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.
The Parameterised Model Checking Problem asks whether an implementation $\text{Impl}(t)$ satisfies a specification $\text{Spec}(t)$ 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 $\langle c_1, \ldots, c_k \rangle$ 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 $\phi$ that maps all (sufficiently large) instantiations $T$ of the parameter to some fixed type $\hat{T}$ and allows us to deduce that if $\text{Spec}(\hat{T})$ is refined by $\phi(\text{Impl}(T))$, then $\text{Spec}(T)$ is refined by $\text{Impl}(T)$. We can then combine this with our extended counter abstraction techniques and conclude that if the abstract model satisfies $\text{Spec}(\hat{T})$, 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

1 **Introduction**  
1.1 Motivation ............................................. 1  
1.2 Concurrent reactive systems .......................... 2  
1.3 Theorem proving and model checking ................. 3  
1.4 Parameterised Model Checking Problem ............... 4  
1.5 Introduction to CSP .................................... 5  
1.5.1 Syntax ............................................ 6  
1.5.2 Denotational models and refinement ............... 9  
1.5.3 Tool support ..................................... 11  
1.6 Global assumptions ................................... 12  
1.7 Aims and contributions ................................. 13  
1.8 Thesis overview ..................................... 16  

2 **Related work** .......................................... 18  
2.1 Predicate abstraction .................................. 20  
2.2 Counter abstraction ................................... 23  
2.3 Data independence .................................... 25  
2.3.1 Data independence and CSP ....................... 26  
2.4 Induction .............................................. 27  
2.5 Counterexample-guided abstraction refinement ....... 28  
2.6 Conclusions .......................................... 30  

3 **Counter abstraction of systems with no identity awareness** .......... 32  
3.1 Decidability results .................................... 35  
3.2 Counter abstraction with unbounded counters ........ 42  
3.2.1 Constructing the counter state machine .......... 42  
3.2.2 Denotational equivalence results .................. 44  
3.3 Counter abstraction with finite thresholds ............ 45  
3.3.1 Constructing the counter state machine .......... 45  
3.3.2 Refinement results ................................ 49  
3.3.3 Threshold function cull ........................... 55  
3.4 Conclusions .......................................... 58
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.
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 \( f : I \rightarrow O \) 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 environ-
ment 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. Verifi-
cation 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 verifi-
cation 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, BCMID90, 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, and Grumberg [CES81, 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 \( P \sqsubseteq Q \) 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 \( \{1 \ldots 11\} \). 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 \( \Sigma \) to mean the alphabet of that process. A special, internal event, denoted by \( \tau \), is never included in any alphabet, so for convenience we write \( \Sigma^* \) to mean \( \Sigma \cup \{ \tau \} \). We also write \( \Sigma^* \) to mean the set of all finite sequences (possibly of zero length) of events from \( \Sigma \).

\(^2\)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
  α → Proc  prefix
  Proc □ Proc  external choice
  □ i ∈ I • Proc(i)  replicated external choice
  Proc ▷ Proc  internal choice
  ▷ i ∈ I • Proc(i)  replicated internal choice
  Proc ⊳ Proc  sliding choice (timeout)
  if b then Proc else Proc  conditional choice
  b & Proc  positive conditional choice
  Proc \ X  hiding
  Proc[[b/α]]  renaming by substitution
  Proc[[R]]  relational renaming
  Proc X | Y Proc  alphabetised parallel
  || i ∈ I • [A(i)] Proc(i)  replicated alphabetised parallel
  Proc X || Proc \ X  generalised parallel
  || X i ∈ I • Proc(i)  replicated generalised parallel
  Proc ||| Proc  interleaving
  ||| i ∈ I • 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 \( \alpha \) describes, and then subsequently behaves like \( P \). The construct \( \alpha \) is an expression of the form \( c\{i\}x_1:X_1 \cdots \{i\}x_k:X_k \), where

- \( c \) is a channel name,
- \( \{i\} \in \{\$, ?, \!\} \) is an input/output symbol,
- if \( \{i\} \in \{\$, ?\} \), then \( x_i \) is an input variable, otherwise it is an output value, and
- if \( \{i\} \in \{\$, ?\} \), then \( X_i \) is a type parameter or type of input, otherwise it is \( \text{null} \).

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 \( \text{in}\{\varepsilon\}:\{0, 1\}?y:2, 3!4 \rightarrow \text{out}(x + y) \rightarrow \text{STOP} \) 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 \( \text{in}.v.w.4 \) for \( w \in \{2, 3\} \), and binds the variable \( y \) to the value \( w \); it then performs the event \( \text{out}.(v + w) \) and deadlock. For constructs where \( \{i\} = \! \) 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 \( \text{null} \), we omit it in practice, e.g. we write \( c!v \) instead of \( c!v:\text{null} \). 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 \( P \boxplus Q \) 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 \( \boxplus \) operator is commutative and associative, so we can define an external choice between more than two processes. To do so, we use \( \boxplus \), the replicated version of the operator: \( \boxplus i \in \mathcal{I} \bullet P(i) \) is an external choice between processes \( P(i) \) for each \( i \) in some finite, non-empty indexing set \( \mathcal{I} \). We write \( ?a:A \rightarrow P(a) \) as syntactic sugar for \( \boxplus a \in A \bullet a \rightarrow P(a) \).

\( P \boxplus Q \) 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, \( \boxplus \) is commutative and associative, so its replicated version, \( \propto \), exists: \( \propto i \in \mathcal{I} \bullet P(i) \) 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

\( \text{Standard CSP commonly also uses the } . \text{ symbol, but this is only syntactic sugar and can always be replaced by one of } \$, ?, \! \).
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 \( \text{True} \), then this process behaves like \( P \); otherwise it behaves like \( Q \). The process \( b \& P \) is syntactic sugar for if \( b \) then \( P \) else \( \text{STOP} \), 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, \( \tau \), 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, \( P[ R ] \) is a process that for every pair of visible events \( a \) and \( b \) such that \( a R b \), 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 \( P X \parallel Y Q \) 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: \( \parallel i \in \mathcal{I} \bullet [A(i)] \) \( P(i) \) 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 \( P \parallel X Q \) 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 \( P \parallel X \cap Y Q = P X \parallel Y Q \), 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 \( \parallel \); the process \( \parallel i \in \mathcal{I} \bullet P(i) \) 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, \( P \parallel \{ \} Q \) 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 \( P \parallel Q \)). The replicated version of the interleaving operator, \( ||| \), 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 $T$**

In this model, each process $P$ is described by a set of finite traces (written $\text{traces}(P)$), 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*:

$$P \sqsubseteq_T Q \iff \text{traces}(Q) \subseteq \text{traces}(P).$$
If $P \sqsubseteq_T Q$ and $Q \sqsubseteq_T P$, then we say that $P$ and $Q$ are *traces equivalent*, denoted $P \equiv_T Q$.

**Example 1.5.1.** Let

$P = a \rightarrow a \rightarrow P \square b \rightarrow P$

and

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

Then

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

and

$\text{traces}(Q_1) = \{\langle \rangle\}$,

$\text{traces}(Q_2) = \{\langle \rangle, \langle a \rangle\}$,

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

Hence, $P \sqsubseteq_T Q_1$ and $P \sqsubseteq_T Q_2$, but $P \not\sqsubseteq_T Q_3$.

**End of example.**

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

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

In addition, if $tr$ is a trace of $P$, then $P/\text{tr}$ (pronounced “$P$ after $tr$”) is the process that $P$ becomes after it performs $tr$. So, in particular,

$\text{traces}(P/\text{tr}) = \{\text{tr}' \mid \text{tr}^-\text{tr}' \in \text{traces}(P)\}$.

**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 $\text{failures}(P)$). A failure is a pair $(tr, X)$, where $tr \in \text{traces}(P)$ and $X \subseteq \Sigma$, and represents the behaviour where $P$ can perform trace $tr$ to reach a stable state $P'$ (i.e. $\tau$ 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 $P' \text{ ref } X$.

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

$P \sqsubseteq_F Q \iff \text{traces}(Q) \subseteq \text{traces}(P) \land \text{failures}(Q) \subseteq \text{failures}(P)$.

If $P \sqsubseteq_F Q$ and $Q \sqsubseteq_F P$, then we say that $P$ and $Q$ are *stable failures equivalent*, denoted $P \equiv_F Q$. 

10
Example 1.5.2. Let

\[ P = a \rightarrow STOP \sqcup b \rightarrow STOP \]

and

\[ Q = a \rightarrow STOP \sqcap b \rightarrow STOP. \]

Then

\[ \text{traces}(P) = \text{traces}(Q) = \{ \langle \rangle, \langle a \rangle, \langle b \rangle \} \]

and

\[ \text{failures}(P) = \{ (\langle \rangle, \{ \} ), (\langle a \rangle, \{ a \} ), (\langle b \rangle, \{ a, b \} ), (\langle a \rangle, \{ b \} ), (\langle b \rangle, \{ a \} ), (\langle b \rangle, \{ a, b \} ) \} , \]

\[ \text{failures}(Q) = \{ (\langle \rangle, \{ \} ), (\langle a \rangle, \{ a \} ), (\langle a \rangle, \{ b \} ), (\langle a \rangle, \{ a, b \} ), (\langle a \rangle, \{ b \} ), (\langle b \rangle, \{ a \} ), (\langle b \rangle, \{ a, b \} ) \} , \]

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, \( Q \sqsubseteq_F P \), but \( P \not\sqsubseteq_F Q \). In addition, observe that \( P \sqsubseteq_T Q \) and \( Q \sqsubseteq_T P \), 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 \( \text{traces}(P) \) and \( \text{failures}(P) \)) 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 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 \( P \)

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\textsuperscript{6} [For99] allows one to automatically perform refinement checks. When a CSP\textsubscript{M} script with process definitions, say $P$ and $Q$, is loaded, FDR can automatically test for refinement $P \sqsubseteq M Q$ 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\textsuperscript{7} [LP07, LF08], ARC\textsuperscript{8} [PY96] and PAT\textsuperscript{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 $\text{Proc}(t)$, parameterised by a type $t$, satisfies this assumption if:

(i) all guards of conditional choices within $\text{Proc}(t)$ 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, $\text{Proc}(t)$ contains no name clashes between type $t$ nondeterministic-selection variables of one argument and free variables of another argument, e.g. $c$x:t -> STOP □ d.x -> STOP is not allowed,

(iii) constructs of $\text{Proc}(t)$ 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.

\textsuperscript{6}Available from http://www.fsel.com/fdr2_download.html.

\textsuperscript{7}Available from http://www.stups.uni-duesseldorf.de/ProB/index.php5/Download.

\textsuperscript{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.

\textsuperscript{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

\[
\text{if } P \lor P' \text{ then } Q \text{ else } R \equiv \text{if } P \text{ then } Q \text{ else } (\text{if } P' \text{ then } Q \text{ else } R), \\
\text{if } P \land P' \text{ then } Q \text{ else } R \equiv \text{if } P \text{ then } (\text{if } P' \text{ then } Q \text{ else } R) \text{ else } R.
\]

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

\[
in_1:x:t \rightarrow (out.x \rightarrow \text{STOP} \quad \square \quad in_2:x:t \rightarrow \text{STOP}).
\]

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 \( T = \{0 \ldots n - 1\} \). Our results and techniques extend to all set types \( T \), isomorphic to the set of first \( n \) natural numbers via simple bijections from \( \{0 \ldots n - 1\} \) 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 $Spec(T) \subseteq Impl(T)$ 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

$$Impl(t) = C \left[ \bigg| \bigg| \big| i \in t \cdot N \bigg| \bigg| A_{Sync} \right]$$
where:

- \( N \) models a single, finite-state node process,
- \( A_{\text{Sync}} \) 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 \( \text{Abstr}_z \), based on a threshold function \( z \), that is \( t \)-independent and is such that, by construction,

\[
\text{Abstr}_z \subseteq \text{Impl}(T)
\]

for all sufficiently large \( T \). Then, by testing whether

\[
\text{Spec} \sqsubseteq \text{Abstr}_z
\]

(e.g. using FDR) we can deduce that \( \text{Spec} \sqsubseteq \text{Impl}(T) \) for all sufficiently large \( T \), where \( \text{Spec} \) 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

\[
\text{Impl}(t) = C_t \left[ \prod_{i \in t} N_i(t) \cdot A(i, t) \right],
\]

where:

- \( N_i(t) \) 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 \( N_i(t) \) 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 \( \text{Abstr}_z \), based on a threshold function \( z \), that is \( t \)-independent and is such that, by construction,

\[
\text{Abstr}_z \subseteq_T \phi(\text{Impl}(T))
\]
for all sufficiently large $T$, where $\phi$ is a function that maps each such $T$ to some fixed type $\hat{T}$. Then, we can use FDR to check whether

$$Spec(\hat{T}) \subseteq_T \text{Abstr}_z,$$

and hence deduce that

$$Spec(\hat{T}) \subseteq_T \phi(\text{Impl}(T)).$$

However, this, on its own, is not enough to allow us to infer that $Spec(T) \subseteq_T \text{Impl}(T)$ 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 $\phi$ maps all sufficiently large instantiations of the distinguished type and such that

$$Spec(\hat{T}) \subseteq \phi(\text{Impl}(T)) \Rightarrow Spec(T) \subseteq \text{Impl}(T),$$

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 \textbf{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 \textbf{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 \textbf{SeqNorm} condition, paving the way to the main type reduction theorems for both the traces and stable failures model. The theorems establish a function $\phi$ 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-consumer's problem.

Finally, we conclude and discuss possible directions in which our work can be developed further in Chapter 8.
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 Kahlol 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 unboundedly 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 $\text{Abstr}$ 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 Sâidi 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 \( \gamma \) 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 \( i \in \{2 \ldots k - 1\} \) 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 Murϕ 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 Saıdı 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 $8 \times 8$ 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 $Impl = \text{King}(8, 1) \setminus \text{Moves}$, where

\[
\text{King}(x, y) = \begin{cases} 
  y > 1 & \text{down} \rightarrow \text{King}(x, y - 1) \\
  y < 6 & \text{up} \rightarrow \text{King}(x, y + 1) \\
  x > 2 & \text{left} \rightarrow \text{King}(x - 1, y) \\
  x < 8 & \text{right} \rightarrow \text{King}(x + 1, y) \\
  x = 1 \lor y \geq 7 \rightarrow \text{error} \rightarrow \text{STOP}
\end{cases}
\]

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

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

\[
\text{Spec} = \text{STOP}.
\]
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.

\[
\begin{align*}
P_1 &\equiv y = 1, \\
P_2 &\equiv y > 1 \land y < 6, \\
P_3 &\equiv y = 6, \\
P_4 &\equiv y \geq 7.
\end{align*}
\]

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

\[
\begin{align*}
\overline{\text{King}}(1) &= \text{up} \rightarrow \overline{\text{King}}(2), \\
\overline{\text{King}}(2) &= \text{up} \rightarrow (\overline{\text{King}}(2) \cap \overline{\text{King}}(3)) \\
&\quad \quad \square \\
\overline{\text{King}}(3) &= \text{down} \rightarrow \overline{\text{King}}(2), \\
\overline{\text{King}}(4) &= \text{error} \rightarrow \text{STOP}.
\end{align*}
\]

Then \( \overline{\text{Impl}} = \overline{\text{King}}(1) \setminus \text{Moves} \) 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 $\text{Spec} \subseteq_T \text{Impl}$ if and only if $\text{Spec} \subseteq_T \tilde{\text{Impl}}$. 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 interpret 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 $\text{Impl}(T) \parallel \text{Abstr}(\text{Impl}(t))$ and $\text{Impl}(T \cup \{i\}) \parallel \text{Abstr}(\text{Impl}(t))$ 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 $\text{Abstr}(\text{Impl}(t))$. 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 \( \text{Abstr} \) of an implementation \( \text{Impl} \) that is exact, i.e. such that \( \text{Impl} \) satisfies a given specification if and only if \( \text{Abstr} \) 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 \( \text{dom}(f) = t \), if \( \sigma \) 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 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

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

where

$$
Pipeline(t) = \begin{cases}
\text{input}^t d : t^? \text{ro}^? \{ \mathsf{yes, no} \} \to \text{use}^! d^! \mathsf{ro}^? \$^c : \{ \mathsf{yes, no} \} \\
\quad \rightarrow \begin{cases}
\text{if } c = \text{no} \text{ then } \text{output}^t d \to Pipeline(t) \\
\quad \text{else } \text{output}^t d^? : t \to Pipeline(t).
\end{cases}
\end{cases}
$$

Here, $\text{use}^! d^! \mathsf{ro}^? \$^c : \{ \mathsf{yes, no} \}$ 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 $c = \mathsf{yes}$) or not ($c = \mathsf{no}$).

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).

$$
Spec(t) = \begin{cases}
\text{input}^? d : t^! \mathsf{no} \rightarrow \text{output}^t d^? : t \to Spec(t) \\
\text{input}^? d : t^! \mathsf{yes} \rightarrow \text{output}^t d \rightarrow Spec(t).
\end{cases}
$$

26
Our verification problem is to check whether $Spec(T) \subseteq Impl(T)$ for all non-empty datatypes $T$. We can use FDR to perform the checks for single instances of $T$. However, if we test that $Spec(T_0) \subseteq Impl(T_0)$ 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

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

(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: $(0, n - 1)$, $(n - 1, n)$, $(n, \infty)$, 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 reduction [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 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:

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

(2.2)

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

(2.3)

$$Spec \sqsubseteq Inv(T)$$

(2.4)
We can then deduce that $\text{Spec} \subseteq \text{Impl}(T)$. 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. $\text{Inv}(i) = \text{Inv}(j)$ 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 $\text{Spec} \subseteq \text{Inv}$ 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 $\text{Spec}$ 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 $\text{Spec} \subseteq \text{Impl}(T)$ 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 $\text{Inv}$, 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, $\phi$, 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).
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 $\text{Impl}(T)$, consisting of $\#T$ similar processes, and a specification $\text{Impl}(T)$, is it true that $\text{Impl}(T)$ satisfies $\text{Spec}(T)$ 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

$$\text{Impl}(t) = C[Nodes(t)],$$

where

- $C[\cdot]$ is a CSP context, independent of $t$,
- $Nodes(t) = \bigparallel_{A_{\text{sync}}} i \in t \cdot \mathcal{N}$,
- $\mathcal{N}$ models a single, finite-state node process, and
- $A_{\text{sync}}$ 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.

\[ N = \text{Node(new)} \]
\[ \text{Node(new)} = \text{load} \rightarrow \text{Node(runnable)} \]
\[ \text{Node(runnable)} = \text{run} \rightarrow \text{Node(running)} \]
\[ \text{Node(running)} = \text{deschedule} \rightarrow \text{Node(runnable)} \]
\[ \quad \square \text{block} \rightarrow \text{Node(blocked)} \]
\[ \quad \square \text{terminate} \rightarrow \text{STOP} \]
\[ \text{Node(blocked)} = \text{interrupt} \rightarrow \text{Node(runnable)} \]

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

\[ \text{Core(idle)} = \text{run} \rightarrow \text{Core(busy)} \]
\[ \text{Core(busy)} = \text{deschedule} \rightarrow \text{Core(idle)} \]
\[ \quad \square \text{block} \rightarrow \text{Core(idle)} \]
\[ \quad \square \text{terminate} \rightarrow \text{Core(idle)} \]

Let \text{cores} be the number of processors present in the system (later on, we verify the system for a single value of \text{cores}). The task of the scheduler is to divide CPU time between the different processes. It consists of \text{cores} interleaved processes \text{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.

\[ \text{Scheduler} = \big|| | i \in \{1 \ldots \text{cores}\} \bullet \text{Core}(idle) \]

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.

\[ \text{Nodes}(t) = \big|| \{\text{load}\} i \in t \bullet N \]

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.

\[ \text{Impl}(t) = (\text{Nodes}(t) \parallel \text{Scheduler}) \]

\[ \text{Alpha} = \{\text{run}, \text{deschedule}, \text{block}, \text{terminate}\} \]

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

\[ \text{Spec} = \text{load} \rightarrow \text{Spec}'(0) \]

\[ \text{Spec}'(x) = x < \text{cores} \& (\text{run} \rightarrow \text{Spec}'(x + 1) \cap \text{STOP}) \]

\[ \Box x > 0 \& \text{stopRun} \rightarrow \text{Spec}'(x - 1) \]

\[ \Box (\text{interrupt} \rightarrow \text{Spec}'(x) \cap \text{STOP}) . \]

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

\[ \forall T \bullet \text{Spec} \subseteq F \text{Impl}(T) . \]  

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 \( \mathcal{L} = (S, s_0, L, \rightarrow) \), where \( S \) is a set of states, \( s_0 \in S \) is an initial node, \( L \) is a set of labels, and \( \rightarrow \subseteq S \times L \times S \) 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:

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, $\text{Nodes}(T)$. 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 $\text{Nodes}(T)$. 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 $\text{Spec}$ in order to conclude that $\text{Impl}(T)$ refines $\text{Spec}$ 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

\footnote{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 $\text{Spec} \subseteq C \left[ \bigcap_{i \in T} N \right]$ for all $T$, where $C[\cdot]$, $N$ and $\text{Spec}$ 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 $T$,

$$Spec \sqsubseteq_T Impl(T) \iff \text{Chaos}(\Sigma \setminus \{fail\}) \sqsubseteq_T Impl(T) \parallel_{\Sigma \setminus \{fail\}} WDT.$$ 

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 $T$,

$$Spec \sqsubseteq_F Impl(T) \iff \left( \text{Chaos}(\Sigma \setminus \{fail\}) \sqsubseteq_T Impl(T) \parallel_{\Sigma \setminus \{fail\}} WDT \right) \land Impl(T) \parallel WDF \text{ is deadlock-free}.$$ 

**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 $Spec' = \text{Chaos}(\Sigma \setminus \{fail\})$, Lemma 3.1.5 implies that for all $T$,

$$Spec \sqsubseteq_T C \left[ \left\lfloor \left\lfloor i \in T \bullet N \right\rfloor \right\rfloor \right] \parallel_{\Sigma \setminus \{fail\}} WDT.$$ 

Let $st_1, \ldots, st_k$ be the states of $N$, where $st_1$ is the initial state. We simulate $C \left[ \left\lfloor \left\lfloor i \in T \bullet N \right\rfloor \right\rfloor \right] \parallel_{\Sigma \setminus \{fail\}} WDT$ for all non-empty and finite $T$ using a single $k$-dimensional VASS $\mathcal{V}(N, C, WDT)$ 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 $DF(\Sigma) \sqsubseteq_F P$, where $DF(A) = \bigcap a \in A \bullet a \rightarrow DF(A)$ is the most nondeterministic process with alphabet $A$. 

37
• We want to talk about the states of context \( C[\cdot] \). Formally, we mean these to be the states of \( C[\text{Run}(\Sigma \setminus \{\text{fail}\})] \), 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 \( C[\cdot] i \in T \cdot \mathcal{N} \) \( \Sigma\setminus\{\text{fail}\} \) \( WDT \), 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 \( (q_0, (0, \ldots, 0), (st_0', st_0'w)) \), where \( st_0' \) and \( st_0'w \) are the initial states of \( C[\text{Run}(\Sigma \setminus \{\text{fail}\})] \) 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, \( s_{t1} \) and then evolve a state corresponding the initial state of \( C[\cdot] i \in \{0 \ldots n - 1\} \cdot \mathcal{N} \) \( \Sigma\setminus\{\text{fail}\} \) \( WDT \).

We have that \( C[\cdot] i \in T \cdot \mathcal{N} \) \( \Sigma\setminus\{\text{fail}\} \) \( WDT \) can perform some trace and reach a state in which the event \( \text{fail} \) 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 \( \text{fail} \). This means that the problem of deciding if \( \text{Spec'} \models T C[\cdot] i \in T \cdot \mathcal{N} \) \( \Sigma\setminus\{\text{fail}\} \) \( WDT \) for all \( T \) is equivalent to coverability of all the states \( ((st_c, st_w), (0, \ldots, 0)) \) such that \( st_w \) can communicate \( \text{fail} \) (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 \( T \),

\[
\text{Spec'} \models_T C[\cdot] i \in T \cdot \mathcal{N} \iff \begin{cases} 
\text{Spec'} \models_T C[\cdot] i \in T \cdot \mathcal{N} \models WDT \land \\
C[\cdot] i \in T \cdot \mathcal{N} \models WDF \text{ is deadlock-free}
\end{cases}
\]

(3.3)

where \( \text{Spec'} \) is as before. We have already shown that verifying if \( \text{Spec'} \models_T \text{Impl}(T) \models WDT \) for all \( T \) is decidable. We can now simulate
Since the deadlock-freedom problem is decidable over vector addition systems\(^4\) [EN94], the problem of testing if \( C \models i \in T \cdot \mathcal{N} \parallel WDF \) 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

\[
C \mid i \in A_{\text{Sync}} \cdot N,
\]

where \( C[;] \) and \( \mathcal{N} \) are as before and \( A_{\text{Sync}} \) 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 \( \text{Spec} \models i \in T \cdot \mathcal{N} \) for all \( T \), where \( \text{Spec} \) 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 \( i \in T \cdot \mathcal{N} \).

