(v, a, \Gamma')$ is replaced by an arbitrary value in $\{B \,..\, B + W_{\mathcal{N}}\}$"

is more specific than

> "every value v in $inputs^t_?(a)$ such that $\neg remembered(v, a, \Gamma')$ is replaced by an arbitrary value in $\hat{T}$".

Hence, the result follows from Lemma 6.4.4. ∎

We now present a result that says that if:

- node processes can remember no more than $W_{\mathcal{N}}$ identities in addition to their own, where $W_{\mathcal{N}} \geq 1$,
- all β- and γ-events contain no payloads with node identities
- the size of T is at least $B + 1$, and
- a parallel composition of nodes in states st_i, for $i \in T$, can perform some event a to reach a parallel composition of nodes in states st'_i, for $i \in T$,

then a $\phi(a)$-labelled transition is available between corresponding states of $ABS_{\infty}^{T,B,W_{\mathcal{N}}}$. The two main differences between the following proposition and Proposition 6.3.8 are the following:

- Node processes no longer have to satisfy $\mathbf{NoEqT}^t$. The presence of equality and inequality tests requires distinction of possibly more than two identities greater or equal to B when a counter state machine is constructed, as we illustrated in Example 6.4.1.
- Node processes have to remember at least one identity (in addition to their own) at some point.
- We no longer assume that β- and γ-events can contain payload identities. Otherwise the result does not hold, as we will demonstrate in Example 6.4.8.

Proposition 6.4.7. *Suppose that $W_{\mathcal{N}} \geq 1$ and that all β- and γ-events contain no payloads with node identities. Let T be an instantiation of type t such that $\#T \geq B+1$. For all i in T suppose that*

$$\begin{array}{rl} & \Big\| \; i \in T \bullet [A(i,T)] \; (cst_i, \Gamma_i, T) \\ \xrightarrow{a} & \Big\| \; i \in T \bullet [A(i,T)] \; (cst'_i, \Gamma'_i, T) \end{array}$$

for some control states cst_i and environments Γ_i. Let $count$ and $count'$ be states of $\zeta_{\infty}^{T}(\mathcal{N}_B(\{0\,..\,B+W_{\mathcal{N}}\}))$ such that for any state st we have that

$$count[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst_i, \Gamma_i, T) \approx st\}$$

and

$$count'[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst'_i, \Gamma'_i, T) \approx st\}.$$

Then, for all i in $\{0\,..\,B-1\}$, given injective partial $(B+W_{\mathcal{N}})$-precollapsing functions $\psi_i : T \nrightarrow \{0\,..\,B+W_{\mathcal{N}}\}$ such that $\mathrm{dom}(\psi_i) = \mathrm{Im}(\Gamma_i)$, there exist injective partial $(B+W_{\mathcal{N}})$-precollapsing functions $\psi'_i : T \nrightarrow \{0\,..\,B+W_{\mathcal{N}}\}$ such that $\mathrm{dom}(\psi'_i) = \mathrm{Im}(\Gamma'_i)$, and $\psi'_i(v) = \psi_i(v)$ for all v in $\mathrm{Im}(\Gamma_i) \cap \mathrm{Im}(\Gamma'_i)$ and, on letting

$$s = \left(\begin{array}{c} \big(\big\| \; i \in \{0\,..\,B-1\} \bullet \\ \quad [A(i,\{0\,..\,B+W^{\mathcal{N}}\})] \; (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\})\big) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W^{\mathcal{N}}\}) \big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ count \end{array} \right)$$

and

$$s' = \left(\begin{array}{c} \big(\big\| \; i \in \{0\,..\,B-1\} \bullet \\ \quad [A(i,\{0\,..\,B+W_{\mathcal{N}}\})] \; (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_{\mathcal{N}}\})\big) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_{\mathcal{N}}\}) \big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ count' \end{array} \right),$$

we have that $\phi(s) \xrightarrow{\phi(a)} \phi(s')$.

Proof: Suppose that

$$\begin{array}{ll} & \Big\| \; i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T) \\ \xrightarrow{a} & \Big\| \; i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T). \end{array}$$

Let *count* and *count'* be as in the statement of the proposition. For every i in T, let $\psi_i : T \nrightarrow \{0\,..\,B + W_{\mathcal{N}}\}$ be an injective partial $(B + W_{\mathcal{N}})$-precollapsing function such that $\mathrm{dom}(\psi_i) = \mathrm{Im}(\Gamma_i)$. We prove the result using a case analysis on the kind of event that a can be.

Case 1. Suppose that a is an α-event of the form $c.f_\alpha(me, \mathit{payloadIDs})$ for some me in T (i.e. a is a private event of $\mathcal{N}_{me}(T)$).
Then

$$(cst_{me}, \Gamma_{me}, T) \xrightarrow{a}_{me} (cst'_{me}, \Gamma'_{me}, T) \tag{6.46}$$

and

$$\forall\, i \in T \setminus \{me\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T). \tag{6.47}$$

We now consider the various possibilities for the value of me.

Subcase 1. Suppose that me is in $\{0\,..\,B-1\}$.
Then (6.47) implies that, on letting $\psi'_i = \psi_i$ for all i in $\{0\,..\,B-1\} \setminus \{me\}$,

$$\begin{array}{l} \forall\, i \in \{0\,..\,B-1\} \setminus \{me\} \bullet \\ \quad (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B + W_{\mathcal{N}}\}) = (cst_i, \psi_i(\Gamma_i), \{0\,..\,B + W_{\mathcal{N}}\}). \end{array} \tag{6.48}$$

Using (6.46), Corollary 6.4.5 implies that there exists an injective partial $(B + W_{\mathcal{N}})$-precollapsing function ψ'_{me} such that $\mathrm{dom}(\psi'_{me}) = \mathrm{Im}(\Gamma'_{me})$, $\psi'_{me}(v) = \psi_{me}(v)$ for all v in $\mathrm{Im}(\Gamma_{me}) \cap \mathrm{Im}(\Gamma'_{me})$, and

$$\begin{array}{ll} & (cst_{me}, \psi_{me}(\Gamma_{me}), \{0\,..\,B + W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}}_{\psi_{me}(me)} & (cst'_{me}, \psi'_{me}(\Gamma'_{me}), \{0\,..\,B + W_{\mathcal{N}}\}), \end{array}$$

where $\hat{a}$ is like a, except that

- every value v in $\mathit{outputs}^t(a)$ is replaced by $\psi_{me}(v)$,
- every value v in $\mathit{inputs}^t_?(a)$ such that $\mathit{remembered}(v, a, \Gamma'_{me})$ is replaced by $\psi'_{me}(v)$, and
- every value v in $\mathit{inputs}^t_?(a) \cap \{B\,..\,\#T - 1\}$ such that $\neg \mathit{remembered}(v, a, \Gamma'_{me})$ is replaced by an arbitrary value in $\{B\,..\,B + W_{\mathcal{N}}\}$.

However, since me is in $\{0\,..\,B-1\}$, $\psi_{me}(me) = me$, so

$$\begin{array}{ll} & (cst_{me}, \psi_{me}(\Gamma_{me}), \{0\,..\,B + W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}}_{me} & (cst'_{me}, \psi'_{me}(\Gamma'_{me}), \{0\,..\,B + W_{\mathcal{N}}\}). \end{array} \tag{6.49}$$

From (6.47) we also have that

$$\forall\, i \in \{B\,..\,\#T-1\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T),$$

which means that

$$count' = count. \tag{6.50}$$

Finally, using ideas similar to those used in Lemma 6.3.4 and Lemma 6.3.7, we can show that

$$\hat{a} \in A(me, \{0\,..\,B+W_{\mathcal{N}}\}) \setminus \bigcup_{i \in \{0..B+W_{\mathcal{N}}\} \setminus \{me\}} A(i, \{0\,..\,B+W_{\mathcal{N}}\}).$$

Therefore, combining (6.48), (6.49) and (6.50) we have that

$$s \xrightarrow{\hat{a}} s'.$$

Hence, and since $\phi(\hat{a}) = \phi(a)$,

$$\phi(s) \xrightarrow{\phi(a)} \phi(s'),$$

as required.

Subcase 2. Suppose that me is in $\{B\,..\,\#T-1\}$.
Then me is not in $\{0\,..\,B-1\}$, so from (6.47) we have that

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T).$$

Hence, on letting $\psi'_i = \psi_i$ for all i in $\{0\,..\,B-1\}$,

$$\begin{array}{l} \forall\, i \in \{0\,..\,B-1\} \bullet \\ \qquad (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_{\mathcal{N}}\}) = (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\}). \end{array} \tag{6.51}$$

Let $\psi_{me} : T \nrightarrow \{0\,..\,B+W_{\mathcal{N}}\}$ be an injective partial $(B+W_{\mathcal{N}})$-precollapsing function such that $\mathrm{dom}(\psi_{me}) = \mathrm{Im}(\Gamma_{me})$. Using (6.46), Corollary 6.4.5 implies that there exists an injective partial $(B+W_{\mathcal{N}})$-precollapsing function ψ'_{me} such that $\mathrm{dom}(\psi'_{me}) = \mathrm{Im}(\Gamma'_{me})$, $\psi'_{me}(v) = \psi_{me}(v)$ for all v in $\mathrm{Im}(\Gamma_{me}) \cap \mathrm{Im}(\Gamma'_{me})$, and

$$\begin{array}{ll} & (cst_{me}, \psi_{me}(\Gamma_{me}), \{0\,..\,B+W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}}_{\psi_{me}(me)} & (cst'_{me}, \psi'_{me}(\Gamma'_{me}), \{0\,..\,B+W_{\mathcal{N}}\}), \end{array}$$

where $\hat{a}$ is like a, except that

- every value v in $outputs^t(a)$ is replaced by $\psi_{me}(v)$,
- every value v in $inputs^t_?(a)$ such that $remembered(v, a, \Gamma'_{me})$ is replaced by $\psi'_{me}(v)$, and
- every value v in $inputs^t_?(a) \cap \{B\,..\,\#T-1\}$ such that $\neg remembered(v, a, \Gamma'_{me})$ is replaced by an arbitrary value in $\{B\,..\,B+W_{\mathcal{N}}\}$.

However, me is in $\{B\,..\,\#T-1\}$ and also in $\mathrm{dom}(\psi_{me}) = \mathrm{Im}(\Gamma_{me})$ (since $\Gamma_{me}(myId)$ must equal me), so the definition of ψ_{me} implies that $\psi_{me}(me) = x$ for some $x \in \{B\,..\,B + W_{\mathcal{N}}\}$. Hence

$$\begin{array}{ll} & (cst_{me}, \psi_{me}(\Gamma_{me}), \{0\,..\,B + W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}}_x & (cst'_{me}, \psi'_{me}(\Gamma'_{me}), \{0\,..\,B + W_{\mathcal{N}}\}). \end{array} \tag{6.52}$$

In addition, by (6.47), it must be that

$$\forall\, i \in \{B\,..\,\#T-1\} \setminus \{me\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T).$$

Hence,

$$count' = count[(cst_{me}, \Gamma_{me}, T)^{--}, (cst'_{me}, \Gamma'_{me}, T)^{++}].$$

However, by Remark 6.3.3 we have that $\phi(\psi_{me}(\Gamma)) = \phi(\psi'_{me}(\Gamma)) = \phi(\Gamma)$ for every environment Γ. In addition, $\phi(\{0\,..\,B + W_{\mathcal{N}}\}) = \phi(T)$. Hence, by the definition of $\approx$ (Definition 6.1.11),

$$[(cst_{me}, \Gamma_{me}, T)] = [(cst_{me}, \psi_{me}(\Gamma_{me}), \{0\,..\,B + W_{\mathcal{N}}\})] \tag{6.53}$$

and

$$[(cst'_{me}, \Gamma'_{me}, T)] = [(cst'_{me}, \psi'_{me}(\Gamma'_{me}), \{0\,..\,B + W_{\mathcal{N}}\})]. \tag{6.54}$$

Therefore,

$$\begin{aligned} count' = count[&(cst_{me}, \psi_{me}(\Gamma_{me}), \{0\,..\,B + W_{\mathcal{N}}\})^{--}, \\ &(cst'_{me}, \psi'_{me}(\Gamma'_{me}), \{0\,..\,B + W_{\mathcal{N}}\})^{++}]. \end{aligned}$$

Further, $count[(cst_{me}, \Gamma_{me}, T)] \geq 1$, which, thanks to (6.53), means that $count[(cst_{me}, \psi_{me}(\Gamma_{me}), \{0\,..\,B + W_{\mathcal{N}}\})] \geq 1$. Hence, from (6.52) we can infer

$$count \xrightarrow{\hat{a}}_\infty count', \tag{6.55}$$

using the definition of $\longrightarrow_\infty$ (6.5). Finally, using ideas similar to those used in Lemma 6.3.4 and Lemma 6.3.7, we can show that

$$\hat{a} \in \left(\bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i, \{0\,..\,B + W_{\mathcal{N}}\})\right) \setminus \bigcup_{i=0}^{B-1} A(i, \{0\,..\,B + W_{\mathcal{N}}\}).$$

Therefore, combining (6.51) and (6.55) we have that

$$s \xrightarrow{\hat{a}} s'.$$

Hence, and since $\phi(\hat{a}) = \phi(a)$,

$$\phi(s) \xrightarrow{\phi(a)} \phi(s'),$$

as required.

Case 2. Suppose that $a = \tau$.
This case is almost identical to Case 1, above.

Case 3. Suppose that $a = c.f_\beta(x, y)$ for some distinct identities x and y in T (i.e. a is an event shared by $\mathcal{N}_x(T)$ and $\mathcal{N}_y(T)$).
Here, we consider the most interesting case, where x is in $\{0\,..\,B-1\}$ and y is in $\{B\,..\,\#T-1\}$.

The assumptions of this case imply that

$$(cst_x, \Gamma_x, T) \xrightarrow{a}_x (cst'_x, \Gamma'_x, T), \tag{6.56}$$

matching a visible symbolic event ϵ_x (see Defintion 4.6.1),

$$(cst_y, \Gamma_y, T) \xrightarrow{a}_y (cst'_y, \Gamma'_y, T), \tag{6.57}$$

matching a visible symbolic event ϵ_y, and

$$\forall\, i \in T \setminus \{x, y\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma, T). \tag{6.58}$$

Since y is not in $\{0\,..\,B-1\}$, from (6.58) we have that

$$\forall\, i \in \{0\,..\,B-1\} \setminus \{x\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T),$$

so, on letting $\psi'_i = \psi_i$ for all i in $\{0\,..\,B-1\}$,

$$\begin{array}{l}\forall\, i \in \{0\,..\,B-1\} \setminus \{x\} \bullet \\ \quad (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_{\mathcal{N}}\}) = (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\}).\end{array} \tag{6.59}$$

Using Corollary 6.4.5 we can infer from (6.56) that there exists an injective partial $(B + W_{\mathcal{N}})$-precollapsing function ψ'_x such that $\mathrm{dom}(\psi'_x) = \mathrm{Im}(\Gamma'_x)$, $\psi'_x(v) = \psi_x(v)$ for all v in $\mathrm{Im}(\Gamma_x) \cap \mathrm{Im}(\Gamma'_x)$, and

$$\begin{array}{ll} & (cst_x, \psi_x(\Gamma_x), \{0\,..\,B+W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}_x}_{\psi_x(x)} & (cst'_x, \psi'_x(\Gamma'_x), \{0\,..\,B+W_{\mathcal{N}}\}),\end{array}$$

where $\hat{a}_x$ is like a, except that

- every value v in $outputs^t(a, \epsilon_x)$ is replaced by $\psi_x(v)$,
- every value v in $inputs^t_?(a, \epsilon_x)$ such that $remembered(v, a, \epsilon_x, \Gamma'_x)$ is replaced by $\psi'_x(v)$, and
- every value v in $inputs^t_?(a, \epsilon_x) \cap \{B\,..\,\#T-1\}$ such that $\neg remembered(v, a, \epsilon_x, \Gamma'_x)$ is replaced by an arbitrary value in $\{B\,..\,B+W_{\mathcal{N}}\}$.

However, $\psi_x(x) = x$ (since x is in $\{0\,..\,B-1\}$), so

$$\begin{array}{ll} & (cst_x, \psi_x(\Gamma_x), \{0\,..\,B+W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}_x}_x & (cst'_x, \psi'_x(\Gamma'_x), \{0\,..\,B+W_{\mathcal{N}}\}).\end{array} \tag{6.60}$$

Also, since y is in $\{B\,..\,\#T-1\}$, the properties of $(B + W_{\mathcal{N}})$-precollapsing functions imply that $\psi_x(y)$ and $\psi'_x(y)$ are in $\{B\,..\,B+W_{\mathcal{N}}\}$. Therefore $\hat{a}_x = c.f_\beta(x, \widehat{y})$, where $\widehat{y}$ equals

- $\psi_x(y)$ if $y \in \mathit{outputs}^t(a, \epsilon_x)$,
- $\psi'_x(y)$ if $y \in \mathit{inputs}^t_?(a, \epsilon_x)$ and $\mathit{remembered}(y, a, \epsilon_x, \Gamma'_x)$, or
- some value in $\{B \,..\, B + W_{\mathcal{N}}\}$ if $y \in \mathit{inputs}^t_?(a, \epsilon_x)$ and $\neg \mathit{remembered}(y, a, \epsilon_x, \Gamma'_x)$.

The above three possibilities are pairwise disjoint thanks to Remark 4.1.3 (otherwise a contradiction to the assumptions about the alphabets of node processes is reached), so $\widehat{y}$ is well-defined. Note that in all cases we have that $\widehat{y}$ is in $\{B \,..\, B + W_{\mathcal{N}}\}$.

Let $\psi_y : T \nrightarrow \{0 \,..\, B + W_{\mathcal{N}}\}$ be an injective partial $(B + W_{\mathcal{N}})$-precollapsing function such that $\mathrm{dom}(\psi_y) = \mathrm{Im}(\Gamma_y)$. Using Corollary 6.4.5 we can infer from (6.57) that there exists an injective partial $(B + W_{\mathcal{N}})$-precollapsing function ψ'_y such that $\mathrm{dom}(\psi'_y) = \mathrm{Im}(\Gamma'_y)$, $\psi'_y(v) = \psi_y(v)$ for all v in $\mathrm{Im}(\Gamma_y) \cap \mathrm{Im}(\Gamma'_y)$ and

$$\begin{array}{ll} & (\mathit{cst}_y, \psi_y(\Gamma_y), \{0 \,..\, B + W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}_y}_{\psi_y(y)} & (\mathit{cst}'_y, \psi'_y(\Gamma'_y), \{0 \,..\, B + W_{\mathcal{N}}\}), \end{array} \tag{6.61}$$

where $\hat{a}_y$ is like a, except that

- every value v in $\mathit{outputs}^t(a, \epsilon_y)$ is replaced by $\psi_y(v)$,
- every value v in $\mathit{inputs}^t_?(a, \epsilon_y)$ such that $\mathit{remembered}(v, a, \epsilon_y, \Gamma'_y)$ is replaced by $\psi'_y(v)$, and
- every value v in $\mathit{inputs}^t_?(a, \epsilon_y) \cap \{B \,..\, \#T - 1\}$ such that $\neg \mathit{remembered}(v, a, \epsilon_y, \Gamma'_y)$ is replaced by an arbitrary value in $\{B \,..\, B + W_{\mathcal{N}}\}$.

We have that $\psi_y(x) = \psi'_y(x) = x$, since x is in $\{0 \,..\, B - 1\}$. In addition, clearly y must be in $\mathit{outputs}^t(a, \epsilon_y)$. Therefore,

$$\hat{a}_y = c.f_\beta(x, \psi_y(y)).$$

Let $\pi : \{0 \,..\, B + W_{\mathcal{N}}\} \to \{0 \,..\, B + W_{\mathcal{N}}\}$ be a bijection such that $\pi(v) = v$ for all v in $\{0 \,..\, B - 1\}$ and $\pi(\psi_y(y)) = \widehat{y}$. Since y is in $\{B \,..\, \#T - 1\}$ and y is in $\mathrm{dom}(\psi_y)$ (as $\Gamma_y(\mathit{myId})$ must equal y), the definition of ψ_y implies that $\psi_y(y)$ is in $\{B \,..\, B + W_{\mathcal{N}}\}$. In addition, $\widehat{y}$ is in $\{B \,..\, B + W_{\mathcal{N}}\}$. So π is well-defined. Then

$$\pi(\hat{a}_y) = c.f_\beta(\pi(x), \pi(\psi_y(y))) = c.f_\beta(x, \widehat{y}) = \hat{a}_x.$$

Therefore, using Corollary 6.4.6 (with $\pi' = \pi$), we can infer from (6.61) that

$$\begin{array}{ll} & (\mathit{cst}_y, \pi(\psi_y(\Gamma_y)), \{0 \,..\, B + W_{\mathcal{N}}\}) \\ \xrightarrow{\hat{a}_x}_{\pi(\psi_y(y))} & (\mathit{cst}'_y, \pi(\psi'_y(\Gamma'_y)), \{0 \,..\, B + W_{\mathcal{N}}\}). \end{array} \tag{6.62}$$

In addition, since x is not in $\{B \,..\, \#T - 1\}$, (6.58) implies that

$$\forall\, i \in \{B \,..\, \#T - 1\} \setminus \{y\} \bullet (\mathit{cst}'_i, \Gamma'_i, T) = (\mathit{cst}_i, \Gamma_i, T).$$

Hence,

$$\mathit{count}' = \mathit{count}[(\mathit{cst}_y, \Gamma_y, T)^{--}, (\mathit{cst}'_y, \Gamma'_y, T)^{++}]. \tag{6.63}$$

However, by Remark 6.3.3 and thanks to the properties of π we have that $\phi(\pi(\psi'_y(\Gamma))) = \phi(\pi(\psi_y(\Gamma))) = \phi(\Gamma)$ for every environment Γ. In addition, $\phi(\{0\,..\,B + W_{\mathcal{N}}\}) = \phi(T)$. Hence, by the definition of the $\approx$ equivalence relation (Definition 6.1.11)

$$[(cst_y, \Gamma_y, T)] = [(cst_y, \pi(\psi_y(\Gamma_y)), \{0\,..\,B + W_{\mathcal{N}}\})] \tag{6.64}$$

and

$$[(cst'_y, \Gamma'_y, T)] = [(cst'_y, \pi(\psi'_y(\Gamma'_y)), \{0\,..\,B + W_{\mathcal{N}}\})]. \tag{6.65}$$

Therefore, from (6.63) we have that

$$\begin{aligned} count' = count[&(cst_y, \pi(\psi_y(\Gamma_y)), \{0\,..\,B + W_{\mathcal{N}}\})^{--},\\ &(cst'_y, \pi(\psi'_y(\Gamma'_y)), \{0\,..\,B + W_{\mathcal{N}}\})^{++}]. \end{aligned}$$

Further, $count[(cst_y, \Gamma_y, T)] \geq 1$, which, thanks to (6.64), means that $count[(cst_y, \pi(\psi_y(\Gamma_y)), \{0\,..\,B + W_{\mathcal{N}}\})] \geq 1$. Hence, from (6.62), combined with the fact that $\pi(\psi_y(y))$ is in $\{B\,..\,B + W_{\mathcal{N}}\}$, we can infer that

$$count \xrightarrow{\hat{a}_x}_{\infty} count', \tag{6.66}$$

using the definition of $\longrightarrow_{\infty}$ (6.5).

Finally, using ideas similar to those used in Lemma 6.3.4 and Lemma 6.3.7, we can show that

$$\begin{aligned} \hat{a}_x \in \Big(&A(x, \{0\,..\,B + W_{\mathcal{N}}\}) \cap \textstyle\bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i, \{0\,..\,B + W_{\mathcal{N}}\})\Big)\\ &\setminus \textstyle\bigcup_{i \in \{0..B-1\}\setminus\{x\}} A(i, \{0\,..\,B + W_{\mathcal{N}}\}), \end{aligned}$$

Hence, and from (6.60), (6.59) and (6.66),

$$s \xrightarrow{\hat{a}_x} s'.$$

Therefore, since $\phi(\hat{a}_x) = \phi(a)$,

$$\phi(s) \xrightarrow{\phi(a)} \phi(s'),$$

as required.