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,
• $\Gamma$ is a finite alphabet,
• $\delta : (Q \times \Gamma \times \{True, False\}^2) \rightarrow (Q \times \{*, +, -\} \times \{1, 2\})$ 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 $\Gamma$, 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

$$Cell(i) = up.i \rightarrow Cell'(i)$$

$$\square$$

$$zero.i \rightarrow Cell(i)$$

$$Cell'(i) = down.i \rightarrow Cell(i).$$

Then,

$$Counter(i, t) = \{id \in t \cdot Cell(i) \}$$

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 $M$ in a way that the event $halt$ is communicated if and only if $M$ is in an accepting state. Then, for all $T$,

$$\exists \text{tr} \bullet Ctrl \parallel \{\text{zero, up, down}\} \parallel \{\text{zero}\} \xrightarrow{tr^\rightarrow(halt)} (Counter(1, T) \parallel Counter(2, T))$$

$$\Leftrightarrow M \text{ halts with both counters never exceeding } \# T.$$  \hfill (3.4)

If we let $N = Cell(1) \parallel Cell(2)$,

then for all $T$,

$$Counter(1, T) \parallel Counter(2, T) \equiv_T \{i \in T \bullet N\}$$

Hence, (3.4) implies that for all $T$,

$$\exists \text{tr} \bullet Ctrl \parallel \{\text{zero, up, down}\} \parallel \{\text{zero}\} \xrightarrow{tr^\rightarrow(halt)} \Leftrightarrow M \text{ halts with both counters never exceeding } \# T.$$  \hfill (3.5)

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

$$Spec \not\subseteq_T Ctrl \{\text{zero, up, down}\} \{\text{zero}\} \Leftrightarrow M \text{ halts with both counters never exceeding } \# T.$$  \hfill (3.5)

This means that

$$\forall T \bullet Spec \subseteq_T Ctrl \{\text{zero, up, down}\} \{\text{zero}\} \Leftrightarrow M \text{ does not halt.}$$  \hfill (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

$$Spec \subseteq_T Ctrl \{\text{zero, up, down}\} \{\text{zero}\} \Leftrightarrow Spec' \subseteq_T \{\text{zero}\} \Leftrightarrow i \in T \bullet N.$$  \hfill (3.5)

This, combined with (3.5) implies that

$$\forall T \bullet Spec' \subseteq_T \{\text{zero}\} \Leftrightarrow i \in T \bullet N.$$  \hfill (3.5)

Since $M$ 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

$$\{\text{zero}\} \Leftrightarrow i \in T \bullet N$$  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 $N$ be represented using an LTS $(S, s_0, \Sigma^r, \rightarrow)$. 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 $f : T \rightarrow \{1 \ldots k\}$ 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 $\parallel A_{Sync} i \in T \cdot st_{f(i)}$ (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 $N$, generates a counter state machine $\zeta^T_\infty(N) = (A_\infty(T), a_\infty(T), \Sigma^r_\infty, \rightarrow_\infty(T))$ of an abstract model $\zeta^T_\infty(N)$ that is bisimilar to $Nodes(T)$. Whenever $T$ is clear from context, we omit it from the right-hand side and simply write $(A_\infty, a_\infty, \Sigma^r_\infty, \rightarrow_\infty)$.

Let $T$ be fixed. The states in $A_\infty$ are $k$-tuples of non-negative integer counters $(c_1, \ldots, c_k)$ such that $\Sigma^k_{j=1} c_j = \#T$. This means that $(0, \ldots, 0)$ is not a state in $A_\infty$, 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 $\{1 \ldots k\}$ and where each $c_i$ counts the number of nodes in state $st_i$, i.e.

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

The initial state $a_\infty$ 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, $\Sigma^r_\infty$, it is enough to observe that the transformed system can perform only those events that a single node process can. Hence, we let

$\Sigma^r_\infty = \Sigma^r$.

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

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

When $a$ is a private visible event of a node process or a $\tau$, 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 $\{1 \ldots k\}$ 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 $\Sigma^* \setminus A_{Sync}$,

\[
(c_1, \ldots, c_k) \xrightarrow{a} (c'_1, \ldots, c'_k) \iff \exists i, j \in \{1 \ldots k\} \cdot st_i \xrightarrow{a} st_j \land c_i \geq 1 \\
\land (i \neq j \land c'_i = c_i - 1 \land c'_j = c_j + 1 \\
\land (\forall h \in \{1 \ldots k\} \setminus \{i, j\} \cdot c'_h = c_h) \\
\lor i = j \land \forall h \in \{1 \ldots k\} \cdot c'_h = c_h). \tag{3.6}
\]

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 $\{1 \ldots k\}$, 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 $\{1 \ldots c_i\}$, and

• for every $i$ in $\{1 \ldots k\}$, precisely $c'_i$ of all the target states are equal to $st_i$.

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

\[
(c_1, \ldots, c_k) \xrightarrow{e} (c'_1, \ldots, c'_k) \iff \exists tgt : \mathbb{N} \times \mathbb{N} \rightarrow S \cdot \\
(\forall i \in \{1 \ldots k\}, j \in \{1 \ldots c_i\} \cdot st_i \xrightarrow{e} tgt^i_j) \\
\land \forall h \in \{1 \ldots k\} \cdot c'_h = \#\{(i, j) \mid i \in \{1 \ldots k\} \land j \in \{1 \ldots c_i\} \land tgt^i_j = st_h\}. \tag{3.7}
\]

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

**Example 3.2.1.** Recall the $N$ process syntax we defined in Example 3.0.1. For any given $T$ we can use the method described above to construct $\zeta^T_{\infty}(N)$ using the LTS of $N$ (Figure 3.1). Figure 3.2 shows $\zeta^{\{1,2\}}_{\infty}(Node)$, where the states of $N$ 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.*
3.2.2 Denotational equivalence results

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

- the $\parallel$ operator is commutative and associative, so the behaviour of $\text{Nodes}(T)$ 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 $t$, $\text{Nodes}(T)$ and $\zeta_T^\infty(\mathcal{N})$ are strongly bisimilar.

**Proof sketch:** By showing that

$$B \triangleq \left\{ \left\{ \parallel j \in T \bullet \text{st}_{f(j)}, (c_1, \ldots, c_k) \right\} \mid f : T \to \{1 \ldots k\} \right. \left. \wedge \forall i \in \{1 \ldots k\} \bullet c_i = \#\{j \in T \mid f(j) = i\} \right\}$$

44
is a strong bisimulation relation between the states of $Nodes(T)$ and the states of $\zeta_T^\infty(N)$. The bisimilarity of $Nodes(T)$ and $\zeta_T^\infty(N)$ follows then immediately by observing that their initial states ($\parallel j \in T \bullet s_0$ and $(\# T, 0, \ldots, 0)$, respectively) are related under $B$.

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

- $\zeta_T^\infty(N) \equiv_T Nodes(T)$, and
- $\zeta_T^\infty(N) \equiv_F Nodes(T)$.

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

$$\zeta_z(N) = (A_z, a_z, \Sigma_z^r, \rightarrow_z),$$

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 $\{1 \ldots k\}$, $c_i$ is an integer between 0 and $z_i$, and

(ii) either there exists $i$ in $\{1 \ldots k\}$ 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 $\zeta^r_\infty(N)$, i.e.
\[ \Sigma^r_z = \Sigma^r_\infty = \Sigma^r. \]

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

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 $\Sigma^r_z \setminus A_{\mathsf{Sync}}$. 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 $\{1 \ldots k\}$, 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 $\Sigma^r_z \setminus A_{\mathsf{Sync}}$ we define

\[ (c_1, \ldots, c_k) \overset{a}{\longrightarrow}_z (c'_1, \ldots, c'_k) \]

\[ \iff \exists i, j \in \{1 \ldots k\} \bullet \]

\[ st_i \overset{a}{\longrightarrow} st_j \land (c_i \geq 1 \lor c_i \geq z_i) \]

\[ \land ((i \neq j \land (c'_i = \max\{c_i - 1, 0\} \land c'_j = \min\{c_j + 1, z_j\} \land c_i = z_i = c'_i \land c'_j = \min\{c_j + 1, z_j\}) \]

\[ \land (\forall h \in \{1 \ldots k\} \setminus \{i, j\} \bullet c'_h = c_h)) \]

\[ \lor i = j \land \forall h \in \{1 \ldots k\} \bullet c'_h = c_h). \]

46
The following example illustrates the above definition and the rationale behind clause (ii) of Definition 3.3.1.

**Example 3.3.2.** Let $s_{t1}$, $s_{t2}$ and $s_{t3}$ be states of a node process such that

$$s_{t1} \xrightarrow{a} s_{t2} \quad \text{and} \quad s_{t2} \xrightarrow{b} s_{t3}.$$ 

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

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

However,

$$(2,0,0) \xrightarrow{a} z \xrightarrow{a} (0,1,0) \not\xrightarrow{b} z (0,0,1),$$

as, according to Definition 3.3.1, $(0,0,1)$ is not a state in $A_z$. Since $z_1 = 1$, 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 $A_{\text{Sync}}$. We want $(c_1, \ldots, c_k)$ and $(c'_1, \ldots, c'_k)$ to be related by $e \xrightarrow{\forall} z$ 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 $\{1 \ldots k\}$, a number of processes indicated by $c_i$ can move from state $s_{t_i}$ and a number of processes indicated by $c'_i$ can move into state $s_{t_i}$. Recall that if for some $i$ in $\{1 \ldots k\}$, the counter value $c_i$ is equal to its threshold $z_i$, then the actual number of processes in the state $s_{t_i}$, 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 $v : \{1 \ldots k\} \rightarrow \mathbb{N}$ such that

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

(ii) if, given $i$ in $\{1 \ldots k\}$, $c_i$ 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 $\{1 \ldots k\}$, precisely $v_i$ node processes within the model are in state $s_{t_i}$, 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 $s_{t_i}$ and if $c'_i = z_i$, then at least $c'_i$ of all the target states are equal to $s_{t_i}$. 

47
Formally, for every \( a \) in \( A_{	ext{sync}} \), we define

\[
(c_1, \ldots, c_k) \xrightarrow{a} v (c'_1, \ldots, c'_k)
\]

\[ \Leftrightarrow \exists v : \{1 \ldots k\} \rightarrow \mathbb{N} \bullet \]

\[ \exists tgt : \mathbb{N} \times \mathbb{N} \rightarrow S \bullet \]

\[ \sum_{i=1}^{k} v_i \geq \max\{z_1,1\} \]

\[ \land (\forall i \in \{1 \ldots k\} \bullet v_i = c_i < z_i \lor c_i = z_i \leq v_i) \quad \text{(3.10)} \]

\[ \land (\forall i \in \{1 \ldots k\}, j \in \{1 \ldots v_i\} \bullet st_i \xrightarrow{a} tgt_i) \]

\[ \land (\forall l \in \{1 \ldots k\} \bullet \]

\[ c'_i = \min\{z_l, \#(i,j) \mid i \in \{1 \ldots k\} \land j \in \{1 \ldots v_i\} \]

\[ \land tgt_i = st_l\} \} \].

Observe that the two parts of the definition of \( \rightarrow_z \) 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 \( i \in \{1 \ldots k\} \), \( \max\{\sum_{j \in \text{targets}(i)} c'_j, z_i, z_1, 1\} \) is an upper bound for the values of \( v_i \) that need to be considered in the body of (3.10), where \( \text{targets}(i) = \{j \in \{1 \ldots k\} \mid st_i \xrightarrow{e} st_j\} \).

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

\[ v_I > \max\{\sum_{j \in \text{targets}(I)} c'_j, z_I, z_1, 1\} \].

(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 \( \{1 \ldots k\} \), \( w_i = v_i - 1 \) 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 \( v_I > z_I \) 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 \( v_I > \sum_{j \in \text{targets}(I)} c'_j \), 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 \( (0, 1, 0, 1, 0) \) indicates
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 $\zeta^T_\infty(N)$ and $\zeta_z(N)$. 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 $\{1 \ldots k\}$ to $\mathbb{N}$. Suppose $s_1 = (c_1, \ldots, c_k)$ and $s_2 = (d_1, \ldots, d_k)$ are two states in $A_{\infty}^z(T)$ and $A_z$, respectively. Then $s_1$ and $s_2$ are $z$-corresponding, written $cor_z(s_1, s_2)$, if\(^6\)

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

The following lemma says that, given a threshold function $z$, two states in $A_z$ are related by $\xrightarrow{a}_z$ if and only if there exist two $z$-corresponding states in $A_{\infty}^z(T)$, 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 $\{1 \ldots k\}$ to $\mathbb{N}$. Let $d, d'$ be two states in $A_z$. Then, for all events $a$ in $\Sigma^z$ we have that

$$d \xrightarrow{a}_z d' \iff \exists T \bullet \#T \geq z_1 \land \exists c, c' \in A_{\infty}^z(T) \bullet cor_z(c, d) \land cor_z(c', d') \land c \xrightarrow{a}_{\infty}(T) c'.$$

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

$$d \xrightarrow{a}_z d' \iff \exists T \bullet \#T \geq \max\{z_1, 1\} \land \exists c, c' \in A_{\infty}^z(T) \bullet cor_z(c, d) \land cor_z(c', d') \land c \xrightarrow{a}_{\infty}(T) c'.$$ (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 $\{1 \ldots k\}$ to $\mathbb{N}$. Let $T$ be an instantiation of type $t$ such that $\#T \geq z_1$. Then, for all events $a$ in $\Sigma^z = \Sigma^\tau$, $$(c_1, \ldots, c_k) \xrightarrow{a}_{\infty}(T) (c'_1, \ldots, c'_k),$$

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 $\{1 \ldots k\}$ to $\mathbb{N}$. Let $T$ be an instantiation of type $t$ such that $\#T \geq z_1$. Then

$$\xi_z(N) \sqsubseteq_T \xi^\tau_z(N).$$

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

\[(\# T \cap z_1, 0, \ldots, 0) \xrightarrow{tr} (c'_1 \cap z_1, \ldots, c'_k \cap z_k).\]

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

Lemma 3.3.9. Let \(z\) be a function from \((1..k)\) to \(\mathbb{N}\) and let \(a\) be an event in \(\Sigma^z\).

Suppose that

(i) if \(a \notin A_{\text{Sync}}\), then for all \(i\) in \((1..k)\), if \(st_i \xrightarrow{a} \), then \(z_i \geq 1\),
(ii) if \(a \in A_{\text{Sync}}\), then for all \(i\) in \((1..k)\), \(z_i \geq 1\), and
(iii) for all \(i\) in \((1..k)\), \(c_i \cap z_i = d_i \cap z_i\).

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 \(A_{\text{Sync}}\).

**Case 1.** Suppose that \(a\) is in \(\Sigma^z \setminus A_{\text{Sync}}\).

Then, by (3.6), \(a\) is available in \((c_1, \ldots, c_k)\) if and only if there exists \(i\) in \((1..k)\) 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 \((1..k)\) 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 \(c_i \cap z_i = d_i \cap z_i\) (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 \(A_{\text{Sync}}\).

Then, by (3.7), \(a\) is available in \((c_1, \ldots, c_k)\) if and only if for every \(i\) in \((1..k)\), 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 \((1..k)\), if \(d_i > 0\), then \(a\) is available in \(st_i\). However, for all \(i\) in \((1..k)\), \(z_i \geq 1\) (by assumption (ii)) and \(c_i \cap z_i = d_i \cap z_i\) (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 \(\Sigma^z\). 

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

Then

$$
\zeta_z(N) \subseteq F \zeta_z^T(N).
$$

**Proof:** By Proposition 3.3.8 we have that $\text{traces}(\zeta_z^T(N)) \subseteq \text{traces}(\zeta_z(N))$.

Let $(tr, X) \in \text{failures}(\zeta_z^T(N))$. Then, there exist $c_1, \ldots, c_k$ such that

$$
\zeta_z^T(N) \xrightarrow{tr} (c_1, \ldots, c_k) \quad \text{and} \quad (c_1, \ldots, c_k) \text{ ref } X.
$$

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

$$
(c_1, \ldots, c_k) \not\xrightarrow{a} \infty.
$$

By Proposition 3.3.8 we have that

$$
\zeta_z(N) \xrightarrow{tr} (c_1 \cap z_1, \ldots, c_k \cap z_k).
$$

Let $c'_1, \ldots, c'_k$ be arbitrary integers such that $c_i \cap z_i = c'_i \cap z_i$ for all $i$ in $\{1 \ldots k\}$. If $a \in A_{Sync}$, then $A_{Sync} \neq \{\}$, so by assumption (ii), $z_i \geq 1$ for all $i$ in $\{1 \ldots k\}$ and if $a \not\in A_{Sync}$, then assumptions (i) and (ii) imply that for all $i$ in $\{1 \ldots k\}$, either $z_i \geq 1$ or $st_i = \text{STOP}$. Hence, from (3.13) we can infer using Lemma 3.3.9 that

$$
(c'_1, \ldots, c'_k) \not\xrightarrow{a} \infty.
$$

Therefore, Lemma 3.3.6 implies that

$$
(c_1 \cap z_1, \ldots, c_k \cap z_k) \not\xrightarrow{a} z.
$$

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

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 $\text{Spec}$ and $C[\zeta_z(N)]$ are all independent of $T$, the theorem allows us to perform a single refinement in order to deduce the truth of the refinement $\text{Spec} \sqsubseteq \text{Impl}(T)$ 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 $\{1 \ldots k\}$ to $\mathbb{N}$ and

$$
\text{Spec} \sqsubseteq_T C[\zeta_z(N)],
$$

then

$$
\text{Spec} \sqsubseteq_T C[\text{Nodes}(T)],
$$
(ii) if $z$ is a function from $\{1 \ldots k\}$ to $\mathbb{N}$ such that

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

and

$$Spec \sqsubseteq_F C[\zeta_z(N)],$$

then

$$Spec \sqsubseteq_F C[\text{Nodes}(T)].$$

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

$$\zeta_z(N) \sqsubseteq_T \text{Nodes}(T).$$

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

$$C[\zeta_z(N)] \sqsubseteq_T C[\text{Nodes}(T)].$$

Therefore, if $Spec \sqsubseteq_T C[\zeta_z(N)]$, then, by transitivity of refinement,

$$Spec \sqsubseteq_T C[\text{Nodes}(T)],$$

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 $\zeta_z(N)$, which we represent using a CSP process, called $\text{CANodes}_z$. Below are the first three branches of the definition; the other branches are similar and can be found in Appendix A.

$$\text{CANodes}_z = \text{CANodes}'_z(1, 0, 0, 0, 0)$$

$$\text{CANodes}'_z(c_1, c_2, c_3, c_4, c_5) =$$

- $c_1 \geq 1 \& \text{load} \rightarrow \text{CANodes}'_z(0, 1, c_3, c_4, c_5)$
- $c_2 \geq 1 \& \text{run} \rightarrow (\text{CANodes}'_z(c_1, 0, (c_3 + 1) \sqcap 2, c_4, c_5) \sqcap \text{CANodes}'_z(c_1, 1, (c_3 + 1) \sqcap 2, c_4, c_5))$
- $c_3 \geq 1 \& \text{deschedule} \rightarrow$
  - (if $c_3 = 2$ then $\text{CANodes}'_z(c_1, (c_2 + 1) \sqcap 1, 1, c_4, c_5)$
    - $\sqcap \text{CANodes}'_z(c_1, (c_2 + 1) \sqcap 1, 2, c_4, c_5)$
  - else $\text{CANodes}'_z(c_1, (c_2 + 1) \sqcap 1, 0, c_4, c_5)$

...
<table>
<thead>
<tr>
<th>$# T$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>23</td>
<td>91</td>
<td>253</td>
<td>1324</td>
<td>4861</td>
<td>17497</td>
<td>61966</td>
<td>216514</td>
<td>747955</td>
</tr>
</tbody>
</table>

Table 3.1: Number of states of $\text{Impl}(T)$ 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 $s_i$.

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

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

where

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

We can now use FDR to verify that

$$\text{Spec} \subseteq F C[\text{CANodes}_z],$$

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 $T$),

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

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 $C[\text{CANodes}_z]$ 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.

---

\(^7\)If $T$ is such that $\# T \geq \text{cores}$, then the exact number of states that FDR checks is given by the formula $1 + \sum_{t=0}^{\text{cores}} \left(\binom{\text{cores}}{t}\right)\left(\begin{array}{c}3 \# T - 1 \end{array}\right)^t$. 

End of example.

54
3.3.3 Threshold function cull

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

$$(\langle \text{load, run, run, stopRun}, \Sigma \rangle) \in \text{failures}(C[\text{CANodes}_z]),$$

but

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

Then, there is no $T$ such that $((\text{load, run, run, stopRun}, \Sigma)$ is a stable failure of $\text{Impl}(T)$, 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:

- $\zeta_z(N)$ performs a sequence of events that implies that $n$ processes move into the $i$-th state,
- $z_i = n$, and
- $\zeta_z(N)$ 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 $\text{Spec} \sqsubseteq C[\zeta_z(N)]$ 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 $\text{Spec}$ and $C[:]$ are finite-state and $A_{\text{Sync}} = \{\}$, then there exists a finite threshold function $z$ such that $\text{Spec} \sqsubseteq C[\zeta_z(N)]$.

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.

$$N = in \rightarrow out \rightarrow STOP$$

$$\text{Impl}(t) = ||| i \in t \bullet N.$$
Suppose that we want to verify that the implementation never performs more `out’s than `in’s. Hence, we let

\[
\text{Spec} = \text{Spec}'(x)
\]

\[
\text{Spec}'(x) = \text{in} \rightarrow \text{Spec}'(x + 1)
\]

\[x > 0 \& \text{ out} \rightarrow \text{Spec}'(x - 1).
\]

Observe that `Spec is infinite-state.

The counter state machine \(\varsigma_z(N)\) can be represented using a CSP process defined as follows.

\[
\text{CANodes}_z = (z_1, 0, 0)
\]

\[
\text{CANodes}'_z(c_1, c_2, c_3) =
\]

\[
c_1 \geq z_1 \lor c_1 \geq 1 \& \text{ in} \rightarrow
\]

\[
(\text{if } c_1 = z_1 \text{ then } \text{CANodes}'_z((c_1 - 1) \sqcup 0, (c_2 + 1) \cap z_2, c_3)
\]

\[
\quad \cap \text{CANodes}'_z(c_1, (c_2 + 1) \cap z_2, c_3)
\]

\[
\quad \text{else } \text{CANodes}'_z(c_1 - 1, (c_2 + 1) \cap z_2, c_3))
\]

\[
\square
\]

\[
c_2 \geq z_2 \lor c_2 \geq 1 \& \text{ out} \rightarrow
\]

\[
(\text{if } c_2 = z_2 \text{ then } \text{CANodes}'_z(c_1, c_2 - 1) \sqcup z_2, (c_3 + 1) \cap z_3)
\]

\[
\quad \cap \text{CANodes}'_z(c_1, c_2, (c_3 + 1) \cap z_3)
\]

\[
\quad \text{else } \text{CANodes}'_z(c_1, c_2 - 1, (c_3 + 1) \cap z_3))
\]

Clearly, \(\text{Spec} \subseteq_T \text{Impl}(T)\) for all \(T\), but there is no threshold function \(z\) such that \(\text{Spec} \nsubseteq_T \text{CANodes}_z\) as \(\text{CANodes}_z\) 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.

\textbf{End of example.}

For a given verification problem, if there exists a minimal finite threshold function \(z_{\text{min}}\) such that \(\text{Spec} \subseteq \varsigma_{z_{\text{min}}}(N)\), then it is desirable to find a threshold function \(z\) which is as small as possible and covers \(z_{\text{min}}\), i.e. is such that \(z_i \geq z_{i_{\text{min}}}\) 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 $n = \#(tr \mid (\Sigma \setminus A_{Sync}))$, where $\Sigma$ 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 $n = \#(tr \mid (\Sigma \setminus A_{Sync})) + \#X$, where $\Sigma$ 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 $\#(tr \mid (\Sigma \setminus A_{Sync}))$ 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 $\Theta$ 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 $\Theta_1$. 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 $\Theta_2$. An even better (but even more expensive) heuristic, call it $\Theta_3$, is like $\Theta_2$, but it only increases by 1 the thresholds of counters in which a discontinuity occurred when behaviour $ce$ was formed.

### Algorithm 1

Threshold function $cull$

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for $i \leftarrow 1, k$ do</td>
</tr>
<tr>
<td>2</td>
<td>if $st_i = STOP$ or $M = T$ then</td>
</tr>
<tr>
<td>3</td>
<td>$z_i \leftarrow 0$</td>
</tr>
<tr>
<td>4</td>
<td>else</td>
</tr>
<tr>
<td>5</td>
<td>$z_i \leftarrow 1$</td>
</tr>
<tr>
<td>6</td>
<td>end if</td>
</tr>
<tr>
<td>7</td>
<td>end for</td>
</tr>
<tr>
<td>8</td>
<td>while $Spec \not\subseteq M \subseteq (\mathcal{N})$ do</td>
</tr>
<tr>
<td>9</td>
<td>$ce \leftarrow$ counterexample</td>
</tr>
<tr>
<td>10</td>
<td>if $\exists T \bullet admissible(ce, Impl(T))$ then</td>
</tr>
<tr>
<td>11</td>
<td>return no</td>
</tr>
<tr>
<td>12</td>
<td>else</td>
</tr>
<tr>
<td>13</td>
<td>$z \leftarrow \Theta(ce, z)$</td>
</tr>
<tr>
<td>14</td>
<td>end if</td>
</tr>
<tr>
<td>15</td>
<td>end while</td>
</tr>
<tr>
<td>16</td>
<td>return yes</td>
</tr>
</tbody>
</table>

---

*Given $tr \in \Sigma$ and provided that $A \subseteq \Sigma$, $tr \mid A$ (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 $\text{Spec} \subseteq C[\zeta(N)]$ exists, termination of Algorithm 1 depends on the choice of heuristic $\Theta$. For example, if $\Theta$ 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 $\Theta_1$, $\Theta_2$ and $\Theta_3$, 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 $f(z_j) = \sum_{i=1}^{k} \max(Z_i - z_j, 0)$. We say that a heuristic $\Theta$ is decreasing if $f$ is decreasing when $\Theta$ is used. The algorithm terminates when we find a finite $j$ such that for all $i \in \{1..k\}$, $z_j \geq Z_i$, i.e. when $f(z_j) = 0$. For this to happen, it is enough if the used heuristic $\Theta$ is decreasing, since $f(z_j)$ is always a finite, non-negative integer. If $\Theta = \Theta_1$, then for all $j$, $z_j^{j+1} = z_j + 1$ for all $i \in \{1..k\}$, which means that $f(z_j^{j+1}) < f(z_j)$. 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 $z_j < Z_i$ and the $i$-th counter attains $z_j^i$, which means that $z_j^{j+1} = z_j^i + 1$, i.e. $f(z_j^{j+1}) < f(z_j)$ (and discontinuity occurs in the $i$-th counter when $\Theta_3$ 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 values1. 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.

1In 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. $\text{card}(t)$.  

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

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

where $\text{snd}(x_1, x_2, x_3) = x_2$, as

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

The following example shows a process that satisfies data independence.

**Example 4.1.2.** Let

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

62
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.

\textit{End of example.}

\textbf{Remark 4.1.3.} Clauses (v) and (vi) of Definition 4.1.1 together imply that, for all constructs $c$ 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 $\xi_i = !$, in which case $X_i = \text{null}$.

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 \textbf{SeqNorm} condition, defined below, achieves this without a major expressiveness reduction.