Case 4. Suppose that $a = c.f_\gamma$ (i.e. a is an event shared by all nodes).
Then

$$\forall\, i \in T \bullet (cst_i, \Gamma_i, T) \xrightarrow{a}_i (cst'_i, \Gamma'_i, T).$$

Hence and since there are no values of type t within a, we can infer, using Corollary 6.4.5, that for every i in T there exists an injective partial $(B+W_{\mathcal{N}})$-precollapsing function ψ'_i such that $\mathrm{dom}(\psi'_i) = \mathrm{Im}(\Gamma'_i)$, $\psi'_i(v) = \psi_i(v)$ for all i in $\mathrm{Im}(\Gamma) \cap \mathrm{Im}(\Gamma')$ and

$$\forall\, i \in T \bullet (cst_i, \psi_i(\Gamma_i), \{0\,..\,B + W_{\mathcal{N}}\}) \xrightarrow{a}_{\psi_i(i)} (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B + W_{\mathcal{N}}\}).$$

Therefore, since for every i in $\{0\,..\,B-1\}$ we have that $\psi_i(i) = i$,

$$\begin{array}{l}\forall\, i \in \{0\,..\,B-1\} \bullet \\ \qquad (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\}) \xrightarrow{a}_i (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_{\mathcal{N}}\}),\end{array} \tag{6.67}$$

and, since for every i in $\{B\,..\,\#T-1\}$ we have that $\psi_i(i)$ is in $\{B\,..\,B+W_{\mathcal{N}}\}$,

$$\begin{array}{l}\forall\, i \in \{B\,..\,\#T-1\} \bullet \\ \qquad (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\}) \xrightarrow{a}_{x_i} (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_{\mathcal{N}}\}),\end{array} \tag{6.68}$$

for some x_i in $\{B\,..\,B+W_{\mathcal{N}}\}$.

For every i in $\{B\,..\,\#T-1\}$ and every environment Γ, Remark 6.3.3 implies that $\phi(\psi'_i(\Gamma)) = \phi(\psi_i(\Gamma)) = \phi(\Gamma)$. In addition, $\phi(\{0\,..\,B+W_{\mathcal{N}}\}) = \phi(T)$. Hence,

$$\forall\, i \in \{B\,..\,\#T-1\} \bullet [(cst_i, \Gamma_i, T)] = [(cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\})] \tag{6.69}$$

and

$$\forall\, i \in \{B\,..\,\#T-1\} \bullet [(cst'_i, \Gamma'_i, T)] = [(cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_{\mathcal{N}}\})]. \tag{6.70}$$

From (6.69) and by the definition of *count* we have that

$$\begin{array}{l}\forall\, i \in \{1\,..\,k\} \bullet \\ \quad \begin{array}{rl} count[st_i] &= \#\{j \in \{B\,..\,\#T-1\} \mid (cst_j, \Gamma_j, T) \approx st_i\} \\ &= \#\{j \in \{B\,..\,\#T-1\} \mid (cst_j, \psi_j(\Gamma_j), \{0\,..\,B+W_{\mathcal{N}}\}) \approx st_i\}. \end{array}\end{array}$$

Hence, for every i in $\{1\,..\,k\}$ there are $count[st_i]$ many states

$$\begin{array}{rcl} src^i_1 &=& (cst_{j^i_1}, \psi_{j^i_1}(\Gamma_{j^i_1}), \{0\,..\,B+W_{\mathcal{N}}\}), \\ src^i_2 &=& (cst_{j^i_2}, \psi_{j^i_2}(\Gamma_{j^i_2}), \{0\,..\,B+W_{\mathcal{N}}\}), \\ &\vdots& \\ src^i_{count[st_i]} &=& (cst_{j^i_{count[st_i]}}, \psi_{j^i_{count[st_i]}}(\Gamma_{j^i_{count[st_i]}}), \{0\,..\,B+W_{\mathcal{N}}\}), \end{array} \tag{6.71}$$

all related to $[st_i]$ by $\approx$, where for all i in $\{1\,..\,k\}$ and for all l in $\{1\,..\,count[st_i]\}$, the indices j^i_l are in $\{B\,..\,\#T-1\}$ and are all distinct. From (6.68) we have that

$$\forall\, i \in \{1\,..\,k\} \forall\, l \in \{1\,..\,count[st_i]\} \bullet src^i_l \xrightarrow{a}_{x^i_l} tgt^i_l, \tag{6.72}$$

where each x^i_l is in $\{B\,..\,B+W_{\mathcal{N}}\}$ and where, for all i in $\{1\,..\,k\}$,

$$\begin{array}{rcl} tgt^i_1 &=& (cst'_{j^i_1}, \psi'_{j^i_1}(\Gamma'_{j^i_1}), \{0\,..\,B+W_{\mathcal{N}}\}), \\ tgt^i_2 &=& (cst'_{j^i_2}, \psi'_{j^i_2}(\Gamma'_{j^i_2}), \{0\,..\,B+W_{\mathcal{N}}\}), \\ &\vdots& \\ tgt^i_{count[st_i]} &=& (cst'_{j^i_{count[st_i]}}, \psi'_{j^i_{count[st_i]}}(\Gamma'_{j^i_{count[st_i]}}), \{0\,..\,B+W_{\mathcal{N}}\}), \end{array} \tag{6.73}$$

with each j^i_l as above. Further,

$$\begin{array}{rl} & \wr (cst'_B, \psi'_B(\Gamma'_B), \{0\,..\,B+W_{\mathcal{N}}\}), \\ & \ldots, \\ & (cst'_{\#T-1}, \psi'_{\#T-1}(\Gamma'_{\#T-1}), \{0\,..\,B+1\}) \wr \\ = & \wr\, tgt^i_l \mid i \in \{1\,..\,k\}, l \in \{1\,..\,count[st_i]\} \wr, \end{array}$$

and

$$\#\{B \,..\, \#T - 1\} = \#\{(i, l) \mid i \in \{1 \,..\, k\} \wedge l \in \{1 \,..\, count[st_i]\}\}.$$

Hecem, from (6.70), we have that

$$\begin{aligned} &\forall\, st \in S \bullet \\ &\quad count'[st] = \#\{j \in \{B \,..\, \#T - 1\} \mid (cst'_j, \Gamma'_j, T) \approx st\} \\ &\qquad\qquad = \#\{j \in \{B \,..\, \#T - 1\} \mid (cst'_j, \psi(\Gamma'_j), \psi(T)) \approx st\} \\ &\qquad\qquad = \#\{(i, l) \mid i \in \{1 \,..\, k\} \wedge l \in \{1 \,..\, count[st_i]\} \wedge tgt^i_l \approx st\}. \end{aligned} \tag{6.74}$$

Combining (6.71)–(6.74) and the assumption of the proposition that $\#T \geq B+1$, we use the definition of $\longrightarrow_\infty$ (6.7) to infer that

$$count \xrightarrow{a}_\infty count'. \tag{6.75}$$

Finally, the definition of a γ-event implies that

$$\forall\, i \in \{0 \,..\, B + W_{\mathcal{N}}\} \bullet a \in A(i, \{0 \,..\, B + W_{\mathcal{N}}\}).$$

Hence, and from (6.67) and (6.75) we have that

$$s \xrightarrow{a} s',$$

which implies that

$$\phi(s) \xrightarrow{\phi(a)} \phi(s'),$$

as required. This completes our proof. ∎

In Proposition 6.4.7 we made the assumption that all β- and γ-events contain no payload identities, i.e. that every β-event is of the form $c.f_\beta(me, other)$ for some distinct identities me and $other$ in T and every γ-event is of the form $c.f_\gamma$. The following example shows that node identities present in payloads of β- events break the proposition. An example showing the same for γ-events is similar.

Example 6.4.8. Let

$$\begin{aligned} \mathcal{N}_{myId}(t) = env!myId?x{:}t \rightarrow (&sync!myId?other{:}t!x \rightarrow STOP \\ &\Box \\ &a \rightarrow sync?other{:}t!myId!x \rightarrow STOP). \end{aligned}$$

Observe that constructs $sync!myId?other{:}t!x$ and $sync?other{:}myId!x$ can generate β-events that contain payload of type t (i.e. the identity x). Since $\mathcal{N}_{myId}(t)$ remembers only at most one node identity (x) in addition to its own, we have that $W_{\mathcal{N}} = 1$. Let $B = 2$ and $T = \{0 \,..\, 2\}$. Also, let

$$\begin{aligned} cst_0 = &sync!myId?other{:}t!x \rightarrow STOP \\ &\Box \\ &a \rightarrow sync?other{:}t!myId!x \rightarrow STOP, \end{aligned}$$

$cst_1 = sync?other{:}t!myId!x \rightarrow STOP,$

$\forall\, i \in \{0,1\} \bullet cst'_i = STOP$

and

$$\Gamma_0 = \{myId \mapsto 0, x \mapsto 2\}, \qquad \Gamma'_0 = \{myId \mapsto 0\},$$
$$\Gamma_1 = \{myId \mapsto 1, x \mapsto 2\}, \qquad \Gamma'_1 = \{myId \mapsto 1\}.$$

Then, for all environments Γ_2 and Γ'_2 such that $\Gamma_2 = \Gamma'_2$ and all control states cst_2 and cst'_2 such that $cst_2 = cst'_2$, we have that

$$\Big\|\ i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T)$$
$$\xrightarrow{sync.0.1.2} \Big\|\ i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T).$$

Now, let $\psi_0, \psi_1 : T \nrightarrow \{0\,..\,B + W_{\mathcal{N}}\}$ be such that

$$\psi_0(0) = 0, \qquad \psi_0(2) = B,$$
$$\psi_1(1) = 1, \qquad \psi_1(2) = B + 1.$$

Then for both $i = 0$ and $i = 1$, ψ_i is an injective partial $(B + W_{\mathcal{N}})$-precollapsing function such that $\mathrm{dom}(\psi_i) = \mathrm{Im}(\Gamma_i)$. However,

$$\psi_0(\Gamma_0) = \{myId \mapsto 0, x \mapsto B\},$$
$$\psi_1(\Gamma_1) = \{myId \mapsto 1, x \mapsto B + 1\},$$

so

$$\Big\|\ i \in \{0,1\} \bullet [A(i, \{0\,..\,B + W_{\mathcal{N}}\})]\ (cst_i, \psi_i(\Gamma_i), \{0\,..\,B + W_{\mathcal{N}}\})$$

cannot do $\phi(sync.0.1.2) = sync.0.1.2$. Hence, if s is defined as in Proposition 6.4.7, then $\phi(s) \xrightarrow{\phi(sync.0.1.2)}\!\!\!\!\!\!/\ $.

End of example.

Our next result says that given nodes that can remember at least one identity (in addition to their own), if $\phi(Nodes(T))$ can perform a sequence of events s and reach some state P, then $ABS_\infty^{T,B,W_{\mathcal{N}}}$ can also perform s and reach a state corresponding to P.

Lemma 6.4.9. *Suppose that $W_{\mathcal{N}} \geq 1$ and suppose that all β- and γ-events contain no payloads with node identities. Let T be an instantiation of type t such that $\#T \geq B + 1$. Then if*

$$\phi(Nodes(T)) \xrightarrow{s} \phi\left(\Big\|\ i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T)\right)$$

for some control states cst_i and environments Γ_i, then for all i in $\{0\,..\,B-1\}$, there exist an injective partial $(B + W_{\mathcal{N}})$-precollapsing function $\psi_i : T \nrightarrow \{0\,..\,B + W_{\mathcal{N}}\}$

such that $\mathrm{dom}(\psi_i) = \mathrm{Im}(\Gamma_i)$ *and*

$$ABS_{\infty}^{T,B,W_{\mathcal{N}}} \xrightarrow{s} \phi \left(\begin{array}{c} \left(\Big\| \; i \in \{0\,..\,B-1\} \bullet \right. \\ \left. [A(i,\{0\,..\,B+W_{\mathcal{N}}\})]\ (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\})\right) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_{\mathcal{N}}\}) \Big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ count \end{array} \right),$$

where $count$ *is a state of* $\zeta_{\infty}^{T}(\mathcal{N}_B(\{0\,..\,B+W_{\mathcal{N}}\}))$ *such that for all states* st*,*

$$count[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst_i, \Gamma_i, T) \approx st\}.$$

Proof: We prove the result by an induction on the length of s.

Base Case. Suppose that $s = \langle\rangle$.

Then, for all i in $\{0\,..\,B-1\}$, $\Gamma_i = \{myId \mapsto i\}$. Now, for every i in $\{0\,..\,B-1\}$ let $\psi_i : T \twoheadrightarrow \{0\,..\,B+W_{\mathcal{N}}\}$ be such that $\mathrm{dom}(\psi_i) = \{i\}$ and $\psi_i(i) = i$. Then, for every i in $\{0\,..\,B-1\}$, ψ_i is an injective partial $(B+W_{\mathcal{N}})$-precollapsing function from T to $\{0\,..\,B+W_{\mathcal{N}}\}$ such that $\mathrm{dom}(\psi_i) = \mathrm{Im}(\Gamma_i)$.

Observe that $(cst_i, \Gamma_i, T) = \mathcal{N}_i(T)$ for all i in T. Therefore, by Remark 6.1.4 we have that

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\}) = \mathcal{N}_{\psi_i(i)}(\{0\,..\,B+W_{\mathcal{N}}\}).$$

However, for all i in $\{0\,..\,B-1\}$, $\psi_i(i) = i$. Hence,

$$\forall\, i \in \{0\,..\,B-1\} \bullet (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\}) = \mathcal{N}_i(\{0\,..\,B+W_{\mathcal{N}}\})$$

Also, by the definition of *count*,

$$count = (\#T - B, 0, \ldots, 0).$$

Therefore, since

$$ABS_{\infty}^{T,B,W_{\mathcal{N}}} = \phi \left(\begin{array}{c} \Big\| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B+W_{\mathcal{N}}\})]\ \mathcal{N}_i(\{0\,..\,B+W_{\mathcal{N}}\}) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_{\mathcal{N}}\}) \Big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ (\#T - B, 0, \ldots, 0) \end{array} \right),$$

we have that

$$ABS_{\infty}^{T,B,W_{\mathcal{N}}} \xrightarrow{\langle\rangle} \phi \left(\begin{array}{c} \left(\Big\| \; i \in \{0\,..\,B-1\} \bullet \right. \\ \left. [A(i,\{0\,..\,B+W_{\mathcal{N}}\})]\ (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\})\right) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_{\mathcal{N}}\}) \Big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ count \end{array} \right),$$

establishing the result in the base case.

Inductive Case. Suppose the result holds for sequence s.

Let $sa = s^\frown\langle a\rangle$ for some event a. Suppose that

$$\phi(Nodes(T)) \stackrel{sa}{\longmapsto} \phi\left(\Big\| \ i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T)\right)$$

for some control states cst'_i and environments Γ'_i. Then, for every i in T there exists a control state cst_i and an environment Γ_i such that

$$\begin{aligned}\phi(Nodes(T)) &\stackrel{s}{\longmapsto} \phi\left(\Big\| \ i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T)\right) \\ &\stackrel{a}{\longrightarrow} \phi\left(\Big\| \ i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T)\right).\end{aligned} \tag{6.76}$$

Then, by the inductive hypothesis, for i in $\{0\,..\,B-1\}$, there exist injective partial $(B+W_{\mathcal{N}})$-precollapsing functions $\psi_i : T \nrightarrow \{0..B+W_{\mathcal{N}}\}$ such that $\mathrm{dom}(\psi_i) = \mathrm{Im}(\Gamma_i)$ and

$$\begin{aligned}&ABS^{T,B,W_{\mathcal{N}}}_{\infty} \\ &\stackrel{s}{\longmapsto} \phi\left(\begin{array}{c}\left(\Big\| \ i \in \{0\,..\,B-1\} \bullet \right. \\ \left. [A(i,\{0\,..\,B+W_{\mathcal{N}}\})]\ (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\})\right) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_{\mathcal{N}}\}) \Big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ count\end{array}\right),\end{aligned} \tag{6.77}$$

where $count$ is a state of $\zeta^T_\infty(\mathcal{N}_B(\{0\,..\,B+W_{\mathcal{N}}\}))$ such that for all states st,

$$count[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst_i, \Gamma_i, T) \approx st\}.$$

From (6.76) we can infer that there exists an event a' such that $\phi(a') = a$ and

$$\Big\| \ i \in T \bullet [A(i,T)]\ (cst_i, \Gamma_i, T) \ \stackrel{a'}{\longrightarrow}\ \Big\| \ i \in T \bullet [A(i,T)]\ (cst'_i, \Gamma'_i, T).$$

Hence, by Proposition 6.4.7 we have that for i in $\{0\,..\,B-1\}$, there exist injective partial $(B+W_{\mathcal{N}})$-precollapsing functions $\psi'_i : T \nrightarrow \{0..B+W_{\mathcal{N}}\}$ such that $\mathrm{dom}(\psi'_i) = \mathrm{Im}(\Gamma'_i)$ and

$$\begin{aligned}&\phi\left(\begin{array}{c}\left(\Big\| \ i \in \{0\,..\,B-1\} \bullet \right. \\ \left. [A(i,\{0\,..\,B+W_{\mathcal{N}}\})]\ (cst_i, \psi_i(\Gamma_i), \{0\,..\,B+W_{\mathcal{N}}\})\right) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_{\mathcal{N}}\}) \Big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ count\end{array}\right) \\ \stackrel{\phi(a')}{\longrightarrow}\ &\phi\left(\begin{array}{c}\left(\Big\| \ i \in \{0\,..\,B-1\} \bullet \right. \\ \left. [A(i,\{0\,..\,B+W_{\mathcal{N}}\})]\ (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_{\mathcal{N}}\})\right) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_{\mathcal{N}}\}) \Big\| \bigcup_{i=B}^{B+W_{\mathcal{N}}} A(i,\{0..B+W_{\mathcal{N}}\}) \\ count'\end{array}\right),\end{aligned}$$

where $count'$ is a state of $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+W_\mathcal{N}\}))$ such that for all states st,

$$count'[st] = \#\{i \in \{B\,..\,\#T-1\} \mid (cst'_i, \Gamma'_i, T) \approx st\}.$$

By combining the above with (6.77) and by recalling that $\phi(a') = a$, we get that

$$ABS_\infty^{T,B,W_\mathcal{N}} \xrightarrow{sa} \phi \left(\begin{array}{c} \left(\!\!\left\| \; i \in \{0\,..\,B-1\} \bullet \right.\right. \\ \quad [A(i,\{0\,..\,B+W_\mathcal{N}\})]\ (cst'_i, \psi'_i(\Gamma'_i), \{0\,..\,B+W_\mathcal{N}\})\big) \\ \bigcup_{i=0}^{B-1} A(i,\{0..B+W_\mathcal{N}\}) \Big\|_{\bigcup_{i=B}^{B+W_\mathcal{N}} A(i,\{0..B+W_\mathcal{N}\})} \\ count' \end{array} \right),$$

which completes our inductive proof. ∎

The following corollary is a simple, but important consequence of Lemma 6.4.9.

Corollary 6.4.10. *Suppose that $W_\mathcal{N} \geq 1$ and suppose that all β- and γ-events contain no payloads with node identities. Let T be an instantiation of type t such that $\#T \geq B+1$. Then*

$$ABS_\infty^{T,B,W_\mathcal{N}} \sqsubseteq_\mathrm{T} \phi(Nodes(T)).$$

Proof: Let tr be a trace of $\phi(Nodes(T))$ and let s be a sequence of events such that $s \setminus \{\tau\} = tr$ and

$$\phi(Nodes(T)) \xrightarrow{s} .$$

Then, Lemma 6.4.9 implies that

$$ABS_\infty^{T,B,W_\mathcal{N}} \xrightarrow{s},$$

which means that tr is a trace of $ABS_\infty^{T,B,W_\mathcal{N}}$. ∎

6.5 Counter abstraction with finite thresholds

In the previous sections we showed how to create an abstract state machine based on integer counters that is traces-refined by the parallel composition of all the node processes, renamed under ϕ. Even though the alphabet of such a counter machine is fixed and finite, its state spaces still depends upon the size of T, since each counter can take values between 0 and $\#T - B$.

Let n be a non-negative integer. In this section we create an abstract model $ABS_z^{B,n}$ such that, for certain values of n (to be established later) and T large enough,

$$ABS_z^{B,n} \sqsubseteq \phi(Nodes(T)). \tag{6.78}$$

$ABS_z^{B,n}$ is very similar to $ABS_\infty^{T,B,n}$, but is no longer dependent upon the size of T. To achieve this we introduce a *threshold function* z that defines upper bounds on the values that counters can take. Informally, z_j forms a cap on the counter c_j for

equivalence class $[st_j]$; we interpret $c_j = z_j$ as meaning that c_j *or more* processes are in a state from $[st_j]$.

In Section 6.5.1 we will define a counter state machine $\zeta_z(\mathcal{N}_B(\{0\,..\,B+m\}))$ that abstracts the nodes $\mathcal{N}_B(\{0\,..\,B+m\}), \ldots, \mathcal{N}_{\#T-1}(\{0\,..\,B+m\})$. The full abstract model, for $n = 0$, and $n = m$ for $m > 0$, respectively, is:

$$ABS_z^{B,0} \mathrel{\widehat{=}} \left(\begin{array}{c} \| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B\})]\; \mathcal{N}_i(\{0\,..\,B\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B\})} \|_{A_\zeta^B} \\ \zeta_z(\mathcal{N}_B(\{0\,..\,B+1\}))[\![\mathcal{R}]\!] \end{array} \right)$$

and

$$ABS_z^{B,m} \mathrel{\widehat{=}} \phi \left(\begin{array}{c} \| \; i \in \{0\,..\,B-1\} \bullet [A(i,\{0\,..\,B+m\})]\; \mathcal{N}_i(\{0\,..\,B+m\}) \\ {\scriptstyle \bigcup_{i=0}^{B-1} A(i,\{0..B+m\})} \|_{\bigcup_{i=B}^{B+m} A(i,\{0..B+m\})} \\ \zeta_z(\mathcal{N}_B(\{0\,..\,B+m\})) \end{array} \right),$$

where $\mathcal{R}$ is, as before, a renaming that replaces every instance of $B+1$ by B.

In Section 6.5.2 we prove that, given large enough T, $ABS_z^{B,0}$ and $ABS_z^{B,W_\mathcal{N}}$ form traces anti-refinements of $ABS_\infty^{T,B,0}$ and $ABS_\infty^{T,B,W_\mathcal{N}}$, respectively. Hence, using the results from Section 6.3 and Section 6.4, they form anti-refinements of $\phi(Nodes(T))$ for sufficiently large instantiations T of type t.

6.5.1 Constructing the counter state machine

The techniques described in this section are similar to those presented in Section 3.3.1. Recall that

- $(S(i), s_0(i), \Sigma(i,\{0\,..\,B+m\}) \cup \{\tau\}, \longrightarrow_i)$ is the state machine of node $\mathcal{N}_i(\{0\,..\,B+m\})$ for i in $\{B\,..\,B+m\}$,
- $S = \bigcup\{S(i) \mid i \in \{B\,..\,B+m\}\}$ is the set of states of nodes $\mathcal{N}_B(\{0\,..\,B+m\}), \ldots, \mathcal{N}_{B+m}(\{0\,..\,B+m\})$,
- $E = \{[st_1], \ldots, [st_k]\} = S/{\approx}$, and
- $[st_1]$ is the equivalence class within E that contains all the initial states $s_0(i)$ for i in $\{B\,..\,B+m\}$.

Let z be a function from $\{1..k\}$ to $\mathbb{N}$. For every i in $\{1..k\}$, z_i is a threshold for the counter corresponding to $[st_i]$. Given a state st, we make the notational convention that $z[st] = z_i$, where i is the unique element in $\{1\,..\,k\}$ such that $[st] = [st_i]$.

Our aim is to create an abstract state machine

$$\zeta_z(\mathcal{N}_B(\{0\,..\,B+m\})) = (A_z, a_z, \Sigma_z^\tau, \longrightarrow_z),$$

which is t-independent and which, for all sufficiently large T, forms an abstraction of $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+m\}))$.