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

\begin{align*}
\text{Channels}(\text{STOP}) & \equiv \{\}, \\
\text{Channels}(c_{\xi_1}x_1 : X_1 \ldots \xi_k x_k : X_k \to P) & \equiv \{c\}, \\
\text{Channels}(P \oplus Q) & \equiv \text{Channels}(P) \cup \text{Channels}(Q), \\
\text{Channels}(P \otimes Q) & \equiv \text{Channels}(P) \cup \text{Channels}(Q), \\
\text{Channels}(P \triangleright Q) & \equiv \text{Channels}(P) \cup \text{Channels}(Q), \\
\text{Channels}(X) & \equiv \text{Channels}(P), \quad \text{if } E(X) = P.
\end{align*}

\textbf{Definition 4.1.4.} A process syntax $\text{Proc}(t)$ satisfies \textbf{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 $P(t) \parallel Q(t)$, internal choices $P(t) \cap Q(t)$ and sliding choices $P(t) \triangleright Q(t)$ within $\text{Proc}(t)$ we have that

\begin{itemize}
  \item $\text{Channels}(P(t)) \cap \text{Channels}(Q(t)) = \{\}$
  \item every conditional choice on $t$ in $P(t)$ and $Q(t)$ is after a prefix,
\end{itemize}

Our definition of \textbf{SeqNorm} is similar to definitions of \textbf{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 \textbf{SeqNorm}.

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

We have decided to include sequentiality as part of the definition of \textbf{SeqNorm}, since we mainly assume \textbf{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 \( \sqcap, \sqcap \) 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

\[
\text{Proc}(t) = \text{in}?x:t?y:t \rightarrow \text{if } x = y \text{ then } \text{STOP} \\
\text{else } \text{out}.x \rightarrow \text{STOP} \sqcap \text{out}.y \rightarrow \text{STOP}.
\]

Then, once \( \text{Proc}(t) \) 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 \( \sqcap, \sqcap \) 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. \( \sqcap x \in X \bullet (c_1.x \rightarrow \text{STOP} \sqcup c_2.x \rightarrow \text{STOP}) \).

**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 \( \sqcap, \sqcap \) and \( \triangleright \))
over conditional choice) can be used to convert it to an equivalent process definition, satisfying this assumption:

\[ P \equiv (b \text{ if } Q \text{ else } R) \equiv (b \text{ if } P \text{ else } Q) \text{ else } (P \text{ else } R), \]
\[ (b \text{ if } P \text{ else } Q) \equiv (b \text{ if } P \text{ else } Q) \text{ else } (P \text{ else } R), \]

where \( \equiv \) is one of \( \sqcap, \sqcup \) or \( \triangleright \).

\textbf{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 \textbf{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. \( \text{Proc}(t) = \alpha(t) \rightarrow \text{Proc}'(t) \). 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 \( \alpha \) of the form \( c\xi_1:x_1 \ldots \xi_k:x_k \), we define a number of useful functions that return index sets of variables and values within \( \alpha \), based on their type and the kind of input or output they model:

\[
\begin{align*}
$^t(\alpha) & \triangleq \{ i \in \{1 \ldots k\} | \xi_i = \$ \land X_i = t \}, \\
$^{\text{non-}t}(\alpha) & \triangleq \{ i \in \{1 \ldots k\} | \xi_i = \$ \land X_i \neq t \}, \\
\gamma(\alpha) & \triangleq \gamma^t(\alpha) \cup \gamma^{\text{non-}t}(\alpha), \\
?^t(\alpha) & \triangleq \{ i \in \{1 \ldots k\} | \xi_i = ? \land X_i = t \}, \\
?^{\text{non-}t}(\alpha) & \triangleq \{ i \in \{1 \ldots k\} | \xi_i = ? \land X_i \neq t \}, \\
\gamma(\alpha) & \triangleq \gamma^t(\alpha) \cup \gamma^{\text{non-}t}(\alpha), \\
!^t(\alpha) & \triangleq \{ i \in \{1 \ldots k\} | \xi_i = ! \land x_i \text{ is of type } t \}, \\
!^{\text{non-}t}(\alpha) & \triangleq \{ i \in \{1 \ldots k\} | \xi_i = ! \land x_i \text{ is not of type } t \}, \\
!(\alpha) & \triangleq !(\alpha) \cup !^{\text{non-}t}(\alpha).
\end{align*}
\]

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

\[ \dag(\alpha) = \{ i \in \{1 \ldots k\} | \xi_i = \dag \}. \]
Sometimes we will want to modify constructs. The following functions make it easier. Let \( \alpha = c \$_1 x_1 : X_1 \ldots \$_k x_k : X_k \) and let \( \uparrow \) be one of either \( t \) or \( \text{non-}t \). Then, we define:

- \( \text{Replace}_t^{\$ \mapsto \uparrow} (\alpha) \) to be a construct like \( \alpha \), but where for every \( i \) in \( \$^{\uparrow}(\alpha) \) the \( \$_i \) symbol (which must be a \$) is replaced by a \(!\) and \( X_i \) is replaced by \( \text{null} \), and

- \( \text{Replace}^{\text{non-}t} \equiv \text{Replace}_t^{\$ \mapsto \uparrow} \circ \text{Replace}^{\text{non-}t} \).

**Example 4.1.8.** Let

\[
\epsilon = c \$ x_1 : t \# x_2 : t \$ x_3 : X \! x_4 ,
\]

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

\[
\text{Replace}_t^{\$ \mapsto \uparrow} (\epsilon) = c ! x_1 ? x_2 : t \$ x_3 : X ! x_4 ,
\]

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

and

\[
\text{Replace}^{\text{non-}t} (\epsilon) = c ! x_1 ? x_2 : t ! x_3 ! x_4 .
\]

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

- \( \text{Value} \) to be the set of all values,

- \( \text{Var} \) to be the set of all variable names (we assume \( \text{Var} \cap \text{Value} = \{ \} \)), and

- \( \text{FV}(P) \) 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 \( \tau \), 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 \( P \xrightarrow{a} \) 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 \( \{0 \ldots n - 1\} \), \( P_i \xrightarrow{a} P_{i+1} \). In particular, \( P \xrightarrow{a} \) (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 \( P(T) \parallel Q(T) \), both \( P(T) \) and \( Q(T) \) are switched on, so any \( \tau \)'s available immediately in \( P(T) \) or \( Q(T) \) must be promoted, leading to the following rules.

\[
\begin{align*}
P(T) \xrightarrow{\tau} P'(T) & \quad & Q(T) \xrightarrow{\tau} Q'(T) \\
P(T) \parallel Q(T) \xrightarrow{\tau} P'(T) \parallel Q(T) & \quad & P(T) \parallel Q(T) \xrightarrow{\tau} P(T) \parallel Q'(T)
\end{align*}
\]

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

\[
\begin{align*}
P(T) \xrightarrow{a} P'(T) & \quad & Q(T) \xrightarrow{a} Q'(T) \\
(P(T) \parallel Q(T) \xrightarrow{a} P'(T)) & \quad & (P(T) \parallel Q(T) \xrightarrow{a} Q'(T))
\end{align*}
\]

Any \( a \neq \tau \)

**Internal choice**

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

\[
\begin{align*}
P(T) \parallel Q(T) & \xrightarrow{\tau} P(T) \\
P(T) \parallel Q(T) & \xrightarrow{\tau} Q(T)
\end{align*}
\]
Sliding choice (timeout)
A sliding choice may be, at any time, resolved to the second argument by a single internal action:

\[ P(T) \triangleright Q(T) \xrightarrow{\tau} Q(T) \]

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

\[ P(T) \xrightarrow{\tau} P'(T) \]
\[ P(T) \triangleright Q(T) \xrightarrow{\tau} P'(T) \triangleright Q(T) \]

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:

\[ P(T) \xrightarrow{a} P'(T) \]
\[ P(T) \triangleright Q(T) \xrightarrow{a} P'(T) \]

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.

\[ P(T) \xrightarrow{\tau} P'(T) \]
\[ P(T) \parallel X(T) \xrightarrow{\tau} P'(T) \parallel X(T) \]
\[ Q(T) \xrightarrow{\tau} Q'(T) \]
\[ Q(T) \parallel X(T) \xrightarrow{\tau} Q'(T) \parallel X(T) \]

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.

\[ P(T) \xrightarrow{e} P'(T) \]
\[ P(T) \parallel X(T) \xrightarrow{e} P'(T) \parallel X(T) \]
\[ Q(T) \xrightarrow{e} Q'(T) \]
\[ Q(T) \parallel X(T) \xrightarrow{e} Q'(T) \parallel X(T) \]

\[ e \in \Sigma \setminus X(T) \]
Finally, all events in the synchronisation set require participation of both arguments in order to be performed.

\[
\frac{P(T) \xrightarrow{e} P'(T) \quad X(T) \quad Q(T) \xrightarrow{e} Q'(T) \quad X(T)}{P(T) \parallel X(T) \quad Q(T) \parallel X(T) \quad Q'(T) \parallel X(T) \quad e \in \Sigma \setminus X(T)}
\]

**Hiding**

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

\[
\frac{P(T) \xrightarrow{a} P'(T) \quad X(T) \quad a \not\in X(T)}{P(T) \setminus X(T) \xrightarrow{a} P'(T) \setminus X(T) \quad a \not\in X(T)}
\]

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

\[
\frac{P(T) \xrightarrow{e} P'(T) \quad X(T) \quad e \in X(T)}{P(T) \setminus X(T) \xrightarrow{e} P'(T) \setminus X(T) \quad e \in X(T)}
\]

**Renaming**

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

\[
\frac{P(T) \xrightarrow{\tau} P'(T) \quad X(T) \quad \mathcal{R}(T) \quad \mathcal{R}(T)}{P(T)\mathcal{R}(T) \xrightarrow{\tau} P'(T)\mathcal{R}(T)}
\]

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

\[
\frac{P(T) \xrightarrow{e} P'(T) \quad X(T) \quad \mathcal{R}(T) \quad \mathcal{R}(T)}{P(T)\mathcal{R}(T) \xrightarrow{e} P'(T)\mathcal{R}(T) \quad e\mathcal{R}(T)e'}
\]

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 $\alpha$ be a construct of the form $c\beta_1 x_1 : X_1 \ldots \beta_k x_k : X_k$ 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:

$$\frac{}{\alpha \rightarrow P(T) \xrightarrow{c.v_1 \ldots v_k} P(T)[v_i/x_i \mid i \in \#$},}$$

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

$$Comms(c\beta_1 x_1 : X_1 \ldots \beta_k x_k : X_k) = \{c.v_1 \ldots v_k \mid \forall i \in \{1 \ldots k\} \bullet (\beta_i = ? \land v_i \in X_i) \lor (\beta_i = ! \land v_i = x_i)\}.$$

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 \rightarrow P(T) \xrightarrow{c.v_1 \ldots v_k} P(T)[v_i/x_i \mid i \in \#$, 
\#}\}$

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.

$$\frac{\text{dom}(v) = \#$, \forall i \in \#$, v_i \in X_i}{\alpha \rightarrow P(T) \xrightarrow{\tau} \text{Replace}_{\#$,!}(\alpha) \rightarrow P(T)[v_i/x_i \mid i \in \#]}$$
Observe that this rule is equivalent to defining $\alpha \rightarrow P(T)$ as

$$\sqcap (x_i : X_i \mid i \in \$,\non-t(\alpha)) \bullet \text{Replace}_{\$\non-t(\alpha)}(\alpha) \rightarrow P(T),$$

where, given an indexing set $I = \{i_1, \ldots, i_n\}$, we use $\sqcap (x_i : X_i \mid i \in I) \bullet P(x_{i_1}, \ldots, x_{i_n})$ as a shorthand notation for $\sqcap (x_{i_1}, \ldots, x_{i_n}) \in X_{i_1} \times \cdots \times X_{i_n} \bullet P(x_{i_1}, \ldots, x_{i_n})$.

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)

\[
\begin{align*}
\text{dom}(v) &= \$^\non-t(\alpha) \land \forall i \in \$^\non-t(\alpha) \bullet v_i \in X_i \quad [\#\$^{\non-t}(\alpha) > 0] \\
\alpha &\rightarrow P(T) \\
\tau &\rightarrow (\text{Replace}_{\$\non-t(\alpha)}(\alpha) \rightarrow P(T)) [v_i/x_i \mid i \in \$^{\non-t}(\alpha)]
\end{align*}
\]

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

\[
\begin{align*}
v &\in \$^t(\alpha) \rightarrow T \quad [\#\$^{\non-t}(\alpha) = 0 \\
\land \#\$^t(\alpha) > 0] \\
\alpha &\rightarrow P(T) \\
\tau &\rightarrow (\text{Replace}_{\$\non-t(\alpha)}(\alpha) \rightarrow P(T)) [v_i/x_i \mid i \in \$^t(\alpha)]
\end{align*}
\]

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

$$\sqcap (x_i : X_i \mid i \in \$^{\non-t}(\alpha)) \bullet (\sqcap (x_i : T \mid i \in \$^t(\alpha)) \bullet \text{Replace}_{\$\non-t(\alpha)}(\alpha) \rightarrow P(T)).$$

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 $\tau$ 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 $\text{Proc}(t) = \text{chn}$\$x:\{a, b\}$\$y: t \rightarrow \text{STOP}$. Suppose we use the standard operational semantics to work out the transition graph of $\text{Proc}(\{0, 1, 2\})$. 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.

\[
P(T) \xrightarrow{a} P'(T) \\
\text{(if } True \text{ then } P(T) \text{ else } Q(T)) \xrightarrow{a} P'(T)
\]

\[
Q(T) \xrightarrow{a} Q'(T) \\
\text{(if } False \text{ then } P(T) \text{ else } Q(T)) \xrightarrow{a} Q'(T)
\]

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

\[
P_j(T) \xrightarrow{\tau} P'_j(T) \\
\square i \in I(T) \bullet P_i(T) \\
\quad \wedge \forall i \in I(T) \setminus \{j\} \bullet P'_i(T) = P_i(T)
\]

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.
Replicated internal choice
For any non-empty indexing set $I(T)$, $\bigcap i \in I(T) \bullet P_i(T)$ 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:

\[
\bigcap i \in I(T) \bullet P_i(T) \xrightarrow{\tau} P_j(T) \quad [j \in I(T)]
\]

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

\[
\forall i \in I(T) \bullet P_i(T) \xrightarrow{e} P'_i(T) \quad [e \in X(T)]
\]

\[
\big\| i \in I(T) \bullet P_i(T) \xrightarrow{e} P'_i(T) \big\| i \in I(T) \bullet P_i(T) \quad [e \in X(T)]
\]

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.

\[
\big\| i \in I(T) \bullet P_i(T) \xrightarrow{a} P'_i(T) \big\| i \in I(T) \bullet P_i(T) \quad [a \in \Sigma^\tau \setminus X(T) \land j \in I(T) \\
\land \forall i \in I(T) \setminus \{j\} \bullet P'_i(T) = P_i(T)]
\]

\[
\big\| i \in I(T) \bullet P_i(T) \xrightarrow{a} P'_i(T) \big\| i \in I(T) \bullet P_i(T)
\]

The above rules also provide the semantics of the replicated interleaving operator, since $\big\| i \in I(T) \bullet P_i(T) = \big\| i \in I(T) \bullet P_i(T)$. 

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:
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:

\[
\forall i \in \mathcal{J}(T) \cdot P_i(T) \xrightarrow{e} P'_i(T) \quad \left[ \mathcal{J}(T) \subseteq \mathcal{I}(T) \wedge e \in \left( \bigcap_{i \in \mathcal{J}(T)} A_i(T) \right) \setminus \bigcup_{i \in \mathcal{I}(T) \setminus \mathcal{J}(T)} A_i(T) \wedge \forall j \in \mathcal{I}(T) \setminus \mathcal{J}(T) \cdot P'_j(T) = P_j(T) \right].
\]

**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.

\[
E(X) = P \\
X(T) \xrightarrow{\tau} P(T)
\]

**Example 4.2.2.** Let

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

Figure 4.2 represents the operational semantics for $\text{Proc}(T)$ 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 $d!x \rightarrow \text{STOP}$ else $\text{STOP}$. The first $\tau$ transitions correspond to Prefix Rule 2a; the second $\tau$ 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.
(from [Ros97, Chapter 7]). Given two states, say $P(T)$ and $Q(T)$, and a sequence of events (visible or invisible) $s = \langle a_i \mid i \in \{1 \ldots n\} \rangle$ for some $n \geq 0$, we write

$$P(T) \xrightarrow{s} Q(T)$$

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

$$P(T) \xrightarrow{\tau}$$

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

$$P(T) \xRightarrow{tr} Q(T)$$

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

$$P(T) \xRightarrow{tr}$$

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

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

$$\text{traces}(P(T)) = \{\text{tr} \in \Sigma^* \mid \exists Q(T) \bullet P(T) \xRightarrow{tr} Q(T)\}$$

and

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

where $Q(T) \text{ ref } X$ 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 $Q(T) \xrightarrow{a} Q'(T)$ 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 \textit{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 \textit{process syntaxes}, derived from original process syntaxes. This is why we call each resulting transition graph a \textit{semi-symbolic labelled transition system} (SSLTS). For the same reasons we call our operational semantics \textit{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 \textit{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 $\rightarrow_s$, i.e. $P(t) \xrightarrow{\alpha}_s Q(t)$ denotes the fact that there is an $\alpha$-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 $P(t) \xrightarrow{\alpha}_s Q(t)$. 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 $\{0 \ldots n - 1\}$, $P_i(t) \xrightarrow{\alpha}_s P_{i+1}(t)$. In particular, $P(t) \xrightarrow{\alpha}_s^* P(t)$ (taking $n = 0$). Finally, $P(t) \xrightarrow{\alpha}_s^*$ means that there exists some symbolic state $Q(t)$ such that $P(t) \xrightarrow{\alpha}_s^* Q(t)$.

We distinguish the following three types of symbolic transitions.

\textbf{Invisible} The invisible symbolic transitions are in a direct correspondence with the standard invisible transitions and are labelled with $\tau$’s (which in this setting are be called \textit{invisible} (or \textit{internal}) \textit{symbolic events}).

\textbf{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 \textit{visible symbolic event}, a construct of the form \( c\xi_1x_1:X_1\ldots\xi_kx_k:X_k \) for \( k > 0 \), where
- \( c \) is a channel name,
- \( \xi_i \in \{$, ?, !$\} \) 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, \( \xi_i \), is either \$ or \?, otherwise it is \textit{null}.

We let \textit{Visible} denote the set of all visible symbolic events.

\textbf{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 \textit{True} or \textit{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 \textit{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 \textit{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 \"\( x = x \)\".

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

\textbf{Example 4.3.2.} Recall the following process definition that we introduced in Example 4.2.1:

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

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

\textit{End of example.}

We will usually use \( \alpha \) and its derivatives (\( \alpha', \alpha_1, \text{etc.} \)) to denote labels whose kind is unknown or indifferent and \( \epsilon \) and its derivatives (\( \epsilon', \epsilon_1, \text{etc.} \)) to denote visible symbolic events.

\subsection{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 \textbf{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 $\alpha$ be a construct of the form $c_{\bar{x}1: X_1} \ldots c_{\bar{x}k: X_k}$ 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 $\alpha$ 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 $\alpha$.

**Symbolic Prefix Rule 1**

$$
\begin{align*}
\epsilon & = c_{\bar{x}'1: X'_1} \ldots c_{\bar{x}'k: X'_k} \\
(\alpha \rightarrow P(t)) & \xrightarrow{\tau} P(t)[x'_i/x_i \mid i \in \text{?non-t(\alpha)}] \\
& \epsilon \in \text{Comms}^{\text{non-t}(\alpha)} \\
& \land \#\text{\text{?non-t}(\alpha)} = 0
\end{align*}
$$

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

$$
\text{Comms}^{\text{non-t}(c_{\bar{x}1: X_1} \ldots c_{\bar{x}k: X_k})} =
\{ c_{\bar{x}'1: X'_1} \ldots c_{\bar{x}'k: X'_k} \mid \forall i \in \{1 \ldots k\} \bullet
\begin{align*}
& \text{?} \land X_i \neq t \land \text{?} \land X'_i = \text{null} \\
& \text{?} \land \text{?} \land X'_i = X_i \land x'_i = x_i \\
& \text{?} \land \text{?} \land X'_i = X_i \land x'_i = x_i
\end{align*}
\}
$$

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 $\tau$ transition. The values being chosen are substituted for the variables of the selections.

**Symbolic Prefix Rule 2**

$$
\begin{align*}
\text{dom}(v) & = \text{\text{?non-t}(\alpha)} \land \forall i \in \text{\text{?non-t}(\alpha)} \bullet v_i \in X_i \\
\alpha & \rightarrow P(t) \\
& \xrightarrow{\tau} (\text{Replace}_{\text{\text{?non-t}(\alpha)}}^{\text{\text{?non-t}(\alpha)}} \rightarrow P(t)) [v_i/x_i \mid i \in \text{\text{?non-t}(\alpha)}]
\end{align*}
$$
**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 $\text{Cond}$ must be promoted by the $\square$ operator in the same way $\tau$’s are, leading to the following two firing rules.

\[
P(t) \overset{\alpha}{\rightarrow} s P'(t) \\
\text{if } \alpha \in \text{Cond} \cup \{\tau\}
\]

\[
Q(t) \overset{\alpha}{\rightarrow} s Q'(t) \\
\text{if } \alpha \in \text{Cond} \cup \{\tau\}
\]

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

\[
P(t) \overset{\varepsilon}{\rightarrow} s P'(t) \\
\text{if } \varepsilon \notin \text{Cond} \cup \{\tau\}
\]

\[
Q(t) \overset{\varepsilon}{\rightarrow} s Q'(t) \\
\text{if } \varepsilon \notin \text{Cond} \cup \{\tau\}
\]

**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.

\[
P(t) \sqcap Q(t) \overset{\tau}{\rightarrow} s P(t) \\
P(t) \sqcap Q(t) \overset{\tau}{\rightarrow} s Q(t)
\]

**Sliding choice (timeout)**

The transitions rules for the $\triangleright$ operator are similar to the ones from the standard operational semantics.

\[
P(t) \triangleright Q(t) \overset{\tau}{\rightarrow} s Q(t)
\]

\[
P(t) \overset{\alpha}{\rightarrow} s P'(t) \\
\text{if } \alpha \in \text{Cond} \cup \{\tau\}
\]

\[
P(t) \overset{\varepsilon}{\rightarrow} s P'(t) \\
\text{if } \varepsilon \notin \text{Cond} \cup \{\tau\}
\]
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 $\text{Cond}$) can be fully evaluated at the time of SSLTS generation. Hence, we have the following rules, similar to the standard ones.

\[
\begin{align*}
P(t) & \xrightarrow{\alpha} P'(t) \\
(\text{if True then } P(t) & \text{ else } Q(t)) \xrightarrow{\alpha} P'(t) \\
Q(t) & \xrightarrow{\alpha} Q'(t) \\
(\text{if False then } P(t) & \text{ else } Q(t)) \xrightarrow{\alpha} Q'(t)
\end{align*}
\]

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

\[
\begin{align*}
(\text{if } \text{cond} & \text{ then } P(t) \text{ else } Q(t)) \xrightarrow{\text{cond}} P(t) \\
(\text{if } \text{cond} & \text{ then } P(t) \text{ else } Q(t)) \xrightarrow{\neg \text{cond}} Q(t)
\end{align*}
\]

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.

\[
E(X) = P \\
X(t) \xrightarrow{\tau} P(t)
\]

**Example 4.3.3.** Recall the following process from Example 4.2.2:

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

\text{else STOP}

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

End of example.
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 $t$:

\[
\begin{align*}
COPY(t) &= \text{in}?x:t \rightarrow \text{out}.x \rightarrow COPY(t), \\
COPY'(t) &= \text{in}?y:t \rightarrow \text{out}.y \rightarrow COPY'(t), \\
COPY''(t) &= \text{in}?x:t \rightarrow \text{out}.x \rightarrow \text{in}?y:t \rightarrow \text{out}.y \rightarrow COPY'(t).
\end{align*}
\]

Then, we consider $COPY(t)$ and $COPY''(t)$ to be identical, but not $COPY(t)$ and $COPY'(t)$, since $COPY(t)$ contains additional $\tau$’s that correspond to the lookup of process identifier $COPY(t)$. In addition, symbolic states $\text{out}.x \rightarrow COPY(t)$ and $\text{out}.y \rightarrow COPY'(t)$ 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 $\text{Proc}(t)$ be some process syntax and let $\mathcal{S} = (S, s_0, L, \rightarrow_s)$ be the SSLTS obtained by applying the SSOS to $\text{Proc}(t)$. Given two symbolic states $P(t)$ and $Q(t)$ in $\mathcal{S}$ and a sequence of symbolic events $\sigma = \{\alpha_i \mid i \in \{1 \ldots n\}\}$, we say that

\[
P(t) \xrightarrow{\sigma_s} Q(t)
\]
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 $\{0 \ldots n - 1\}$ we have that $P_i(t) \xrightarrow{\alpha_{i+1}} P_{i+1}(t)$. We also write
\[ P(t) \xrightarrow{\sigma}_s \]
to mean that there exists a symbolic state $Q(t)$ such that $P(t) \xrightarrow{\sigma}_s Q(t)$. If $P(t) \xrightarrow{\sigma}_s$, then $\sigma$ is called a symbolic trace of $P(t)$. Therefore, a symbolic trace of $\text{Proc}(t)$ is a sequence of labels of symbolic events that form a path, starting at $s_0$, through the $S$. We let $\text{SymbolicTraces}(\text{Proc}(t))$ denote the set of all symbolic traces of $\text{Proc}(t)$. Observe that symbolic traces are quite different from standard traces as they may contain invisible symbolic events ($\tau$'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 $\sigma, \rho$ and their derivatives ($\sigma', \rho_1$, 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 $\text{Env}(t) \equiv \text{Var} \rightarrow t$. Then an environment is a partial function $\Gamma \in \text{Env}(T)$ for some instantiation $T$ of type $t$.