In order to ensure that we can uniquely define the initial state, a_z (see below), for the rest of this section we assume that $\#T \geq B + \max\{z_1 + 1\}$ (B and z_1 are usually

small in practice, so all the refinement checks for $\#T < B + \max\{z_1 + 1\}$ can be performed directly). Then the states in A_z are k-tuples of counters $(c_1, \ldots, c_k)$ such that

(i) for every i in $\{1 \,..\, k\}$, c_i is an integer between 0 and z_i, and

(ii) either there exists i in $\{1 \,..\, k\}$ such that $c_i = z_i$ or $\sum_{i=1}^{k} c_i \geq z_1$.

Observe that requirement (ii) means that we only allow tuples of counters that imply a possibility of the presence of at least z_1 nodes with identities in $\{B \,..\, B + m\}$. This, together with the definition of the initial state, below, ensures counters never imply the presence of fewer processes than we started with.

The addition of the threshold function requires a new definition of the initial state. Since $\#T \geq B + z_1$, there must be at least z_1 nodes with identities greater or equal to B, all in their initial states. Hence, we let

$$a_z = (z_1, 0, \ldots, 0).$$

The alphabet of the abstract state machine is the same as the alphabet of every counter state machine with unbounded counters, i.e.

$$\Sigma_z = \bigcup\{\Sigma(i, \{0 \,..\, B + m\}) \mid i \in \{B \,..\, B + m\}\}$$

and

$$\Sigma_z^\tau = \Sigma_z \cup \{\tau\}.$$

Finally, we define the transition relation, $\longrightarrow_z$. Let *count* and *count*$'$ be two states in A_z. Intuitively, there is an a-transition available between *count* and *count*$'$ if there are some local states where a is available, the values of the counters of *count* indicate there are enough processes in those states, and the counters of *count*$'$ are updated correctly, without exceeding their thresholds, to reflect the number of processes in corresponding equivalence classes after a is executed. We define the transition relation $\xrightarrow{a}_z$ by considering the type of event that a can be.

Let $a = \tau$. Then, there is a τ-transition between *count* and *count*$'$ if the following hold.

- There is a local state st in $S(me)$, for some me in $\{B \,..\, B + m\}$, in which τ can be executed to reach some state st'.
- Let i in $\{1..k\}$ be such that $st \in [st_i]$. If $z_i > 0$, then the counter corresponding to $[st_i]$ is at least 1. If $z_i = 0$, there can always be an arbitrary number of processes in states in $[st_i]$, so we always enable the transition (this means that the counter model may form a strict anti-refinement of the concrete model).
- The counters of *count*$'$ are updated correctly. If $[st] \neq [st']$, then we decrease *count*$[st]$ by 1, without going below 0, to get *count*$'[st]$, and increase *count*$[st']$ by 1, without going over threshold $z[st']$, to get *count*$'[st']$, with both operations happening atomically. In addition, if *count*$[st]$ has reached its threshold, $z[st]$,

then a second transition is available, since there could be more than $z[st]$ nodes in states in $[st]$, so after a happens there would still be at least $z[st]$ nodes in states in $[st]$ (i.e. $count'[st]$ must equal $count[st]$) and $count[st']$ is increased by 1, without going over threshold $z[st']$, to get $count'[st']$ (this again means that the counter model may form a strict anti-refinement of the concrete model). If, however, $[st] = [st']$, then all counters remain unchanged (i.e. $count' = count$).

For brevity, given a tuple of counters $count$, we let

$$count[st^{++}{}_z] = count[st \leftarrow \min\{count[st] + 1, z[st]\}],$$

and

$$count[st^{--}{}_z] = count[st \leftarrow \max\{count[st] - 1, 0\}].$$

Then, formally:

$$\begin{array}{ll} & count \xrightarrow{\tau}_z count' \\ \Leftrightarrow & \exists\, me \in \{B \,..\, B + m\} \bullet \\ & \quad \exists\, st, st' \in S(me) \bullet \\ & \qquad st \xrightarrow{\tau}_{me} st' \wedge count[st] \geq \min\{z[st], 1\} \\ & \qquad \wedge\, (count' = count[st^{--}{}_z, st'^{++}{}_z] \\ & \qquad\quad \vee\, count[st] = z[st] \wedge count' = count[st'^{++}{}_z]). \end{array} \tag{6.79}$$

Next, let me be some identity in $\{B \,..\, B + m\}$. Let a be either

- an α-event of the form $c.f_\alpha(me, payloadIDs)$, or
- a β-event of the form $c.f_\beta(me, other, payloadIDs)$ or $c.f_\beta(other, me, payloadIDs)$, where $other \in \{0 \,..\, B - 1\}$.

This case is almost identical to the case above. Formally, for every a as above,

$$\begin{array}{ll} & count \xrightarrow{a}_z count' \\ \Leftrightarrow & \exists\, st, st' \in S(me) \bullet \\ & \quad st \xrightarrow{a}_{me} st' \wedge count[st] \geq \min\{z[st], 1\} \\ & \quad \wedge\, (count' = count[st^{--}{}_z, st'^{++}{}_z] \\ & \qquad \vee\, count[st] = z[st] \wedge count' = count[st'^{++}{}_z]). \end{array} \tag{6.80}$$

Now, let me and $other$ be two distinct identities in $\{B \,..\, B + m\}$. Let a be a β-event of the form $c.f_\beta(me, other, payloadIDs)$. Then, there is an a-labelled transition between $count$ and $count'$ if the following hold.

- There are concrete states st_{me} in $S(me)$ and st_{other} in $S(other)$, in which a can be executed to reach states st'_{me} and st'_{other}, respectively.
- If $[st_{me}] = [st_{other}]$, then the counters of $count$ corresponding to $[st_{me}]$ and $[st_{other}]$ are at least 2, or their threshold is 0 or 1 and has been reached; if $[st_{me}] \neq [st_{other}]$ then each of $count[st_{me}]$ and $count[st_{other}]$ either is at least 1 or its threshold is 0.

- The counters of $count'$ are updated correctly: we decrease the counters corresponding to $[st_{me}]$ and $[st_{other}]$ by 1, without going below 0, and increase the counters corresponding to $[st'_{me}]$ and $[st'_{other}]$ by 1, without going over their thresholds, with all the operations happening atomically; if the counter corresponding to either $[st_{me}]$ or $[st_{other}]$ is at its limit, then we also have the option to leave it unmodified.

Formally, for every a as above,

$$
\begin{array}{ll}
 & count \xrightarrow{a}_z count' \\
\Leftrightarrow & \exists\, st_{me}, st'_{me} \in S(me), st_{other}, st'_{other} \in S(other) \bullet \\
 & \quad st_{me} \xrightarrow{a}_{me} st'_{me} \wedge st_{other} \xrightarrow{a}_{other} st'_{other} \\
 & \quad \wedge\ count[st_{me}] \geq \min\{z[st_{me}], p\} \\
 & \quad \wedge\ count[st_{other}] \geq \min\{z[st_{other}], p\} \\
 & \quad \wedge\ (count' = count[st_{me}{}^{--}{}_z, st_{other}{}^{--}{}_z, st'_{me}{}^{++}{}_z, st'_{other}{}^{++}{}_z] \\
 & \qquad \vee\ count[st_{me}] = z[st_{me}] \\
 & \qquad\quad \wedge\ count' = count[st_{other}{}^{--}{}_z, st'_{me}{}^{++}{}_z, st'_{other}{}^{++}{}_z] \\
 & \qquad \vee\ count[st_{other}] = z[st_{other}] \\
 & \qquad\quad \wedge\ count' = count[st_{me}{}^{--}{}_z, st'_{me}{}^{++}{}_z, st'_{other}{}^{++}{}_z] \\
 & \qquad \vee\ count[st_{me}] = z[st_{me}] \wedge count[st_{other}] = z[st_{other}] \\
 & \qquad\quad \wedge\ count' = count[st'_{me}{}^{++}{}_z, st'_{other}{}^{++}{}_z]), \\
 & \quad \text{where } p = 2 \text{ if } [st_{me}] = [st_{other}] \text{ and } p = 1 \text{ otherwise.}
\end{array}
\tag{6.81}
$$

Finally, let a be a γ-event. We want $count$ and $count'$ to be related by $\xrightarrow{a}_z$ if and only if there exist a-transitions that allow simultaneous movement of all node processes modelled by $count$ such that for every i in $\{1\,..\,k\}$, a number of processes indicated by $count[st_i]$ can move from states in $[st_i]$, and a number of processes indicated by $count'[st_i]$ can move into states in $[st_i]$. Recall that if for i in $\{1\,..\,k\}$, the counter value $count[st_i]$ is equal to its threshold z_i, then the actual number of processes in states within $[st_i]$, call it v_i, is greater or equal to $count[st_i]$. It must be that $\sum_{i=1}^{k} v_i = \#T - B \geq \max\{z_1, 1\}$. Therefore, a movement of all the processes with identities greater or equal to B can happen if and only if there exists $v : \{1\,..\,k\} \to \mathbb{N}$ such that:

(i) $\sum_{i=1}^{k} v_i \geq \max\{z_1, 1\}$,

(ii) for i in $\{1\,..\,k\}$, if $count[st_i] < z_i$, then $v_i = count[st_i]$, and if $count[st_i] = z_i$, then $v_i \geq count[st_i]$, and

(iii) there exists a concrete model such that for every i in $\{1\,..\,k\}$, precisely v_i node processes within the model are in states within $[st_i]$, where they can perform a to reach some target states; if $count'[st_i] < z_i$, then precisely $count'[st_i]$ of all the target states are in $[st_i]$; and if $count'[st_i] = z_i$, then at least $count'[st_i]$ of all the target states are in $[st_i]$.

This is illustrated in Figure 6.2, where we seek to find transitions from states on the left-hand side to the states on the right-hand side such that

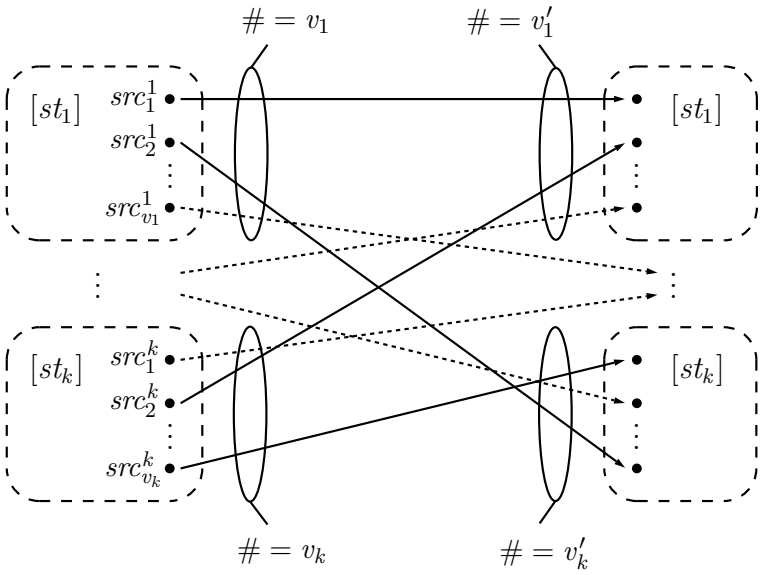


Figure 6.2: Seeking transitions between the states of S such that for every i in $\{1..k\}$, $count[st_i]$ processes can move from states in $[st_i]$, and $count'[st_i]$ processes can move into states in $[st_i]$.

- for every i in $\{1..k\}$ there are v_i transitions leaving equivalence class $[st_i]$, where either $v_i = count[st_i] < z_i$ or $v_i \geq count[st_i] = z_i$, and
- for every i in $\{1 \,..\, k\}$ there are v'_i transitions entering equivalence class $[st_i]$, where either $v'_i = count'[st_i] < z_i$ or $v_i \geq count'[st_i] = z_i$.

Formally, for every γ-event a in Σ_z,

$$
\begin{array}{l}
count \xrightarrow{a}_z count' \qquad (6.82)\\
\Leftrightarrow\\
\exists\, v : \{1 \,..\, k\} \to \mathbb{N} \bullet\\
\quad \exists\, src, tgt : \mathbb{N} \times \mathbb{N} \to S \bullet\\
\qquad \sum_{i=1}^{k} v_i \geq \max\{z_1, 1\}\\
\qquad \wedge\ (\forall\, i \in \{1 \,..\, k\} \bullet v_i = count[st_i] < z_i \vee count[st_i] = z_i \leq v_i)\\
\qquad \wedge\ (\forall\, i \in \{1 \,..\, k\}, j \in \{1 \,..\, v_i\} \bullet\\
\qquad\qquad src^i_j \approx st_i \wedge \exists\, me \in \{B \,..\, B + m\} \bullet src^i_j \xrightarrow{a}_{me} tgt^i_j)\\
\qquad \wedge\ \forall\, st \in S \bullet count'[st] = \min\{z[st], \#\{(i,j) \mid i \in \{1 \,..\, k\} \wedge j \in \{1 \,..\, v_i\}\\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ tgt^i_j \approx st\}\}.
\end{array}
$$

By clause (iii) of Remark 6.1.9, the four parts of the definition of $\longrightarrow_z$ are pairwise disjoint, so they combine to a well-defined definition of the transition relation.

The way (6.82) is formulated requires one to check infinitely many possibilities for v in order to deduce transitions in an abstract model. This is a significant practical problem, which we solve by providing upper bounds for the values of v that need to be considered.

Proposition 6.5.1. *For every i in $\{1\,..\,k\}$, $\max\{\sum_{j\in targets(i)} count'[st_j], z_i, z_1, 1\}$ is an upper bound for the values of v_i that need to be considered in the body of (6.82), where $targets(i) = \{j \in \{1\,..\,k\} \mid \exists\, st, st' \in S, me \in \{B\,..\,B+m\} \bullet st \approx st_i \wedge st \xrightarrow{e}_{me} st' \wedge st' \approx st_j\}$.*

Proof: Similar to the proof of Proposition 6.5.1. ∎

Example 6.5.2. Recall the $\mathcal{N}_{myId}(t)$ process from Example 6.2.1. Let z be threshold functions such that $z_1 = z_2 = z_3 = 1$. We use the method described above to construct $\zeta_z(\mathcal{N}_1(\{0,1,2\}))$, as shown in Figure 6.3.

End of example.

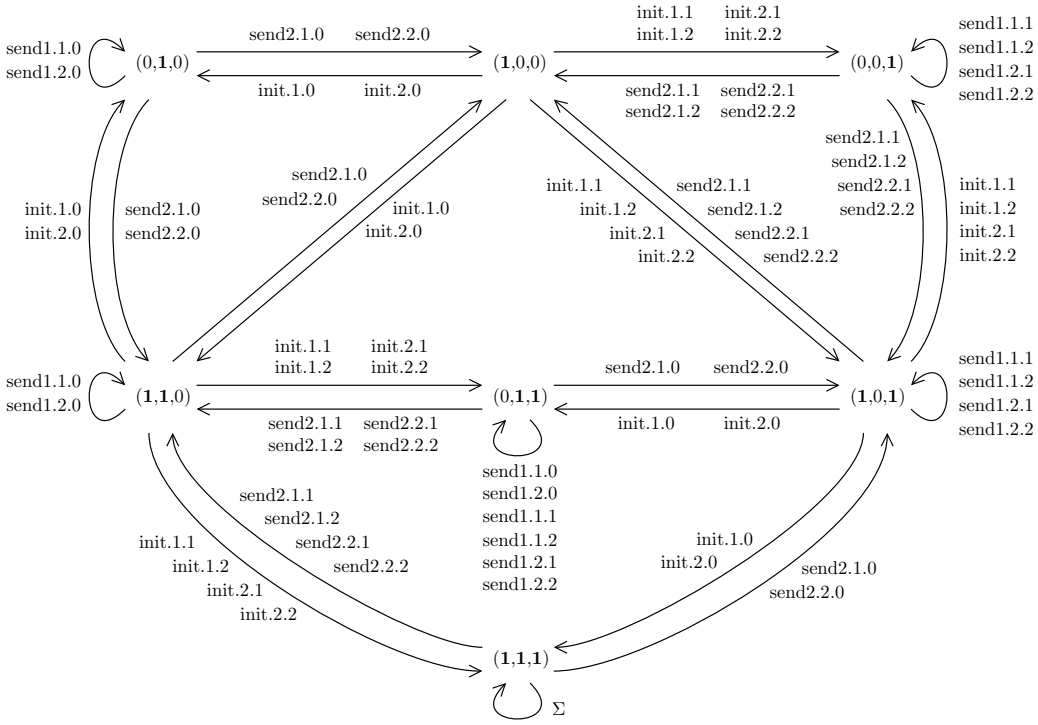


Figure 6.3: Counter state machine $\zeta_z(\mathcal{N}_1(\{0,1,2\}))$ for $z_1 = z_2 = z_3 = 1$ and $\mathcal{N}_B(\{0\,..\,B+m\})$ defined as in Example 6.2.1. Counters that have attained their thresholds are indicated using bold font.

6.5.2 Refinement results

It is intuitive to expect that counter abstracting a process using thresholds creates an anti-refinement of a counter machine, for the same process, without thresholds. We prove this formally in this section. This, combined with the results of Section 6.3 and Section 6.4 shows that models obtained using counter abstraction with thresholds are abstractions of ϕ-renamed parallel compositions of nodes.

Recall Definition 3.3.5 that says that, given type T and function $z : \{1\,..\,k\} \to \mathbb{N}$, states $s_1 = (c_1, \ldots, c_k)$ in $A_\infty(T)$, and $s_2 = (d_1, \ldots, d_k)$ in A_z, s_1 and s_2 are *z-corresponding*, written $cor_z(s_1, s_2)$, if $d_i = c_i \sqcap z_i$ for all $i \in \{1\,..\,k\}$.

The following lemma says that, given a threshold function z, two states in $\mathcal{A}_z$ are related by $\xrightarrow{a}_z$ if and only if there exist two z-corresponding states in $\mathcal{A}_\infty(T)$, related by $\xrightarrow{a}_\infty(T)$, for some T of size at least $B + \max\{z_1, 1\}$.

Lemma 6.5.3. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let d, d' be two states in A_z. Then for all events a in Σ_z^τ we have that*

$$\begin{array}{ll} & d \xrightarrow{a}_z d' \\ \Leftrightarrow & \exists\, T \bullet \#T \geq B + \max\{z_1, 1\} \\ & \qquad \wedge\, \exists\, c, c' \in A_\infty(T) \bullet cor_z(c, d) \wedge cor_z(c', d') \wedge c \xrightarrow{a}_\infty(T)\, c'. \end{array}$$

Proof: Similar to the proof of Lemma 3.3.6. ∎

Corollary 6.5.4. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let T be an instantiation of type t of size at least $B + \max\{z_1, 1\}$. Then, for all events a in $\Sigma_z^\tau = \Sigma_\infty^\tau(T)$, if*

$$(c_1, \ldots c_k) \xrightarrow{a}_\infty(T)\, (c'_1, \ldots, c'_k),$$

then

$$(c_1 \sqcap z_1, \ldots, c_k \sqcap z_k) \xrightarrow{a}_z (c'_1 \sqcap z_1, \ldots, c'_k \sqcap z_k).$$

Proposition 6.5.5. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let T be an instantiation of type t of size at least $B + \max\{z_1, 1\}$. Then*

$$\zeta_z(\mathcal{N}_B(\{0\,..\,B + m\})) \sqsubseteq_{\mathrm{T}} \zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B + m\})).$$

Proof: By an easy induction on the number of transitions corresponding to a trace tr, we can prove, using Corollary 6.5.4, that if

$$(\#T - B, 0, \ldots 0) \stackrel{tr}{\Longrightarrow} (c'_1, \ldots, c'_k),$$

then

$$((\#T - B) \sqcap z_1, 0, \ldots, 0) \stackrel{tr}{\Longrightarrow} (c'_1 \sqcap z_1, \ldots, c'_k \sqcap z_k).$$

Observe that $(\#T - B, 0, \ldots 0)$ is the initial state of $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B + m\}))$. In addition, we assumed that $\#T \geq B + \max\{z_1, 1\}$, so $(\#T - B) \sqcap z_1 = z_1$. However, $(z_1, 0, \ldots, 0)$ is the initial state of $\zeta_z(\mathcal{N}_B(\{0\,..\,B + m\}))$. Therefore, tr is in $traces(\zeta_z(\mathcal{N}_B(\{0\,..\,B + m\})))$, which implies the result. ∎

Corollary 6.5.6. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let T be an instantiation of type t of size at least $B + \max\{z_1, 1\}$. Then for all non-negative integers n, we have that*

$$ABS_z^{B,n} \sqsubseteq_{\mathrm{T}} ABS_\infty^{T,B,n}.$$

Proof: Using the definitions of $ABS_\infty^{T,B,0}$ (p. 150) and $ABS_z^{B,0}$ (p. 189), as well as $ABS_\infty^{T,B,m}$ (p. 150) and $ABS_z^{B,m}$ (p. 189) for positive m, the result follows from Proposition 6.5.5 and monotonicity of CSP operators. ∎

Theorem 6.5.7. *Let z be a function from $\{1\,..\,k\}$ to $\mathbb{N}$. Let T be an instantiation of type t of size at least $B + \max\{z_1, 1\}$. Then*

(i) if $\mathcal{N}_{myId}(t)$ satisfies $\boldsymbol{NoEqT}^t$, then

$$ABS_z^{B,0} \sqsubseteq_{\mathrm{T}} \phi(Nodes(T))$$

(ii) if $W_{\mathcal{N}} \geq 1$, and β- and γ-events contain no payloads with node identities, then

$$ABS_z^{B,W_{\mathcal{N}}} \sqsubseteq_{\mathrm{T}} \phi(Nodes(T)).$$

Proof: Follows from Corollary 6.5.6, Corollary 6.3.10 and Corollary 6.4.10 by transitivity of refinement. ∎

6.6 Using counter abstraction with full identity awareness in verification

Counter abstraction techniques, as defined in the previous sections, allow us only to verify problems of the form $Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(Nodes(T))$. However, in Section 6.1 we allowed the implementations under our consideration to have node processes running in contexts of the form

$$C_{X(t),Y(t),Z(t)}[\cdot] = \Big((\cdot \setminus X(t)) \underset{Y(t)}{\parallel} Ctrl(t)\Big) \setminus Z(t)$$

for some polymorphic definitions of sets of events $X(t)$, $Y(t)$ and $Z(t)$ and where $Ctrl(t)$ is data independent and satisfies $\mathbf{PosConjEqT}^t_{\mathrm{T}}$. Let

$$Impl(T) = C_{X(t),Y(t),Z(t)}[Nodes(T)].$$

Hence, we would like to be able to prove problems of the form

$$Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T) \tag{6.83}$$

for all sufficiently large T.

In order to make the refinement check independent of T, we abstract the context and look at

$$C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}[ABS_z^{B,n}].$$

We can then use a model checker to verify that the above traces-refines $Spec(\phi(T))$, which, thanks to Theorem 6.5.7, tells us that

$$Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}[\phi(Nodes(T))].$$

To complete the proof we will need that

$$C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}[\phi(P(T))] \sqsubseteq_{\mathrm{T}} \phi(C_{X(T),Y(T),Z(T)}[P(T)]) \tag{6.84}$$

for all processes syntaxes $P(t)$; we prove this in Proposition 6.6.7, below. Finally, (6.83) will follow from Theorem 5.2.15.

We now aim to show that (6.84) holds by proving a stronger result, namely that given some type T, some process syntax $P(t)$, and *any* function f such that $\mathrm{dom}(f) = T$,

$$C_{X(f(T)),Y(f(T)),Z(f(T))}[f(P(T))] \sqsubseteq_{\mathrm{T}} f(C_{X(T),Y(T),Z(T)}[P(T)]) \tag{6.85}$$

We proceed by showing corresponding results for parallel composition and hiding.

Firstly, given types $X_1(t), \ldots, X_n(t)$, for each $i \in \{1 \,..\, n\}$ and $x \in X_i(t)$, we define

$$f_i(x) = \begin{cases} f(x) & \text{if } X_i(t) = t \\ x & \text{otherwise.} \end{cases}$$

We then lift the application of f to events in the following way. Given an event $c.x_1 \ldots x_n$, where for every i in $\{1 \,..\, n\}$, x_i is of type $X_i(t)$, we define

$$f(c.x_1 \ldots x_n) = c.f_1(x_1) \ldots f_n(x_n).$$

We then lift the application of f to sets in the natural way. We then have the following result.

Lemma 6.6.1. *If $X(t)$ is a set definition that is polymorphic in t (see Definition 6.1.1), then $X(f(T)) = f(X(T))$ for all types T and for all functions f such that* $\mathrm{dom}(f) = T$.

Proof: Firstly, we prove the result for simply polymorphic sets, i.e. $X(t) = \{c.x_1 \ldots x_n \mid x_1 \in X_1(t), \ldots, x_n \in X_n(t)\}$, where c is some channel name, $x_1, \ldots, x_n$ are all distinct, and for all i in $\{1 \,..\, n\}$, either $X_i(t) = t$ or $X_i(t)$ is not related to t in any way:

$$\begin{array}{ll} & X(f(T)) \\ = & \{c.y_1 \ldots y_n \mid y_1 \in f_1(X_1(T)), \ldots, y_n \in f_n(X_n(T))\} \\ = & \{c.f_1(x_1) \ldots f_n(x_n) \mid x_1 \in X_1(T), \ldots, x_n \in X_n(T)\} \\ = & \{f(c.x_1 \ldots x_n) \mid x_1 \in X_1(T), \ldots, x_n \in X_n(T)\} \\ = & f(X(T)). \end{array}$$

The result for arbitrary polymorphic set definitions then follows from distributivity of f over set unions. ∎

Next, we present a result from [Ros98] (also in [RB99] and [Bro01]), whose proof follows from Lazić's theory [Laz99].

Lemma 6.6.2. *Suppose that $P(t)$ is data independent and satisfies $\boldsymbol{PosConjEqT}^t_{\mathrm{T}}$. Let T be an instantiation of type t and let f be a function such that* $\mathrm{dom}(f) = T$. *Then*

$$P(f(T)) \equiv_{\mathrm{T}} f(P(T)).$$

We would like to show that distributing the application of every renaming function f with $\mathrm{dom}(f) = T$ over generalised parallel results in a traces anti-refinement, i.e. that

$$f(P) \underset{f(X)}{\parallel} f(Q) \sqsubseteq_{\mathrm{T}} f(P \underset{X}{\parallel} Q) \tag{6.86}$$

for all processes P and Q. Unfortunately, this is not true in general. To demonstrate this, let $P = Q = c?x{:}t \rightarrow STOP$, $X = \{c.1\}$, $T = \{1, 2\}$ and $f(1) = f(2) = 1$. Then $\langle c.1, c.1\rangle \mathit{traces}(f(P \underset{X}{\parallel} Q))$, but $\langle c.1, c.1\rangle \notin \mathit{traces}(f(P) \underset{f(X)}{\parallel} f(Q))$. The reason why the refinement above does not hold is that the application of f created new synchronisations: the processes P and Q do not synchronise on $c.2$, but $f(P)$ and $f(Q)$ synchronise on $f(c.2) = c.1$.

We therefore introduce a condition, which we call $\mathbf{NoNewSync}_X$, that stops new synchronisations from occurring when f is applied to processes that run in generalised parallel. Further, the condition will ensure that when f is applied to hiding, no events are hidden whose corresponding events were not hidden before the application of f.

Definition 6.6.3. Given a set X, we say that a function f satisfies $\mathbf{NoNewSync}_X$ if for all events a in a given alphabet we have that if $f(a)$ is in $f(X)$, then a is in X.

The following lemma establishes that (6.86) holds, provided $\mathbf{NoNewSynx}_X$ is satisfied.

Lemma 6.6.4. *Suppose that f satisfies $\boldsymbol{NoNewSync}_X$. Then*

$$f(P) \underset{f(X)}{\parallel} f(Q) \sqsubseteq_{\mathrm{T}} f(P \underset{X}{\parallel} Q).$$

Proof: We can show that if $f(P \underset{X}{\parallel} Q) \xrightarrow{a} f(P' \underset{X}{\parallel} Q')$, then $f(P) \underset{f(X)}{\parallel} f(Q) \xrightarrow{a} f(P') \underset{f(X)}{\parallel} f(Q')$; this is proved by a simple case analysis on the type of event that a can be and its membership in $\mathit{initials}(P)$ and $\mathit{initials}(Q)$, The result then follows by an induction on the length of an arbitrary trace of $f(P \underset{X}{\parallel} Q)$. ∎

A similar but stronger result (with equality instead of refinement) is true for hiding.

Lemma 6.6.5. *Suppose f satisfies $\boldsymbol{NoNewSync}_X$. Then*

$$f(P) \setminus f(X) \equiv_{\mathrm{T}} f(P \setminus X)$$

for all processes P.

Proof: Similar to the previous proof. ∎

Lemma 6.6.6. *Suppose that $X(t)$ is polymorphic in t. Let T be an instantiation of type t. Then every function f such that $\mathrm{dom}(f) = T$ satisfies $\boldsymbol{NoNewSync}_{X(T)}$.*

Proof: Firstly, we prove the result for simply polymorphic sets. Let $X(t) = \{c.x_1 \ldots x_n \mid x_1 \in X_1(t), \ldots, x_n \in X_n(t)\}$ for some types $X_i(t)$, and let $a = c.x_1 \ldots x_n$. Then

$$
\begin{array}{ll}
 & a \in X(T) \\
\Leftrightarrow & c.x_1 \ldots x_n \in X(T) \\
\Leftrightarrow & \{\text{by the properties of polymorphic set definitions}\} \\
 & c.f_1(x_1) \ldots f_n(x_n) \in X(f(T)) \\
\Leftrightarrow & \{\text{by Lemma 6.6.1}\} \\
 & c.f_1(x_1) \ldots f_n(x_n) \in f(X(T)) \\
\Leftrightarrow & f(c.x_1 \ldots x_n) \in f(X(T)) \\
\Leftrightarrow & f(a) \in f(X(T)).
\end{array}
$$

The result for arbitrary polymorphic set definitions then follows from distributivity of f over set unions. ∎

We are now ready to prove (6.85).

Proposition 6.6.7. *Let T be an instantiation of type t and let f be a function with* $\mathrm{dom}(f) = T$. *Then*

$$C_{X(f(T)),Y(f(T)),Z(f(T))}[f(P(T))] \sqsubseteq_{\mathrm{T}} f(C_{X(T),Y(T),Z(T)}[P(T)])$$

for all process syntaxes $P(t)$.

Proof: Let $P(t)$ be given. From Lemma 6.6.6 we have that f satisfies $\mathbf{NoNewSync}_{X(T)}$, $\mathbf{NoNewSync}_{Y(T)}$ and $\mathbf{NoNewSync}_{Z(T)}$. Then

$$
\begin{array}{ll}
 & C_{X(f(T)),Y(f(T)),Z(f(T))}(f(P(T))) \\
= & \{\text{by the definition of } C_{X(t),Y(t),Z(t)}[\cdot]\} \\
 & \Big(\Big(f(P(T)) \setminus X(f(T))\Big) \underset{Y(f(T))}{\|} \mathit{Ctrl}(f(T))\Big) \setminus Z(f(T)) \\
= & \{\text{by Lemma 6.6.1}\} \\
 & \Big(\Big(f(P(T)) \setminus f(X(T))\Big) \underset{f(Y(T))}{\|} \mathit{Ctrl}(f(T))\Big) \setminus f(Z(T)) \\
\sqsubseteq_{\mathrm{T}} & \{\text{by Lemma 6.6.2, since } \mathit{Ctrl}(t) \text{ satisfies } \mathbf{PosConjEqT}^t_{\mathrm{T}} \text{ (see Section 6.1)}\} \\
 & \Big(\Big(f(P(T)) \setminus f(X(T))\Big) \underset{f(Y(T))}{\|} f(\mathit{Ctrl}(T))\Big) \setminus f(Z(T)) \\
\sqsubseteq_{\mathrm{T}} & \{\text{by Lemma 6.6.5}\} \\
 & \Big(f(P(T) \setminus X(T)) \underset{f(Y(T))}{\|} f(\mathit{Ctrl}(T))\Big) \setminus f(Z(T)) \\
\sqsubseteq_{\mathrm{T}} & \{\text{by Lemma 6.6.4}\} \\
 & f\Big((P(T) \setminus X(T)) \underset{Y(T)}{\|} \mathit{Ctrl}(T)\Big) \setminus f(Z(T))
\end{array}
$$

$$
\begin{array}{ll}
\sqsubseteq_{\mathrm{T}} & \{\text{by Lemma 6.6.5}\} \\
& f\Big(\Big((P(T) \setminus X(T)) \underset{Y(T)}{\parallel} Ctrl(T)\Big) \setminus Z(T)\Big) \\
= & \{\text{by the definition of } C_{X(t),Y(t),Z(t)}[\cdot]\} \\
& f(C_{X(T),Y(T),Z(T)}[P(T)]).
\end{array}
$$

∎

The following is the main result of this thesis. It links together most of the work presented in this and previous chapters. Since, given sufficiently large T, the processes $Spec(\phi(T))$, $C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}\left[ABS_z^{B,0}\right]$ and $C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}\left[ABS_z^{B,W_{\mathcal{N}}}\right]$ are all independent of T, it allows us to perform a single refinement check in order to deduce the truth of the refinement $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$ for infinitely many types T.

Theorem 6.6.8. *Suppose that*

(i) $Spec(t)$ *satisfies* ***SeqNorm*** *and* $\boldsymbol{RevPosConjEqT}^t_{\mathrm{T}}$,

(ii) $Impl(T) = C_{X(T),Y(T),Z(T)}[Nodes(T)]$ *satisfies the conditions from Section 6.1 and* $C_{X(T),Y(T),Z(T)}$ *is a context defined as above,*

(iii) $B \geq \begin{cases} \max\{\#!^t(\alpha) \mid \alpha \text{ is a construct of } Spec(t)\}, & \text{if } Spec(t) \text{ satisfies } \boldsymbol{NoEqT}^t \\ \max\{\#!^t(\sigma,\epsilon)(Spec(t)) \mid \sigma \,\hat{}\, \langle\epsilon\rangle \in SymbolicTraces(Spec(t))\}, & \text{otherwise,} \end{cases}$

(iv) z *is a function from* $\{1 \,..\, k\}$ *to* $\mathbb{N}$,

(v) T *is an instantiation of type* t *of size at least* $B + \max\{z_1, 1\}$, *and*

(vi) ϕ *is a* B*-collapsing function.*

Then

(a) if $\mathcal{N}_{myId}(t)$ *satisfies* $\boldsymbol{NoEqT}^t$ *and*

$$Spec(\{0 \,..\, B\}) \sqsubseteq_{\mathrm{T}} C_{X(\{0..B\}),Y(\{0..B\}),Z(\{0..B\})}\left[ABS_z^{B,0}\right],$$

then

$$Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T),$$

(b) if $\mathcal{N}_{myId}(t)$ *can remember at least one identity (in addition to its own),* β*- and* γ*-events contain no payloads with node identities, and*

$$Spec(\{0 \,..\, B\}) \sqsubseteq_{\mathrm{T}} C_{X(\{0..B\}),Y(\{0..B\}),Z(\{0..B\})}\left[ABS_z^{B,W_{\mathcal{N}}}\right],$$

then

$$Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T).$$

Proof: Let n be one of 0 (for case (a)) or $W_{\mathcal{N}}$ (for case (b)). Since $\#T \geq B+1$, we have that $\phi(T) = \{0\,..\,B\}$. So

$$
\begin{array}{ll}
 & \mathit{Spec}(\phi(T)) \\
\sqsubseteq_{\mathrm{T}} & \{\text{by assumption}\} \\
 & C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}\left[ABS_z^{B,n}\right] \\
\sqsubseteq_{\mathrm{T}} & \{\text{by Theorem 6.5.7 using conditions (iv), (v) and (vi)}\} \\
 & C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}\left[\phi(\mathit{Nodes}(T))\right] \\
= & \{\text{by Proposition 6.6.7}\} \\
 & \phi(C_{X(T),Y(T),Z(T)}\left[\mathit{Nodes}(T)\right]) \\
= & \phi(\mathit{Impl}(T)).
\end{array}
$$

Impl(t) satisfies **TypeSym** by Remark 6.1.2. Hence, using conditions (i), (iii), (v) and (vi), we can infer from Theorem 5.2.15 (if $\mathit{Spec}(t)$ does not satisfy $\mathbf{NoEqT}^t$) and Corollary 5.2.16 (if it does) that

$$\mathit{Spec}(T) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T),$$

as required. ∎

6.7 Conclusions

In this chapter we described how standard counter abstraction techniques, introduced in Chapter 3, can be generalised to systems that contain node identifiers.

We described the family of allowed implementations, each of which consists of a parallel composition of unboundedly many nodes processes and a controller, and we allowed some communication between the nodes or between nodes and the controller to be hidden.

We presented how to build abstract models $ABS_\infty^{T,B,n}$ by modelling nodes with identities less than B explicitly and abstracting remaining nodes using counter state machines based on unbounded integer counters. We demonstrated how these counter state machines can be constructed using only a small number of nodes. We proved that for processes that do not contain equality or inequality tests on t (i.e. satisfy $\mathbf{NoEqT}^t$), $ABS_\infty^{T,B,0}$ forms traces anti-refinements of $\phi(\mathit{Nodes}(T))$ for sufficiently large T. An analogous result was proved for $ABS_\infty^{T,B,W_{\mathcal{N}}}$ when dealing with processes that might contain such tests. We noted that even though the alphabets of $ABS_\infty^{T,B,0}$ and $ABS_\infty^{T,B,W_{\mathcal{N}}}$ no longer depend on T, their states still do.

We improved the abstraction techniques by introducing threshold functions z. We presented how abstract counter state machines based on bounded integer counters can be constructed. Also, we proved that such constructions can be achieved with only a finite number of operations. Using such counter state machines with bounded counters, we defined abstract models $ABS_z^{B,0}$ (for processes that do not contain equality or inequality tests on t) and $ABS_z^{B,W_{\mathcal{N}}}$ (for processes that might do), independent of T. We proved that, for sufficiently large T, the abstract models form traces anti-refinements of $\phi(\mathit{Nodes}(T))$.

Finally, we extended our results to node processes that run in contexts that we allowed within implementations. We combined these results with our type reduction theory (Theorem 5.2.15). This way, we obtained the main result of this thesis (Theorem 6.6.8), which allows us to perform single refinement checks (which may need to be combined with a number of direct checks for some small instances of implementations) in order to deduce traces refinement in positive instances of our subclass of the Parameterised Model Checking Problem.

Our results can be applied in practice as follows.

- A script containing definitions of a node $\mathcal{N}_{myId}(t)$, an implementation and a specification $Spec(t)$ is prepared.
- The constructs of the specification are used to obtain the minimum value for B (cf. condition (iii)).
- Using B and an appropriate value of m (1 for processes without equality and inequality tests on t and $W_{\mathcal{N}}$ for processes that contain such tests), the LTSs of nodes $\mathcal{N}_B(\{0\,..\,B+m\}), \ldots, \mathcal{N}_{B+m}(\{0\,..\,B+m\})$ need to be obtained.
- A threshold function z is chosen.
- The techniques from Section 6.5 are used to construct the counter state machine $\zeta_z(\mathcal{N}_B(\{0\,..\,B+m\}))$.
- The counter state machine is defined as a CSP process and combined with a parallel composition of explicitly modelled nodes with identities in $\{0\,..\,B-1\}$ (with awareness of node identities from a collapsed type) to give a CSP definition of the abstract model $ABS_z^{B,0}$ or $ABS_z^{B,W_{\mathcal{N}}}$.
- The counter state machine is embedded in a suitable context.
- A model checker is used to perform a refinement check of the abstract model against $Spec(\{0\,..\,B\})$.
- If the refinement holds, then (under the assumptions required by our theory), Theorem 6.6.8 can be used to deduce that all implementations for sufficiently large instantiations T of type t satisfy the specification.
- This can be combined with direct refinement checks for smaller types to give a verification proof for all sizes of the implementation.
- If the refinement does not hold, then the counterexample returned by the model checker needs to be inspected: either it represents a genuine counterexample against the unabstracted implementation, or it is a false-negative caused by over-abstraction. The latter case is typically caused by choosing a too small threshold function, so a new threshold can be selected, and the process repeated.

In Chapter 7 we will describe a tool that greatly helps with automation of the process described in the previous paragraph. Also, note that Algorithm 1 can be used for finding small thresholds using a CEGAR approach (however, due to implications of Theorem 6.1.3, it is only a semialgorithm in this case).

Chapter 7

Case study and tool support

In this chapter we present TomCAT, a tool that automates the production of counter abstraction models based on the techniques of both Chapter 3 and Chapter 3. In addition, In Section 7.2 we demonstrate how the results of this thesis apply in practice by verifying a variation of the produce-consumers problem.

7.1 Tool support

We have created a tool[1], called TomCAT, which automatically produces CSP definitions of counter abstraction models described in Chapter 3 and Chapter 6. Given an input script in an appropriate format (templates are provided for convenience), TomCAT reads the following from the command line:

- the denotational model to be used for refinement checking, in the case when no node identifiers are used, or
- the parameters B, the number of processes modelled explicitly, and m, the number of processes whose state machines are used for constructing the counter abstraction, in the case when node identifiers are used[2] (for more details on these parameters see Chapter 6).

It then uses the backend of FDR to create a concrete state machine of the node process (if no node identifiers are used) or of the node processes with identities $B, \ldots, B+m$ (if the script contains node identifiers). TomCAT then creates an output CSP script, which extends the input one with the definitions of counter-abstraction models. Before an output file produced by TomCAT can be run on FDR, the user needs to complete the script by providing the values for counter thresholds. In addition, if the input script contains node identifiers, the user also needs to provide the equivalence classes[3] of node states under $\approx_\phi$ (see Section 6.1).

[1] Available from `http://www.comlab.ox.ac.uk/tomasz.mazur/tools/TomCAT`.

[2] For a given specification/implementation pair, the calculation of B and m is fully automatable, but this has not been implemented in the current version of TomCAT.

[3] In principle, it would be enough for the user to define only the $\sim$ relation (see Section 6.1), as this, combined with the value of B, is sufficient to define $\approx_\phi$. However, this is tricky to do automatically in practice and has not been implemented in the current version of TomCAT.

In Example 3.3.12 we presented a CSP definition of an abstract state machine for the system from Example 3.0.1. However, counters defined in such a way are very inefficient to compile in FDR, as they use a single sequential process with $\prod_{i=1}^{k}(z_i + 1)$ states, where z is the given threshold function. The definitions produced by TomCAT are CSP-equivalent to those in Example 3.3.12, but are more efficient to compile: they use parallel compositions of k separate processes, one for each counter, giving a total of $\sum_{i=1}^{k}(z_i + 1)$ states to compile. TomCAT's input and output scripts (with some formatting modifications to aid readability) for the verification problem from Example 3.3.12 can be found in Appendix C.

7.2 A case study - verification of a producer-consumers problem

In this section we present a variation of the producer-consumer problem [Sta98, Tan87] to illustrate how the theory from Chapter 6 can applied.

7.2.1 Problem description

The implementations we model consist of a single producer, a single slot that can hold at most one item of data (i.e. a buffer of size one) and a number of nodes (consumers) with identities of the distinguished type t.

The producer receives data from the environment using channel *input* and deposits it into the slot using channel *put* (see Figure 7.1(a)). Every node can retrieve data from the slot using channel *get* and pass it to the environment using channel *output*.

In our system we use three binary semaphores:

- *EmptySlotSemaphore* is used to wake up the producer whenever the slot is ready to accept data.
- *FullSlotSemaphore* is used to wake up a node whenever the slot contains a piece of data.
- *Mutex* semaphore is used to guarantee mutual exclusion of critical sections of node processes[4].

The dashed arrows on Figure 7.1(b) show how the semaphores are used within the system. We implement the three semaphores as follows.

$$
\begin{aligned}
&EmptySlotSemaphore(state) = \\
&\quad state = up \;\&\; emptySlotDown \rightarrow EmptySlotSemaphore(down) \\
&\quad \Box \\
&\quad emptySlotUp \rightarrow EmptySlotSemaphore(up)
\end{aligned}
$$

[4]Our use of the *Mutex* semaphore is different from that in the literature on the producer-consumer problem, where it protects the buffer from being simultaneously accessed by the producer and the consumer. The atomicity of CSP events automatically guarantees such a buffer protection, without the need for a semaphore.

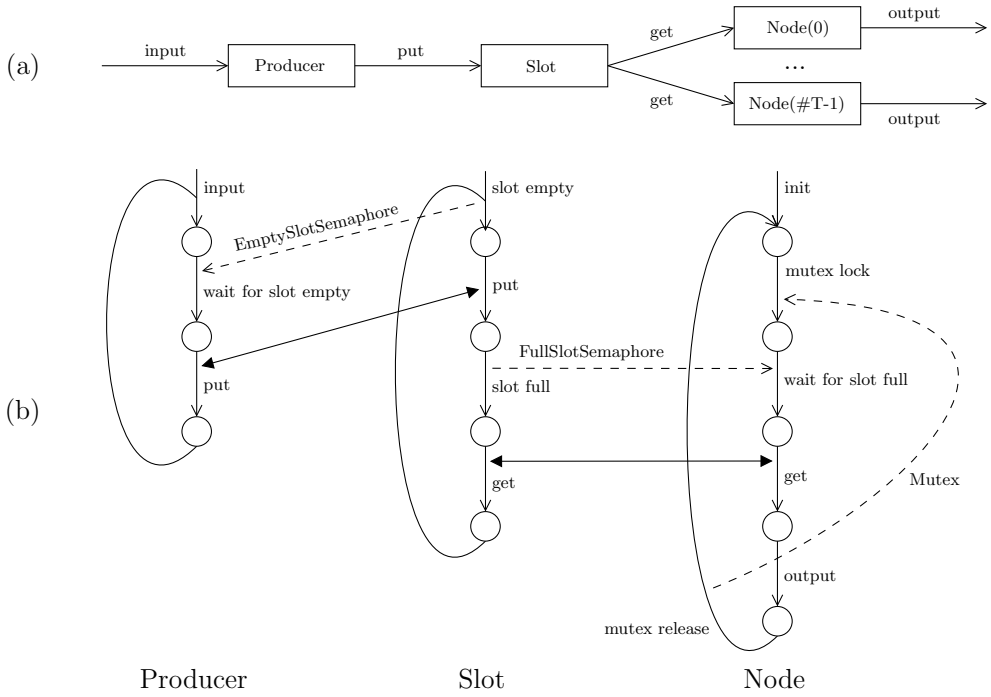


Figure 7.1: Implementation diagram: an overview of (a) the data flow between processes, and (b) the synchronisations between processes, where double-ended arrows denote synchronised transitions and dashed arrows denote the use of a semaphore.

$$
\begin{aligned}
&\mathit{FullSlotSemaphore}(\mathit{state}, t) = \\
&\quad \mathit{state} = \mathit{up}\ \&\ \mathit{fullSlotDown}?i{:}t \rightarrow \mathit{FullSlotSemaphore}(\mathit{down}, t) \\
&\quad \Box \\
&\quad \mathit{fullSlotUp} \rightarrow \mathit{FullSlotSemaphore}(\mathit{up}, t)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Mutex}(\mathit{state}, t) = \\
&\quad \mathit{state} = \mathit{up}\ \&\ \mathit{mutexDown}?i{:}t \rightarrow \mathit{Mutex}(\mathit{down}, t) \\
&\quad \Box \\
&\quad \mathit{mutexUp}?i{:}t \rightarrow \mathit{Mutex}(\mathit{up}, t)
\end{aligned}
$$

The *FullSlotSemaphore* and *Mutex* semaphores use events with parameters in order to allow single nodes to synchronise on them.