We adopt the notational convention that for all $v$ in $\text{Value}$, $\Gamma(v) = v$. 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 $\Gamma$ is in $\text{Env}(T)$, 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 dom($\Gamma$)). In particular, given a process definition $P(t)$ we define
\[ P(t)[\Gamma] \equiv P(t)[\Gamma(x)/x \mid x \in \text{dom}(\Gamma)] \]

Note that for all environments $\Gamma$ and all symbolic events $\alpha$ we have that
$\dagger(\Gamma(\alpha)) = \dagger(\alpha)$ and $?\dagger(\Gamma(\alpha)) = ?\dagger(\alpha)$
for † ∈ \{t, non-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 \( f : T \rightarrow T' \) and an environment \( \Gamma \) in \( \text{Env}(T) \), we define
\[ f(\Gamma) = \{ x \mapsto f(v) \mid \Gamma(x) = v \}. \]
Observe that \( f(\Gamma) \) is an environment in \( \text{Env}(T') \).

The states of the LTS \( C \) that COSE generates from a given process syntax \( \text{Proc}(t) \) are configurations \( (P(t), \Gamma, T) \), where:
- \( P(t) \) is a symbolic state, equal to or slightly modified from a state of the SSLTS \( S \) of \( \text{Proc}(t) \),
- \( \Gamma \) is an environment in \( \text{Env}(T) \), 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 \( C \) is defined to be the configuration \( (P_0(t), \{\}, T) \), where \( P_0(t) \) is the initial state of \( S \). To emphasise the fact that COSE is a concrete operational semantics we denote the transition relation using the same symbol \( (\rightarrow) \) 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 \( \Gamma \) in \( \text{Env}(T) \).

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:
- \( P(t)[\Gamma] \equiv_a P'(t)[\Gamma'] \), and
- \( T = T' \),
where \( \equiv_a \) 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:
\[ \text{COPY}(t) = \text{in}\?x:t \rightarrow \text{out}.x \rightarrow \text{COPY}(t), \]
\[ \text{COPY}'(t) = \text{in}\?y:t \rightarrow \text{out}.y \rightarrow \text{COPY}'(t). \]
Then,
\[ (\text{COPY}(t), \{\}, T) = (\text{COPY}'(t), \{\}, T) \]
for all \( T \), since \( \text{COPY}(t)[\{\}] \) is equal to \( \text{COPY}'(t)[\{\}] \) with bound variable \( y \) renamed to \( x \). Also, we have that
\[ (\text{out}.x \rightarrow \text{COPY}(t), \{x \mapsto 0\}, T) = (\text{out}.y \rightarrow \text{COPY}'(t), \{x \mapsto 0, y \mapsto 0\}, T), \]
since
\[(\text{out}.x \to \text{COPY}(t))[\{x \mapsto 0\}]\]
\[= \text{out}.0 \to \text{COPY}(t)[\{x \mapsto 0\}]\]
\[\equiv_a \text{out}.0 \to \text{COPY}'(t)[\{x \mapsto 0, y \mapsto 0\}]\]
\[= (\text{out}.y \to \text{COPY}'(t))[\{x \mapsto 0, y \mapsto 0\}]\].

End of example.

Remark 4.4.3. Observe that
\[P(t)[\Gamma] = P(t)[\text{FV}(P(t)) \triangleleft \Gamma]\] for all symbolic states \(P(t)\) and all environments \(\Gamma\). Therefore, configurations \((P(t), \Gamma, T)\) and \((P(t), \text{FV}(P(t)) \triangleleft \Gamma, T)\) are identical. From now on we always assume environment minimality within configurations, which we achieve by restricting the environment \(\Gamma\) 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 \textbf{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 \(Q(t) = \text{Replace}_t^t(c, P(t))\) be a symbolic state like \(P(t)\), except every transition from \(P(t)\) labelled with a visible symbolic transition \(\epsilon\) on channel \(c\) is replaced with an identical transition in \(Q(t)\), but labelled with \(\text{Replace}_t^t(\epsilon)\) instead, i.e. \(P(t) \xrightarrow{\epsilon} P'(t)\) if and only if \(Q(t) \xrightarrow{\text{Replace}_t^t(\epsilon)} P'(t)\). 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 non-deterministic 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

\[
\begin{align*}
P(t) & \xrightarrow{\epsilon} Q(t) \\
\epsilon = c^t_{\exists_1 x_1 : X_1} \ldots c^t_{\exists_k x_k : X_k} \land v \in S^t(\epsilon) & \to T \\
(P(t), \Gamma, T) & \xrightarrow{\tau} (\text{Replace}_t^t(c, P(t)), \Gamma \oplus \{x_i \mapsto v_i \mid i \in S^t(\epsilon)\}, T)
\end{align*}
\]

\(^3\)Given a function \(f\) and a set \(S \subseteq \text{dom}(f)\), \(S \circ f\) denotes the function obtained from \(f\) by restricting its domain to the elements of \(S\), i.e.
\(S \circ f = \{x \mapsto f(x) \mid x \in S\}\).
When we defined \textbf{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.

\textbf{Example 4.4.4.} Let

\[ \text{Proc}(t) = c_1 ?x:t \rightarrow (c_2 ?x:y:t \rightarrow \text{STOP} \quad \square c_1 !x \rightarrow \text{STOP}) \]

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

\[ \text{End of example.} \]

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 \( t \).

\textbf{Translation Rule 2}

\[ P(t) \xrightarrow{\epsilon} Q(t) \]

\[ \epsilon = c_{\delta_1 x_1 : X_1} \ldots c_{\delta_k x_k : X_k} \]

\[ \left( \text{dom}(v) = \{1 \ldots k\} \land (\forall i \in ?^t(\epsilon) \bullet v_i \in T) \land \forall i \in !^t(\epsilon) \bullet v_i = \Gamma(x_i) \right) \]

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

\[ \left[ \#^t(\epsilon) = 0 \right] . \]

\textbf{Example 4.4.5.} Let

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

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

\[ \text{Proc}(t) \xrightarrow{\text{chn} !a y:z:t} \text{Proc}'(t)[a/x], \quad (4.1) \]

\[ \text{Proc}(t) \xrightarrow{\text{chn} !b y:z:t} \text{Proc}'(t)[b/x]. \quad (4.2) \]

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

\[ \text{conf}_0 = (\text{Replace}_{\delta_{\delta_1}}(\epsilon, \text{Proc}(t)), \{y \mapsto 0\}, T), \]

\[ \text{conf}_1 = (\text{Replace}_{\delta_{\delta_1}}(\epsilon, \text{Proc}(t)), \{y \mapsto 1\}, T). \]

\[ 85 \]
Now, from (4.1) and (4.2) and the definition of Replace,

\[
\begin{align*}
\text{Replace}^1_{\tau \to t}(c, \text{Proc}(t)) & \xrightarrow{\text{chn}l\text{a}ly?z:t} \text{Proc}'(t)[a/x], \\
\text{Replace}^1_{\tau \to t}(c, \text{Proc}(t)) & \xrightarrow{\text{chn}l\text{b}ly?z:t} \text{Proc}'(t)[b/x].
\end{align*}
\]

Hence, using Translation Rule 2, we can deduce

\[
\begin{align*}
\text{conf}_0 & \xrightarrow{\text{chn}, 0, 0, 0} (\text{Proc}'(t)[a/x], \{y \mapsto 0, z \mapsto 0\}, T), \\
\text{conf}_0 & \xrightarrow{\text{chn}, 0, 0, 1} (\text{Proc}'(t)[a/x], \{y \mapsto 0, z \mapsto 1\}, T), \\
\text{conf}_0 & \xrightarrow{\text{chn}, k, 0, 0} (\text{Proc}'(t)[b/x], \{y \mapsto 0, z \mapsto 0\}, T), \\
\text{conf}_0 & \xrightarrow{\text{chn}, k, 0, 1} (\text{Proc}'(t)[b/x], \{y \mapsto 0, z \mapsto 1\}, T);
\end{align*}
\]

and similarly for \(\text{conf}_1\).

**End of example.**

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

\[
\begin{align*}
P(t) & \xrightarrow{c} Q(t) \land \epsilon = c\xi_1 x_1 : X_1 \ldots \xi_k x_k : X_k \\
& \land \text{dom}(v) = \{1 \ldots k\} \land (\forall i \in S^t(\epsilon) \cup ?^t(\epsilon) \bullet v_i \in T) \land (\forall i \in !^t(\epsilon) \bullet v_i = \Gamma(x_i)),
\end{align*}
\]

then

\[
(P(t), \Gamma, T) \xrightarrow{\text{chn}l\text{a}ly?'z', \ldots, 'z'} (Q(t), \Gamma \oplus \{x_i \mapsto v_i \mid i \in S^t(\epsilon) \cup ?^t(\epsilon)\}, T),
\]

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

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**

\[
P(t) \xrightarrow{\tau} Q(t) \quad (P(t), \Gamma, T) \xrightarrow{\tau} (Q(t), \Gamma, T)
\]

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**

\[
P(t) \xrightarrow{\text{cond}} Q(t) \quad (Q(t), \Gamma, T) \xrightarrow{a} (R(t), \Gamma', T) \quad [\text{cond} \in \text{Cond} \land \llbracket \text{cond} \rrbracket_\Gamma],
\]

where \(\llbracket \text{cond} \rrbracket_\Gamma\) denotes the truth value of the proposition obtained from \(\text{cond}\) by substituting all free variables of type \(t\) with their corresponding values contained within the environment \(\Gamma\). 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 $\text{Proc}(t)$ is some process syntax that satisfies $\text{SeqNorm}$. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the LTSs generated from $\text{Proc}(t)$, 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

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

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:

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

and

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

for every process syntax $\text{Proc}(t)$, instantiation $T$ of $t$ and environment $\Gamma$ in $\text{Env}(T)$.

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

(ii) \(\sigma\) generates\(\Gamma\) \(tr\) \(\iff\) \(\langle\tau\rangle\) \(\sigma\) generates\(\Gamma\) \(tr\),

(iii) \(\sigma\) generates\(\Gamma\) \(tr\) \(\land\) \(\llbracket\text{cond}\rrbracket\) \(\Gamma\) \(\iff\) \(\langle\text{cond}\rangle\) \(\sigma\) generates\(\Gamma\) \(tr\),

(iv) \(e \in \text{Insts}(\epsilon) \land \sigma\) generates\(\Gamma\) \(\text{Match}(\epsilon, e)\) \(tr\) \(\iff\) \(\langle\epsilon\rangle\) \(\hat{\sigma}\) generates\(\Gamma\) \(\langle\epsilon\rangle\) \(tr\),

where if \(\epsilon\) is a visible symbolic event of the form \(c\hat{\eta}_1x_1: X_1 \ldots \hat{\eta}_kx_k: X_k\) (recall that, by Remark 4.3.1, \(\text{\$\non-t}(\epsilon) = \\{\}\)), then

\[
\text{Insts}(\epsilon) = \{c.v_1 \ldots v_k \mid \forall i \in \text{\$t}(\epsilon) \cup \text{?-t}(\epsilon) \bullet v_i \in T \land \forall i \in \text{!t}(\epsilon) \bullet v_i = \Gamma(x_i)\}
\]

and

\[
\text{Match}(\epsilon, c.v_1 \ldots v_k) = \{x_i \mapsto v_i \mid i \in \text{\$t}(\epsilon) \cup \text{?-t}(\epsilon)\}.
\]

Observe that rule (ii) indicates that we essentially ignore any \(\tau\)'s present in the symbolic traces.

For brevity we write \(\sigma, \sigma'\) generates\(\Gamma\) \(tr\) to mean that both \(\sigma\) generates\(\Gamma\) \(tr\) and \(\sigma'\) 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 \(\text{Proc}(T)\) immediately after a trace \(tr\) matches a visible symbolic event \(\epsilon\) if there exists a symbolic trace \(\sigma\) such that \(\sigma\langle\epsilon\rangle\) is in \(\text{SymbolicTraces}(\text{Proc}(t))\) and \(\sigma\langle\epsilon\rangle\) generates\(\Gamma\) \(\langle\epsilon\rangle\) \(tr\).

### 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 (SSLTs). 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 \textbf{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

\[ \text{Spec}(\hat{T}) \subseteq \text{Impl}(\hat{T}) \text{ implies that } \forall T \supseteq \hat{T} \bullet \text{Spec}(T) \subseteq \text{Impl}(T), \]

where \( \hat{T} = \{0 \ldots B\} \) is a small and fixed type for some non-negative integer \( B \), our theory shows that (provided certain criteria are met)

\[ \text{Spec}(\hat{T}) \subseteq \phi(\text{Impl}(T)) \text{ implies that } \text{Spec}(T) \subseteq \text{Impl}(T), \] (5.1)

for all \( T \) such that \( \hat{T} \subseteq T \), where \( \phi \) is a \( B \)-collapsing function:

**Definition 5.0.1.** A \( B \)-collapsing function is a function \( \phi : T \rightarrow \{0 \ldots B\} \) such that

- \( \phi(v) = v \) for all \( v \) in \( \{0 \ldots B - 1\} \),
- \( \phi(v) = B \) for all \( v \) in \( \{B \ldots \#T - 1\} \).

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 $\phi(\text{Impl}(T))$, even though it can only communicate values of type $t$ that are in $\phi(T)$, still depends on $T$. For example, if

$$\text{Impl}(t) = \left\lfloor \left\lfloor \left\lfloor i \in t \bullet c.i \rightarrow \text{STOP} \right\rfloor \right\rfloor$$

and $\phi(v) = 0$ for all $v$, then for every $T$, $\phi(\text{Impl}(T))$ 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 $\text{Abstr}$ such that

$$\text{Abstr} \sqsubseteq \phi(\text{Impl}(T))$$

for all sufficiently large $T$. Then, it is enough to test

$$\text{Spec}(\hat{T}) \sqsubseteq \text{Abstr}$$

to deduce that

$$\text{Spec}(\hat{T}) \sqsubseteq \phi(\text{Impl}(T))$$

and then use (5.1). We will present an abstraction method for constructing suitable $\text{Abstr}$ 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 $\text{SeqNorm}$ 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 $\text{SeqNorm}$ 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 \( \text{Proc}(t) \), let \( \alpha_1, \alpha_2, \ldots \) be the prefix constructs of \( \text{Proc}(t) \) (here, two prefix constructs are regarded as different if they appear in different places in \( \text{Proc}(t) \)). Let \( T \) be a given instantiation of type \( t \). Then, for every pair of a trace \( tr \) and a visible event \( e \) such that \( tr \hat{\langle} e \hat{\rangle} \) is a trace of \( \text{Proc}(T) \), there must be at least one \( \alpha_i \) that gives rise to \( e \) immediately after \( tr \). We then say that \( \alpha_i \) is matched by \( e \) (or that \( e \) matches \( \alpha_i \)). We define visible symbolic events to match syntax constructs in an analogous way.

**Example 5.1.2.** Let

\[
P(t) = c?x.t \rightarrow STOP
\]

\[
c\$x.t \rightarrow c.x \rightarrow STOP.
\]

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 \( \text{SeqNorm} \). We will show (in Corollary 5.1.15) that for processes that do satisfy \( \text{SeqNorm} \), each event (after a given trace) matches a unique construct.

The following lemma shows that (for a process that satisfies \( \text{SeqNorm} \)), each visible or conditional symbolic event leads to a unique symbolic state.

**Lemma 5.1.3.** Suppose that \( \text{Proc}(t) \) satisfies \( \text{SeqNorm} \). Let \( e \) be a visible or conditional symbolic event and suppose that

\[
\text{Proc}(t) \stackrel{\sigma_1}{\rightarrow_s} \text{Proc}'_1(t) \quad \text{and} \quad \text{Proc}(t) \stackrel{\sigma_2}{\rightarrow_s} \text{Proc}'_2(t),
\]

where \( \sigma_1 = \tau^a \hat{\langle} e \hat{\rangle} \) for \( a \geq 0 \) and \( \sigma_2 = \tau^b \hat{\langle} e \hat{\rangle} \) for \( b \geq 0 \). Then \( \text{Proc}'_1(t) = \text{Proc}'_2(t) \).

A main reason for introducing the \( \text{SeqNorm} \) 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

\[
\text{Proc}(t) = \sqcap x \in X \bullet a \rightarrow c.x \rightarrow STOP.
\]

Then any sensible firing rule for \( \sqcap \) would imply that after performing the symbolic trace \( \langle a \rangle \), \( \text{Proc}(t) \) 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

$$\text{Proc}_{x,y}(t) = (\text{if } x = y \text{ then } a \rightarrow \text{STOP} \text{ else STOP})$$

$$\text{if } x = y \text{ then } b \rightarrow \text{STOP} \text{ else STOP}$$

can do the conditional symbolic event "$x = y$" and reach either the symbolic state $a \rightarrow \text{STOP}$ or the symbolic state $(\text{if } x = y \text{ then } a \rightarrow \text{STOP} \text{ else STOP}) \circ b \rightarrow \text{STOP}$. This is why we banned all the behaviours described above by assuming that process syntax satisfies \text{SeqNorm}.\text{\textsuperscript{1}}

**Proof of Lemma 5.1.3:** We prove the result by a structural induction on $\text{Proc}(t)$. The most interesting cases are those for prefix and external choice.

**STOP**

This case holds trivially.

**Prefix**

Suppose $\text{Proc}(t) = \alpha \rightarrow \text{Proc}'(t)$ for some construct $\alpha = c_1^x x_1 : X_1, \ldots, c_k^x x_k : X_k$ and some process syntax $\text{Proc}'(t)$. Clearly $\epsilon$ must be a visible symbolic event matching $\alpha$, say $c_1^x x_1 : X_1', \ldots, c_k^x x_k' : X_k'$. We consider two different cases, corresponding to the number of nondeterministic selections over non-$t$ types of $\alpha$.

**Case 1.** Suppose that $\#\$\text{non-t}(\alpha) = 0$.

Then, by Symbolic Prefix Rule 1 (p. 78), it must be that

$$\sigma_1 = \sigma_2 = \langle \epsilon \rangle$$

with $\epsilon$ in $\text{Comms}^{\text{non-t}}(\alpha)$ and

$$\text{Proc}'_1(t) = \text{Proc}'_2(t) = \text{Proc}'(t)[x_i/x_i \mid i \in ?^{\text{non-t}}(\alpha)]$$.

**Case 2.** Suppose that $\#\$\text{non-t}(\alpha) > 0$.

Then, Symbolic Prefix Rule 2 (p. 78) implies that the only symbolic transitions in $\text{Proc}(t)$ are

$$\text{Proc}(t) \xrightarrow{\tau} s \left(\text{Replace}_{\text{non-t}}(\alpha) \rightarrow \text{Proc}'(t)\right) [v_i/x_i \mid i \in \$^{\text{non-t}}(\alpha)]$$

\text{\textsuperscript{1}}\text{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 $\text{dom}(v) = \{1 \ldots k\}$ and such that if $i$ is in $\text{\$non-t(}\alpha\text{\$)$, then $v_i$ is in $X_i$. We are guaranteed that

$$\#\text{\$non-t(}\text{\{\text{\$non-t(}\alpha\text{\$)\}} = 0,$$

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 $j \in \{1, 2\}$,

$$\text{Proc}_j(t) = (\text{Proc}(t)[v_j^i/x_i | i \in \text{\$non-t(}\alpha\text{\$})][x'_i/x_i | i \in \text{\?non-t(}\alpha\text{\$})]$$

and

$$\epsilon \in \text{Comms}^{\text{\$non-t(}\text{(\text{\$non-t(}\alpha\text{\$)\})}}.$$ 

Then, thanks to the definition of $\text{Comms}^{\text{\$non-t(}\text{(\text{\$non-t(}\alpha\text{\$)\})}}$, $v^1$ and $v^2$ are equal under domain restriction to $\text{\$non-t(}\alpha\text{\$)}$. Therefore, $\text{Proc}_1(t) = \text{Proc}_2(t)$.

**External choice**

Suppose that $\text{Proc}(t) = P(t) \boxdot Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. Since $\text{Proc}(t)$ satisfies $\text{SeqNorm}$, we know that neither $P(t)$ nor $Q(t)$ contains a conditional choice on $t$ before a prefix. Therefore $\epsilon$ cannot be a conditional symbolic event, so must be a visible symbolic event. $\text{SeqNorm}$ implies that the channels of the initial visible symbolic events of $P(t)$ and $Q(t)$ are disjoint, so we have that either

$$P(t) \xrightarrow{\tau_s} \text{or} \quad Q(t) \xrightarrow{\tau_s},$$

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

$$P(t) \xrightarrow{\tau_s} s \quad \text{or} \quad Q(t) \xrightarrow{\tau_s} s.$$

Even though there may be some $\tau$’s, contributed by $Q(t)$, in the symbolic trace of $P(t)$ leading to $\text{Proc}_1(t)$, the uniqueness of $P'(t)$ implies that $\text{Proc}_1'(t) = \text{Proc}_2'(t) = P'(t)$.

**Internal choice**

Suppose that $\text{Proc}(t) = P(t) \cap Q(t)$ for some process syntax $P(t)$ and $Q(t)$. Then the only symbolic transitions from $\text{Proc}(t)$ are

$$\text{Proc}(t) \xrightarrow{\tau_s} P(t) \quad \text{and} \quad \text{Proc}(t) \xrightarrow{\tau_s} Q(t).$$

Therefore $\sigma_1 = \langle \tau \rangle^* \rho_1$ and $\sigma_2 = \langle \tau \rangle^* \rho_2$ for some $\rho_1, \rho_2 \in \text{SymbolicTraces}(P(t)) \cup \text{SymbolicTraces}(Q(t))$ such that $\rho_1, \rho_2 \in \{\tau\}^* \langle \epsilon \rangle$. 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 \( \text{Proc}(t) = \text{if } \text{cond} \text{ then } P(t) \text{ else } Q(t) \) for some boolean condition \( \text{cond} \) and some process syntaxes \( P(t) \) and \( Q(t) \). If \( \text{cond} \) is not in \( \text{Cond} \), then \( \text{cond} \) immediately evaluates to \( \text{True} \) or \( \text{False} \) and the result is implied by the inductive hypothesis for \( P(t) \) or \( Q(t) \), respectively. If \( \text{cond} \) is in \( \text{Cond} \), then \( \sigma_1 = \sigma_2 = \langle \text{cond} \rangle \) and \( \text{Proc'}_1(t) = \text{Proc'}_2(t) = P(t) \), or \( \sigma_1 = \sigma_2 = \langle \neg \text{cond} \rangle \) and \( \text{Proc'}_1(t) = \text{Proc'}_2(t) = Q(t) \).

Binding
Suppose that \( \text{Proc}(t) = X(t) \) 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

\[
\sigma_1 = \langle \tau \rangle^* \rho_1 \quad \text{and} \quad \sigma_2 = \langle \tau \rangle^* \rho_2
\]

for some \( \rho_1, \rho_2 \) 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 \( \sigma \) and \( \sigma' \) be two symbolic traces. Then \( \sigma \) and \( \sigma' \) are non-\( \tau \) equivalent, written \( \sigma \equiv_{\text{non-}\tau} \sigma' \), 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 \( \text{Proc}(t) \) satisfies \textbf{SeqNorm}. Suppose further that

\[
\text{Proc}(t) \xrightarrow{\sigma} P(t) \quad \text{and} \quad \text{Proc}(t) \xrightarrow{\sigma'} Q(t).
\]

Then if neither \( \sigma \) nor \( \sigma' \) ends with a \( \tau \) and \( \sigma \equiv_{\text{non-}\tau} \sigma' \), then \( P(t) = Q(t) \).

**Proof:** By induction on the number of visible and conditional symbolic events of \( \sigma \) and \( \sigma' \) (which must be equal, since \( \sigma \equiv_{\text{non-}\tau} \sigma' \)), 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 \( \text{Proc}(t) \) satisfies \textbf{SeqNorm}. Then, if \( \text{Proc}(t) \xrightarrow{\sigma} \xrightarrow{\epsilon} \) and \( \text{Proc}(t) \xrightarrow{\sigma'} \xrightarrow{\epsilon'} \), where \( \sigma, \sigma' \in (\text{Cond} \cup \{ \tau \})^* \) are such that \( \sigma \equiv_{\text{non-}\tau} \sigma' \) and \( \epsilon = c;_{x_1:X_1} \cdots ;_{x_k:X_k} \) and \( \epsilon' = c';_{x'_1:X'_1} \cdots ;_{x'_k:X'_k} \) are visible symbolic events, then

(i) if the channels of \( \epsilon \) and \( \epsilon' \) are identical (i.e. \( c = c' \)), then the parts of \( \epsilon \) and \( \epsilon' \) involving type \( t \) are equal, i.e.

\[
\forall i \in $t(\epsilon) \cup ?t(\epsilon) \cup !t(\epsilon) = $t(\epsilon') \cup ?t(\epsilon') \cup !t(\epsilon')
\]

and

\[
\forall i \in $t(\epsilon) \cup ?t(\epsilon) \cup !t(\epsilon) \cdot \tilde{8}_i = \tilde{8}'_i \land x_i = x'_i \land X_i = X'_i
\]

95
(ii) if $\text{Insts}_\Gamma(\epsilon) \cap \text{Insts}_\Gamma(\epsilon') \neq \{\} \quad \text{for some environment } \Gamma$, then $\epsilon = \epsilon'$.

**Proof:** Firstly, observe that if $\text{Insts}_\Gamma(\epsilon) \cap \text{Insts}_\Gamma(\epsilon') \neq \{\}$, then the channels of $\epsilon$ and $\epsilon'$ must be the same. Hence in both cases $\epsilon = \epsilon'$. Since every channel has a fixed structure of the communication along it, the number of components of $\epsilon$ and $\epsilon'$ must be identical, i.e. $k = l$. We now prove both clauses using a structural induction on $\text{Proc}(t)$. The most interesting case is that for prefix.

**STOP**

The result holds trivially in this case.

**Prefix**

Suppose that $\text{Proc}(t) = \alpha \rightarrow \text{Proc}'(t)$ for some construct $\alpha$ of the form $c_{\xi_1} x_1 : X_1 \ldots c_{\xi_k} x_k : X_k$ and some process syntax $\text{Proc}'(t)$. We perform a case analysis on the number of nondeterministic selections over non-$t$ types of $\alpha$.