Let $\mathit{Data} = \{0, 1\}$ be the type of data items input from and output to the environment (we will later use data independence techniques to extend our verification results to arbitrary *Data*). The producer accepts data of type *Data* from the environment using the *input* channel. It then waits for the *EmptySlotSemaphore* to be raised (which signals that the slot is ready to accept data), so that it can lower it. Finally, it deposits the data it received from the environment into the slot and loops back to its initial state.

$$\mathit{Producer} = \mathit{input}?d{:}\mathit{Data} \rightarrow \mathit{emptySlotDown} \rightarrow \mathit{put}.d \rightarrow \mathit{Producer}$$

We model a node process in a slightly contrived way in order to obtain alphabets that contain α-, β- and γ-events. All nodes (consumers) are first simultaneously initialised (this introduces a γ-event into nodes' alphabets). A node can then obtain the mutex lock, either directly, by lowering the *Mutex* semaphore, or by receiving it from another node that is willing to give up its mutex lock (this introduces β-events into nodes' alphabets). Once in possession of the mutex lock, the node waits for *FullSlotSemaphore* to be raised (which signals that there is data available in the slot), retrieves the data stored in the slot using the *get* channel and outputs it to the environment using the *output* channel. Finally, it can transfer the mutex lock to another node or release it by raising the *Mutex* semaphore.

It is important to note that whenever a node transfers a mutex lock to or from another node, it should not be able to transfer it to or from itself. Otherwise there would be a possibility of multiple mutex locks existing within the system and mutex locks being obtained without them being first granted by the *Mutex* semaphore. A natural way to ban such mutex self-exchanges is to use selections of node identities from $t \setminus \{i\}$ when events on channel *exchangeMutex* are used. However, this breaks the assumptions of data independence (Definition 4.1.1), which all nodes have to satisfy. Hence, we allow the node with identity i to perform the event $exchangeMutex.i.i$, but we disable this event by excluding it from the alphabet of the the node when composing it with the rest of the nodes in alphabetised parallel.

We model the node with identity i as follows.

$$
\begin{aligned}
Node(i,t) &= init \rightarrow Node'(i,t) \\
Node'(i,t) &= mutexDown.i \rightarrow Node''(i,t) \\
&\quad \Box \\
&\quad exchangeMutex?j{:}t!i \rightarrow Node''(i,t) \\
Node''(i,t) &= fullSlotDown.i \rightarrow get.i?d{:}Data \rightarrow output.i.d \rightarrow Node'''(i,t) \\
Node'''(i,t) &= exchangeMutex!i?j{:}t \rightarrow Node'(i,t) \\
&\quad \Box \\
&\quad mutexUp.i \rightarrow Node'(i,t)
\end{aligned}
$$

We let $Nodes(t)$ model the parallel composition of all node processes.

$$
\begin{aligned}
Nodes(t) &= \Big\| \; i \in t \bullet [A(i,t)]\, Node(i,t) \\
A(i,t) &= \{init, mutexUp.i, mutexDown.i, fullSlotDown.i\} \\
&\quad \cup \{get.i.d, output.i.d \mid d \in Data\} \\
&\quad \cup \{exchangeMutex.i.j, exchangeMutex.j.i \mid j \in t \setminus \{i\}\}
\end{aligned}
$$

We now model the slot. When the slot is empty, it wakes up the producer by raising *EmptySlotSemaphore* and accepts any data input on the *put* channel. When the slot contains data d, it wakes up the node holding the mutex lock by raising *FullSlotSemaphore* and sends d to the node using the *get* channel.

$$EmptySlot(t) = emptySlotUp \rightarrow put?d{:}Data \rightarrow FullSlot(d,t)$$

$$FullSlot(d,t) = fullSlotUp \rightarrow get?i{:}t!d \rightarrow EmptySlot(t)$$

To respect the nomenclature we used in Section 6.6, we let $Ctrl(t)$ model the part of our implementation not including the node processes.

$$Ctrl(t) = Mutex(up,t) \,|||\, \left(\begin{array}{l} (Producer \underset{\{|put|\}}{\parallel} EmptySlot(t)) \underset{\alpha Sem}{\parallel} \\ \qquad (EmptySlotSemaphore(down) \,|||\, FullSlotSemaphore(down,t)) \end{array} \right)$$

$$\alpha Sem = \{emptySlotDown, emptySlotUp, fullSlotUp\}$$

Our implementation consists of the nodes running in parallel with the controller process and with all events not on the *input* or *output* channel hidden. Hence, we let

$$Impl(t) = C_{X(T),Y(T),Z(T)}[Nodes(T)]$$

where

$$C_{X(t),Y(t),Z(t)}[\cdot] = \Big((\ \cdot\ \setminus X(t)) \underset{Y(t)}{\parallel} Ctrl(t) \Big) \setminus Z(t)$$

$$X(t) = \{init\}$$

$$Y(t) = \{|\ mutexUp, mutexDown, fullSlotDown, get\ |\}$$

$$Z(t) = \{|\ mutexUp, mutexDown, exchangeMutex, fullSlotUp, \\ fullSlotDown, emptySlotUp, emptySlotDown, put, get\ |\}.$$

We want to verify that for every instantiation T of type t, the implementation traces-refines a three-place buffer, i.e. that all inputs are output in the same order without any loss, and that never more than three items can be input without an output happening. The following is a process that models a three-place buffer.

$$Buff(\langle\rangle, t) = input?d{:}Data \rightarrow Buff(\langle d\rangle, t)$$

$$Buff(\langle d\rangle \hat{}\, seq, t) =$$
$$\quad output?i{:}t!d \rightarrow Buff(seq, t)$$
$$\quad \Box$$
$$\quad length(seq) < 2 \ \&\ input?d'{:}Data \rightarrow Buff(\langle d\rangle \hat{}\, seq \hat{}\, \langle d'\rangle, t)$$

Using the above, we let our specification be the following.

$$Spec(t) = Buff(\langle\rangle, t)$$

Then, our verification becomes: given type $Data$, is it true that

$$Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$$

for every instantiation T of type t?

7.2.2 Verification using TomCAT and FDR

We begin the verification of the producer-consumers problem by creating an input script for TomCAT using the definitions from the previous section, written in machine-readable CSP (see Appendix C.2.1).

By analysing $Spec(t)$ we can see that

$$\max\{\#!^t(\alpha) \mid \alpha \text{ is a construct of } Spec(t)\} = 0,$$

so we let $B = 0$ and we let ϕ be a B-collapsing function.

We use TomCAT to produce an output CSP script (which, with some formatting modifications to aid readability, can be found in Appendix C.2.2). As mentioned in Section 7.1, before the script can be executed using FDR, we need to complete it with a description of equivalence classes of states of node processes under ϕ and a definition of a threshold function.

The output script contains the following definition, where x is a node identity:

$$\begin{array}{l} Transitions(x) = \\ \quad \text{if } x = 0 \text{ then} \\ \quad\quad \{(0, init, 1), (1, exchangeMutex.0.0, 2), (1, exchangeMutex.1.0, 2), \\ \quad\quad\ (1, mutexDown.0, 2), (2, fullSlotDown.0, 3), (3, get.0.1, 4), \\ \quad\quad\ (3, get.0.0, 5), (4, output.0.1, 6), (5, output.0.0, 6), \\ \quad\quad\ (6, exchangeMutex.0.0, 6), (6, mutexUp.0, 1), (6, exchangeMutex.0.1, 1)\} \\ \quad \text{else if } x = 1 \text{ then} \\ \quad\quad \{(7, init, 8), (8, exchangeMutex.1.1, 9), (8, exchangeMutex.0.1, 9), \\ \quad\quad\ (8, mutexDown.1, 9), (9, fullSlotDown.1, 10), (10, get.1.1, 11), \\ \quad\quad\ (10, get.1.0, 12), (11, output.1.1, 13), (12, output.1.0, 13), \\ \quad\quad\ (13, exchangeMutex.1.1, 13), (13, exchangeMutex.1.0, 8), (13, mutexUp.1, 8)\} \\ \quad \text{else if } x = 2 \text{ then } \ldots \text{ else } \{\}. \end{array}$$

The first two branches of the definition, for $x = 0$ (i.e. $x = B$) and $x = 1$ (i.e. $x = B + 1$) describe LTSs of $Node(B, \{0\,..\,B+1\})$ and $Node(B+1, \{0\,..\,B+1\})$, respectively. The branch for $x = 3$ only plays a role in distinguishing between beta and gamma events, so we have omitted its details.

These LTSs are shown in Figure 7.2. It is easy to see that the equivalence classes that need to be added to the output script are[5]:

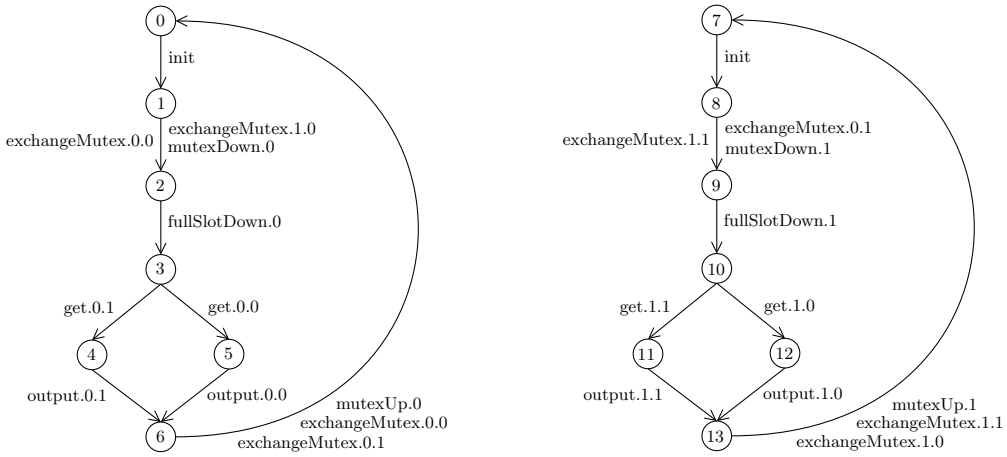
$$E = \langle\{0, 7\}, \{1, 8\}, \{2, 9\}, \{3, 10\}, \{4, 11\}, \{5, 12\}, \{6, 13\}\rangle.$$

We define E as a *sequence*; in each sequence of counters, the counters will follow the same order.

We need to complete the output script with values of the threshold function. Suppose that we choose the following threshold function to be added to the script:

$$Threshold = \langle 1, 1, 1, 1, 0, 1, 1\rangle,$$

[5]Here, each equivalence class is of the form $\{i, i+7\}$ for $i \in \{0\,..\,6\}$. However, this is a coincidence and in other examples we have obtained less orderly behaved equivalence classes.



(a) (b)

Figure 7.2: LTSs of (a) $Node(B, \{0\,..\,B+1\})$ and (b) $Node(B+1, \{0\,..\,B+1\})$.

i.e. we let $z(5) = 0$ and for all i in $\{1\,..\,7\} \setminus \{5\}$, $z(i) = 1$. Then, we have that $Spec(\{0..B\}) \sqsubseteq_{\mathrm{T}} C_{\{0..B\},\{0..B\},\{0..B\}}\left[ABS_z^{B,0}\right]$, with the following counterexample returned by FDR:

$$tr = \langle output.0.1 \rangle.$$

It is not surprising that the refinement does not hold, since, by letting $z(5) = 0$, we say that at all times there are 0 or more nodes in the state where $output.0.1$ is available. So $output.0.1$ is immediately available within the abstract model. However, this is a spurious counterexample, as no concrete implementation can ever perform tr.

If we change the threshold function to

$$Threshold = \langle 1, 1, 1, 1, 1, 1, 1 \rangle,$$

then we still have that $Spec(\{0..B\}) \not\sqsubseteq_{\mathrm{T}} C_{\{0..B\},\{0..B\},\{0..B\}}\left[ABS_z^{B,0}\right]$, with the following counterexample:

$$tr = \langle input.1, output.1.1, output.1.1 \rangle.$$

This corresponds to the following trace that is performed by the abstract model without the hiding of events.

$$tr' = \langle init, emptySlotUp, input.1, emptySlotDown, put.1, fullSlotUp, \\ mutexDown.1, fullSlotDown.1, get.1.1, output.1.1, output.1.1 \rangle$$

However, no concrete implementation without hiding can perform tr', so tr is yet another spurious counterexample. This behaviour is present in our abstract model due to the fact once a node is in a state, say st where it possesses the mutex lock (i.e. any state other than $Node$ or $Node'$), the corresponding counter is 1. Since all

the thresholds are equal to 1, the counter indicates that there could be *more* than one process in *st*. This leads to a model where a single mutex lock can be cloned and held by multiple processes.

To eliminate such behaviours, we set the threshold to the following:

$$\mathit{Threshold} = \langle 0, 2, 2, 2, 2, 2, 2 \rangle,$$

i.e. we are not interested in how many processes there are before initialisation, and we count processes in all other states using the $\{0, 1, 2 \text{ or more}\}$ domain. This time FDR confirms that the refinement

$$\mathit{Spec}(\{0..B\}) \sqsubseteq_{\mathrm{T}} C_{\{0..B\},\{0..B\},\{0..B\}} \left[ABS_z^{B,0} \right] \tag{7.1}$$

holds. The check takes less than 1 second, and visits 536 states. This is, in fact, the minimal threshold function for which the refinement holds.

It is easy to check that the assumptions of Theorem 6.6.8 are met. Hence, using clause (ii) of the theorem we can infer from (7.1) that $\mathit{Spec}(T) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T)$ for all instantiations T of type t such that $\#T \geq B + \max\{z_1, 1\} = 1$, i.e. the refinement holds for all non-empty types T. This solves our verification problem for the fixed type $\mathit{Data} = \{0, 1\}$ of data items.

Finally, we can extend this result to more general types *Data* using Lazić's data independence theory [Laz99]. In our model, both $\mathit{Spec}(t)$ and $\mathit{Impl}(t)$ satisfy $\mathbf{NoEqT}^t$ (with respect to the *Data* type) and $\mathit{Spec}(t)$ satisfies the **Norm** condition (also with respect to the *Data* type) from [Ros97, Chapter 15]; so, by Theorem 15.2.2 from [Ros97], the refinement holds for all finite and infinite types *Data*.

More case studies (using the counter abstraction techniques presented in Chapter 3) can be found in [Tho10].

Chapter 8

Conclusions

In this final chapter we conclude with a summary of the work presented in this thesis, a discussion of some observations on counter abstraction and a list of ways in which our work could be improved and extended.

8.1 Summary

Given a specification $Spec(t)$ and an implementation $Impl(t)$, direct model checking can help us to find bugs in the implementation for a finite (and small) number of instantiations T of parameter t. However, one is often interested in uniform verification, i.e. in proving correctness for all T.

Lazić's theory of data independence [Laz99] (see Section 2.3 and Section 4.1.1) for the CSP process algebra solves the problem of uniform verification of parameterised systems with the parameter being a datatype. Inspired by these results, we have developed a type reduction theory (presented in Chapter 5 with the key theorems being Theorem 5.2.15 and Theorem 5.2.24), that establishes function ϕ that collapses all sufficiently large types T to a fixed type $\hat{T} = \{0..B\}$ for some non-negative integer B (the value to be chosen for B is determined in the traces model by Theorem 5.2.15 and in the stable failures model by Theorem 5.2.24), treating all identities in $\{0..B-1\}$ faithfully, but mapping all other identities onto B. The theory then proves that for all T such that $\hat{T} \subseteq T$,

$$Spec(\hat{T}) \sqsubseteq \phi(Impl(T)) \text{ implies that } Spec(T) \sqsubseteq Impl(T) \tag{8.1}$$

with both refinements in either the traces or the stable failures model. In order for the above to hold, the system to be verified has to satisfy certain conditions, the most important of which include a normality condition, **SeqNorm** (see Definition 4.1.4), for specifications and a type symmetry condition, **TypeSym** (see Definition 5.2.1), for implementations.

Our type reduction theory makes an extensive use of symbolic representation of process behaviour, which allows us to use known behaviours of one instantiation of a specification to deduce behaviours of another one. In Chapter 4 we presented a symbolic operational semantics for CSP processes that satisfy **SeqNorm** and we provided a set of translation rules that allow us to concretise symbolic transition graphs.

We also showed that, crucially, the combination of the symbolic operational semantics and the translation rules is congruent to a fairly standard operational semantics.

Since the process $\phi(\mathit{Impl}(T))$ used in (8.1) still depends on T, the type reduction theory, on its own, does not resolve the problem of an infinite number of refinement checks needed to solve a given verification problem. However, the usefulness of the theory comes from the fact that it can be combined with an abstraction method that produces models Abstr such that for all sufficiently large T,

$$\mathit{Abstr} \sqsubseteq \phi(\mathit{Impl}(T)). \tag{8.2}$$

Then, by testing that $\mathit{Spec}(\hat{T}) \sqsubseteq \mathit{Abstr}$ we can deduce from transitivity of refinement and (8.1) that $\mathit{Spec}(T) \sqsubseteq \mathit{Impl}(T)$ holds for all sufficiently large T (and the verification problem can be solved directly for all smaller T).

We presented one such abstraction method, based on the ideas of counter abstraction. Counter abstraction works by counting how many, rather than which, node processes are in a given state: for nodes with k local states, an abstract state $(c_1, \ldots, c_k)$ models a global state where c_i processes are in the i-th state. Then, a threshold function z is used to cap the values of each counter. If for some i, counter c_i reaches its threshold, z_i, then this is interpreted as there being z_i or more nodes in the i-th state. The addition of thresholds makes abstract models independent of the instantiation of the parameter.

We introduced the ideas of counter abstraction in Chapter 3 and showed how they apply to verification problems where no use of node identifiers is made (and therefore where all node processes are identical). Since in every such verification problem the implementation cannot communicate values of type t, we were able to show that the abstract models Abstr_z, based on a threshold function z, satisfy

$$\mathit{Abstr}_z \sqsubseteq_{\mathrm{T}} \mathit{Impl}(T) \qquad \text{and} \qquad \mathit{Abstr}_z \sqsubseteq_{\mathrm{F}} \mathit{Impl}(T)$$

for all sufficiently large T. In addition, the lack of node identifiers means that specifications are independent of parameter t. This means that testing that $\mathit{Spec} \sqsubseteq \mathit{Abstr}_z$ is enough to deduce that $\mathit{Spec}(T) \sqsubseteq \mathit{Impl}(T)$ hold for all sufficiently large T, without the need for our type reduction theory.

The addition of node identifiers breaks one of the assumptions on which canonical counter abstraction techniques are built, namely that the state space of a given node does not change when varying the parameter. In Chapter 6 we showed how to solve this problem. We extended standard counter abstraction techniques by counting processes within equivalence classes of a relation $\approx_\phi$. This relation identifies states that hold identities that are identified under the collapsing function ϕ. We showed how to build abstract models, Abstr_z, such that (8.2) holds. This, combined with our type reduction theory allows us to deduce that if a single refinement check (namely, $\mathit{Spec}(\hat{T}) \sqsubseteq \mathit{Abstr}_z$) holds, then the original verification problem has a positive answer for all sufficiently large instantiations of the parameter.

To aid the creation of counter abstraction models based on the techniques of both Chapter 3 and Chapter 6 we have implemented an automated tool, TomCAT. In addition, we have demonstrated how our results apply in practice by providing a case study, in which we verified a variation of the producer-consumers problem.

8.2 Observations on counter abstraction

Thompson's project

In 2010, Richard Thompson, an undergraduate student at Oxford University Computing Laboratory completed a final year project [Tho10], based on the counter abstraction techniques that we presented in Chapter 3.

Thompson verifies liveness properties of the form "if a node performs a, then some node eventually performs b" by hiding all events except for b after a is performed and testing if b is not refused and the process does not diverge. Such tests would usually be performed in the failures/divergences model, but our theory does not yet support it (see under "Additional CSP denotational models" in Section 8.3). Instead, he performs refinement checks of the form $Spec \sqsubseteq_{\mathrm{F}} Abstr$ and deduces that $Spec \setminus X \sqsubseteq_{\mathrm{FD}} Abstr \setminus X$ (where $\sqsubseteq_{\mathrm{FD}}$ denotes refinement in the failures/divergences model) holds for an appropriate set of events X (the validity of such a deduction is warranted by the fact that $Spec$ and $Abstr$ are both divergence-free).

In Chapter 3 we assumed that the context within which node processes run is independent of the distinguished type. Thompson shows how to verify implementations that contain a controller (which he calls a *guard*) with a finite number of parameters that count node processes in particular states. The main idea is to abstract all variables of the controller that depend on t in a way that synchronises their values with the values of the counters in $\zeta_z(\mathcal{N})$. In particular, if a counter corresponding to some state st of a node has attained its threshold and a transition that represents a node moving out of state st is performed, then the nondeterministic choices of whether to decrement the counter and whether to decrement the corresponding variable of the controller have to be resolved in the same way. This is achieved by a clever use of renaming and synchronisation.

In Chapter 3 we also assumed that specifications are independent of the distinguished type. Thompson demonstrates how to perform verification of safety and liveness properties quantified over all nodes. Such properties are naturally expressed using specification processes that count nodes in some relevant states (making them dependent on the instantiation of the parameter). The main idea is to keep a single node or a pair of nodes unabstracted and transform the specification into one that verifies the same property, but only on this explicitly modelled part of the system (while running in parallel with a counter abstraction of the rest of the system). Then, by appealing to the symmetry of the implementation, one can deduce that the property must hold for all nodes.

Finally, Thompson shows how to counter abstract a system containing two independent types of nodes by counter abstracting them separately and combining them into a single abstract model.

Specifications based on counter values

One of the advantages of counter abstraction is the availability of explicit information about the number of processes in all states; when modelling counters using CSP processes, each counter has the awareness of its current value. Suppose that we

extend the definition of the counter corresponding to the i-th state whose value if *value* in the following way:

$$\begin{aligned} Counter_i(value) = &\ll \text{counter definition as before} \gg \\ &\Box \\ &counter.i.v \rightarrow Counter_i(value), \end{aligned}$$

where *counter* is a channel not used in the rest of the definition of the counter. We can then easily define specifications that talk about the number of processes in a given state. For example, suppose that each node process has 3 local states. Then

$$Spec = Chaos(\{|\ counter.1, counter.3\ |\} \cup \{counter.2.i \mid i \in \{0\,..\,2\}\}), \tag{8.3}$$

if used in a traces refinement check, tests if always at most two node processes are in the second state. Similarly,

$$\begin{aligned} Spec = &\ Chaos(\{|\ counter.1, counter.3\ |\} \cup \{a, counter.2.0, counter.2.1\}) \\ &\Box \\ &c2.2 \rightarrow a \rightarrow Spec, \end{aligned}$$

if used in a stable failures refinement check, tests if whenever two nodes are in the second state, then a and only a is offered. Using counter-based specification feels like a very natural way of capturing behaviours that depend on the number of processes in a given state; if we did not allow counters to communicate their values, then, assuming that node processes cannot perform τ's to move to or from the second state, the specification that we expressed in (8.3) would have to be expressed as e.g.

$$\begin{aligned} Spec' &= Chaos(\Sigma \setminus (Moveto2 \cup Movefrom2)) \ \Box\ ?a{:}Moveto2 \rightarrow Spec' \\ Spec' &= Chaos(\Sigma \setminus (Moveto2 \cup Movefrom2)) \\ &\quad \Box\ ?a{:}Movefrom2 \rightarrow Spec \ \Box\ ?a{:}Moveto2 \rightarrow Spec'' \\ Spec'' &= Chaos(\Sigma \setminus (Moveto2 \cup Movefrom2)) \ \Box\ ?a{:}Movefrom2 \rightarrow Spec', \end{aligned}$$

where *Moveto*2 and *Movefrom*2 are sets of labels of, respectively, the incoming and outgoing transitions of the node's second state.

8.3 Future work

In this section we discuss some directions in which the work presented in this thesis could be improved and extended.

Additional CSP operators

In Section 1.5.1 we described the CSP syntax used in our work. Even though we considered a fairly comprehensive set of syntax terms that allow us to express most processes that one uses in practice, the following terms could be added.

- The interrupt operator, $\triangle$.
  The process $P \triangle Q$ initially behaves like P, but at some point Q may take over by performing one of its initial event. This operator adds no extra expressiveness over the traces, stable failures (only under the divergence-freedom assumption) and failures/divergences models, as it can always be expressed using the operators we included in the language used throughout this thesis (this construction[1] is based on a similar one in [Ros08c]):

$$P \triangle Q = \Big((P[\![\mathcal{R}_P]\!] \,|||\, Q[\![\mathcal{R}_Q]\!]) \underset{\Sigma(P)\cup\Sigma(Q)}{\lVert} Reg \Big)[\![\mathcal{R}]\!], \tag{8.4}$$

  where

$$\mathcal{R}_P = \{(x, p.x) \mid x \in \Sigma(P)\}$$
$$\mathcal{R}_Q = \{(x, q.x) \mid x \in \Sigma(Q)\}$$
$$\mathcal{R} = \{(p.x, x), (q.x, x) \mid x \in \Sigma(P) \cup \Sigma(Q)\}$$

  and

$$Reg = p?x{:}\Sigma(P) \rightarrow Reg \;\Box\; q?x{:}\Sigma(Q) \rightarrow Run(\{|\ q\ |\}).$$

  Here, P is allowed to perform all of its communication until Q performs one of its initial events, at which point Reg only allows Q to communicate, blocking P's visible events indefinitely.

  For stable failures refinement checks of possibly divergent processes, our operational semantics and theoretical results would need to be extended with cases for $\triangle$.

- The throws operator, Θ_A.
  This operator (only recently added to the CSP language [Ros08a, Ros08c]) allows P to give up control to Q by communicating an event in A (interestingly, this is the only CSP operator that allows a process to perform an action that turns this process off). Over the traces model and the stable failures model with the divergence-freedom assumption this operator does not add any expressiveness, because we can express Θ_A using $\triangle$ (see above) and operators of the CSP language we used in this thesis (this construction is based on a similar one in [Ros08c]):

$$P \;\Theta_A\; Q = \Big((P[\![\mathcal{R}_P]\!] \triangle c \rightarrow Q[\![\mathcal{R}_Q]\!]) \underset{\Sigma(P)}{\lVert} Reg \Big)[\![\mathcal{R}]\!] \setminus \{c\},$$

  where $\mathcal{R}_P$, $\mathcal{R}_Q$ and $\mathcal{R}$ are as in the construction for $\triangle$, above, and

$$Reg = p?x{:}(\Sigma(P) \setminus A) \rightarrow Reg \;\Box\; p?x{:}A \rightarrow c \rightarrow STOP.$$

[1] Interestingly, Computer Science undergraduates at Oxford were asked to derive this construction (which they had not seen before) in their second-year Concurrency exam in 2008.

Here, we allow P to freely communicate all events not in A. However, once it communicates an event in A, then *Reg* blocks all of P's visible action, allowing for the interrupt to happen (trigger by the c event, which we subsequently hide) and giving the control to Q.

For failures/divergences refinement checks and for refinement checks of possibly divergent processes in the stable failures model, additional treatment of Θ_A is needed to adapt our techniques to the extended CSP language.

- The atomic process *SKIP* and the sequential composition operator, ; .
  SKIP is, in a sense, similar to *STOP*, but while the latter is a synonym of deadlock and cannot perform any action, *SKIP* is a synonym of successful termination: the only action it can perform is a special termination event, ✓. The process P; Q behaves like P until it successfully terminates (i.e. performs ✓), which triggers it to behave like Q. We decided to omit *SKIP* and ; from our syntax, since otherwise we would need to include special treatment of ✓ in all the semantic models. This would significantly add to the length and complexity of our results. We have already done some work extending our theories (mostly those presented in Chapter 3) to processes with ✓ and we believe that the results of this thesis still hold for process syntaxes that include *SKIP* and ; .

- The atomic process div.
  This process is a synonym of divergence (livelock), i.e. it can immediately engage in performing an infinite, unbroken sequence of τ's. This process does not add extra expressiveness, as it is equivalent to e.g. $P = (a \rightarrow P) \setminus \{a\}$.

Additional CSP denotational models

The counter abstraction techniques presented in Chapter 6 apply only to verification within the traces model. We would like to extend them to the stable failures model in the future. Unfortunately, our refinement results from Section 6.3 and Section 6.4 do not directly extend from the traces to the stable failures model, since there can be stable failures of a concrete implementation that are not stable failures of the abstraction defined in this thesis. The following example illustrates this issue.

Example 8.3.1. Let

$$\mathcal{N}_{myId}(t) = in!myId?i{:}t \rightarrow out.i \rightarrow STOP.$$

Suppose that B is some fixed non-negative integer. Let ϕ be a B-collapsing function and let T be a type such that $\#T \geq B + 2$. Observe that for all i, $out.i$ is in the alphabet of every node, so it is a γ-event (see Section 6.1). We then have that

$$(\langle in.i.B \mid i \in \{0 \mathinner{..} B-1\}\rangle^\frown \langle in.i.i \mid i \in \{B \mathinner{..} \#T-1\}\rangle, \Sigma) \in failures(Nodes(T)),$$

so

$$(\langle in.i.B \mid i \in \{0 \mathinner{..} B-1\}\rangle^\frown (in.B.B)^{\#T-B}, \phi(\Sigma)) \in failures(\phi(Nodes(T))),$$

where $Nodes(t)$ is defined as in Section 6.1. However, after performing the trace

$$tr = \langle in.i.B \mid i \in \{0\,..\,B-1\}\rangle^\frown (in.B.B)^{\#T-B},$$

the counter that corresponds to the equivalence class $[(out.i \rightarrow STOP, \{myId \mapsto B, i \mapsto B\}, T)]$ within the counter state machine $\zeta_\infty^T(\mathcal{N}_B(\{0\,..\,B+1\}))$ must be equal to $\#T - B$ and all other counters must be 0. Hence, by the definition of the abstract transition relation, (6.7), $\zeta_\infty^T(\mathcal{N}_B(\{0..B+1\}))$ cannot refuse $out.B$ after performing tr. Hence,

$$(tr, \phi(\Sigma)) \notin failures(ABS_\infty^{T,B,0}).$$

End of example.

The above example implies that our abstraction techniques do not, in general, guarantee that $ABS_\infty^{T,B,n} \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$ for $n = 0$ and $n = W_{\mathcal{N}}$ (which would be the equivalent of Corollary 6.3.10 and Corollary 6.4.10, respectively, within the stable failures model). One method that could potentially be used to ensure that such refinements hold by construction is to distinguish more nodes within abstract counter state machines by taking ϕ to be a $(B + \max\{n, 1\})$-collapsing function (rather than a B-collapsing function) when counting processes in the equivalence classes of the $\approx_\phi$ relation. An alternative approach would be to investigate methods for adding the lost refusals back into abstractions (e.g. by introducing copies of some relevant states, with identical incoming transitions, but with all outgoing transitions labelled with certain events removed).

We have not yet considered extensions of our techniques to the failures/divergence model in great detail. However, we can provide some observations. Firstly, in Chapter 3 we proved bisimilarity between parallel compositions of nodes that do not use node identifiers, $Nodes(T)$, and their counter abstracted models without thresholds, $\zeta_\infty^T(\mathcal{N})$ (Proposition 3.2.2). This implies that $\zeta_\infty^T(\mathcal{N}) \equiv_{\mathrm{FD}} Nodes(T)$ (thus extending Corollary 3.2.3 to the failures/divergences model). In addition, given a threshold function z, Proposition 3.3.10 shows that, under certain conditions, $\zeta_z(\mathcal{N}) \sqsubseteq_{\mathrm{F}} \zeta_\infty^T(\mathcal{N})$. Since addition of thresholds cannot remove any divergences, it must be that $\zeta_z(\mathcal{N}) \sqsubseteq_{\mathrm{FD}} \zeta_\infty^T(\mathcal{N})$ and hence $\zeta_z(\mathcal{N}) \sqsubseteq_{\mathrm{FD}} Nodes(T)$ for all sufficiently large T. Unfortunately, similar ideas cannot be used when working with systems that make use of node identifiers, as a bisimilarity like the one described above no longer holds (see Example 8.3.1 for a counterexample).

In addition, a caveat is in order. Suppose that a node can perform a τ in some state st. Then, the counter abstraction model can diverge once it reaches a state where the counter corresponding to st attains its threshold, say z_{st} (this is because the abstract model can then perform an infinite number of τ's by always assuming the value of the counter to mean "more than z_{st} processes in state st" and therefore never decreasing the counter). This is not a problem in the stable failures models, as even though it leads to an instability, which could, theoretically, cause some failure not to be recorded, the abstract model always has the option to interprate the value of the counter to mean "exactly z_{st} processes in state st" and decrease it, avoiding the divergence (in Chapter 3 we required thresholds of counters corresponding to

non-deadlocked states to be at least 1 when the stable failures model was used to always enable such a decrease). However, such divergences within abstract models may lead to spurious counterexamples when performing uniform verification in the failures/divergences model.

Finally, we observe that usually the only divergences property one is interested in is full divergence-freedom. In practice, this might be an easier problem to verify for all instantiations of the distinguished type than verifying failures/divergences refinement. Once a system is shown to be divergence-free, a refinement check in the stable failures model implies refinement in the failures/divergences model.

Proving Conjecture 3.3.13

In Section 3.3.3 we provided a CEGAR-style algorithm for finding small threshold functions z such that $\zeta_z(\mathcal{N})$ is an exact abstraction, i.e. such that

$$\forall\, T \bullet Spec \sqsubseteq C\left[\,\Big\|_{A_{Sync}} \; i \in T \bullet \mathcal{N}\right] \tag{8.5}$$

if and only if

$$Spec \sqsubseteq \zeta_z(\mathcal{N}). \tag{8.6}$$

Theorem 3.1.7 showed that verifying if (8.5) holds is undecidable. However, if we assume that $Spec$ and $\mathcal{N}$ are finite-state, and $A_{Sync} = \{\}$ (i.e. nodes are interleaved), then the problem becomes decidable (Theorem 3.1.1). We conjectured (Conjecture 3.3.13) that for every verification problem within this decidable family, there exists a finite threshold function z such that (8.6) holds. This means that for every such verification problem, our algorithm is guaranteed to terminate, provided a decreasing heuristic is used (see Section 3.3.3). We base our belief on the fact that the only way for there being no finite z such that (8.6) holds is if there are infinitely many discontinuities (see Section 3.3.3), in which case the algorithm can loop forever, generating an infinite sequence of spurious counterexamples. However, we believe that infinitely many discontinuities can only happen if either the control state or the specification is infinite-state (which would contradict our assumptions). However, this statement needs a formal proof.

Context generalisation for counter abstraction with node identities

In Chapter 3 we proved refinement results for node processes that run in general, t-independent contexts. The contexts that were considered in Chapter 6 were of the form $\Big((\,\cdot \setminus X(t)) \underset{Y(t)}{\|} Ctrl(t)\Big) \setminus Z(t)$. The majority of implementations one uses in practice are of this form; however, it would be desirable to extend Theorem 6.6.8 to more general contexts.

Extending symbolic operational semantics

The operational semantics that we presented in Section 4.3 served an important purpose in proving the results of our type reduction theory (see Chapter 5). However, the type reduction theory assumes that processes satisfy the **SeqNorm** condition (see Section 4.1.2). Therefore, for brevity, we provided inference rules only for those operators that **SeqNorm** allows. To increase the generality, it would be desirable to formalise transition rules for parallel compositions, renaming, hiding and replicated choices in the future.

Combining counter abstraction and compression

We believe that there is scope for combining our counter abstraction techniques with FDR compression algorithms [RGG$^+$95, Ros10] to achieve higher efficiency of abstract models. These compression algorithms work best when events that are irrelevant to the specification are hidden, and when we hide communication between parts of a system before composing them with the rest of the system. The former is pretty self-explanatory. One way of achieving the latter would be to look for events that are in the alphabets of only a few counters, compose those counters in parallel first, and hide the common events. We would then iteratively keep adding more counters to the network, while hiding the events not in the alphabets of the remaining counters; this process is automatable [Ros10]. Although compression may dramatically decrease the number of states that need to be explored, it adds some overhead to the refinement checking process. In practice, the balance between the two is system-specific. Therefore, a series of case studies verifying different types of implementations (based on different kinds of interaction between counters in abstract models) need to be performed in order to decide whether the benefits of compression are worthwhile in our case.

Additional case studies

Example 3.3.12 demonstrated how counter abstraction techniques from Chapter 3 apply in practice. Two, larger case studies, based on the same techniques, can be found in [Tho10]. In Chapter 7 we presented a case study that uses counter abstraction techniques from Chapter 6. However, it would be beneficial to test our ideas in practice further, especially by verifying linear or ring-based networks, leader election protocols or security protocols. Examples that require the use of multiple applications of counter abstractions for different parts of a given system would also be interesting, as would be examples that combine our type reduction theory with other abstraction methods. In particular, it would also be interesting to see how well the techniques that Thompson used in [Tho10] (see under "Thompson's project" in Section 8.2 for a summary) apply to systems other than those to which they were originally applied and how well they transfer to the counter abstraction techniques for systems with node identifiers from Chapter 6.

TomCAT improvements

We have implemented TomCAT in order to automate the process of creating counter abstraction models defined as in Chapter 3 and Chapter 6. For systems that use no identifiers the tool is fully automatic, i.e. it only requires the user to provide the necessary information (a denotational model to be used in assertions and a threshold function) to create a complete, runnable CSP script. However, as already noted in Section 7.1, in the case of implementations that make use of node identifiers, the user needs to provide equivalence classes of states under the $\approx_\phi$ relation (Definition 6.1.11). In our theoretical work in Chapter 6, is is enough to define the $\sim$ relation and the value of B to work out the equivalence classes of $\approx_\phi$. This is made possible by tracking the values of type t that are stored within every state. However, the representation of a state machine that TomCAT receives from the backend of FDR is such that this information is not readily available. Theoretically, we could obtain it by considering all events available after a given state is reached and comparing them to syntax terms to deduce what values had to be stored in that state. However, we have not yet found an easy way to implement it in practice (and we also believe that it would be better to build this into the FDR, where the syntax is available), so this feature has been omitted in the current version of TomCAT.

There are also a number of small code optimisations that we should include in a future release of TomCAT, e.g. the shell scripts that drives the backend of FDR should be rewritten so that whenever state machines of n processes are needed, a single session in FDR is created, during which all the n state machines are generated and stored in a temporary file (instead of invoking n sessions, each generating a single state machine).

Index of notation

Sets, functions and relations

Alphabets

Abstract models

Operational semantics

Symbolic operational semantics

Denotational semantics

Conditions

Abbreviations

Miscellaneous

Bibliography

[ADI06] R. Alur, T. Dang, and F. Ivančić. Counterexample-guided predicate abstraction of hybrid systems. *Theoretical Computer Science*, 354(2):250–271, 2006.

[AE04] S. Abraham and R. J. Efford. Final report on the August 14, 2003 blackout in the United States and Canada: causes and recommendations. Technical report, U.S.-Canada Power System Outage Task Force, April 2004.

[AHI98] K. Ajami, S. Haddad, and J-M. Ilié. Exploiting symmetry in linear time temporal logic model checking: one step beyond. In *TACAS'98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67, 1998.

[AK86] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL'81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 164–176. ACM, 1981.

[Bar93] G. Barrett. Model checking in practice — the T9000 virtual channel processor. In *FME'93: Proceedings of the 1st International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 129–147. Springer-Verlag, 1993.

[BB04] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *CAV'04: Proceedings of the 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, 2004.

[BB06] J. C. M. Baeten and M. Bravetti. A generic process algebra. *Electronic Notes in Theoretical Computer Science*, 162:65–71, 2006.

[BBJ02] G. S. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and logic*. Cambridge University Press, March 2002.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *DAC'90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 46–51. ACM, 1990.

[Ben01] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *DAC'01: Proceedings of the 38th annual Design Automation Conference*, pages 244–248. ACM, 2001.

[BG05] S. Barner and O. Grumberg. Combining symmetry reduction and underapproximation for symbolic model checking. *Formal Methods in System Design*, 27(1-2):29–66, 2005.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[BK84] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.

[BLR00] P. J. Broadfoot, G. Lowe, and A. W. Roscoe. Automating data independence. In *ESORICS'00: Proceedings of the 6th European Symposium on Research in Computer Security*, volume 1895 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 2000.

[BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[BR00a] T. Ball and S. K. Rajamani. Bebop: a symbolic model checker for boolean programs. In *SPIN'00: Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer-Verlag, 2000.

[BR00b] T. Ball and S. K. Rajamani. Boolean programs: a model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, March 2000.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN'01: Proceedings of the 8th international SPIN workshop on Model checking of software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, 2001.

[BR02a] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *ACM SIGPLAN Notices*, 37(1):1–3, 2002.

[BR02b] P. J. Broadfoot and A. W. Roscoe. Capturing parallel attacks within the data independence framework. In *CSFW'02: Proceedings of the 15th IEEE workshop on Computer Security Foundations*, page 147. IEEE Computer Society, 2002.

[Bro01] P. J. Broadfoot. *Data independence in the model checking of security protocols*. DPhil thesis, University of Oxford, September 2001.

[BSV93] F. Balarin and A. L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *CAV'93: Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 29–40. Springer-Verlag, 1993.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260–261, 1969.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[CCG+03] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE'03: Proceedings of the 25th International Conference on Software Engineering*, pages 385–395. IEEE Computer Society, 2003.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Logic of Programs Workshop*, volume 125 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CEFJ96] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFH+03] E. M. Clarke, A. Fehnker, Z. Han, B. H. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science*, 14(4):583–604, 2003.

[CG87] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *PODC'87: Proceedings of the 6th Annual Association for Computing Machinery Symposium on Principles of Distributed Computing*, pages 294–303. ACM Press, 1987.

[CGH+95] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In *CHDL'93: Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications*, volume 6 of *Formal Methods in System Design*, pages 217–232. Springer-Verlag, 1995.

[CGJ+00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00: Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.

[CGJ+03] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGKS02] E. M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, 2002.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[CGP02] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 431–441. ACM, 2002.

[cit10] Most cited computer science articles (generated from CiteSeer$^{\mathrm{X}}$), `http://citeseerx.ist.psu.edu/stats/articles`, February 2010.

[CR98] S. Creese and A. W. Roscoe. TTP: A case study in combining induction and data independence. Technical Report PRG-TR-1-99, University of Oxford, 1998.

[CR99a] S. Creese and J. N. Reed. Verifying end-to-end protocols using induction with CSP/FDR. In *IPPS/SPDP'99: Proceedings of the IPPS/SPDP Workshop on Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 1243–1257. Springer-Verlag, 1999.

[CR99b] S. Creese and A. W. Roscoe. Formal verification of arbitrary network topologies. In *PDPTA'99: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 1999.

[CR99c] S. Creese and A. W. Roscoe. Verifying an infinite family of inductions simultaneously using data independence and FDR. In *IFIP'99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, pages 437–452, 1999.

[CR00] S. Creese and A. W. Roscoe. Data independent induction over structured networks. In *PDPTA'00: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000.

[Cre01] S. Creese. *Data independent induction: CSP model checking of arbitrary sized networks*. DPhil thesis, University of Oxford, 2001.

[Das03] S. Das. *Predicate abstraction*. PhD thesis, Stanford University, 2003.

[DD01] S. Das and D. L. Dill. Successive approximation of abstract transition relations. In *LICS'01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 51–60. IEEE Computer Society, 2001.

[DD02] S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. In *FMCAD'02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 19–32. Springer-Verlag, 2002.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD'92: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525. IEEE Computer Society, 1992.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV'99: Proceedings of the 11th International Conference on Computer Aided Verification*, volume 163 of *Lecture Notes in Computer Science*, pages 160–171. Springer-Verlag, 1999.

[DGL06] A. Dimovski, D. R. Ghica, and R. S. Lazić. A counterexample-guided refinement tool for open procedural programs. In *SPIN'06: Proceedings*

of the 13th International SPIN Workshop on Model Checking of Software, volume 3925 of *Lecture Notes in Computer Science*, pages 288–292. Springer-Verlag, 2006.

[Dil96] D. L. Dill. The Murphi verification system. In *CAV'96: Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer Verlag, 1996.

[DM05] A. F. Donaldson and A. Miller. Automatic symmetry detection for model checking using computational group theory. In *FM'05: Proceedings of the 13th International Symposium on Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 481–496, 2005.

[EHT00] E. A. Emerson, J. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *LICS'00: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, pages 121–131. IEEE Computer Society, 2000.

[EK00] E. A. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *CADE'00: Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pages 236–255. Springer-Verlag, 2000.

[EK04] E. A. Emerson and V. Kahlon. Parameterized model checking of ring-based message passing systems. In *CSL'04: Proceedings of the 18th International Workshop on Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 325–339. Springer-Verlag, 2004.

[EN94] J. Esparza and M. Nielsen. Decidability issues for Petri nets. *Petri Nets Newsletter*, 52:245–262, 1994.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *POPL'95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–94. ACM, 1995.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic verification of parameterized synchronous systems (extended abstract). In *CAV'96: Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 87–98. Springer-Verlag, 1996.

[ER64] C. C. Elgot and A. Robinson. Random-access stored-program machines, an approach to programming languages. *Journal of the ACM*, 11(4):365–399, 1964.

[ES93] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *CAV'93: Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478. Springer-Verlag, 1993.

[ES97] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: an automata-theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4):617–638, 1997.

[ET98] E. A. Emerson and R. J. Trefler. Model checking real-time properties of symmetric systems. In *MFCS'98: Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 427–436. Springer-Verlag, 1998.

[ET99] E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: new techniques for symmetry reduction in model checking. In *CHARME'99: Proceedings of the Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 142–156. Springer-Verlag, 1999.

[EW03] E. A. Emerson and T. Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In *CHARME'03: Proceedings of the Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 216–230. Springer-Verlag, 2003.

[EW05] E. A. Emerson and T. Wahl. Dynamic symmetry reduction. In *TACAS'05: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis*, volume 3440 of *Lecture Notes in Computer Science*, pages 382–396. Springer-Verlag, 2005.

[Fit96] M. Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag, 1996.

[For99] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement — FDR 2 user manual,* `http://www.fsel.com/fdr2_manual.html`, 1999.

[For03] Formal Systems (Europe) Ltd. *Process Behaviour Explorer — ProBE user manual,* `http://www.fsel.com/probe\_manual.html`, 2003.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL'97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 174–186. ACM, 1997.

[GS92] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97: Proceedings of the 9th International Conference on Computer*

Aided Verification, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[GS99] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design: An International Journal*, 15(3):217–238, November 1999.

[Hac74] M. Hack. The recursive equivalence of the reachability problem and the liveness problem for petri nets and vector addition systems. In *SWAT'74: Proceedings of the 15th Annual Symposium on Switching and Automata Theory*, pages 156–164. IEEE Computer Society, 1974.

[HB95] R. Hojati and R. K. Brayton. Automatic datapath abstraction in hardware systems. In *CAV'95: Proceedings of the 7th International Conference on Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 1995.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[HLP01] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a spacecraft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, 1985.

[HP76] J. E. Hopcroft and J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. Technical Report TR76-280, Cornell University, 1976.

[HP85] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Logics and models of concurrent systems. Springer-Verlag, 1985.

[ID96] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996.

[KM69] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[KM82] T. Kasai and R. E. Miller. Homomorphisms between models of parallel computation. *Journal of Computer and System Sciences*, 25(3):285–331, 1982.

[KM89] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. In *PODC'89: Proceedings of the 8th annual ACM Symposium on Principles of Distributed Computing*, pages 239–247. ACM, 1989.

[KNS01] M. Z. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using Cadence SMV and PRISM. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 194–206, London, UK, 2001. Springer-Verlag.

[KP00] Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.

[Kun97] R. Kunzig. Europe's dream - explosion of Ariane 5 rocket - Special issue: the coming age of exploration - interview. *Discover*, 18:96–103, May 1997.

[Kur94] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[Lam61] J. Lambek. How to program an infinite abacus. *Canadian Mathematical Bulletin*, 4:295–302, 1961.

[Lam74] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lan96] G. L. Lann. The Ariane 5 flight 501 failure - a case study in system engineering for computing systems. Technical Report RR-3079, Institut National de Recherche en Informatique et en Automatique, 1996.

[Laz99] R. S. Lazić. *A semantic study of data independence with applications to model checking*. DPhil thesis, University of Oxford, 1999.