**Case 1.** Suppose that $\#_{\text{non-}t}(\alpha) = 0$.

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

$$\sigma = \sigma' = ()$$

and that both $\epsilon$ and $\epsilon'$ are in $\text{Comms}_{\text{non-}t}(\alpha)$. By the definition of $\text{Comms}_{\text{non-}t}$ (p. 78), $\epsilon$ and $\epsilon'$ may differ only in the values of deterministic inputs of non-$t$ types of $\alpha$. Hence, clause (i) of the lemma holds. To prove clause (ii), we let $c.v_1 \ldots v_k$ be a common member of $\text{Insts}_\Gamma(\epsilon)$ and $\text{Insts}_\Gamma(\epsilon')$. Then the definition of $\text{Insts}_\Gamma$ (p. 88) implies that

$$\forall i \in \{!\epsilon\} \bullet \xi_i = \xi'_i = ! \land x_i = x'_i \land v_i = \Gamma(x_i) \land X_i = X'_i = \text{null}.$$  \hfill (5.2)

The definition of $\text{Comms}_{\text{non-}t}$ implies that $\text{?}_{\text{non-}t}(\alpha) \subseteq \{!\epsilon\}$, so

$$\forall i \in \text{?}_{\text{non-}t}(\alpha) \bullet \xi_i = \xi'_i = ! \land x_i = x'_i \land X_i = X'_i = \text{null}.$$  \hfill (5.2)

This, combined with clause (i) of the lemma, (5.2) and the fact that $\#_{\text{non-}t}(\alpha) = 0$, implies that $\epsilon = \epsilon'$.

**Case 2.** Suppose that $\#_{\text{non-}t}(\alpha) > 0$.

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 $\text{Proc}(t)$ are

$$\text{Proc}(t) \xrightarrow{s} (\text{Replace}_{\#_{\text{non-}t}(\alpha)} \rightarrow \text{Proc}'(t)) [v_i/x_i \mid i \in \#_{\text{non-}t}(\alpha)]$$

$$= \text{Replace}_{\#_{\text{non-}t}(\alpha)}[v_i/x_i \mid i \in \#_{\text{non-}t}(\alpha)]$$

$$\rightarrow \text{Proc}'(t) [v_i/x_i \mid i \in \#_{\text{non-}t}(\alpha)]$$

for functions $v$ such that $\text{dom}(v) = \#_{\text{non-}t}(\alpha)$, and if $i$ is in $\#_{\text{non-}t}(\alpha)$, then $v_i$ is in $X_i$.

Clearly,

$$\#_{\text{non-}t}(\text{Replace}_{\#_{\text{non-}t}(\alpha)}[v_i/x_i \mid i \in \#_{\text{non-}t}(\alpha)]) = 0$$

96
for every such function \(v\), so Symbolic Prefix Rule 1 (p. 78) implies that
\[
\begin{align*}
\epsilon &\in \text{Comms}^{\text{non-t}}\left(\left(\text{Replace}_{\text{SeqNorm}}^{\text{non-t}}(\alpha)\right)\left[v_i/x_i \mid i \in \#^{\text{non-t}}(\alpha)\right]\right), \\
\epsilon' &\in \text{Comms}^{\text{non-t}}\left(\left(\text{Replace}_{\text{SeqNorm}}^{\text{non-t}}(\alpha)\right)\left[v'_i/x_i \mid i \in \#^{\text{non-t}}(\alpha)\right]\right)
\end{align*}
\]
for some functions \(v\) and \(v'\) such that \(\text{dom}(v) = \text{dom}(v') = \#^{\text{non-t}}(\alpha)\) and if \(i\) is in \(\#^{\text{non-t}}(\alpha)\), then \(v_i\) and \(v'_i\) are in \(X_i\). The definition of \(\text{Comms}^{\text{non-t}}\) (p. 78) implies that the parts of \(\epsilon\) and \(\epsilon'\) 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 \(\text{Insts}_t(\epsilon)\) and \(\text{Insts}_t(\epsilon')\). Then, the definition of \(\text{Insts}_t\) (p. 88) implies that
\[
\forall i \in \text{dom}(\epsilon) \bullet \ #_i = \ #_i' = ! \wedge x_i = x'_i \wedge \Gamma(x_i) \wedge X_i = X'_i = \text{null}.
\] (5.3)
However, the definition of \(\text{Comms}^{\text{non-t}}\) implies that
\[
\begin{align*}
?^{\text{non-t}}(\alpha) &= \ ?^{\text{non-t}}\left(\left(\text{Replace}_{\text{SeqNorm}}^{\text{non-t}}(\alpha)\right)\left[v_i/x_i \mid i \in \#^{\text{non-t}}(\alpha)\right]\right) \subseteq !^{\epsilon}, \\
\#^{\text{non-t}}(\alpha) &\subseteq !^{\text{non-t}}\left(\left(\text{Replace}_{\text{SeqNorm}}^{\text{non-t}}(\alpha)\right)\left[v_i/x_i \mid i \in \#^{\text{non-t}}(\alpha)\right]\right) \subseteq !^{\epsilon}.
\end{align*}
\]
Hence,
\[
\forall i \in \#^{\text{non-t}}(\alpha) \cup ?^{\text{non-t}}(\alpha) \bullet \ #_i = \ #_i' \wedge x_i = x'_i \wedge X_i = X'_i.
\]
This, combined with clause (i) of the lemma and (5.3), implies that \(\epsilon = \epsilon'\).

**External choice**
Suppose that \(\text{Proc}(t) = P(t) \Box Q(t)\) for some process syntaxes \(P(t)\) and \(Q(t)\). Since \(\text{Proc}(t)\) satisfies \textbf{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, \textbf{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 \(\epsilon\) and \(\epsilon'\) 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 \(\text{Proc}(t) = P(t) \cap Q(t)\) for some process syntaxes \(P(t)\) and \(Q(t)\). Since \(\text{Proc}(t)\) satisfies \textbf{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 \(\cap\) (p. 79) imply that the only symbolic transitions from \(\text{Proc}(t)\) are
\[
\text{Proc}(t) \xrightarrow{\tau} s \quad P(t) \quad \text{and} \quad \text{Proc}(t) \xrightarrow{\tau} s \quad Q(t).
\]
Therefore \( \sigma = \langle \tau \rangle \hat{\rho} \) and \( \sigma' = \langle \tau \rangle \hat{\rho}' \) 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 \( \text{Proc}(t) = \text{if} \ cond \ \text{then} \ P(t) \ \text{else} \ Q(t) \) for some boolean condition \( \text{cond} \) and some process syntaxes \( P(t) \) and \( Q(t) \). If \( \text{cond} \) is not in \( \text{Cond} \), then \( \text{cond} \) immediately evaluates to either \( \text{True} \) or \( \text{False} \) and the result is implied by the inductive hypothesis for \( P(t) \) or \( Q(t) \), respectively. So suppose that \( \text{cond} \) is in \( \text{Cond} \). Then, the SSOS firing rules for conditional choice (p. 80) imply that the only symbolic transitions in \( \text{Proc}(t) \) are

\[
\text{Proc}(t) \xrightarrow{\text{cond}} s \quad P(t) \quad \text{and} \quad \text{Proc}(t) \xrightarrow{\neg \text{cond}} s \quad Q(t).
\]

The fact that \( \sigma \equiv_{\text{non-}\tau} \sigma' \) implies that either

\[
\sigma = \langle \text{cond}\rangle \hat{\rho} \quad \text{and} \quad \sigma' = \langle \text{cond}\rangle \hat{\rho}'
\]

or

\[
\sigma = \langle \neg \text{cond}\rangle \hat{\rho} \quad \text{and} \quad \sigma' = \langle \neg \text{cond}\rangle \hat{\rho}'
\]

for some \( \rho, \rho' \in (\text{Cond} \cup \{ \tau \})^* \) such that \( \rho \equiv_{\text{non-}\tau} \rho' \). Without loss of generality we assume the former. Then

\[
P(t) \xrightarrow{\rho} s \quad \epsilon_s \quad \text{and} \quad P(t) \xrightarrow{\rho'} s \quad \epsilon'_s.
\]

The result is now implied by the inductive hypothesis for \( P(t) \).

**Binding**

Suppose that \( \text{Proc}(t) = X(t) \) 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

\[
\sigma_1 = \langle \tau \rangle \hat{\rho}_1 \quad \text{and} \quad \sigma_2 = \langle \tau \rangle \hat{\rho}_2
\]

for some \( \rho_1, \rho_2 \in \text{SymbolicTraces}(P(t)) \cap (\text{Cond} \cup \{ \tau \})^* \) such that \( \rho_1 \equiv_{\text{non-}\tau} \rho_2 \). Then, the result is implied by the inductive hypothesis for \( P(t) \).

**Definition 5.1.7.** Let \( \epsilon = c \otimes x_1 : X_1 \ldots \otimes x_k : X_k \) and \( \epsilon' = c' \otimes x'_1 : X'_1 \ldots \otimes x'_k : X'_k \) be two visible symbolic events. Then \( \epsilon \) and \( \epsilon' \) are \( \text{non-t equivalent} \), written \( \epsilon \equiv_{\text{non-t}} \epsilon' \), if the channels and non-\( t \) parts of \( \epsilon \) and \( \epsilon' \) are identical, i.e. if \( c = c' \) and

\[
\forall i \in \text{non-t}(\epsilon) \cup \text{non-t}(\epsilon') \quad \otimes_{\text{non-t}(\epsilon)} \cup \text{non-t}(\epsilon) \cup \text{non-t}(\epsilon') \cup \text{non-t}(\epsilon') \quad \bullet
\]

\[
\text{S}_i = \text{S}_i' \land x_i = x'_i \land X_i = X'_i.
\]
We lift $\equiv_{\text{non-}t}$ to sequences of visible symbolic events using pointwise application. Finally, we define two symbolic traces, $\sigma$ and $\sigma'$ to be non-$t$ equivalent, written $\sigma \equiv_{\text{non-}t} \sigma'$ if the restrictions of $\sigma$ and $\sigma'$ 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 $\tau$, and
- are non-$t$ equivalent,

are unique up to $\tau$’s, and lead to unique symbolic states.

**Lemma 5.1.8.** Suppose that Proc$(t)$ satisfies SeqNorm. Suppose further that

$$\text{Proc}(t) \xrightarrow{\sigma} s \text{ Proc}(t) \text{ and } \text{Proc}(t) \xrightarrow{\sigma'} s \text{ Proc}(t),$$

where $\sigma, \sigma \in \text{SymbolicTraces}(\text{Proc}(t)) \cap (\text{Visible} \cup \{\tau\})^*$ do not end with a $\tau$ and $\sigma \equiv_{\text{non-}t} \sigma'$. Then $\sigma \equiv_{\text{non-}t} \sigma'$ and $P(t) = P'(t)$.

**Proof:** We prove the result by an induction on the number of visible events in $\sigma$ and $\sigma'$ (which must be equal).

**Base case.** Suppose that $\sigma = \sigma' = \langle \rangle$. Then, clearly $\sigma \equiv_{\text{non-}t} \sigma'$ and $P(t) = P'(t) = \text{Proc}(t)$.

**Inductive case.** Suppose that $\sigma = \sigma_0^\prec \tau^a \prec (\epsilon)$ and $\sigma' = \sigma'_0^\prec \tau^b \prec (\epsilon')$ for some symbolic traces $\sigma_0$ and $\sigma'_0$ that do not end with a $\tau$, integers $a, b \geq 0$ and visible symbolic events $\epsilon$ and $\epsilon'$. Then

$$\text{Proc}(t) \xrightarrow{\sigma_0} s \text{ Proc}(t) \xrightarrow{\tau^a \prec (\epsilon)} s \text{ Proc}(t) \text{ and } \text{Proc}(t) \xrightarrow{\sigma'_0} s \text{ Proc}(t) \xrightarrow{\tau^b \prec (\epsilon')} s \text{ Proc}(t)$$

for some symbolic states $P_0(t)$ and $P'_0(t)$. Since $\sigma \equiv_{\text{non-}t} \sigma'$, we have that $\sigma_0 \equiv_{\text{non-}t} \sigma'_0$ and $\epsilon \equiv_{\text{non-}t} \epsilon'$. Therefore, by the inductive hypothesis $\sigma_0 \equiv_{\text{non-}t} \sigma'_0$ and $P_0(t) = P'_0(t)$. In addition, by Lemma 5.1.6, the parts of $\epsilon$ and $\epsilon'$ involving type $t$ must be identical. Hence, $\epsilon = \epsilon'$. This, combined with the fact that $\sigma_0 \equiv_{\text{non-}t} \sigma'_0$, implies that $\sigma \equiv_{\text{non-}t} \sigma'$. 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 $\tau$’s), then all its initial events (after $\tau$’s) must be that conditional or its negation.

**Lemma 5.1.9.** Suppose that Proc$(t)$ satisfies SeqNorm. Then, if

$$\text{Proc}(t) \xrightarrow{\sigma} s \text{ cond} \xrightarrow{\alpha} s \text{ and } \text{Proc}(t) \xrightarrow{\sigma'} s \text{ cond} \xrightarrow{\alpha} s,$$

where $\sigma, \sigma' \in \{\tau\}^*$, cond $\in \text{Cond}$ and $\alpha \neq \tau$, then $\alpha \in \{\text{cond}, \neg\text{cond}\}$.
This means that the only transitions available in $\text{ProcSymbolicTraces}$ are $\text{ProcSymbolicTraces}_\sigma$. 

**Base case.** Suppose that the result holds for all process syntaxes and some $\kappa$ of conditional symbolic events within a given symbolic trace. We prove the result using an induction on $\kappa$.

**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 $\sigma = \tau a \cdot (\text{cond}) \cdot \rho$ for some $a \geq 0$, some $\text{cond} \in \text{Cond}$ and some $\rho \in (\text{Cond} \cup \{\tau\})^*$ with $\kappa(\rho) = k$. Arguing similarly as in the base case, $\kappa(\sigma') > 0$. Therefore, $\sigma' = \tau b \cdot (\text{cond}') \cdot \rho'$ for some $b \geq 0$, some $\text{cond}' \in \text{Cond}$ and some $\rho' \in (\text{Cond} \cup \{\tau\})^*$. By Lemma 5.1.9, $\epsilon \in \{\text{cond}, \neg \text{cond}\}$. This is a contradiction as $\epsilon \in \text{Visible}$. Therefore $\kappa(\sigma') = 0$, which means that $\sigma' \in \{\tau\}^*$. Hence $\sigma \equiv_{\text{non-}\tau} \sigma'$.

**Proof:** Since $\text{Proc}(t)$ satisfies $\text{SeqNorm}$, 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) $\text{Proc}(t)$ is a conditional choice on $t$, where the boolean condition is equal to $\text{cond}$ or $\neg \text{cond}$;

(ii) $\text{Proc}(t)$ is a process identifier bound by the global environment $E$ to a conditional choice like that in clause (i) or (iii); or

(iii) $\text{Proc}(t)$ is a conditional choice whose boolean condition immediately evaluates to $\text{True}$ or $\text{False}$ and appropriate branch is a process syntax as in clause (i) or (ii).

This means that the only transitions available in $\text{Proc}(t)$ are $\text{Proc}(t) \xrightarrow{\tau} \text{ProcSymbolicTraces}_s \text{cond}_s$ and $\text{Proc}(t) \xrightarrow{\tau} \text{ProcSymbolicTraces}_s \neg \text{cond}_s$.

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 $\text{True}$ in the initial environment.

**Lemma 5.1.10.** Suppose that $\text{Proc}(t)$ satisfies $\text{SeqNorm}$. Let $\sigma \cdot \langle \epsilon \rangle, \sigma' \cdot \langle \epsilon' \rangle \in \text{SymbolicTraces}(\text{Proc}(t))$ be such that $\sigma, \sigma' \in (\text{Cond} \cup \{\tau\})^*$, $\epsilon, \epsilon' \in \text{Visible}$, $\sigma \cdot \langle \epsilon \rangle$ generates$_\Gamma \langle \epsilon \rangle$ and $\sigma' \cdot \langle \epsilon' \rangle$ generates$_\Gamma \langle \epsilon' \rangle$ for some environment $\Gamma$ and some visible events $\epsilon$ and $\epsilon'$. Then $\sigma \equiv_{\text{non-}\tau} \sigma'$.

**Proof:** Let $\kappa : \text{SymbolicTraces}(\text{Proc}(t)) \rightarrow \mathbb{N}$ 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 $\text{Proc}(t) \rightarrow^* \langle \epsilon \rangle$. Suppose $\kappa(\sigma') > 0$. Then $\sigma' = \tau a \cdot (\text{cond}) \cdot \rho$ for some $a \geq 0$, some $\text{cond} \in \text{Cond}$ and some symbolic trace $\rho \in (\text{Cond} \cup \{\tau\})^*$. Therefore $\text{Proc}(t) \rightarrow^* \text{ProcSymbolicTraces}_s \text{cond}_s$. By Lemma 5.1.9, $\epsilon \in \{\text{cond}, \neg \text{cond}\}$. This is a contradiction as $\epsilon \in \text{Visible}$. Therefore $\kappa(\sigma') = 0$, which means that $\sigma' \in \{\tau\}^*$. Hence $\sigma \equiv_{\text{non-}\tau} \sigma'$.

**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 $\sigma = \tau a \cdot (\text{cond}) \cdot \rho$ for some $a \geq 0$, some $\text{cond} \in \text{Cond}$ and some $\rho \in (\text{Cond} \cup \{\tau\})^*$ with $\kappa(\rho) = k$. Arguing similarly as in the base case, $\kappa(\sigma') > 0$. Therefore, $\sigma' = \tau b \cdot (\text{cond}') \cdot \rho'$ for some $b \geq 0$, some $\text{cond}' \in \text{Cond}$ and some $\rho' \in (\text{Cond} \cup \{\tau\})^*$. By Lemma 5.1.9, $\text{cond}' \in \{\text{cond}, \neg \text{cond}\}$. Since $\sigma \cdot \langle \epsilon \rangle$ generates$_\Gamma \langle \epsilon \rangle$ and $\sigma' \cdot \langle \epsilon' \rangle$ generates$_\Gamma \langle \epsilon' \rangle$, $\text{cond}$ and $\text{cond}'$ must both evaluate to $\text{True}$ within $\Gamma$, because there are no visible symbolic events within $\sigma$ and $\sigma'$ before $\text{cond}$ and $\text{cond}'$, respectively, that could modify the environment $\Gamma$. So it must be that $\text{cond} = \text{cond}'$. By Lemma 5.1.3, there is a unique state $P(t)$ such that $\text{Proc}(t) \rightarrow^* \text{ProcSymbolicTraces}_s P(t)$. Hence, $\rho \cdot \langle \epsilon \rangle$ and $\rho' \cdot \langle \epsilon' \rangle$ are both symbolic traces of $P(t)$.
such that \( \rho^* \langle e \rangle \) generates\( \Gamma \) \( \langle e \rangle \) and \( \rho'^* \langle e' \rangle \) generates\( \Gamma \) \( \langle e' \rangle \). In addition, \( \kappa(\rho) = k \). Therefore, by the inductive hypothesis, \( \rho \equiv_{\text{non-}\tau} \rho' \), which implies that \( \sigma \equiv_{\text{non-}\tau} \sigma' \).

The following lemma shows that each concrete visible transition leads to a unique process syntax and environment.

**Lemma 5.1.11.** Suppose that \( \text{Proc}(t) \) satisfies SeqNorm. Suppose further that

\[
(\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{\tau}^* \langle c \rangle \quad \text{and} 
(\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{\tau}^* \langle c' \rangle,
\]

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 \( \sigma, \sigma' \) and visible symbolic events \( \epsilon = c \epsilon_1 x_1 : X_1 \ldots \epsilon_k x_k : X_k \) and \( \epsilon' = c' \epsilon'_1 x'_1 : X'_1 \ldots \epsilon'_k x'_k : X'_k \) such that

- \( \text{Proc}(t) \xrightarrow{\sigma} \epsilon \xrightarrow{s} P(t) \) and \( \text{Proc}(t) \xrightarrow{\sigma'} \epsilon' \xrightarrow{s} Q(t) \), and
- \( \sigma^* \langle \epsilon \rangle \) generates\( \Gamma_{\text{init}} \) \( \langle \epsilon \rangle \) and \( \sigma'^* \langle \epsilon' \rangle \) generates\( \Gamma_{\text{init}} \) \( \langle \epsilon \rangle \).

Then it must be that \( \sigma, \sigma' \in (\text{Cond} \cup \{ \tau \})^* \). So, by Lemma 5.1.10, \( \sigma \equiv_{\text{non-}\tau} \sigma' \).

From the definition of the generates relation (p. 87) we have that \( e \in \text{Insts}_{\text{init}}(\epsilon) \cap \text{Insts}_{\text{init}}(\epsilon') \), so Lemma 5.1.6 implies that \( \epsilon = \epsilon' \). Hence, \( \sigma^* \langle \epsilon \rangle \equiv_{\text{non-}\tau} \sigma'^* \langle \epsilon' \rangle \) 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

\[
\Gamma = \Gamma_{\text{init}} \oplus \{ x_i \mapsto v_i \mid i \in \{ s(\epsilon) \cup ?(\epsilon) \} \}
\]

and

\[
\Gamma' = \Gamma_{\text{init}} \oplus \{ x'_i \mapsto v_i \mid i \in \{ s(\epsilon') \cup ?(\epsilon') \} \}.
\]

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 \( \tau \). The following lemma establishes this formally.

**Lemma 5.1.12.** Suppose that \( \text{Proc}(t) \) satisfies SeqNorm. Suppose further that

\[
(\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{s} (P(t), \Gamma) \quad \text{and} 
(\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{s'} (Q(t), \Gamma'),
\]

where \( s, s' \) do not end with a \( \tau \) 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 \textbf{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 \( \text{Proc}(t) \) satisfies \textbf{SeqNorm}. Then, if \( \sigma, \sigma' \in \text{SymbolicTraces}(\text{Proc}(t)) \) are such that \( \sigma \text{ generates}_\Gamma tr \) and \( \sigma' \text{ generates}_\Gamma tr \), and either \( \sigma = \sigma' = \langle \rangle \) or both \( \sigma \) and \( \sigma' \) end in a visible symbolic event, then \( \sigma \equiv_{\text{non-}\tau} \sigma' \).

**Proof:** We prove the result by an induction on the length of \( tr \).

**Base case.** Suppose that \( tr = \langle \rangle \). Then \( \sigma \text{ generates}_\Gamma \langle \rangle \) and \( \sigma' \text{ generates}_\Gamma \langle \rangle \). Therefore, by the definition of the \text{generates} relation (p. 87), \( \sigma \) and \( \sigma' \) cannot contain visible symbolic events. Hence, by the assumptions of this proposition, \( \sigma = \sigma' = \langle \rangle \), which means that \( \sigma \equiv_{\text{non-}\tau} \sigma' \).

**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 \( tr_{k+1} = tr_k \cdot \langle e \rangle \). There must also exist symbolic traces \( \sigma_1, \sigma'_1, \sigma_2, \) and \( \sigma'_2 \) such that

- either \( \sigma_1 = \sigma'_1 = \langle \rangle \) or both \( \sigma_1 \) and \( \sigma'_1 \) end in a visible symbolic event,
- \( \sigma = \sigma_1 \cdot \sigma_2 \) and \( \sigma' = \sigma'_1 \cdot \sigma'_2 \), and
- \( \sigma_1 \text{ generates}_\Gamma tr_k \) and \( \sigma'_1 \text{ generates}_\Gamma tr_k \).

Then, by the inductive hypothesis, \( \sigma_1 \equiv_{\text{non-}\tau} \sigma'_1 \). Hence, if \( P(t) \) and \( Q(t) \) are symbolic states such that \( \text{Proc}(t) \xrightarrow{\sigma_1}_s P(t) \) and \( \text{Proc}(t) \xrightarrow{\sigma'_1}_s Q(t) \), 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 \( (\text{Proc}(t), \Gamma) \xrightarrow{\omega} (P(t), \Gamma) \) for some unique environment \( \Gamma \) (uniqueness being guaranteed by Lemma 5.1.12). We now have that \( \sigma_2 \text{ generates}_\Gamma \langle e \rangle \) and \( \sigma'_2 \text{ generates}_\Gamma \langle e \rangle \). Hence, \( \sigma_2, \sigma'_2 \neq \langle \rangle \). Therefore, both \( \sigma_2 \) and \( \sigma'_2 \) must end in a visible symbolic event (since they are suffixes of \( \sigma \) and \( \sigma' \)). So, \( \sigma_2 = \rho \cdot \langle e \rangle \) and \( \sigma'_2 = \rho' \cdot \langle e' \rangle \) for some symbolic traces \( \rho, \rho' \in (\text{Cond} \cup \{ \tau \})^* \) and some visible symbolic events \( e \) and \( e' \). Hence, by Lemma 5.1.10, \( \rho \equiv_{\text{non-}\tau} \rho' \). Also from the definition of \text{generates} we have that \( e \in \text{Insts}_\Gamma(\rho) \cap \text{Insts}_\Gamma(\rho') \), so, by Lemma 5.1.6, \( e = e' \). Therefore, \( \sigma_2 \equiv_{\text{non-}\tau} \sigma'_2 \), and hence \( \sigma \equiv_{\text{non-}\tau} \sigma' \).

Observe that the part “either \( \sigma = \sigma' = \langle \rangle \) or both \( \sigma \) and \( \sigma' \) 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

\[
\text{Proc}(t) = \text{in?x:t?y:t} \rightarrow \text{if } x = y \text{ then } a \rightarrow \text{STOP} \\
\text{else STOP}
\]

102
Let $\sigma = \langle \text{in} ? x : t ? y : t, x = y \rangle$ and $\sigma' = \langle \text{in} ? x : t ? y : t, x = y \rangle$. Then $\sigma$ and $\sigma'$ are both symbolic traces of $\text{Proc}(t)$, and $\sigma, \sigma'$ generates $\{ \langle \text{in} .0 . \rangle \}$, but $\sigma \not\equiv_{\text{non-}\tau} \sigma'$. 

End of example.

The following corollary is an immediate consequence of Proposition 5.1.13 and says that, for a process that satisfies $\text{SeqNorm}$, 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 $\text{Proc}(t)$ satisfies $\text{SeqNorm}$. Then, if $\text{tr} \hat{\langle} e \rangle$ is a trace of $(\text{Proc}(t), \Gamma, T)$ for some type $T$ and some environment $\Gamma$, and $e$ matches constructs $\alpha$ and $\alpha'$, then $\alpha = \alpha'$.