[LB04] S. Lahiri and R. Bryant. Constructing quantified invariants via predicate abstraction. In G. Levi and B. Steffen, editors, *VMCAI'04: Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 267–281. Springer-Verlag, 2004.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS'01: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112. Springer-Verlag, 2001.

[LF08] M. Leuschel and M. Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *ICFEM'08: Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, volume 5256 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 2008.

[LHR01] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized networks of processes. *Theoretical Computer Science*, 256(1–2):113–144, 2001.

[LNR04a] R. S. Lazić, T. C. Newcomb, and A. W. Roscoe. On model checking data-independent systems with arrays with whole-array operations. In *Proceedings of the Symposium on the Occasion of 25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 275–291. Springer-Verlag, 2004.

[LNR04b] R. S. Lazić, T. C. Newcomb, and A. W. Roscoe. On model checking data-independent systems with arrays without reset. *Theory and Practice of Logic Programming*, 4(5-6):659–693, 2004.

[LO06] G. Lowe and J. Ouaknine. On timed models and full abstraction. In *MFPS'06: Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 497–519. Elsevier, 2006.

[Low95] G. Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *TACAS'96: Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

[Low98] G. Lowe. Casper: a compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[Low04] G. Lowe. On the application of counterexample-guided abstraction refinement and data independence to the parameterised model checking problem. In *AVIS'04: Proceedings of the 3rd International Workshop on Automatic Verification of Infinite-State Systems*, 2004.

[LP07] M. Leuschel and D. Plagge. Seven at one stroke: LTL model checking for B, Z, CSP, and more. In *AVoCS'07: Proceedings of the 7th International Workshop on Automated Verification of Critical Systems*, September 2007.

[LR98] R. S. Lazić and A. W. Roscoe. Verifying determinism of concurrent systems which use unbounded arrays. Technical Report TR-2-98, University of Oxford, 1998.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[Lub84] B. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Informatica*, 1984.

[Maz08] T. Mazur. Formal verification of not fully symmetric systems using counter abstraction. In *MOVEP'08: Proceedings of the MOdelling and VErifying Process*, 2008.

[MC85] B. Mishra and E. M. Clarke. Hierarchical verification of asynchronous circuits using temporal logic. *Theoretical Computer Science*, 38(2-3):269–291, 1985.

[McM92] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[McM98] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV'98: Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.

[McM99] K. L. McMillan. Verification of infinite state systems by compositional model checking. In Laurence Pierre and Thomas Kropf, editors, *CHARME'99: Proceedings of the Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*. Springer, 1999.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[Min61] M. L. Minsky. Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines. *Annals of Mathematics*, 74(3):437–455, 1961.

[Min67] M. L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, 1967.

[ML09] T. Mazur and G. Lowe. Counter abstraction in the CSP/FDR setting. In *AVoCS'07: Proceedings of the 7th International Workshop on Automated Verification of Critical Systems*, volume 250 of *Electronic Notes in Theoretical Computer Science*, pages 171–186. Elsevier, 2009.

[MP95] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, 1995.

[MPW89] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100, 1989.

[MQS00] Kenneth L. McMillan, Shaz Qadeer, and James B. Saxe. Induction in compositional model checking. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 312–327, London, UK, 2000. Springer-Verlag.

[MS91] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocols. In *ICPDC'91: Proceedings of the ISSM International Conference on Parallel and Distributed Computing*, pages 242–251, 1991.

[New03] T. C. Newcomb. *Model checking data-independent systems with arrays*. DPhil thesis, University of Oxford, 2003.

[PB05] E. Pek and N. Bogunović. Predicate abstraction in protocol verification. In *ConTEL'05: Proceedings of the 8th International Conference on Telecommunications*, volume 2, pages 627–632, June 2005.

[PD95] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):773–787, 1995.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Volume 1, 1966, English translation.

[Pie96] B. C. Pierce. *Foundational Calculi for Programming Languages*, chapter 139. The CRC Handbook of Computer Science and Engineering. CRC Press, 1996.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *SFCS'77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[PRI] *PRISM user manual,* `http://www.prismmodelchecker.org/manual/`.

[PXZ02] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag, 2002.

[PY96] A. N. Parashkevov and J. Yantchev. ARC — a tool for efficient refinement and equivalence checking for CSP. In *ICA3PP'96: Proceedings of the IEEE International Conference on Algorithms and Architectures for Parallel Processing*, pages 68–75. IEEE Computer Society, 1996.

[Qad03] S. Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):730–741, 2003.

[Rac78] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6:223–231, 1978.

[RB99] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2-3):147–190, 1999.

[RGG+95] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson, and B. Scattergood. Hierarchical compression for model-checking CSP or how to check 10^{20} dining philosophers for deadlock. In *TACAS'95: Proceedings of the 1st International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 133–152. Springer-Verlag, 1995.

[RGM+03] A. W. Roscoe, M. H. Goldsmith, N. Moffat, T. Whitworth, and I. Zakiuddin. Watchdog transformations for property-oriented model checking. In *FME'03: Proceedings of the 12th Formal Methods Europe Symposium*, volume 2805 of *Lecture Notes in Computer Science*, pages 600–616. Springer-Verlag, 2003.

[RH07] A. W. Roscoe and D. Hopkins. SVA, a tool for analysing shared-variable programs. In *AVoCS'07: Proceedings of the 7th International Workshop On Automated Verification Of Critical Systems*, pages 177–183, 2007. to appear.

[RJDR98] J. N. Reed, D. M. Jackson, B. Deianov, and G. M. Reed. Automated formal analysis of networks: FDR models of arbitrary topologies and flow-control mechanisms. In *FASE'98: Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag, 1998.

[RL01] A. W. Roscoe and R. S. Lazić. What can you decide about resetable arrays? Technical Report DSSE-TR-2001-3 (VCL'01: Proceedings of the 2nd International Workshop on Verification and Computational Logic), University of Southampton, 2001.

[RL05] G. T. Rohrmair and G. Lowe. Using data-independence in the analysis of intrusion detection systems. *Theoretical Computer Science*, 340(1):82–101, 2005.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[Ros97] A. W. Roscoe. *The theory and practice of concurrency*. Prentice-Hall, 1997.

[Ros98] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *CSFW'98: Proceedings of the 11th IEEE workshop on Computer Security Foundations*, pages 84–95. IEEE Computer Society, 1998.

[Ros01] A. W. Roscoe. Compiling shared variable programs into CSP. In *Proceedings of PROGRESS workshop 2001*, 2001.

[Ros08a] A. W. Roscoe. On the expressiveness of CSP. 2008. Draft of October, 2008.

[Ros08b] A. W. Roscoe. Revivals, stuckness and the hierarchy of CSP models. 2008. Draft of December, 2008.

[Ros08c] A. W. Roscoe. The three platonic models of divergence-strict CSP. In *ICTAC'08: Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 23–49. Springer-Verlag, 2008.

[Ros10] A. W. Roscoe. *Understanding concurrent systems*. Springer-Verlag, 2010.

[RRS07] J. N. Reed, A. W. Roscoe, and J. E. Sinclair. Responsiveness and stable revivals. *Formal Aspects of Computing*, 19(3):303–319, 2007.

[RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of automated reasoning, volume I and II*. Elsevier, 2001.

[Sab88] K. Sabnani. An algorithmic technique for protocol verification. *IEEE Transactions on Communications*, 36:924–931, August 1988.

[SBD02] A. Stump, C. Barrett, and D. L. Dill. CVC: a Cooperating Validity Checker. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, volume 400-423 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[Sca98] B. Scattergood. *The semantics and implementation of machine readable CSP*. DPhil thesis, University of Oxford, 1998.

[Sip96] M. Sipser. *Introduction to the theory of computation*. International Thomson Publishing, 1996.

[SLD08] J. Sun, Y. Liu, and J. S. Dong. Model checking CSP revisited: Introducing a Process Analysis Toolkit. In *ISoLa'08: Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 307–322. Springer-Verlag, 2008.

[SLR+09] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In *FM'09: Proceedings of the 2nd World Congress on Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 123–139. Springer-Verlag, 2009.

[SS63] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963.

[Sta98] W. Stallings. *Operating systems: internals and design principles (3rd ed.)*. Prentice-Hall, Inc., 1998.

[SW01] D. Sangiorgi and D. Walker. *Pi-Calculus: A theory of mobile processes*. Cambridge University Press, 2001.

[Tan87] A. S. Tanenbaum. *Operating systems: design and implementation.* Prentice-Hall, 1987.

[Tho10] R. Thompson. On the application of counter abstraction to the parameterised model checking problem using CSP and FDR. Master's thesis, University of Oxford, 2010.

[van97] R. J. van Glabbeek. Notes on the methodology of CCS and CSP. *Theoretical Computer Science*, 177(2):329–349, 1997.

[van00] H. van Vliet. *Software engineering: principles and practice.* John Wiley & Sons, Inc., 2000.

[Ver94] I. Vernier. Parameterized evaluation of CTL-X formulae. In *Proceedings of the 1st International Conference on Temporal Logic workshop*, 1994.

[Wah07] T. Wahl. Adaptive symmetry reduction. In *CAV'07: Proceedings of the 19th international conference on Computer aided verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 393–405. Springer-Verlag, 2007.

[WGC05] O. Wei, A. Gurfinkel, and M. Chechik. Identification and counter abstraction for full virtual symmetry. In *CHARME'05: Proceedings of Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 285–300. Springer-Verlag, 2005.

[WL90] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *AVMFSS'90: Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1990.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL'86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 184–193. ACM, 1986.

[Xu05] J. Xu. *Automatic verification of parameterized systems.* PhD thesis, New York University, 2005.

Appendix A

CSP definition of $\zeta_z(\mathcal{N})$ for Example 3.3.12

$$
\begin{aligned}
&CANodes'_z(c_1, c_2, c_3, c_4, c_5) = \\
&\quad c_1 \geq 1 \;\&\; load \rightarrow CANodes'_z(0, 1, c_3, c_4, c_5) \\
&\quad \Box \\
&\quad c_2 \geq 1 \;\&\; run \rightarrow (CANodes'_z(c_1, 0, (c_3 + 1) \sqcap 2, c_4, c_5) \\
&\qquad\qquad \sqcap\; CANodes'_z(c_1, 1, (c_3 + 1) \sqcap 2, c_4, c_5)) \\
&\quad \Box \\
&\quad c_3 \geq 1 \;\&\; deschedule \rightarrow \\
&\qquad (\text{if } c_3 = 2 \text{ then } CANodes'_z(c_1, (c_2 + 1) \sqcap 1, 1, c_4, c_5) \\
&\qquad\qquad \sqcap\; CANodes'_z(c_1, (c_2 + 1) \sqcap 1, 2, c_4, c_5) \\
&\qquad\quad \text{else } CANodes'_z(c_1, (c_2 + 1) \sqcap 1, 0, c_4, c_5)) \\
&\quad \Box \\
&\quad c_3 \geq 1 \;\&\; block \rightarrow \\
&\qquad (\text{if } c_3 = 2 \text{ then } CANodes'_z(c_1, c_2, 1, (c_4 + 1) \sqcap z_4, c_5) \\
&\qquad\qquad \sqcap\; CANodes'_z(c_1, c_2, 2, (c_4 + 1) \sqcap z_4, c_5) \\
&\qquad\quad \text{else } CANodes'_z(c_1, c_2, 0, (c_4 + 1) \sqcap z_4, c_5)) \\
&\quad \Box \\
&\quad c_3 \geq 1 \;\&\; terminate \rightarrow \\
&\qquad (\text{if } c_3 = 2 \text{ then } CANodes'_z(c_1, c_2, 1, c_4, (c_5 + 1) \sqcap 1) \\
&\qquad\qquad \sqcap\; CANodes'_z(c_1, c_2, 2, c_4, (c_5 + 1) \sqcap 1) \\
&\qquad\quad \text{else } CANodes'_z(c_1, c_2, 0, c_4, (c_5 + 1) \sqcap 1)) \\
&\quad \Box \\
&\quad c_4 \geq 1 \;\&\; interrupt \rightarrow (CANodes'_z(c_1, (c_2 + 1) \sqcap 1, c_3, 0, c_5) \\
&\qquad\qquad\qquad \sqcap\; CANodes'_z(c_1, (c_2 + 1) \sqcap 1, c_3, 1, c_5))
\end{aligned}
$$

Appendix B

The $\langle f[\cdot]$-ren-app$\rangle$ law

Given a renaming function F and renaming relation $\mathcal{R}$, we define

$$\mathcal{R}[\![F]\!] = \{(F(a), F(b)) \mid a\mathcal{R}b\}.$$

Then we have the following result.

Lemma B.0.2 ($\langle f[\cdot]$-ren-app$\rangle$). *Given a process P, an injective renaming F and a relational renaming $\mathcal{R}$, we have that*

$$F(P[\![\mathcal{R}]\!]) = (F(P))[\![\mathcal{R}[\![F]\!]]\!].$$

Proof: We have that for any events a and b

$$\begin{array}{ll} & a(F \circ \mathcal{R})b \\ \Leftrightarrow & \{\text{definition of relation composition}\} \\ & \exists\, c \bullet a\mathcal{R}c \wedge cFb \\ \Leftrightarrow & \{\text{F is a function}\} \\ & \exists\, c \bullet a\mathcal{R}c \wedge b = F(c) \\ \Leftrightarrow & \{\text{definition of } \mathcal{R}[\![F]\!]\} \\ & F(a)(\mathcal{R}[\![F]\!])b \\ \Leftrightarrow & \{\text{F is a function}\} \\ & \exists\, d \bullet aFd \wedge d(\mathcal{R}[\![F]\!])b \\ \Leftrightarrow & \{\text{definition of relation composition}\} \\ & a((\mathcal{R}[\![F]\!]) \circ F)b. \end{array}$$

This means that $F \circ \mathcal{R} = (\mathcal{R}[\![F]\!]) \circ F$. Hence

$$\begin{array}{ll} & F(P[\![\mathcal{R}]\!]) \\ = & P[\![F \circ \mathcal{R}]\!] \\ = & P[\![(\mathcal{R}[\![F]\!]) \circ F]\!] \\ = & (F(P))[\![\mathcal{R}[\![F]\!]]\!], \end{array}$$

which is what we had to prove. ∎

Appendix C

CSP scripts

C.1 Example 3.0.1

C.1.1 Source code

```
1  -- This script models a process scheduling mechanism in an
2  -- operating system for a multiprocessor machine.
3
4  -- The implementation consists of a number of identical nodes,
5  -- each representing a single process requesting CPU access,
6  -- and a scheduler.
7
8  -- The specification says that there should never be more
9  -- processes with CPU access than there are CPUs. Further,
10 -- every process with CPU access cannot be denied giving up
11 -- the access.
12
13
14 T = {0..4} -- an example instantiation T of type t
15           -- this is not needed for TomCAT, but allows one to
16           -- directly verify a single size of the implementation
17           -- by executing the input script in FDR
18
19 cores = 2  -- the number of CPUs
20
21 datatype NodeStates = new | runnable | running | blocked
22 datatype CoreStates = idle | busy
23
24 channel load,run,block,terminate,deschedule,interrupt,stopRun
25
26
27 -- A single node process
28
29 Node = Node'(new)
30
31 Node'(state) = state == new & load -> Node'(runnable)
32              []
33              state == runnable & run -> Node'(running)
34              []
35              state == running & (block -> Node'(blocked)
```

```
36                                      []
37                                      terminate -> STOP
38                                      []
39                                      deschedule -> Node'(runnable))
40              []
41              state == blocked & interrupt -> Node'(runnable)
42
43
44  -- Parallel composition of all node processes
45
46  Nodes = [|ASync|] i:T @ Node
47
48
49  --  The set of events on which all nodes have to synchronise
50
51  ASync = {load}
52
53
54  -- A single CPU
55
56  Core(idle) = run -> Core(busy)
57  Core(busy) = deschedule -> Core(idle)
58               []
59               block -> Core(idle)
60               []
61               terminate -> Core(idle)
62
63
64  -- The most nondeterministic scheduler
65
66  Scheduler = |||i:{0..(cores-1)} @ Core(idle)
67
68
69  -- Composing the implementation
70
71  C(P) = (P [|{|run,terminate,deschedule,block|}|] Scheduler)
72             [[deschedule<-stopRun,block<-stopRun,terminate<-stopRun]]
73
74  Impl = C(Nodes)
75
76
77  -- Defining the specification
78
79  Spec = Spec'(0)
80
81  Spec'(x) = x < cores & (run -> Spec'(x+1) |~| STOP)
82             []
83             x > 0 & stopRun -> Spec'(x-1)
84             []
85             (load -> Spec'(x) |~| interrupt -> Spec'(x) |~| STOP)
86
87  -- An assertion for FDR
88
89  assert Spec [F= Impl
```

C.1.2 TomCAT's output

```
1  DoAlphaType = if empty(AlphaTransitions) then {null}
2                                           else AlphaTransitions
3  GammaEventsNonEmpty = if empty(GammaEvents) then {null} else GammaEvents
4  channel doAlpha : DoAlphaType
5  channel doGamma : GammaEventsNonEmpty.Seq({(s0,s1,c) |
6                                        s0 <- States, s1 <- States,
7                                        c <- {0..max(max(sumz,z(0)),1)}})
8  channel TAU_
9
10 T = {0..4} -- an example instantiation T of type t
11            -- this is not needed for TomCAT, but allows one to
12            -- directly verify a single size of the implementation
13            -- by executing the input script in FDR
14
15 cores = 2
16
17 datatype NodeStates = new | runnable | running | blocked
18 datatype CoreStates = idle | busy
19
20 channel load,run,block,terminate,deschedule,interrupt,stopRun
21
22
23 -- A single node process
24
25 Node = Node'(new)
26
27 Node'(state) = state == new & load -> Node'(runnable)
28              []
29              state == runnable & run -> Node'(running)
30              []
31              state == running & (block -> Node'(blocked)
32                                  []
33                                  terminate -> STOP
34                                  []
35                                  deschedule -> Node'(runnable))
36              []
37              state == blocked & interrupt -> Node'(runnable)
38
39
40 -- Parallel composition of all node processes
41
42 Nodes = [|ASync|] i:T @ Node
43
44
45 --  The set of events on which all nodes have to synchronise
46
47 ASync = {load}
48
49
50 -- A single CPU
51
52 Core(idle) = run -> Core(busy)
53 Core(busy) = deschedule -> Core(idle)
```

```
54                []
55                block -> Core(idle)
56                []
57                terminate -> Core(idle)
58
59
60  -- The most nondeterministic scheduler
61
62  Scheduler = |||i:{0..(cores-1)} @ Core(idle)
63
64
65  -- Composing the implementation
66
67  C(P) = (P [|{|run,terminate,deschedule,block|}|] Scheduler)
68                [[deschedule<-stopRun,block<-stopRun,terminate<-stopRun]]
69
70  Impl = C(Nodes)
71
72
73  -- Defining the specification
74
75  Spec = Spec'(0)
76
77  Spec'(x) = x < cores & (run -> Spec'(x+1) |~| STOP)
78             []
79             x > 0 & stopRun -> Spec'(x-1)
80             []
81             (load -> Spec'(x) |~| interrupt -> Spec'(x) |~| STOP)
82
83  -- An assertion for FDR
84
85  assert Spec [F= Impl
86
87
88  -- ### THRESHOLDS ###
89
90  -- Define a sequence of thresholds for the states of Node, e.g.
       Thresholds = <2,0,3,0,1>
91
92  Thresholds = <ENTER THRESHOLD SEQUENCE HERE>
93
94  -- z function allows us to extract values of thresholds
95
96  z(x) =
97   let
98       zsub(0,<n>^ns) = n
99       zsub(x,<n>^ns) = zsub(x-1,ns)
100  within
101      zsub(x,Thresholds)
102
103 sumz =
104  let
105      sumzsub(total,0) = total + z(0)
106      sumzsub(total,x) = sumzsub(total + z(x),x-1)
107  within
```

```
108         sumzsub(0,card(States)-1)
109
110
111 -- ### STATES AND TRANSITIONS ###
112
113 States={0..4}
114
115 Transitions = {(0,load,1),(1,run,2),(2,terminate,3),(2,deschedule,1),
116                (2,block,4),(4,interrupt,1)}
117
118 initState = 0
119
120
121 -- ### ALPHA TRANSITIONS ###
122
123 AlphaTransitions = {(s0,e,s1) | (s0,e,s1) <- Transitions,
124                                 not member(e,ASync)}
125
126 IncomingAlpha(st) = {(s0,e,s1) | (s0,e,s1) <- AlphaTransitions,
127                                  s0 != st and s1 == st}
128 SelfloopsAlpha(st) = {(s0,e,s1) | (s0,e,s1) <- AlphaTransitions,
129                                   s0 == st and s1 == st}
130 OutgoingAlpha(st) = {(s0,e,s1) | (s0,e,s1) <- AlphaTransitions,
131                                  s0 == st and s1 != st}
132
133
134 -- ### GAMMA-EVENTS AND TRANSITIONS ###
135
136 GammaEvents = ASync
137
138 GammaTransitions = {(s0,e,s1) | (s0,e,s1) <- Transitions,
139                                 member(e,GammaEvents)}
140
141 GenerateDoGammaTransitions =
142  let
143   GammaSeq = makeSeq(GammaTransitions)
144  within
145   {(e,<(s0,s1) | (s0,e',s1) <- GammaSeq, e == e'>) | e <- GammaEvents}
146
147 lex((s0,e,s1),(s0',e',s1')) = s0' < s0 or (s0' == s0 and s1' <= s1)
148
149 stateMin(S) = { x | x <- S, { y | y <- S, lex(x,y)} == {x}}
150
151 makeSeq({}) = <>
152 makeSeq(S) =
153  let
154      first = pick(stateMin(S))
155  within
156      <first>^makeSeq(diff(S,{first}))
157
158 getSeq(e) = pick({seq | (e',seq) <- GenerateDoGammaTransitions,
159                         e == e'})
160
161 allCounts(<>) = {<>}
162 allCounts(<(s0,s1)>^ps) =
```

```
163   {<(s0,s1,c)>^ts | ts <- allCounts(ps),
164                     c <- {0..max(max(sumz,z(0)),1)}}
165
166
167  -- ### AUXILIARY FUNCTIONS ###
168
169  min(a,b) = if a < b then a else b
170  max(a,b) = if a < b then b else a
171  pick({}) = <>
172  pick({x}) = x
173
174
175  -- ### SINGLE COUNTER ###
176
177  Counter(st, c) =
178   let
179       sumActual(<>) = 0
180       sumActual(<(s0,s1,c)>^ps) = c + sumActual(ps)
181       v(st,<>) = 0 -- v specifies an actual number of processes that a
              counter represents
182       v(st,<(s0,s1,c)>^ps) = if st == s0 then c + v(st,ps) else v(st,ps)
183       firstState(st,<>) = false
184       firstState(st,<(s0,s1,c)>^ps) = if st == s0 then true
185                                                   else firstState(st,ps)
186       sumzTargets(st,<>) = 0
187       sumzTargets(st,<(s0,s1,c)>^ps) =
188            if st == s0 then z(s1) + sumzTargets(st,ps)
189                        else sumzTargets(st,ps)
190       sumSecond(st,<>) = 0 -- calculates how many processes move into
              state st
191       sumSecond(st,<(s0,s1,c)>^ps) =
192            if st == s1 then c + sumSecond(st,ps) else sumSecond(st,ps)
193   within
194       (doAlpha?_:IncomingAlpha(st) -> Counter(st, min(c+1,z(st))))
195       []
196       (c >= min(z(st),1) & doAlpha?_:SelfloopsAlpha(st) -> Counter(st, c)
              )
197       []
198       (c >= min(z(st),1) & doAlpha?_:OutgoingAlpha(st) ->
199            (if c == z(st) then Counter(st, c) |~| Counter(st, c-1)
200                           else Counter(st, c-1)))
201       []
202       doGamma?e:GammaEvents?seq:{seq |
203            seq <- allCounts(getSeq(e)), sumActual(seq) > 0 and
204            (not firstState(st,seq) or (firstState(st,seq) and
205             v(st,seq) <= sumzTargets(st,seq) and
206             if c == z(st) then v(st,seq) >= c else v(st,seq) == c)} ->
207                 Counter(st,min(sumSecond(st,seq),z(st)))
208
209  AAlpha(st) = {doAlpha.a |
210    a <- Union({IncomingAlpha(st),SelfloopsAlpha(st), OutgoingAlpha(st)})}
211  AGamma(st) = {doGamma.a.seq | a <- GammaEvents,
212                                seq <- allCounts(getSeq(a))}
213
214  AlphabetOfCounter(st) = union(AAlpha(st), AGamma(st))
```

```
215
216
217 -- ### PARALLEL COMPOSITION OF ALL COUNTERS ###
218
219 Counters = || st:States @ [AlphabetOfCounter(st)]
220                     Counter(st, if st == initState then z(st) else 0)
221
222 CANodes =
223  ((Counters
224     [[ doAlpha.(s0,e1,s1) <- e1 | (s0,e1,s1) <- AlphaTransitions ]])
225      \ {TAU_})
226       [[doGamma.e2.seq <- e2 | e2 <- GammaEvents,
227                                seq <- allCounts(getSeq(e2)) ]]
228
229
230 -- ### ASSERTION ###
231
232 assert Spec [F= C(CANodes)
```

C.2 Case study

C.2.1 Source code

```
1  -- This script models a producer-consumer problem.
2
3  -- The implementation consists of a single producer,
4  -- a single slot that can hold at most one item of data and
5  -- a number of nodes (consumers) with identities of type t.
6  -- In our system we use three binary semaphores: Mutex,
7  -- SlotEmptySemaphore and SlotFullSemaphore. The Mutex
8  -- semaphore is used to achieve mutual exclusion of
9  -- critical sections of node processes. We use
10 -- SlotEmptySemaphore to wake up the producer whenever
11 -- the slot is ready to accept data. Finally,
12 -- SlotEmptySemaphore is used to wake up nodes whenever
13 -- the slot contains a piece of data.
14
15 -- We want to verify that every implementation
16 -- traces-refines a three-place buffer.
17
18
19 T = {0..2}    -- an example instantiation T of type t
20              -- must be a superset of {0..B+m}
21              -- so that the types of channels are well-defined
22
23 Data = {0,1} -- type of data being input from / output to
24              -- the environment
25
26 datatype Semaphore = up | down
27
28 channel init, fullSlotUp, emptySlotUp, emptySlotDown
29 channel input, put: Data
30 channel mutexDown, mutexUp, fullSlotDown : T
31 channel get : T.Data
```

```
32 channel exchangeMutex : T.T
33 channel output : T.Data
34
35
36 -- Defining the three semaphores
37
38 Mutex(state,t) =
39  state == up & mutexDown?i:t -> Mutex(down,t)
40  []
41  mutexUp?i:t -> Mutex(up,t)
42
43 FullSlotSemaphore(state,t) =
44  state == up & fullSlotDown?i:t -> FullSlotSemaphore(down,t)
45  []
46  fullSlotUp -> FullSlotSemaphore(up,t)
47
48 EmptySlotSemaphore(state) =
49  state == up & emptySlotDown -> EmptySlotSemaphore(down)
50  []
51  emptySlotUp -> EmptySlotSemaphore(up)
52
53
54 -- Defining the producer
55
56 Producer = input?d:Data -> emptySlotDown -> put.d -> Producer
57
58
59 -- Defining the nodes
60
61 Node(i,t) = init -> Node'(i,t)
62
63 Node'(i,t) =
64  mutexDown.i -> Node''(i,t)
65  []
66  exchangeMutex?j:t!i -> Node''(i,t)
67
68 Node''(i,t) =
69  fullSlotDown.i -> get.i?d:Data -> output.i.d -> Node'''(i,t)
70
71 Node'''(i,t) =
72  exchangeMutex.i?j:t -> Node'(i,t)
73  []
74  mutexUp.i -> Node'(i,t)
75
76 A(i,t) = {init,mutexDown.i,fullSlotDown.i,get.i.d,output.i.d,mutexUp.i,
77           exchangeMutex.i.j,exchangeMutex.j.i | j <- diff(t,{i}), d <-
                Data}
78
79 Nodes(t) = || i:t @ [A(i,t)] Node(i,t)
80
81
82 -- Defining the slot
83
84 EmptySlot(t) = emptySlotUp -> put?d:Data -> FullSlot(d,t)
85
```

```
86  FullSlot(d,t) = fullSlotUp -> get?i:t!d -> EmptySlot(t)
87
88
89  -- Defining the controller
90
91  Controller(t) =
92   Mutex(up,t) |||  ((Producer [|{|put|}|] EmptySlot(t))
93                       [|{emptySlotDown,emptySlotUp,fullSlotUp}|]
94                        (EmptySlotSemaphore(down)
95                           |||  FullSlotSemaphore(down,t)))
96
97
98  -- Composing all parts together
99
100 X(t) = {init}
101 Y(t) = {|mutexUp,mutexDown,fullSlotDown,get|}
102 Z(t) = {|mutexUp,mutexDown,exchangeMutex,fullSlotUp,fullSlotDown,
103           emptySlotUp,emptySlotDown,put,get|}
104
105 C(P,t) = ((P \ X(t)) [|Y(t)|] Controller(t)) \ Z(t)
106
107 Impl(t) = C(Nodes(t),t)
108
109
110 -- Defining the specification
111
112 Buff(<>,t) = input?d:Data -> Buff(<d>,t)
113 Buff(<d>^seq,t) =
114  output?i:t!d -> Buff(seq,t)
115  []
116  length(seq) < 2 & input?d':Data -> Buff(<d>^seq^<d'>,t)
117
118 Spec(t) = Buff(<>,t)
119
120
121 -- an assertion for FDR
122
123 assert Spec(T) [T= Impl(T)
```

C.2.2 TomCAT's output

```
1  DoAlphaType = if empty(AlphaTransitions) then {null}
2                                           else AlphaTransitions
3  GammaEventsNonEmpty = if empty(GammaEvents) then {null} else GammaEvents
4  channel doAlpha : DoAlphaType
5  channel doBeta  : BetaEvents.Int.Int.Int.Int
6  channel doGamma : GammaEventsNonEmpty.Seq({(s0,s1,c) | s0 <- States,
7                        s1 <- States, c <- {0..max(max(sumz,z(1)),1)}})
8  channel TAU_
9
10 T = {0..2}    -- an example instantiation T of type t
11               -- must be a superset of {0..B+m}
12               -- so that the types of channels are well-defined
13
```

```
14 Data = {0,1} -- type of data being input from / output to
15               -- the environment
16
17 datatype Semaphore = up | down
18
19 channel init, fullSlotUp, emptySlotUp, emptySlotDown
20 channel input, put: Data
21 channel mutexDown, mutexUp, fullSlotDown : T
22 channel get : T.Data
23 channel exchangeMutex : T.T
24 channel output : T.Data
25
26
27 -- Defining the three semaphores
28
29 Mutex(state,t) =
30  state == up & mutexDown?i:t -> Mutex(down,t)
31  []
32  mutexUp?i:t -> Mutex(up,t)
33
34 FullSlotSemaphore(state,t) =
35  state == up & fullSlotDown?i:t -> FullSlotSemaphore(down,t)
36  []
37  fullSlotUp -> FullSlotSemaphore(up,t)
38
39 EmptySlotSemaphore(state) =
40  state == up & emptySlotDown -> EmptySlotSemaphore(down)
41  []
42  emptySlotUp -> EmptySlotSemaphore(up)
43
44
45 -- Defining the producer
46
47 Producer = input?d:Data -> emptySlotDown -> put.d -> Producer
48
49
50 -- Defining the nodes
51
52 Node(i,t) = init -> Node'(i,t)
53
54 Node'(i,t) =
55  mutexDown.i -> Node''(i,t)
56  []
57  exchangeMutex?j:t!i -> Node''(i,t)
58
59 Node''(i,t) =
60  fullSlotDown.i -> get.i?d:Data -> output.i.d -> Node'''(i,t)
61
62 Node'''(i,t) =
63  exchangeMutex.i?j:t -> Node'(i,t)
64  []
65  mutexUp.i -> Node'(i,t)
66
67 A(i,t) = {init,mutexDown.i,fullSlotDown.i,get.i.d,output.i.d,mutexUp.i,
```

```
68              exchangeMutex.i.j,exchangeMutex.j.i | j <- diff(t,{i}), d <-
                    Data}
69
70  Nodes(t) = || i:t @ [A(i,t)] Node(i,t)
71
72
73  -- Defining the slot
74
75  EmptySlot(t) = emptySlotUp -> put?d:Data -> FullSlot(d,t)
76
77  FullSlot(d,t) = fullSlotUp -> get?i:t!d -> EmptySlot(t)
78
79
80  -- Defining the controller
81
82  Controller(t) =
83   Mutex(up,t) ||| ((Producer [|{|put|}|] EmptySlot(t))
84                       [|{emptySlotDown,emptySlotUp,fullSlotUp}|]
85                        (EmptySlotSemaphore(down)
86                           ||| FullSlotSemaphore(down,t)))
87
88
89  -- Composing all parts together
90
91  X(t) = {init}
92  Y(t) = {|mutexUp,mutexDown,fullSlotDown,get|}
93  Z(t) = {|mutexUp,mutexDown,exchangeMutex,fullSlotUp,fullSlotDown,
94            emptySlotUp,emptySlotDown,put,get|}
95
96  C(P,t) = ((P \ X(t)) [|Y(t)|] Controller(t)) \ Z(t)
97
98  Impl(t) = C(Nodes(t),t)
99
100
101 -- Defining the specification
102
103 Buff(<>,t) = input?d:Data -> Buff(<d>,t)
104 Buff(<d>^seq,t) =
105  output?i:t!d -> Buff(seq,t)
106  []
107  length(seq) < 2 & input?d':Data -> Buff(<d>^seq^<d'>,t)
108
109 Spec(t) = Buff(<>,t)
110
111
112 -- an assertion for FDR
113
114 assert Spec(T) [T= Impl(T)
115
116
117 -- ### PARAMETERS ###
118
119 B = 0
120 m = 1
121
```

```
122
123 -- ### AUXILIARY FUNCTIONS ###
124
125 phi(x) = if x < B then x else B
126 min(a,b) = if a < b then a else b
127 max(a,b) = if a < b then b else a
128 pick({}) = -1
129 pick({x}) = x
130
131
132 -- ### THRESHOLDS ###
133
134 -- Define a sequence of thresholds for all equivalence classes, e.g.
       Thresholds = <2,0,3,0,1>
135
136 Thresholds = <ENTER THRESHOLD SEQUENCE HERE>
137
138 -- z function allows us to extract values of thresholds
139
140 z(x) =
141  let
142      zsub(1,<n>^ns) = n
143      zsub(x,<n>^ns) = zsub(x-1,ns)
144  within
145      zsub(x,Thresholds)
146
147 -- sumz is the sum of all thresholds
148
149 sumz =
150  let
151      sumzsub(total,1) = total + z(1)
152      sumzsub(total,x) = sumzsub(total + z(x),x-1)
153  within
154      sumzsub(0,k)
155
156
157 -- ### STATES AND TRANSITIONS ###
158
159 States = {0..13}
160
161 -- we need the LTSs of N_B({0..B+m}), ... , N_(B+m)({0..B+m}) and we
       also use N_(B+m+1)({0..B+m}) to properly distinguish between beta-
       and gamma-events
162
163 Transitions_(x_) =
164  if x_ == 0 then {(0,init,1),(1,exchangeMutex.1.0,2),
165                   (1,mutexDown.0,2),(1,exchangeMutex.0.0,2),
166                   (2,fullSlotDown.0,3),(3,get.0.1,4),
167                   (3,get.0.0,5),(4,output.0.1,6),
168                   (5,output.0.0,6),(6,exchangeMutex.0.0,0),
169                   (6,mutexUp.0,1),(6,exchangeMutex.0.1,1)} else
170  if x_ == 1 then {(7,init,8),(8,exchangeMutex.1.1,9),
171                   (8,mutexDown.1,9),(8,exchangeMutex.0.1,9),
172                   (9,fullSlotDown.1,10),(10,get.1.1,11),
173                   (10,get.1.0,12),(11,output.1.1,13),
```

```
174                     (12,output.1.0,13),(13,exchangeMutex.1.1,8),
175                     (13,exchangeMutex.1.0,8),(13,mutexUp.1,8)} else
176   if x_ == 2 then {(14,init,15),(15,exchangeMutex.1.2,16),
177                     (15,mutexDown.2,16),(15,exchangeMutex.0.2,16),
178                     (16,fullSlotDown.2,17),(17,get.2.0,18),
179                     (17,get.2.1,19),(18,output.2.0,20),(19,output.2.1,20),
180                     (20,mutexUp.2,15),(20,exchangeMutex.2.0,15),
181                     (20,exchangeMutex.2.1,15)} else {} -- This set of
                            transitions is only used for alphabet partitioning.
                             Source and target states of these transitions do
                            not have to be included in the equivalence classes,
                             below.
182
183  initState(1) = 0 -- this state needs to be in the first equivalence
         class
184  initState(2) = 7 -- this state needs to be in the first equivalence
         class
185
186  AllTransitions = Union({Transitions(i) | i <- {B..(B+m)}})
187
188  labels(transitions) = {e | (s,e,s') <- transitions}
189
190  TransitionLabels = labels(AllTransitions)
191
192
193  -- ### EQUIVALENCE CLASSES ###
194
195  -- Define equivalence classes by providing a sequence of sets of related
         states, e.g. <{0,1,4,5},{2,6},{3,7}>
196  -- Source and target states of Transitions(3) do not have to be included
         in the equivalence classes.
197
198  E = <ENTER A SEQUENCE OF SETS OF RELATED STATES HERE>
199
200  k = length(E)
201
202  initEc = 1
203
204
205  -- ### ALPHABET PARTITIONING ###
206
207  -- tests if state st is a member of the ec-th equivalence class
208
209  ecmember(st,ec) =
210   let
211       ecmembersub(st,1,<sts>^stss) = member(st,sts)
212       ecmembersub(st,ec,<sts>^stss) = ecmembersub(st,ec-1,stss)
213   within
214       ecmembersub(st,ec,E)
215
216  -- count the number of alphabets a particular event occurs in to decide
         if it is a alpha-, beta- or gamma-event
217
218  countAs(e) =
219   let
```

```
220       countAssub(e,0) = if member(e,labels(Transitions(B))) then 1 else 0
221       countAssub(e,i) = if member(e,labels(Transitions(B+i))) then 1 +
              countAssub(e,i-1) else countAssub(e,i-1)
222   within
223       countAssub(e,m+1)
224
225
226   -- ### ALPHA-EVENTS AND TRANSITIONS ###
227
228   AlphaEvents = {e | e <- TransitionLabels, countAs(e) == 1 or e == TAU_}
229
230   AlphaTransitions = {(s0,e,s1) | (s0,e,s1) <- AllTransitions,
231                                   member(e,AlphaEvents)}
232
233   IncomingAlpha(ec) = {(s0,e,s1) | (s0,e,s1) <- AlphaTransitions,
234                                    not ecmember(s0,ec)
235                                    and ecmember(s1,ec)}
236   SelfloopsAlpha(ec) = {(s0,e,s1) | (s0,e,s1) <- AlphaTransitions,
237                                     ecmember(s0,ec) and ecmember(s1,ec)}
238   OutgoingAlpha(ec) = {(s0,e,s1) | (s0,e,s1) <- AlphaTransitions,
239                                    ecmember(s0,ec)
240                                    and not ecmember(s1,ec)}
241
242
243   -- ### BETA-EVENTS AND TRANSITIONS ###
244
245   BetaEvents = {e | e <- TransitionLabels, countAs(e) == 2 and e != TAU_}
246
247   BetaTransitions = {(s0,e,s1) | (s0,e,s1) <- AllTransitions,
248                                  member(e,BetaEvents)}
249
250   BetaEventsOf(ec) = {e | (s0,e,s1) <- BetaTransitions,
251                           ecmember(s0,ec) or ecmember(s1,ec)}
252
253   BetaTransitionPairs(e,ec) = -- Given a beta-event e and an equivalence
          class ec, BetaTransitionPairs(e,ec) returns a set of 4-tuples with 2
          source states st1 and st2 and 2 target states st1' and st2' such that
           st1 and st2 can both perform e to reach st1' and st2', respectively,
           and at least one of st1, st1', st2, st2' is in equivalence class ec
254    let
255     whichLTS(st,l,st') = pick({i | i <- {B..(B+m)},
256                                    member((st,l,st'),Transitions(i))})
257    within
258     {(st1,st1',st2,st2') | (st1,e1,st1') <- BetaTransitions,
259                            (st2,e2,st2') <- BetaTransitions,
260                            e1 == e and e2 == e,
261                            ecmember(st1,ec) or ecmember(st1',ec)
262                            or ecmember(st2,ec) or ecmember(st2',ec),
263                            whichLTS(st1,e1,st1') != whichLTS(st2,e2,st2'),
264                            st2 >= st1} -- (st1, st1', st2, st2') is the
                                same as (st2, st2', st1, st1') for our
                                purposes, so we only take one of these
265
266
267   -- ### GAMMA-EVENTS AND TRANSITIONS ###
```

```
268
269 GammaEvents = diff(TransitionLabels,union(AlphaEvents,BetaEvents))
270
271 GammaTransitions = {(s0,e,s1) | (s0,e,s1) <- AllTransitions,
272                                  member(e,GammaEvents)}
273
274 GenerateDoGammaTransitions =
275  let
276   GammaSeq = makeSeq(GammaTransitions)
277  within
278   {(e,<(s0,s1) | (s0,e',s1) <- GammaSeq, e == e'>) | e <- GammaEvents}
279
280 lex((s0,e,s1),(s0',e',s1')) = s0' < s0 or (s0' == s0 and s1' <= s1)
281
282 stateMin(S) = { x | x <- S, { y | y <- S, lex(x,y)} == {x}}
283
284 makeSeq({}) = <>
285 makeSeq(S) =
286  let
287      first = pick(stateMin(S))
288  within
289      <first>^makeSeq(diff(S,{first}))
290
291 getSeq(e) = pick({seq | (e',seq) <- GenerateDoGammaTransitions,
292                         e == e'})
293
294 allCounts(<>) = {<>}
295 allCounts(<(s0,s1)>^ps) =
296  {<(s0,s1,c)>^ts | ts <- allCounts(ps), c <- {0..max(max(sumz,z(1)),1)}}
297
298
299 -- ### SINGLE COUNTER ###
300
301 -- a single counter, corresponding to equivalence class ec and with
       value c
302
303 Counter(ec, c) =
304  let
305      compare(st) = if ecmember(st,ec) then 1 else 0
306      ups(st1',st2') = compare(st1') + compare(st2')
307      downs(st1,st2) = compare(st1) + compare(st2)
308      p(st1,st2) = compare(st1) + compare(st2)
309      sumActual(<>) = 0
310      sumActual(<(s0,s1,c)>^ps) = c + sumActual(ps)
311      v(ec,<>) = 0 -- v specifies an actual number of processes that a
           counter represents
312      v(ec,<(s0,s1,c)>^ps) = if ecmember(s0,ec) then c + v(ec,ps)
313                                                else v(ec,ps)
314      firstState(ec,<>) = false
315      firstState(ec,<(s0,s1,c)>^ps) = if ecmember(s0,ec)
316                                      then true
317                                      else firstState(ec,ps)
318      sumSecond(ec,<>) = 0
319      sumSecond(ec,<(s0,s1,c)>^ps) = if ecmember(s1,ec)
320                                      then c + sumSecond(ec,ps)
```

```
321                                          else sumSecond(ec,ps)
322       sumzTargets(ec,<>) = 0
323       sumzTargets(ec,<(s0,s1,c)>^ps) =
324            if ecmember(s0,ec) then z(whichEc(s1)) + sumzTargets(ec,ps)
325                               else sumzTargets(ec,ps)
326       whichEc(st) = pick({ i | i <- {1..k}, ecmember(st,i)})
327   within
328       (doAlpha?_:IncomingAlpha(ec) -> Counter(ec, min(c+1,z(ec))))
329       []
330       (c >= min(z(ec),1) & doAlpha?_:SelfloopsAlpha(ec) -> Counter(ec,c))
331       []
332       (c >= min(z(ec),1) & doAlpha?_:OutgoingAlpha(ec) ->
333          (if c == z(ec) then Counter(ec, c) |~| Counter(ec, c-1)
334                         else Counter(ec, c-1)))
335       []
336       ([] e: BetaEventsOf(ec),
337           (st1,st1',st2,st2'): BetaTransitionPairs(e,ec) @
338             c >= min(p(st1,st2),z(ec)) & doBeta.e.st1.st1'.st2.st2' ->
339               (if c == z(ec)
340                then |~|x:{min((c+ups(st1',st2')
341                                -downs(st1,st2)),z(ec))..z(ec)}
342                     @ Counter(ec,x)
343                else Counter(ec,min(c+ups(st1',st2')
344                                    -downs(st1,st2),z(ec)))
345              )
346       )
347       []
348       doGamma?e:GammaEvents?seq:{seq |
349           seq <- allCounts(getSeq(e)), sumActual(seq) > 0 and
350           (not firstState(ec,seq) or (firstState(ec,seq)
351            and v(ec,seq) <= sumzTargets(ec,seq) and
352            if c == z(ec) then v(ec,seq) >= c
353                          else v(ec,seq) == c))} ->
354         Counter(ec,min(sumSecond(ec,seq),z(ec)))
355
356 -- Defining alphabets of counters
357
358 ECAAlpha(ec) = {doAlpha.a | a <- Union({IncomingAlpha(ec),
359                                         SelfloopsAlpha(ec),
360                                         OutgoingAlpha(ec)})}
361 ECABeta(ec) = {doBeta.a.st1.st1'.st2.st2' | a <- BetaEventsOf(ec),
362                       (st1,st1',st2,st2') <- BetaTransitionPairs(a,ec)}
363 ECAGamma(ec) = {doGamma.a.seq | a <- GammaEvents,
364                                 seq <- allCounts(getSeq(a))}
365
366 ECA(ec) = Union({ECAAlpha(ec),ECABeta(ec),ECAGamma(ec)})
367
368
369 -- ### ABS CONSTRUCTION ###
370
371 Counters = || ec:{1..k} @ [ECA(ec)]
372                 Counter(ec, if ec == initEc then z(ec) else 0)
373
374 -- Renaming events to their original form and converting all TAU_ events
        to real tau's
```

```
375
376 CountersRenamed =
377   (((Counters
378       [[ doAlpha.(s0,e1,s1) <- e1 | (s0,e1,s1) <- AlphaTransitions ]])
379        \ {TAU_})
380         [[ doBeta.e2.st1.st1'.st2.st2' <- e2 |
381          e2 <- BetaEvents, ec <- {1..k},
382          (st1,st1',st2,st2') <- BetaTransitionPairs(e2,ec)]])
383           [[doGamma.e3.seq <- e3 | e3 <- GammaEvents,
384                                    seq <- allCounts(getSeq(e3)) ]]
385
386 -- Abstract model of all node processes
387
388 ABS =
389   ((|| i: {0..(B-1)} @ [A(i,{0..(B+m)})] Node(i,{0..(B+m)}))
390   [Union({A(i,{0..(B+m)}) | i <- {0..(B-1)}})
391    ||
392    Union({A(i,{0..(B+m)}) | i <- {B..(B+m)}})] CountersRenamed)
393   [[ mutexDown.x1 <- mutexDown.phi(x1) | x1 <- {0..(B+m)} ]]
394   [[ mutexUp.x1 <- mutexUp.phi(x1) | x1 <- {0..(B+m)} ]]
395   [[ fullSlotDown.x1 <- fullSlotDown.phi(x1) | x1 <- {0..(B+m)} ]]
396   [[ get.x2.x1 <- get.phi(x2).x1 | x2 <- {0..(B+m)}, x1 <- Data ]]
397   [[ exchangeMutex.x2.x1 <- exchangeMutex.phi(x2).phi(x1) |
398       x2 <- {0..(B+m)}, x1 <- {0..(B+m)} ]]
399   [[ output.x2.x1 <- output.phi(x2).x1 | x2 <- {0..(B+m)},
400                                          x1 <- Data ]]
401
402
403 -- ### ASSERTION ###
404
405 assert Spec({0..B}) [T= C(ABS,{0..B})
```