**Proof:** Suppose for contradiction that $\alpha \neq \alpha'$. Definition 5.1.1 implies that $\alpha$ and $\alpha'$ give rise to $e$ immediately after $\text{tr}$. By Proposition 5.1.13, if $\sigma \hat{\langle} e \rangle$ and $\sigma' \hat{\langle} e' \rangle$ are both symbolic traces of $\text{Proc}(t)$ such that $\sigma \hat{\langle} e \rangle, \sigma' \hat{\langle} e' \rangle$ generates $\{ \text{tr} \hat{\langle} e \rangle \}$ and where $e$ and $e'$ are visible, then $\sigma \equiv_{\text{non-}\tau} \sigma'$ and $e = e'$. We now have that $e$ matches both $\alpha$ and $\alpha'$. The firing rules of SSOS (see Section 4.3.2) imply that $\alpha$ and $\alpha'$ 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, $\text{Proc}(t)$ contains a binary choice with branches sharing a common channel name. This contradicts the fact that $\text{Proc}(t)$ satisfies $\text{SeqNorm}$, 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 $\sigma$ generates $\Gamma \text{tr}$ and $\sigma \equiv_{\text{non-}\tau} \rho$. Then $\rho$ generates $\Gamma \text{tr}$.

**Proof:** By going through the clauses of the definition of the $\text{generates}$ relation (p. 87). Recall that clause (ii) of that definition says that all $\tau$’s in symbolic traces are essentially ignored when they generate concrete traces and observe that the restrictions of $\sigma$ and $\rho$ 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 $\text{SeqNorm}$. Let $T$ be an instantiation of type $t$. Suppose that $(Q(t), \Gamma_{\text{init}}) \sqsubseteq_T (P(t), \Gamma_{\text{init}})$. Then for all visible symbolic events $e, e'$, symbolic traces $\sigma, \sigma'$ and traces $\text{tr}$ such that

(i) $\text{tr} \hat{\langle} e \rangle \in \text{traces}(P(t), \Gamma_{\text{init}}, T)$,

(ii) $\sigma \hat{\langle} e \rangle \in \text{SymbolicTraces}(P(t))$ generates $\Gamma_{\text{init}} \text{tr} \hat{\langle} e \rangle$, and

(iii) $\sigma' \hat{\langle} e' \rangle \in \text{SymbolicTraces}(Q(t))$ generates $\Gamma_{\text{init}} \text{tr} \hat{\langle} e \rangle$,

we have that $!\langle e' \rangle \subseteq !\langle e \rangle$.
Proof: Suppose for a contradiction that there is some $j \in !\langle e' \rangle \setminus !(e)$. Then $j \in \$^i(e) \cup ?^i(e)$ (since $\$^{non-t}(e) = ?^{non-t}(e) = \{\}$ by Remark 4.3.1). This means that the $j$-th variable or value of $e'$ is of type $t$, so $j \in !(e')$. Let $e = e_1\ldots e_k$ and let $e' = e_1 v_1 \ldots e_{j-1} v_{j'} v_{j+1} \ldots v_k$, 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, $tr^{-}\langle e' \rangle \in traces(P(t), \Gamma_{init}, T)$, i.e. $e'$ matches $e$.

Since $(Q(t), \Gamma_{init}, T) \subseteq_T (P(t), \Gamma_{init}, T)$, we must have that $tr^{-}\langle e' \rangle \in traces(Q(t), \Gamma_{init})$. Clause (iii), combined with the fact that $v_{j'}$ is an output for $Q(t)$, different from $v_j$, implies that $\sigma^{-}\langle e' \rangle$ cannot generate $tr^{-}\langle e' \rangle$ within $\Gamma_{init}$. So let $\rho = \sigma_1 \cdot \sigma_2$ and $\rho = \rho_1 \cdot \rho_2$, where $\sigma_1$ and $\rho_1$ 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 $\sigma_1$ generates $\Gamma_{init} tr$ and $\rho_1$ generates $\Gamma_{init} tr$, so by Proposition 5.1.13, $\sigma_1 \equiv_{non-t} \rho_1$. Hence, if $Q(t)$ and $Q''(t)$ are symbolic states such that $Q(t) \xrightarrow{\sigma_1, s} Q'(t)$ and $Q(t) \xrightarrow{\rho_1, s} Q''(t)$, then, thanks to Corollary 5.1.5, we have that $Q'(t) = Q''(t)$.

![Diagram](image)

Figure 5.1: Illustration of the proof of Lemma 5.1.17.

Let $\Gamma$ be an environment reached after $tr$, i.e. such that $(Q(t), \Gamma_{init}, T) \xrightarrow{\tau} (Q'(t), \Gamma, T)$ for some $s$ such that $s \setminus \tau = tr$; by Lemma 5.1.12, $\Gamma$ is unique. Then, we have that

- $\sigma_2^{-}\langle e' \rangle, \rho_2^{-}\langle e'' \rangle \in SymbolicTraces(Q'(t))$,
- $\langle e' \rangle, \langle e'' \rangle \in traces(Q'(t), \Gamma, T)$,
- $\sigma_2^{-}\langle e' \rangle$ generates $\Gamma \langle e \rangle$, and
- $\rho_2^{-}\langle e'' \rangle$ generates $\Gamma \langle e \rangle$.

So, by Lemma 5.1.10, $\sigma_2 \equiv_{non-t} \rho_2$. Let $e' = c_{i_1}^{\langle x_1 \rangle} x_1 \ldots c_{i_k}^{\langle x_k \rangle} x_k$ and $e'' = c_{i_1}^{\langle x_i \rangle} x_i \ldots c_{i_k}^{\langle x_i \rangle} x_i$. Then, since the channels of $e'$ and $e''$ are the same, Lemma 5.1.6 implies that $!\langle e' \rangle = !\langle e'' \rangle$ and $x_i' = x_j''$. Since $\sigma_2^{-}\langle e' \rangle$ generates $\Gamma \langle e \rangle = \langle c.v_1 \ldots v_k \rangle$ and $\rho_2^{-}\langle e'' \rangle$ generates $\Gamma \langle e \rangle = \langle c.v_1 \ldots v_{j-1}, v_{j'}, v_{j+1} \ldots v_k \rangle$ and $j \in !\langle e'' \rangle$ (as $j \in !\langle e' \rangle$), we have that $x_i' = v_j$ and $x_j'' = v_{j'}$. Hence $v_j = v_{j'}$. This is a contradiction, so $!(e') \subseteq !(e)$.

104
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 $\text{Proc}(t)$ satisfies **SeqNorm**. Let $T$ and $\tilde{T}$ be instantiations of type $t$ such that $\tilde{T} \subseteq T$ and let $\Gamma$ be an environment in $\text{Env}(T)$. Then, if

$$\text{(Proc}(t), \Gamma, \text{Proc}(t), \Gamma', \tilde{T}) \xrightarrow{a} \text{(Proc}''(t), \Gamma', \tilde{T})$$

for some symbolic state $\text{Proc}''(t)$ and some environment $\Gamma'$, then

$$\text{(Proc}(t), \Gamma, T) \xrightarrow{a} \text{(Proc}''(t), \Gamma', T).$$

**Proof:** Since $\tilde{T}$ is a subset of $T$, we have that $\Gamma, \Gamma' \in \text{Env}(\tilde{T})$ implies $\Gamma, \Gamma' \in \text{Env}(T)$, since every partial function from $\text{Var}$ to $\tilde{T}$ is also a partial function from $\text{Var}$ 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 $\tau$ 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 $\text{Proc}(t) \xrightarrow{\tau} P(t)$ for some visible symbolic event $\epsilon = c\delta_1 x_1: X_1 \ldots \delta_k x_k: X_k$ such that $\#\delta(i) > 0$ and some symbolic state $P(t)$. In addition, $\text{Proc}''(t) = \text{Replace}^i_{\delta, \epsilon}(c, \text{Proc}(t))$ and $\Gamma' = \Gamma \uplus \{x_i \mapsto v_i \mid i \in \delta^i(\epsilon)\}$, where $v$ is a function in $\delta^i(\epsilon) \rightarrow \tilde{T}$. Then, $v$ is also a function in $\delta^i(\epsilon) \rightarrow T$, so Translation Rule 1 implies that

$$\text{(Proc}(t), \Gamma, T) \xrightarrow{\tau} \text{Replace}^i_{\delta, \epsilon}(c, \text{Proc}(t), \Gamma \uplus \{x_i \mapsto v_i \mid i \in \delta^i(\epsilon)\}, T)
= (\text{Proc}''(t), \Gamma', T).$$

If the transition in (5.4) is a result of Translation Rule 3, then it must be that $\text{Proc}(t) \xrightarrow{\tau} \text{Proc}'(t)$ and $\Gamma = \Gamma'$. Hence, the same rule implies that

$$(\text{Proc}(t), \Gamma, T) \xrightarrow{\tau} (\text{Proc}''(t), \Gamma', T).$$

**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 $\text{Proc}(t) \xrightarrow{\epsilon} \text{Proc}'(t)$ for some visible symbolic event $\epsilon = c\delta_1 x_1: X_1 \ldots \delta_k x_k: X_k$ such that $\#\delta(i) = 0$. In addition, $\Gamma' = \Gamma \uplus \{x_i \mapsto v_i \mid i \in \delta^i(\epsilon)\}$, and for all $i$ in $\delta^i(\epsilon), v_i$ is in $\tilde{T}$. However,
since $\hat{T} \subseteq T$, we have that for all $i$ in $?^i(\epsilon)$, $v_i$ is in $T$. Therefore, Translation Rule 2 implies that

$$(\text{Proc}(t), \Gamma, T) \xrightarrow{a} (\text{Proc}'(t), \Gamma \oplus \{x_i \mapsto v_i \mid i \in ?^i(\epsilon)\}, T) = (\text{Proc}'(t), \Gamma', T).$$

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 Translation Rule 4. Then it must be that $\text{Proc}(t) \xrightarrow{\text{cond}} P(t)$ and $(P(t), \Gamma, \hat{T}) \xrightarrow{a} (\text{Proc}'(t), \Gamma', \hat{T})$. The latter transition requires $k$ applications of Translation Rule 4, so the inductive hypothesis implies that $(P(t), \Gamma, \hat{T}) \xrightarrow{a} (\text{Proc}'(t), \Gamma', \hat{T})$. Hence, Translation Rule 4 implies that $(\text{Proc}(t), \Gamma, T) \xrightarrow{a} (\text{Proc}'(t), \Gamma', T)$, which completes our proof.

**Corollary 5.1.19.** Suppose that $\text{Proc}(t)$ satisfies SeqNorm. Let $T$ and $\hat{T}$ be instantiations of type $T$ such that $\hat{T} \subseteq T$. Then, if

$$(\text{Proc}(t), \Gamma, \hat{T}) \xrightarrow{\text{tr}} (\text{Proc}'(t), \Gamma', \hat{T}),$$

then

$$(\text{Proc}(t), \Gamma, T) \xrightarrow{\text{tr}} (\text{Proc}'(t), \Gamma', T).$$

**Proof:** Let $s$ be a sequence of events such that $(\text{Proc}(t), \Gamma, \hat{T}) \xrightarrow{s} (\text{Proc}'(t), \Gamma', \hat{T})$ and $s \setminus \{\tau\} = \text{tr}$. 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 $\phi$ to be a function from $T$ to $\{0 \ldots B\}$ such that

- $\phi(v) = v$ for all $v$ in $\{0 \ldots B - 1\}$,
- $\phi(v) = B$ for all $v$ in $\{B \ldots \#T - 1\}$.

In other words, $\phi$ replaces all but a fixed finite number of members of $t$ by a single value. Whenever $B$ is clear from context, we call $\phi$ 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

$$\text{Spec}(\hat{T}) \subseteq \phi(\text{Impl}(T)) \text{ implies that } \text{Spec}(T) \subseteq \text{Impl}(T),$$

for all $T$ such that $T \supseteq \hat{T}$, where $\phi : T \to \hat{T}$ is a collapsing function.

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...
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 \( \text{Spec} \) (including \textbf{SeqNorm}) and \( \text{Impl} \) (that it is symmetric in \( t \)); they give a lower bound on the size of \( \hat{T} \) based on the syntax of \( \text{Spec} \).

The rest of this section is structured as follows. Below we lift \( \phi \) 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 \( \text{cond} \), we define \( \phi(\text{cond}) \) to be like \( \text{cond} \), 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 \( \phi \) 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 \( \phi \), we define

\[
\phi^{-1}(v) = \begin{cases} 
\{v' \in T \mid \phi(v') = v\} & \text{if } v \in \{0 \ldots B\} \\
\{v\} & \text{otherwise}
\end{cases}
\]

Also, given \( T \), we lift the definition of \( \phi^{-1} \) to events:

\[
\phi^{-1}(c.v_1 \ldots v_k) = \{c.v'_1 \ldots v'_{\kappa} \mid \forall i \in \{1 \ldots k\} \bullet v'_i \in \phi^{-1}(v_i)\},
\]

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 $\pi : T \rightarrow T$, $Proc(T)$ and $Proc(T)[\pi(e)/e | e \in \Sigma]$ 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 $[\pi(e)/e | e \in \Sigma]$. Then

$$B = \{(P(T)[\Gamma],(P(T)[\pi^{-1}(\Gamma)][\pi])) | P(t) \in CSP_t, \Gamma \in Env(T)\}$$

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 $N_i(t)$ for each $i \in t$:

$$Nodes(t) = \parallel i \in t \bullet [A(i, t)] N_i(t).$$

This process syntax satisfies **TypeSym** provided the node process $N_i(t)$ 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 $N_i(t)$ 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:

$$C_t[\cdot] = (\cdot \parallel Ctrl) \setminus Y,$$

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 $\pi : T \rightarrow T$, $(\pi(tr), \pi(X)) \in failures(Proc(T))$. 

109
Equality tests

For a data independent process syntax, the syntactic condition \textbf{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 \textbf{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 $\text{Proc}(t)$, we say that $\text{Proc}(t)$ satisfies the $\textbf{PosConjEqT}^{t}_{\mathcal{M}}$ condition if for every conditional choice on $t$ of the form

\[
\text{if } \text{cond} \text{ then } P(x_1, \ldots, x_k) \text{ else } Q(x_1, \ldots, x_k)
\]

within $\text{Proc}(t)$, we have that

\begin{itemize}
  \item $\text{cond}$ is a positive conjunction of equality tests on $t$ (which gives rise to the name of the condition), and
  \item $P(v_1, \ldots, v_k) \sqsubseteq_{\mathcal{M}} Q(v_1, \ldots, v_k)$ for all values $v_1, \ldots, v_k$.
\end{itemize}

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 $\textbf{PosConjEqT}^{t}$.

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 $\textbf{PosConjEqT}^{t}_{\mathcal{M}}$. Hence, we have the following definition.

**Definition 5.2.8.** Given a CSP model $\mathcal{M}$ and a process syntax $\text{Proc}(t)$, we say that $\text{Proc}(t)$ satisfies the $\textbf{RevPosConjEqT}^{t}_{\mathcal{M}}$ condition if for every conditional choice on $t$ of the form

\[
\text{if } \text{cond} \text{ then } P(x_1, \ldots, x_k) \text{ else } Q(x_1, \ldots, x_k)
\]

within $\text{Proc}(t)$, we have that

\begin{itemize}
  \item $\text{cond}$ is a positive conjunction of equality tests on $t$, and
  \item $Q(v_1, \ldots, v_k) \sqsubseteq_{\mathcal{M}} P(v_1, \ldots, v_k)$ for all values $v_1, \ldots, v_k$.
\end{itemize}

Whenever $\mathcal{M}$ is clear from the context, we will simply write $\textbf{RevPosConjEqT}^{t}$.

**Example 5.2.9.** The process syntax

\[
\text{Proc}(t) = \text{in} ? x : t ? y : t ? z : t \rightarrow \text{if } x = y \text{ then } \text{out} . x \rightarrow \text{out} . y \rightarrow \text{STOP} \text{ else } \text{out} \& z \rightarrow (\text{out} . y \rightarrow \text{STOP} \sqcap \text{STOP})
\]
satisfies $\text{RevPosConjEqT}_T^t$. However, the process syntax

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

does not satisfy $\text{RevPosConjEqT}_T^t$, because if $x$ and $y$ are distinct values, then $\text{out}.y \rightarrow \text{STOP} \not\subseteq_T \text{out}.x \rightarrow \text{STOP}$. 

End of example.

Most specification processes that one tends to use in practice do not contain conditionals, so vacuously satisfy both $\text{PosConjEqT}_M^t$ and $\text{RevPosConjEqT}_M^t$ for all models $M$. Further, our experience is that many specifications that do contain conditionals satisfy $\text{RevPosConjEqT}_M^t$.

In Section 5.2.1 we presented two similar conditions ($\text{PosConjEqT}$ and $\text{RevPosConjEqT}$) 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 $\text{Proc}(t)$ satisfies condition $\text{NoEqT}_t^t$ 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 $\text{cond}$ then $\text{P}(t)$ else $\text{Q}(t)$’. 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

$$\text{N}_\text{myId}(t) = \text{in}?:i:t \rightarrow \text{if } \text{myId} = i \text{ then } a \rightarrow \text{STOP}$$
$$\text{else } b \rightarrow \text{STOP}$$

does not satisfy $\text{NoEqT}_t^t$, since it contains an explicit equality test on $t$ that requires a map from $\{\text{myId}, i\}$ to some values of type $t$ in order to evaluate the truth of condition $\text{myId} = i$.

Similarly, the process syntax

$$\text{N}_\text{myId}(t) = \text{in}?:x:t \rightarrow \text{out}.x \rightarrow \text{STOP} \parallel \text{in}?:y:t \rightarrow \text{out}.y \rightarrow \text{STOP}$$
$$\{\text{out}\}$$

does not satisfy $\text{NoEqT}_t^t$, since an event on the $\text{out}$ 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

$$\text{N}_\text{myId}(t) = \text{in}?:x:t \rightarrow \text{if } x = x \text{ then } a \rightarrow \text{STOP}$$
$$\text{else } b \rightarrow \text{STOP}$$

does not satisfy $\text{NoEqT}_t^t$, since $x = x$ evaluates to $\text{True}$ 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 $\text{Proc}(t)$ satisfies $\text{SeqNorm}$ and $\text{RevPosConjEqT}$ and given a collapsing function $\phi$, if

- $tr$ is a trace of $(\text{Proc}(t), \Gamma_{\text{init}}, T)$ (for some sufficiently large $T$),
- $\phi(tr) \cdot (e)$ is a trace of $(\text{Proc}(t), \phi(\Gamma_{\text{init}}), T)$, 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 $(\text{Proc}(t), \Gamma_{\text{init}}, T)/tr$. 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

- $\text{Proc}(t)$ satisfies $\text{SeqNorm}$ and $\text{RevPosConjEqT}$,
- $\phi$ is a $B$-collapsing function, and
- $T$ is an instantiation of type $t$ of size at least $B + 1$.

Suppose further that

(i) $tr \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}})$,
(ii) $\phi(tr) \cdot (e) \in \text{traces}(\text{Proc}(t), \phi(\Gamma_{\text{init}}))$ with $e = c.v_1 \ldots v_k$,
(iii) $\sigma \cdot (e) \in \text{SymbolicTraces}(\text{Proc}(t))$ and $\sigma \cdot (e)$ generates $\phi(\Gamma_{\text{init}}) \cdot \phi(tr) \cdot (e)$, where $e$ is a visible symbolic event of the form $c \xi_1 x_1: X_1 \ldots \xi_k x_k: X_k$, and
(iv) $\forall i \in !^t(e) \cdot v_i \in \{0 .. B-1\}$.

Then

$$\forall v' \in \{1 \ldots k\} \rightarrow \text{Value} | (\forall i \in !^t(e) \cup ?^t(e) \cdot v'_i \in T) \land (\forall i \in !(e) \cdot v'_i = v_i) \cdot tr \cdot (c.v'_i \ldots v'_k) \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}}).$$

**Proof:** We prove the result using a structural induction on $\text{Proc}(t)$. The most interesting cases are those for prefix, external choice and conditional choice.

**STOP**

The result holds vacuously in this case.

**Prefix**

Suppose that $\text{Proc}(t) = \alpha \rightarrow P(t)$ for some construct $\alpha = c \xi_1 x_1: X_1 \ldots \xi_k x_k: X_k$ and $P(t)$.
some process syntax $P(t)$. We consider two cases now.

**Subcase 1.** Suppose that $tr = \emptyset$. Then, by Symbolic Prefix Rule 1 (p. 78) and Symbolic Prefix Rule 2 (p. 78), $\sigma \in \{\tau\}^*$. Hence $\text{Proc}(t) \xrightarrow{\tau^*} e \xrightarrow{\tau} s$. Using Translation Rule 3 (p. 86) and Remark 4.4.6 we get that

\[
\forall v' \in \{1 \ldots k\} \rightarrow \text{Value} | \\
(\forall i \in \$^i(e) \cup ?^i(e) \cdot v_i' \in T) \land (\forall i \in !^i(e) \cdot v_i' = \Gamma_{\text{init}}(x_i)) \bullet \\
\langle c.v'_1 \ldots v'_k \rangle \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}})
\]

Since $tr = \emptyset$, assumption (iii) of the proposition and the definition of the generates relation (p. 87) imply together that $\langle e \rangle$ generates$_{\phi(\Gamma_{\text{init}})} (e)$. Therefore, again by the definition of generates and by assumption (iv),

\[
\forall i \in !^i(e) \cdot v_i = (\phi(\Gamma_{\text{init}}))(x_i) \land v_i \in \{0 \ldots B - 1\}. 
\]

(5.7)

Suppose that

\[
v' \in \{1 \ldots k\} \rightarrow \text{Value} \text{ is such that} \\
(\forall i \in \$^i(e) \cup ?^i(e) \cdot v_i' \in T) \land (\forall i \in !^i(e) \cdot v_i' = v_i).
\]

Then, from (5.7) we can infer that $\forall i \in !^i(e) \cdot v_i' = (\phi(\Gamma_{\text{init}}))(x_i) \land v_i' \in \{0 \ldots B - 1\}$. However, the properties of $\phi$ imply that for all variables $\text{var}$ and all values $\text{val}$ we have that

\[
(\phi(\Gamma_{\text{init}}))(\text{var}) = \text{val} \land \text{val} \in \{0 \ldots B - 1\} \Rightarrow \Gamma_{\text{init}}(\text{var}) = \text{val},
\]

so

\[
\forall i \in !^i(e) \cdot v_i' = \Gamma_{\text{init}}(x_i). 
\]

(5.8)

In addition, from the definition of generates, $\forall i \in !^{\text{non-}t}(e) \cdot v_i = \phi(\Gamma_{\text{init}})(x_i)$. But we know that $\forall i \in !^{\text{non-}t}(e) \cdot v_i' = v_i$, so

\[
\forall i \in !^{\text{non-}t}(e) \cdot v_i' = (\phi(\Gamma_{\text{init}}))(x_i) = \Gamma_{\text{init}}(x_i)
\]

with the last equality following from the fact that for all $i$ in $!^{\text{non-}t}(e)$, $x_i$ must be of a non-$t$ type. Hence and from (5.8),

\[
\forall i \in !^i(e) \cdot v_i' = \Gamma_{\text{init}}(x_i).
\]

Therefore, (5.6) implies that $\langle c.v'_1 \ldots v'_k \rangle \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}})$. We have shown that

\[
\forall v' \in \{1 \ldots k\} \rightarrow \text{Value} | (\forall i \in \$^i(e) \cup ?^i(e) \cdot v_i' \in T) \land (\forall i \in !^i(e) \cdot v_i = v_i') \bullet \\
\langle c.v'_1 \ldots v'_k \rangle \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}}),
\]

which is what we wanted to show.

**Subcase 2.** Suppose that $tr = \langle e' \rangle \cdot tr'$ for some visible event $a$ and some trace $tr'$. Then $\phi(tr) = \langle \phi(e') \rangle \cdot \phi(tr')$. So clearly $\phi(tr)$ is non-empty and we know that
\( \sigma \) generates_{\phi(\Gamma_{\text{init}})} \phi(tr). By the definition of generates, it must be that there is at least one visible symbolic event within \( \sigma \). Hence, \( \sigma = \sigma_1^*(e') \sigma_2 \) for some visible symbolic event \( e' \) and some symbolic traces \( \sigma_1 \) and \( \sigma_2 \) such that \( \sigma_1 \) is in \( \{\tau\}^* \) (\( \sigma_1 \) cannot contain any conditional symbolic events because \( \text{Proc}(t) \) is a prefix). Then,

\[
\text{Proc}(t) \xrightarrow{\tau \ast} e' P'(t)
\]

and

\[
\sigma_2^*(e) \in \text{SymbolicTraces}(P'(t)), \tag{5.9}
\]

where \( P'(t) \) is like \( P(t) \), but with some substitutions of concrete values for the non-\( t \) type input variables of \( \alpha \), 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 now deduce the inductive hypothesis for \( P'(t) \), using Translation Rule 3 (p. 86), Remark 4.4.6 and Match \( \phi \) (5.10) gives us condition (ii), observing that \( \sigma_2 \), \( \sigma \) generates \( \sigma \) and \( \sigma_2 \) cannot contain any conditional symbolic events because \( \text{Proc}(t) \) is a prefix. Then,

\[
\text{Proc}(t) \xrightarrow{\tau \ast} e' P'(t)
\]

and

\[
\sigma_2^*(e) \in \text{SymbolicTraces}(P'(t)), \tag{5.9}
\]

where \( P'(t) \) is like \( P(t) \), but with some substitutions of concrete values for the non-\( t \) type input variables of \( \alpha \), 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

\[
(\text{Proc}(t), \phi(\Gamma_{\text{init}})) \xrightarrow{\tau \ast} \phi(e') (P'(t), \phi(\Gamma_{\text{init}}) \oplus \text{Match}(e', \phi(e'))),
\]

where, recall from Section 4.6, \( \text{Match}(e', \phi(e')) \) is a map from type \( t \) input variables of \( e' \) to the corresponding concrete values of \( \phi(e') \). We have that configuration \( (P'(t), \phi(\Gamma_{\text{init}}) \oplus \text{Match}(e', \phi(e'))) \) is unique (thanks to Lemma 5.1.12). We know from assumption (ii) that \( \langle \phi(e') \rangle \cdot \phi(tr') \cdot \langle e \rangle \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}}) \). Hence,

\[
\phi(tr') \cdot \langle e \rangle \in \text{traces}(P'(t), \phi(\Gamma_{\text{init}}) \oplus \text{Match}(e', \phi(e'))). \tag{5.10}
\]

Similarly, since \( \text{Proc}(t) \xrightarrow{\tau \ast} e' P'(t) \) and \( tr = \langle e' \rangle \cdot tr' \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}}) \) (from assumption (i)), using Translation Rule 3 (p. 86), Remark 4.4.6 and Lemma 5.1.12 we can infer that

\[
(\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{\tau \ast} (P'(t), \Gamma_{\text{init}} \oplus \text{Match}(e', e')) \xrightarrow{tr'}.
\tag{5.11}
\]

Hence,

\[
tr' \in \text{traces}(P'(t), \Gamma_{\text{init}} \oplus \text{Match}(e', e')). \tag{5.12}
\]

Finally, assumption (iii) implies that

\[
\sigma^*(e) = \sigma_1^*(e) \sigma_2^*(e) \text{ generates}_{\phi(\Gamma_{\text{init}})} \phi(tr) \cdot \langle e \rangle = \langle \phi(e') \rangle \cdot \phi(tr') \cdot \langle e \rangle.
\]

So, by the definition of generates (p. 87),

\[
\sigma_2^*(e) \text{ generates}_{\phi(\Gamma_{\text{init}}) \oplus \text{Match}(e', e'))} \phi(tr') \cdot \langle e \rangle. \tag{5.13}
\]

We can now deduce the inductive hypothesis for \( P(t) \), with \( tr' \) in place of \( tr \), \( \sigma_2 \) in place of \( \sigma \), and \( \Gamma_{\text{init}} \oplus \text{Match}(e', e') \) in place of \( \Gamma_{\text{init}} \): (5.12) gives us condition (i); (5.10) gives us condition (ii); (5.13) gives us condition (iii). Hence

\[
\forall v' \in \{1 \ldots k\} \to \text{Value} \mid (\forall i \in S(t) \cup \tau(t) \cdot v'_i \in T) \land (\forall i \in !(\epsilon) \cdot v_i = v'_i) \cdot tr' \cdot \langle e, v'_1 \ldots v'_k \rangle \in \text{traces}(P'(t), \Gamma_{\text{init}} \oplus \text{Match}(e', e')).
\]

114
This, combined with (5.11), gives us that
\[ \forall v' \in \{1 \ldots k\} \rightarrow Value \mid (\forall i \in \delta'(e) \cup \delta'(\epsilon) \bullet v'_i \in T) \land (\forall i \in \delta'(\epsilon) \bullet v_i = v'_i) \bullet (e')^*tr' \cdot (c.v'_1 \ldots v'_k) \in traces(Proc(t) , \Gamma_{init}). \]

However, \( tr = (e')^*tr' \), so the result holds.

**External choice**
Suppose that \( Proc(t) = P(t) \parallel Q(t) \) for some process syntaxes \( P(t) \) and \( Q(t) \). \textbf{SeqNorm} implies that the channels of the initial symbolic events of \( P(t) \) and \( Q(t) \) are disjoint, so we have that either
\[ \phi(tr)^*\langle e \rangle \in traces(P(t), \phi(\Gamma_{init})) \quad \text{and} \quad tr \in traces(P(t), \Gamma_{init}) \]
or
\[ \phi(tr)^*\langle e \rangle \in traces(Q(t), \phi(\Gamma_{init})) \quad \text{and} \quad tr \in traces(Q(t), \Gamma_{init}) \]

Without loss of generality we assume the former. Let \( \rho \in SymbolicTraces(P(t)) \) be such that \( \rho \) generates \( \phi(\Gamma_{init}) \) \( \phi(tr) \). Then \( \sigma \) is like \( \rho \), except that it may have some additional \( \tau \)'s (promoted from \( Q(t) \)) before the first visible symbolic event (recall that \textbf{SeqNorm} prevents conditional symbolic events before a visible symbolic event). Hence, \( \sigma = \tau^*a^*\rho \) for some \( a \). Therefore, since \( \sigma^*\langle e \rangle \) generates \( \phi(\Gamma_{init}) \) \( \phi(tr)^*\langle e \rangle \), the definition of the generates relation (p. 87) implies
\[ \rho^*\langle e \rangle \text{ generates } \phi(\Gamma_{init}) \phi(tr)^*\langle e \rangle. \]

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 \( Proc(t) = \text{if} \ cond \ \text{then} \ P(t) \ \text{else} \ Q(t) \) for some process syntax \( P(t) \) and \( Q(t) \). If \( cond \) is not in \( Cond \), then \( cond \) immediately evaluates to \text{True} or \text{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
\[ \sigma = \langle cond \rangle^*\rho \quad \text{or} \quad \sigma = \langle \neg cond \rangle^*\rho \]

for some symbolic trace \( \rho \). We now perform a case analysis on the truth value of the evaluation of \( cond \) within the environments \( \Gamma_{init} \) and \( \phi(\Gamma_{init}) \).
**Case 1.** Suppose that $\llbracket \text{cond} \rrbracket_{\phi(\Gamma_{\text{init}})} = \llbracket \text{cond} \rrbracket_{\Gamma_{\text{init}}} = \text{True}$. Then it must be that $\rho \in \text{SymbolicTraces}(P(t))$. From assumption (iii) we have that

$$\rho^\cdot \langle \epsilon \rangle \text{ generates}_{\phi(\Gamma_{\text{init}})} \phi(\text{tr})^\cdot \langle \epsilon \rangle.$$  

In addition, from assumptions (i) and (ii) we have that

$$\text{tr} \in \text{traces}(P(t), \Gamma_{\text{init}}) \quad \text{and} \quad \phi(\text{tr})^\cdot \langle \epsilon \rangle \in \text{traces}(P(t), \phi(\Gamma_{\text{init}})).$$  

The result is now implied in this case by the inductive hypothesis for $P(t)$ and the fact that $(\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{\phi} (P(t), \Gamma_{\text{init}})$. 

**Case 2.** Suppose that $\llbracket \text{cond} \rrbracket_{\phi(\Gamma_{\text{init}})} = \llbracket \text{cond} \rrbracket_{\Gamma_{\text{init}}} = \text{False}$. This case is like Case 1, above, with $Q(t)$ in place of $P(t)$. 

**Case 3.** Suppose that $\llbracket \text{cond} \rrbracket_{\phi(\Gamma_{\text{init}})} = \text{True} \land \llbracket \text{cond} \rrbracket_{\Gamma_{\text{init}}} = \text{False}$. Then, by assumption (i) and (ii),

$$\text{tr} \in \text{traces}(Q(t), \Gamma_{\text{init}}) \quad \text{and} \quad \phi(\text{tr})^\cdot \langle \epsilon \rangle \in \text{traces}(P(t), \phi(\Gamma_{\text{init}})).$$  

Since $\text{Proc}(t)$ satisfies $\text{RevPosConjEqT}_{\text{T}}$, we have that $(Q(t), \phi(\Gamma_{\text{init}})) \subseteq_{\text{T}} (P(t), \phi(\Gamma_{\text{init}}))$, so

$$\phi(\text{tr})^\cdot \langle \epsilon \rangle \in \text{traces}(Q(t), \phi(\Gamma_{\text{init}})).$$  

Let $\rho^\cdot \langle \epsilon' \rangle \in \text{SymbolicTraces}(Q(t))$ be such that $\rho^\cdot \langle \epsilon' \rangle \text{ generates}_{\phi(\Gamma_{\text{init}})} \phi(\text{tr})^\cdot \langle \epsilon \rangle$. Then, by Lemma 5.1.17, $!\langle \epsilon' \rangle \subseteq !\langle \epsilon \rangle$. Hence, $!\langle \epsilon' \rangle \subseteq !\langle \epsilon \rangle$, so assumption (iv) implies that

$$\forall \ i \in !\langle \epsilon' \rangle \bullet v_i \in \{0 \ldots B - 1\}.$$  

Therefore, by the inductive hypothesis for $Q(t)$,

$$\forall \ v' \in \{1 \ldots k\} \rightarrow \text{Value} \ | \ \begin{array}{l}
(\forall \ i \in \$\langle \epsilon' \rangle \cup ?\langle \epsilon' \rangle \bullet v_i' \in T) \land (\forall \ i \in !\langle \epsilon' \rangle \bullet v_i' = v_i) \bullet \text{tr}^\cdot \langle c.v_1' \ldots v_k' \rangle \in \text{traces}(Q(t), \Gamma_{\text{init}}).\end{array}$$  

(5.14)

Suppose $v' \in \{1 \ldots k\} \rightarrow \text{Value}$ is such that

$$(\forall \ i \in \$\langle \epsilon' \rangle \cup ?\langle \epsilon' \rangle \bullet v_i' \in T) \land \forall \ i \in !\langle \epsilon \rangle \bullet v_i' = v_i.$$  

The fact that $!\langle \epsilon' \rangle \subseteq !\langle \epsilon \rangle$ implies that

$$\forall \ i \in !\langle \epsilon' \rangle \bullet v_i' = v_i.$$  

(5.15)

In addition, we know that both $\epsilon$ and $\epsilon'$ give rise to $c.v_1 \ldots v_k$, so

$$\$\langle \epsilon' \rangle \cup ?\langle \epsilon' \rangle \cup !\langle \epsilon' \rangle = \$\langle \epsilon \rangle \cup ?\langle \epsilon \rangle \cup !\langle \epsilon \rangle,$$  

116
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
\[ \forall v' \in \{1 \ldots k\} \rightarrow Value \ | \ (\forall i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \land (\forall i \in !^t(\epsilon) \bullet v'_i = v_i) \bullet \tr^\prec(c.v'_1 \ldots v'_k) \in \text{traces}(Q(t), \Gamma_{init}). \]

The result now follows, because \( (Proc(t), \Gamma_{init}) \xrightarrow{\circ} (Q(t), \Gamma_{init}). \)

**Case 4.** Suppose that \( \llbracket \text{cond} \rrbracket_{\phi(\Gamma_{init})} = \text{False} \land \llbracket \text{cond} \rrbracket_{\Gamma_{init}} = \text{True}. \)

This case is not possible since \( \text{cond} \) is a conjunction of equality tests and for no function \( \phi \) we can ever have \( x = y \) and \( \phi(x) \neq \phi(y). \)

**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 (p. 80) implies that
\[ \sigma = \langle \tau \rangle^\prec \rho \]
for some \( \rho \) in \( \text{SymbolicTraces}(P(t)) \). Then, assumption (iii) implies that \( \rho^\prec(\epsilon) \) generates \( \phi(\Gamma_{init}) \phi(\tr^\prec(\epsilon)). \) In addition, by the COSE translation rules (see Section 4.4.1), it must be that
\[ \bullet \tr \in \text{traces}(P(t), \Gamma_{init}), \]
\[ \bullet \phi(\tr^\prec(\epsilon)) \in \text{traces}(P(t), \phi(\Gamma_{init})), \]
so the result is now implied by the inductive hypothesis for \( P(t) \) and the fact that \( (Proc(t), \Gamma_{init}) \xrightarrow{\circ} (P(t), \Gamma_{init}). \)

Recall that two symbolic traces \( \sigma \) and \( \sigma' \) 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 \( \sigma \) and \( \sigma' \) (see Definition 5.1.7). We will need the following definition in order to define our threshold.

**Definition 5.2.13.** Let \( Proc(t) \) be a process syntax, \( \sigma \) a symbolic trace and \( \epsilon \) a visible symbolic event. Then \( !^t(\sigma, \epsilon)(Proc(t)) \) is the set of indices of all output variables of type \( t \) in all constructs of \( Proc(t) \) that end symbolic traces that are non-\( t \) equivalent to \( \sigma^\prec(\epsilon) \). Formally,
\[ !^t(\sigma, \epsilon)(Proc(t)) = \bigcup \{!^t(\epsilon') \mid \sigma'^\prec(\epsilon') \in \text{SymbolicTraces}(Proc(t)) \]
\[ \land \sigma'^\prec(\epsilon') \equiv_{\text{non-}t} \sigma^\prec(\epsilon) \}. \]
Example 5.2.14. Let

\[
\text{Proc}(t) = \text{in}!a?x:t?y:t \rightarrow \text{then out}!x$y:t \rightarrow \text{STOP} \\
\text{else out}x:t!y:t \rightarrow \text{STOP},
\]

where \(a\) is of a non-\(t\) type. Then

\[
!(\langle \text{in}!a?x:t?y:t \rangle, \text{out}x:t!y:t)(\text{Proc}(t)) = \{1, 2\},
\]

since both of the constructs on channel \(\text{out}\) end symbolic traces of \(\text{Proc}(t)\) that are non-\(t\) equivalent to the symbolic trace \(\langle \text{in}!a?x:t?y:t, \text{out}x:t!y:t \rangle\), 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

\[
!(\langle \text{in}!b?x?t?y:t \rangle, \text{out}x:t!y:t)(\text{Proc}(t)) = \{\},
\]

since \(\text{Proc}(t)\) cannot perform any symbolic trace that is non-\(t\) equivalent to \(\langle \text{in}!b?x?t?y:t, \text{out}x:t!y:t \rangle\).

End of example.

We now present the first of our two main results of this chapter. The following theorem establishes a threshold \(\text{Thresh}_T\) such that, provided \(\text{Spec}(t)\) and \(\text{Impl}(t)\) fulfill certain requirements, then for all \(B \geq \text{Thresh}_T\), if \(\phi\) is a \(B\)-precollapsing function, then for all \(n \geq B\), if

\[
\text{Spec}(\{0 \ldots B\}) \sqsubseteq_T \phi(\text{Impl}(\{0 \ldots n\})), \tag{5.16}
\]

then

\[
\text{Spec}(\{0 \ldots n\}) \sqsubseteq_T \text{Impl}(\{0 \ldots n\})
\]

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 \(\{0 \ldots B\}\), thanks to the external renaming by \(\phi\)). So, theoretically, we still need to perform an unbounded number of checks (one for each value of \(n\)) in order to verify that \(\text{Spec}(T) \sqsubseteq_T \text{Impl}(T)\) 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 \(\text{Abstr}\) such that \(\text{Abstr} \sqsubseteq_T \phi(\text{Impl}(\{0 \ldots n\}))\) for all \(n \geq B\) for some \(B \geq \text{Thresh}_T\), then it is enough to perform a single refinement check, namely \(\text{Spec}(\{0 \ldots \max\{\text{Thresh}_T, B\}\}) \sqsubseteq_T \text{Abstr}\), in order to conclude that \(\text{Spec}(\{0 \ldots n\}) \sqsubseteq_T \text{Impl}(\{0 \ldots n\})\) for all \(n \geq \max\{\text{Thresh}_T, B\}\). Combining this with manual checks for all \(n \in \{0 \ldots \max\{\text{Thresh}_T, B\} - 1\}\), we obtain the answer to whether \(\text{Spec}(T) \sqsubseteq_T \text{Impl}(T)\) 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 \(\text{Abstr}\) 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) $\text{Spec}(t)$ satisfies $\text{SeqNorm}$ and $\text{RevPosConjEqT}_T$,

(ii) $\text{Impl}(t)$ satisfies $\text{TypeSym}$,

(iii) $\text{Thresh}_T$ is the maximum number of unique indices of output variables reachable on non-$t$ equivalent symbolic traces of $\text{Spec}(t)$:

$$\text{Thresh}_T = \max \{ \#!^i(\sigma, \epsilon)(\text{Spec}(t)) \mid \sigma \cdot (\epsilon) \in \text{SymbolicTraces}(\text{Spec}(t)) \},$$

(iv) $B \geq \text{Thresh}_T$,

(v) $T$ is an instantiation of type $t$ of size at least $B + 1$, and

(vi) $\phi$ is a $B$-collapsing function.

Then if $\text{Spec}(\phi(T)) \sqsubseteq_T \phi(\text{Impl}(T))$, then $\text{Spec}(T) \sqsubseteq_T \text{Impl}(T)$.

**Proof:** Suppose that $\text{Spec}(\phi(T)) \sqsubseteq_T \phi(\text{Impl}(T))$ and assume for a contradiction that $\text{Spec}(T) \not\sqsubseteq_T \text{Impl}(T)$. Consider a shortest trace that demonstrates this non-refinement; this trace is necessarily non-empty, so of the form $tr \cdot (\epsilon)$ such that

$$tr \cdot (\epsilon) \in \text{traces}(\text{Impl}(T)),
\quad tr \in \text{traces}(\text{Spec}(T)),
\quad tr \cdot (\epsilon) \not\in \text{traces}(\text{Spec}(T)).$$

Suppose that $e = c.v_1 \ldots v_k$, and suppose that $tr \cdot (\epsilon)$ is generated by a symbolic trace $\sigma_1 \cdot (\epsilon_1)$, where $\epsilon_1$ is visible.

By assumptions (iii) and (iv), we have that $\#!^i(\sigma_1, \epsilon_1)(\text{Spec}(t)) \leq B$, so let $\pi : T \to T$ be a bijection that maps $\{v_i \mid i \in !^i(\sigma_1, \epsilon_1)(\text{Spec}(t)) \}$ into $\{0 \ldots B - 1\}$. By Corollary 5.2.3, $\text{Spec}(t)$ satisfies $\text{TypeSym}$, so by Remark 5.2.6 we have that

$$\pi(tr) \cdot (\epsilon) \in \text{traces}(\text{Spec}(T)),$$

$$\pi(tr) \in \text{traces}(\text{Spec}(T)),$$

$$\pi(tr) \cdot (\epsilon) \not\in \text{traces}(\text{Spec}(T)).$$

Hence,

$$\phi(\pi(tr)) \cdot (\phi(\pi(\epsilon))) \in \text{traces}(\phi(\text{Impl}(T))).$$

But $\text{Spec}(\phi(T)) \sqsubseteq_T \phi(\text{Impl}(T))$, so

$$\phi(\pi(tr)) \cdot (\phi(\pi(\epsilon))) \in \text{traces}(\text{Spec}(\phi(T))).$$
However, by Corollary 5.1.19, \( \text{Spec}(T) \sqsubseteq_T \text{Spec}(\phi(T)) \), so
\[
\phi(\pi(tr))^\ast(\phi(\pi(e))) \in \text{traces}(\text{Spec}(T)). \tag{5.18}
\]

We can now apply Proposition 5.2.12 to \( \text{Spec}(T) \), 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 \( \sigma \), and taking \( \epsilon \) to be the symbolic event that generates \( \phi(\pi(e)) \); condition (iv) is satisfied since \( !^t(\epsilon) \subseteq !^t(\sigma_1, \epsilon_1)(\text{Spec}(t)) \) (since \( \sigma^\ast(\epsilon) \equiv_{\text{non-}t} \sigma_1^\ast(\epsilon_1) \)), and by construction, all the corresponding fields of \( \phi(\pi(tr)) \) are in \( \{0..B-1\} \). Considering the valuation \( \nu' \) such that \( c.v'_1 \ldots v'_k = \pi(e) \), then allows us to deduce that
\[
\pi(tr)^\ast(\pi(e)) \in \text{traces}(\text{Spec}(T)).
\]
This is a contradiction, which completes our proof. \( \blacksquare \)

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) \( \text{Spec}(t) \) satisfies \textit{SeqNorm} and \textit{NoEqT}^t,
(ii) \( \text{Impl}(t) \) satisfies \textit{TypeSym},
(iii) \( \text{Thresh}_T \) is the maximum number of outputs of type \( t \) in constructs of \( \text{Spec}(t) \):
\[\text{Thresh}_T = \max \{ \#!^t(\alpha) \mid \alpha \text{ is a construct of } \text{Spec}(t) \} ,\]
(iv) \( B \geq \text{Thresh}_T \),
(v) \( T \) is an instantiation of type \( t \) of size at least \( B + 1 \), and
(vi) \( \phi \) is a \( B \)-collapsing function.

Then if \( \text{Spec}(\phi(T)) \sqsubseteq_T \phi(\text{Impl}(T)) \), then \( \text{Spec}(T) \sqsubseteq_T \text{Impl}(T) \).

**Proof:** Let \( \sigma^\ast(\epsilon) \) and \( \sigma'^\ast(\epsilon') \) be symbolic traces of \( \text{Spec}(t) \), with \( \epsilon \) and \( \epsilon' \) visible. Suppose that \( \sigma^\ast(\epsilon) \equiv_{\text{non-}t} \sigma'^\ast(\epsilon') \). Since \( \text{Spec}(t) \) contains no equality tests on \( t \), Lemma 5.1.8 implies that \( \epsilon = \epsilon' \). Hence, \( \#!^t(\sigma, \epsilon)(\text{Spec}(t)) = \#!^t(\epsilon) \). Therefore,
\[
\text{Thresh}_T = \max \{ \#!^t(\alpha) \mid \alpha \text{ is a construct of } \text{Spec}(t) \} \geq \max \{ \#!^t(\sigma, \epsilon)(\text{Spec}(t)) \mid \sigma^\ast(\epsilon) \in \text{SymbolicTraces}(\text{Spec}(t)) \},
\]
with equality in the usual case where every construct is reachable. In addition, observe that every process syntax that satisfies \textit{NoEqT}^t also satisfies \textit{RevPosConjEqT}^T. The result then follows from Theorem 5.2.15. \( \blacksquare \)
Remark 5.2.17. Even if $Spec(t)$ uses a conditional on $t$, thanks to Lemma 5.1.17, $!t(\alpha_{\text{then}}) \supseteq !t(\alpha_{\text{else}})$, where $\alpha_{\text{then}}$ and $\alpha_{\text{else}}$ are the constructs in the “then” and “else” branches, respectively; hence the threshold from Corollary 5.2.16 still applies. It’s only when $Spec(t)$ contains nested conditionals, like for example

$$
\begin{align*}
\text{if } x = y \text{ then } (\text{if } y = z \text{ then } \text{STOP} & \text{ else } c!x$y:t \rightarrow \text{STOP}) \\
\text{else } (\text{if } y = z \text{ then } c$y:x!t & \rightarrow \text{STOP} \\
\text{else } c$t:x$y:t),
\end{align*}
$$

that one needs to consider multiple constructs together.

Remark 5.2.18. For every verification problem, the value of $\text{Thresh}_T$ 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 $\text{Thresh}_T$ 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 $Proc(t)$ from Example 5.2.14, for which $!((in!a?x:t?y:t), out$x:t$x:t)(Proc(t)) = \{1, 2\}$. It is clear that $!t(\sigma, \epsilon)(Proc(t))$ has fewer elements for other values of $\sigma$ and $\epsilon$. Hence, $\text{Thresh}_T = 2$ in this case.

End of example.

Example 5.2.21. Let

$$
Spec(t) = c_1?i:t?j:t \rightarrow (c_2?i:t!a!b \rightarrow c_3.i.i.i \rightarrow \text{STOP} \\
\Box \\
c_3!i!j$l:t \rightarrow \text{STOP}),
$$

where $a$ and $b$ are some values of a non-$t$ type. Then

$\#!(c_1?i:t?j:t) = 0$,  
$\#!(c_2?i:t!a!b) = 0$,  
$\#!(c_3.i.i.i) = 3$,  
$\#!(c_3!i!j$l:t) = 2$,  

Hence, the value of $\text{Thresh}_T$ in Corollary 5.2.16 is $\max\{\#!(\alpha) \mid \alpha \text{ is a construct of } Spec(t)\} = 3$.

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 role, 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.

\[
\text{SERVER}(t) = \text{request_peer}?i:t \rightarrow \text{SERVER}'(i, t)
\]

\[
\text{SERVER}'(i, t) =
\begin{align*}
\text{request_peer}?j:t & \rightarrow \\
\text{if } i = j & \text{ then SERVER}'(i, t) \\
\text{else } & \text{ peer_id}.i.j \rightarrow \text{peer_id}.j.i \rightarrow \text{session_start.i.j} \\
& \quad \rightarrow \text{session_end!i}!j?\text{flag}:\{s, f\} \rightarrow \text{SERVER}(t)
\end{align*}
\]

\[
\text{CLIENT}(i, t) =
\begin{align*}
\text{request_peer}.i & \rightarrow \text{peer_id}!i?j:t \\
& \quad \rightarrow (\text{session_start}.i.j \rightarrow \text{CLIENT}'(i, j, \text{yes}, t))
\end{align*}
\]

\[
\text{CLIENT}'(i, j, \text{principal}, t) =
\begin{align*}
\text{if } \text{principal} = \text{yes} & \text{ then session_end}.i.j.f \rightarrow \text{CLIENT}(i, t) \\
& \quad \rightarrow \text{communicate}.i.j \rightarrow \text{session_end}.i.j.s \rightarrow \text{CLIENT}(i, t) \\
\text{else } & \text{ session_end}.j.i.f \rightarrow \text{CLIENT}(i, t) \\
& \quad \rightarrow \text{communicate}.j.i \rightarrow \text{session_end}.j.i.s \rightarrow \text{CLIENT}(i, t)
\end{align*}
\]

Since clients may communicate with each other, we let \(\text{CLIENTS}(t)\) model a parallel composition of all client processes, each of which synchronizes on all the events it can participate in (which we denote by \(\text{Alpha}(i, t)\)).

\[
\text{CLIENTS}(t) = \parallel i \in t \bullet [\text{Alpha}(i, t)] \text{ CLIENT}(i, t)
\]

\[
\text{Alpha}(i, t) = \{ \text{request_peer}.i, \text{peer_id}.i.j, \text{session_start}.i.j, \text{session_start}.j.i, \\
\text{session_end}.i.j.f, \text{session_end}.j.i.f, \text{flag}, \text{communicate}.i.j, \\
\text{communicate}.j.i \mid j \in t, \text{flag} \in \{s, f\}\}
\]
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.

\[ \text{Impl}(t) = (\text{SERVER}(t) \parallel \text{CLIENTS}(t)) \]

\[ \setminus \{ | \text{request}_\text{x}_\text{y}, \text{peer}_\text{id}, \text{session}_\text{start}, \text{session}_\text{end} | \} \]

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.

\[ \text{Spec}(t) = \text{session}_\text{start}\; i : t \; j : t \rightarrow \text{session}_\text{end} \; ! \; i \; j \; \text{status} : \{s, f\} \rightarrow \text{Spec}(t). \]

The process syntaxes \( \text{Spec}(t) \) and \( \text{Impl}(t) \) meet the assumptions of Theorem 5.2.15. We have that

\[ #^{	ext{pl}}(\emptyset, \text{session}_\text{start}\; i : t \; j : t) = 0 \]
\[ #^{	ext{pl}}((\text{session}_\text{start}\; i : t \; j : t), \text{session}_\text{end}.i.j.s) = 2 \]
\[ #^{	ext{pl}}((\text{session}_\text{start}\; i : t \; j : t), \text{session}_\text{end}.i.j.f) = 2 \]

and \( #^{	ext{pl}}(\sigma, \epsilon) \geq 2 \) for all other \( \sigma^{	ext{pl}}(\epsilon) \in \text{SymbolicTraces}(\text{Spec}(t)) \). Hence, the threshold, calculated as in Theorem 5.2.15, is

\[ \text{Thresh}^T = \max\{ #^{	ext{pl}}(\sigma, \epsilon)(\text{Spec}(t)) \mid \sigma^{	ext{pl}}(\epsilon) \in \text{SymbolicTraces}(\text{Spec}(t)) \} = 2. \]

(Since \( \text{Spec}(t) \) satisfies \( \text{NoEqT}^T \) 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 \( \text{Abstr} \) such that \( \text{Spec}^{\{0,1,2\}} \sqsubseteq^T \text{Abstr} \) and \( \text{Abstr} \sqsubseteq^T \phi(\text{Impl}^{\{0\ldots B-1\}}) \) for all \( n \geq 2 \) (see Chapter 6), then, by Theorem 5.2.15, \( \text{Spec}(T) \sqsubseteq^T \text{Impl}(T) \) 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 \( \text{Proc}(t) \) satisfies \( \text{SeqNorm} \) and \( \text{RevPosConjEqT}^T \) and given a collapsing function \( \phi \), if

- \( \text{tr} \) is a trace of \( (\text{Proc}(t), \Gamma_{\text{init}}, T) \) (for some sufficiently large \( T \)),
- \( (\phi(\text{tr}), X) \) is a failure of \( (\text{Proc}(t), \phi(\Gamma_{\text{init}}), T) \), and
- events in \( \text{initials}(\text{Proc}(t), \Gamma_{\text{init}}, T)/\text{tr} \) do not have outputs of type \( t \) from outside of \( \{0\ldots B - 1\} \),
then \((tr, X)\) is a failure of \((\text{Proc}(t), \Gamma_{\text{init}}, T)/tr\).

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

- \(\text{Proc}(t)\) satisfies \textbf{SeqNorm} and \textbf{RevPosConjEqT}\(_F\),
- \(\phi\) is a \(B\)-collapsing function, and
- \(T\) is an instantiation of type \(t\) of size at least \(B + 1\).

Suppose further that

(i) \(tr \in \text{traces}(\text{Proc}(t), \Gamma_{\text{init}})\),

(ii) \((\phi(tr), X) \in \text{failures}(\text{Proc}(t), \phi(\Gamma_{\text{init}}))\), and

(iii) if \(P\) is a configuration such that \((\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{tr} P\), then every output value of type \(t\) of every event in \(\text{initials}(P)\) is in \(\{0 \ldots B - 1\}\).

Then \((tr, X) \in \text{failures}(\text{Proc}(t), \Gamma_{\text{init}})\).

**Proof:** We prove the result using a structural induction on \(\text{Proc}(t)\).

STOP

If \(\text{Proc}(t) = \text{STOP}\), then \(tr = \langle \rangle\) and \((\langle \rangle, X) \in \text{failures}(\text{STOP}, \Gamma_{\text{init}})\) for all \(X\).

**Prefix**

Suppose that \(\text{Proc}(t) = \alpha \rightarrow \text{Proc}'(t)\) for some construct \(\alpha = c_{\$1,x_1: X_1} \ldots c_{\$k,x_k: X_k}\) and some process syntax \(\text{Proc}'(t)\). We now consider two cases.

**Subcase 1.** Suppose that \(tr = \langle \rangle\).

The fact that \((\phi(tr), X) \in \text{failures}(\text{Proc}(t), \phi(\Gamma_{\text{init}}))\) implies that there exists an environment \(\Gamma\) such that \(\text{dom}(\Gamma) = \$_t(\alpha)\) and

\[(\text{Proc}(t), \phi(\Gamma_{\text{init}})) \xrightarrow{\langle \rangle} (P(t), \phi(\Gamma_{\text{init}}) \oplus \Gamma) \text{ ref } X,
\]

where \(P(t)\) is like \(\text{Proc}(t)\), but with some substitutions of concrete values for the nondeterministic input variables of non-\(t\) types of \(\alpha\) and with the effects of the application of \textit{Replace}_{\$\leftrightarrow t}!, as dictated by the SSOS firing rules for prefix (see Section 4.3.2).

Then,

\[(\text{Proc}(t), \Gamma_{\text{init}}) \xrightarrow{\langle \rangle} (P(t), \Gamma_{\text{init}} \oplus \Gamma)
\]

by resolving the nondeterministic selections of \(\alpha\) (if any) in the same way. We now show that \((P(t), \Gamma_{\text{init}} \oplus \Gamma) \text{ ref } X\). Observe that the only difference between the initial
events of two configurations \((S, \Gamma_1)\) and \((S, \Gamma_2)\) are the output values of type \(t\) that come from the environments \(\Gamma_1\) and \(\Gamma_2\). Therefore,

\[
\text{initials}(P(t), \Gamma_{\text{init}} \oplus \Gamma) = \\
\{ c.v_1' \ldots v_k' | (\forall i \in !'(\alpha) \bullet v_i' = (\Gamma_{\text{init}} \oplus \Gamma)(x_i)) \land \exists c.v_1 \ldots v_k \in \text{initials}(P(t), \phi(\Gamma_{\text{init}}) \oplus \Gamma) \bullet \\
(\forall i \in \{1 \ldots k\} \setminus !'(\alpha) \bullet v_i' = v_i) \}
\]

Let \(v\) be such that \(c.v_1 \ldots v_k\) is in \(\text{initials}(P(t), \phi(\Gamma_{\text{init}}) \oplus \Gamma)\) and let \(v'\) be such that \(c.v_1' \ldots v_k'\) is in \(\text{initials}(P(t), \Gamma_{\text{init}} \oplus \Gamma)\). Also, let \(i \in !'(\alpha)\). Hence, \(v_i = (\phi(\Gamma_{\text{init}}) \oplus \Gamma)(x_i)\) and \(v_i' = (\Gamma_{\text{init}} \oplus \Gamma)(x_i)\). Therefore, by assumption (iii) of the proposition, \(v_i' \in \{0 \ldots B - 1\}\). So, thanks to the properties of \(\phi\), \(\phi(v_i') = v_i\). Hence 

\[
\forall i \in !'(\alpha) \bullet v_i' = (\Gamma_{\text{init}} \oplus \Gamma)(x_i) = (\phi(\Gamma_{\text{init}}) \oplus \Gamma)(x_i) = v_i.
\]

Hence,

\[
\text{initials}(P(t), \Gamma_{\text{init}} \oplus \Gamma) = \text{initials}(P(t), \phi(\Gamma_{\text{init}}) \oplus \Gamma).
\]

Now, since \((P(t), \Gamma_{\text{init}} \oplus \Gamma)\) is stable, \((P(t), \Gamma_{\text{init}} \oplus \Gamma) \text{ ref } Y\) for every \(Y \subseteq \Sigma \setminus \text{initials}(P(t), \Gamma_{\text{init}} \oplus \Gamma)\). However, the fact that \((P(t), \phi(\Gamma_{\text{init}}) \oplus \Gamma) \text{ ref } X\) implies that \(X \subseteq \Sigma \setminus \text{initials}(P(t), \phi(\Gamma_{\text{init}}) \oplus \Gamma)\), so by (5.19) \((P(t), \Gamma_{\text{init}} \oplus \Gamma) \text{ ref } X\). This implies that \((\langle \rangle, X) \in \text{failures}(\text{Proc}(t), \Gamma_{\text{init}})\), as required.

**Subcase 2.** Suppose that \(tr \neq \langle \rangle\).

Then \(tr = (e)^{-1} tr'\) for some visible event \(e\) that matches \(\alpha\) and some trace \(tr'\). Let \(\Gamma = \phi(\Gamma_{\text{init}}) \oplus \text{Match}(\alpha, \phi(e))\) and \(\Gamma' = \Gamma_{\text{init}} \oplus \text{Match}(\alpha, e)\). From the assumptions of the proposition we can infer that

\[
tr' \in \text{traces}(P(t), \Gamma'),
\]

and

\[(\phi(tr'), X) \in \text{failures}(P(t), \Gamma),\]

where \(P(t)\) is like \(\text{Proc}'(t)\), but with some substitutions of concrete values for the non-\(t\) type input variables of \(\alpha\), as dictated by the SSOS firing rules for prefix (see Section 4.3.2). Assumption (iii), combined with the fact that \((\text{Proc}(t), \Gamma_{\text{init}}) \stackrel{(e)}{\longrightarrow} (P(t), \Gamma')\), 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 \(\text{initials}(P)\) is in \(\{0 \ldots B - 1\}\). Observe that \(\Gamma = \phi(\Gamma')\). So, by the inductive hypothesis for \(P(t)\),

\[
(tr', X) \in \text{failures}(P(t), \Gamma'),
\]

which implies that

\[(tr, X) \in \text{failures}(\text{Proc}(t), \Gamma_{\text{init}}).\]
External choice
Suppose that $\text{Proc}(t) = P(t) \boxdot Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. We consider two cases.

Subcase 1. Suppose that $tr = \langle \emptyset \rangle$.
Then, by the assumptions of the proposition and from the denotational semantics of $\boxdot$ (see e.g. [Ros97, Chapter 8]),

$$(\langle \emptyset \rangle, X) \in \text{failures}(P(t), \phi(\Gamma_{\text{init}}))$$

and

$$(\langle \emptyset \rangle, X) \in \text{failures}(Q(t), \phi(\Gamma_{\text{init}})).$$

We have that $\langle \emptyset \rangle \in \text{traces}(P(t), \Gamma_{\text{init}})$ and $\langle \emptyset \rangle \in \text{traces}(Q(t), \Gamma_{\text{init}})$. Assumption (iii), together with the semantics of external choice, implies that if $P$ is a configuration such that $(P(t), \Gamma_{\text{init}}) \xrightarrow{tr} P$ or $(Q(t), \Gamma_{\text{init}}) \xrightarrow{tr} P$, then every output of type $t$ of every event in $\text{initials}(P)$ is in $\{0 \ldots B-1\}$. So, using the inductive hypothesis for $P(t)$ and $Q(t)$, we get that

$$(\langle \emptyset \rangle, X) \in \text{failures}(P(t), \Gamma_{\text{init}}) \quad \text{and} \quad (\langle \emptyset \rangle, X) \in \text{failures}(Q(t), \Gamma_{\text{init}}).$$

Hence, by the denotational semantics of external choice,

$$(\langle \emptyset \rangle, X) \in \text{failures}(\text{Proc}(t), \Gamma_{\text{init}}).$$

Subcase 2. Suppose that $tr \neq \langle \emptyset \rangle$.
Then $\text{SeqNorm}$ implies that the channels of the initial events of $P(t)$ and $Q(t)$ are disjoint, so either

$$(\phi(tr), X) \in \text{failures}(P(t), \phi(\Gamma_{\text{init}})) \quad \text{and} \quad tr \in \text{traces}(P(t), \Gamma_{\text{init}})$$

or

$$(\phi(tr), X) \in \text{failures}(Q(t), \phi(\Gamma_{\text{init}})) \quad \text{and} \quad tr \in \text{traces}(Q(t), \Gamma_{\text{init}}).$$

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 $(P(t), \Gamma_{\text{init}}) \xrightarrow{tr} P$, then every output of type $t$ of every event in $\text{initials}(P)$ is in $\{0 \ldots B-1\}$. Then, the inductive hypothesis for $P(t)$ implies that

$$(tr, X) \in \text{failures}(P(t), \Gamma_{\text{init}}).$$

Hence, by the denotational semantics of external choice,

$$(tr, X) \in \text{failures}(\text{Proc}(t), \Gamma_{\text{init}}).$$

126
**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 $\text{Proc}(t) = P(t) \triangleright Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. We consider two cases.

**Subcase 1.** Suppose that $\text{tr} = \emptyset$.
Then, by the assumptions of the proposition and from the denotational semantics of $\triangleright$ (see e.g. [Ros97, Chapter 8]),

$$(\emptyset, X) \in \text{failures}(Q(t), \phi(\Gamma_{init})).$$

Clearly, $\emptyset \in \text{traces}(Q(t), \Gamma_{init})$. In addition, assumption (iii), combined with the fact that $\text{Proc}(t), \Gamma_{init} \xrightarrow{\emptyset} (Q(t), \Gamma_{init})$, implies that if $P$ is a configuration such that $(Q(t), \Gamma_{init}) \xrightarrow{t} P$, then every output of type $t$ of every event in $\text{initials}(P)$ is in $\{0 \ldots B-1\}$. So, the inductive hypothesis for $Q(t)$ implies that

$$(\emptyset, X) \in \text{failures}(Q(t), \Gamma_{init}).$$

Hence, using the denotational semantics of sliding choice,

$$(\emptyset, X) \in \text{failures}(\text{Proc}(t), \Gamma_{init}).$$

**Subcase 2.** Suppose that $\text{tr} \neq \emptyset$.
This case is identical to Subcase 2 of the external choice case, above.

**Conditional choice**
Suppose that $\text{Proc}(t) = \text{if } \text{cond} \text{ then } P(t) \text{ else } Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. If $\text{cond}$ is not in $\text{Cond}$, then it immediately evaluates to $\text{True}$ or $\text{False}$, in which case the result is implied by the inductive hypothesis for $P(t)$ or $Q(t)$, respectively. For a condition $\text{cond}$ in $\text{Cond}$ we perform a case analysis on the truth value of the evaluation of $\text{cond}$ within environments $\Gamma_{init}$ and $\phi(\Gamma_{init})$.

**Case 1.** Suppose that $\llbracket \text{cond} \rrbracket_{\phi(\Gamma_{init})} = \llbracket \text{cond} \rrbracket_{\Gamma_{init}} = \text{True}$.
Then

$$(\phi(\text{tr}), X) \in \text{failures}(P(t), \phi(\Gamma_{init})) \quad \text{and} \quad \text{tr} \in \text{traces}(P(t), \Gamma_{init}).$$

In addition, assumption (iii), combined with the fact that $\text{Proc}(t), \Gamma_{init} \xrightarrow{\emptyset} (P(t), \Gamma_{init})$, implies that if $P$ is a configuration such that $(P(t), \Gamma_{init}) \xrightarrow{t} P$, then every output of type $t$ of every event in $\text{initials}(P)$ is in $\{0 \ldots B-1\}$. Then, the inductive hypothesis for $P(t)$ implies that

$$(\text{tr}, X) \in \text{failures}(P(t), \Gamma_{init}).$$
Therefore,

\[(tr, X) \in failures(Proc(t), \Gamma_{init}).\]

**Case 2.** Suppose that \(\llbracket \text{cond} \rrbracket_{\phi(\Gamma_{ INIT})} = \llbracket \text{cond} \rrbracket_{\Gamma_{ INIT}} = False.\)

This case is like Case 1, above, with \(Q(t)\) in place of \(P(t)\).

**Case 3.** Suppose that \(\llbracket \text{cond} \rrbracket_{\phi(\Gamma_{INIT})} = True \land \llbracket \text{cond} \rrbracket_{\Gamma_{INIT}} = False.\)

Then

\[(\phi(tr), X) \in failures(P(t), \phi(\Gamma_{INIT})) \quad \text{and} \quad tr \in traces(Q(t), \Gamma_{INIT}).\]

However, \(Proc(t)\) satisfies \(\text{RevPosConjEqT}_F\), so \((Q(t), \phi(\Gamma_{INIT})) \sqsubseteq_F (P(t), \phi(\Gamma_{INIT})).\)

Hence,

\[(\phi(tr), X) \in failures(Q(t), \phi(\Gamma_{INIT})).\]

In addition, assumption (iii), combined with the fact that \((Proc(t), \Gamma_{INIT}) \not\rightarrow (Q(t), \Gamma_{INIT})\), implies that if \(P\) is a configuration such that \((Q(t), \Gamma_{INIT}) \not\rightarrow tr \rightarrow P\), then every output of type \(t\) of every event in \(\text{initials}(P)\) is in \([0..B-1]\). The inductive hypothesis for \(Q(t)\) implies now that

\[(tr, X) \in failures(Q(t), \Gamma_{INIT}),\]

which implies that

\[(tr, X) \in failures(Proc(t), \Gamma_{INIT}).\]

**Case 4.** Suppose that \(\llbracket \text{cond} \rrbracket_{\phi(\Gamma_{INIT})} = False \land \llbracket \text{cond} \rrbracket_{\Gamma_{INIT}} = True.\)

This case is not possible, since \(\text{cond}\) is a conjunction of equality tests and for no function \(\phi\) we can ever have \(x = y\) and \(\phi(x) \neq \phi(y)\).

**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. \(E(X) = P\)). 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 \(\text{Thresh}_F\) such that if \(Spec(t)\) and \(Impl(t)\) fulfil certain requirements, then, for all \(B \geq \text{Thresh}_F\), if \(Spec([0..B]) \sqsubseteq_F \phi(Impl([0..n]))\) implies that \(Spec([0..n]) \sqsubseteq_F Impl([0..n])\) 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 \ldots B\}, thanks to the external renaming by \(\phi\). However, if we can produce a fixed process \(Abstr\) such that \(Abstr \sqsubseteq_F \phi(Impl(\{0 \ldots n\}))\) for all \(n \geq B\) for some \(B \geq \text{Thresh}_F\) by construction, then it is enough to perform a single refinement check, namely \(\text{Spec}(\{0 \ldots \max\{\text{Thresh}_F, B\}\}) \sqsubseteq_F Abstr\), in order for Theorem 5.2.24 to imply that \(\text{Spec}(\{0 \ldots n\}) \sqsubseteq_F \text{Impl}(\{0 \ldots n\})\) for all \(n \geq \max\{\text{Thresh}_F, B\}\). Combining this with manual checks for all \(n \in \{0 \ldots \max\{\text{Thresh}_F, B\} - 1\}\), we obtain the answer to whether \(\text{Spec}(T) \sqsubseteq_F \text{Impl}(T)\) for all instantiations \(T\) of type \(t\).

Recall that, given a symbolic conditional event \(\text{cond}\) and an environment \(\Gamma\), \([\text{cond}]_\Gamma\) denotes the truth value of the proposition obtained from \(\text{cond}\) by substituting all free variables of type \(t\) with their corresponding values contained within \(\Gamma\). We lift the definition of \([\cdot]_\Gamma\) to symbolic traces without visible symbolic events in the following way. Given a symbolic trace \(\sigma\) in \((\text{Cond} \cup \{\tau\})^*\) we let \([\sigma]_\Gamma\) be equal to \(\sqcup\{[\text{cond}]_\Gamma \mid \text{cond in } \sigma, \text{cond } \in \text{Cond}\}\). In addition, given a non-empty sequence \(\text{seq}\) we define \(\text{last(seq)}\) to be the last element of \(\text{seq}\).

**Theorem 5.2.24. (Extendibility of stable failures refinement of systems with replicated components)**

Suppose that

(i) \(\text{Spec}(t)\) satisfies \(\text{SeqNorm}\) and \(\text{RevPosConjEqT}_t\), and is divergence-free and has a finite alphabet for every finite instantiation of type \(t\),

(ii) \(\text{Impl}(t)\) satisfies \(\text{TypeSym}\),

(iii) no construct \(\alpha\) in \(\text{Spec}(t)\) combines nondeterministic inputs of type \(t\) and deterministic input of any type, i.e. if \(#)\$\{\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) \(\text{Thresh}_F = \max\{\text{Thresh}_T, \)

\[
\max\{\text{Thresh}^{\text{\Gamma}}_t(\sigma, \Gamma) + \text{Thresh}^{\ast}_t(\sigma, \Gamma) \mid 
\sigma \in \text{SymbolicTraces}(\text{Spec}(t)) 
\land (\sigma = \langle \rangle \lor \text{last(}\sigma\text{) } \in \text{Visible} \land \Gamma \in \text{Env}(T))\},
\]

where

- \(\text{Thresh}^{\text{\Gamma}}_t(\sigma, \Gamma)\) counts the number of unique output variables of type \(t\) in all the visible symbolic events \(\epsilon\) available in \(\text{Spec}(t)\) immediately after \(\sigma\) such that all conditionals between the last symbolic event of \(\sigma\) and \(\epsilon\) evaluate to True, i.e.

\[
\#\{x_i \mid \text{Spec}(t) \xrightarrow{\sigma} \rho \xrightarrow{\epsilon} \ast \land \rho \in (\text{Cond } \cup \{\tau\})^* \land \Gamma = \text{True} 
\land \epsilon = c_1x_1:X_1 \ldots c_kx_k:X_k \in \text{Visible} \land i \in !^t(\epsilon)\},
\]

- \(\text{Thresh}^{\ast}_t(\sigma, \Gamma)\) counts the number of (not necessarily unique) input variables of type \(t\) in all the visible symbolic events \(\epsilon\) available in \(\text{Spec}(t)\) immediately after \(\sigma\) such that all conditionals between the last symbolic event of \(\sigma\) and \(\epsilon\) evaluate to True, i.e.
\[
\text{Thresh}_T(\sigma, \Gamma) = \Sigma\{\#?^t(\epsilon) \mid \text{Spec}(t) \xrightarrow{\sigma_1} \xrightarrow{\beta} \xrightarrow{\epsilon} \rho \in (\text{Cond} \cup \{\tau\})^* \\
\land \epsilon \in \text{Visible} \land [\rho]_{\Gamma} = \text{True}\},
\]

\(\circ\) \(\text{Thresh}_T\) is as in Theorem 5.2.15,

(vi) \(B \geq \text{Thresh}_F\), and

(vii) \(\phi\) is a \(B\)-collapsing function.

Then, if \(\text{Spec}(\phi(T)) \subseteq_F \phi(\text{Impl}(T))\), then \(\text{Spec}(T) \subseteq_F \text{Impl}(T)\).

**Proof:** Suppose that the refinement \(\text{Spec}(\phi(T)) \subseteq_F \phi(\text{Impl}(T))\) holds and assume for contradiction that \(\text{Spec}(T) \not\subseteq_F \text{Impl}(T)\). Refinement in the stable failures model implies refinement in the traces model, so \(\text{Spec}(\phi(T)) \subseteq_T \phi(\text{Impl}(T))\). Then, by Theorem 5.2.15 (which is applicable since its assumptions are weaker than those of this theorem), \(\text{Spec}(T) \subseteq_T \text{Impl}(T)\).

Consider a minimal counterexample \((tr, X)\) to the refinement \(\text{Spec}(T) \subseteq_F \text{Impl}(T)\), i.e.

\[
(tr, X) \in \text{failures}(\text{Impl}(T)),
\]

\[
(tr, X) \notin \text{failures}(\text{Spec}(T)),
\]

\[
\forall e \in X \bullet (tr, X \setminus \{e\}) \in \text{failures}(\text{Spec}(T)).
\]

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

\[
\text{Spec}(T) \xrightarrow{tr} P_e(T) \land P_e(T) \xrightarrow{ref} (X \setminus \{e\}) \land P_e(T) \xrightarrow{\epsilon} .
\]

This also means that every event in \(X\) is accepted in some stable state of \(\text{Spec}(T)/tr\). Hence,

\[
X \subseteq \text{initials}(\text{Spec}(T)/tr).
\]

We now aim to show that \(X\) is dependent upon at most \(\text{Thresh}_F\) 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 \(\alpha\) of \(\text{Spec}(t)\) (uniqueness follows from Corollary 5.1.15) with \(#\$^t(\alpha) > 0\) (which, by assumption (iii), implies that \(#?^t(\alpha) = 0\)) and show that

\[
\forall v' : \{1 \ldots k\} \to \text{Value} \mid
\left(\forall i \in \$^t(\alpha) \bullet v'_i \in T\right) \land \left(\forall i \in \{1 \ldots k\} \setminus \$^t(\alpha) \bullet v'_i = v_i\right) \bullet
c.v'_1 \ldots v'_k \in X.
\]
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 $\alpha$ are resolved in a way such that the values $v'_i$ are chosen instead of $v_i$ for all $i \in \mathcal{S}'(\alpha)$; call this stable state $P_{e'}(T)$ (see Figure 5.2 for an example). The initials of $P_{e'}(T)$ are the same 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, $e' \not\in X$ 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 $\alpha$ of $Spec(t)$ (uniqueness guaranteed by Corollary 5.1.15) with $\#?(\alpha) > 0$ (which, by assumption (iii), implies that $\#?\mathcal{S}(\alpha) = 0$) and show that

\[
\forall v' : \{1 \ldots k\} \rightarrow \text{Value} \mid
\begin{align*}
(\forall i \in \{1 \ldots k\} \setminus ?(\alpha) \bullet v'_i = v_i) & \land (\exists i \in ?(\alpha) \bullet v'_i \neq v_i) \\
\bullet c.v'_1 \ldots v'_k & \not\in X.
\end{align*}
\tag{5.26}
\]

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.

\[
(Spec(t), \{\}, T) \xrightarrow{\Delta} (Spec'(t), \Gamma, T)
\tag{5.27}
\]

\footnote{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 $\tau$ and $s \setminus \{\tau\} = tr$. Observe that

$$(Spec(t), \{\}, T) \xrightarrow{tr} (Spec'(t), \Gamma, T).$$

So, thanks to the uniqueness of $Spec'(t)$ and $\Gamma$, and since $s$ does not end with a $\tau$,

initials($Spec(T)/tr$) = initials($Spec'(t), \Gamma, T$).

Let $S = S_1 \cup S_2$, where

$$S_1 = \{v_i \mid e = c.v_1 \ldots v_k \in initials(Spec(T)/tr) \land e \text{ matches a unique construct } \alpha \text{ of } Spec(t) \land i \in !'(\alpha)\}$$

and

$$S_2 = \{v_i \mid c.v_1 \ldots v_k \in X \land i \in \{1 \ldots k\} \land v_i \in T \land \exists v' \in T \bullet c.v_1 \ldots v_{i-1}, v', v_{i+1} \ldots v_k \notin X\}.$$ 

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 $\Gamma$. 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 $\sigma = \emptyset$ or last($\sigma$) $\in$ Visible and $Spec(t) \xrightarrow{\sigma \rightarrow s}$ 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

$$\text{Threshold}_{\sigma}(\sigma, \Gamma) = \sum \{\#	ext{?}(\epsilon) \mid Spec(t) \xrightarrow{\sigma \rightarrow s} \xrightarrow{\epsilon} s \land \rho \in (\text{Cond} \cup \{\tau\})^* \land \epsilon \in \text{Visible} \land \|\rho\|_\Gamma = \text{True}\}.$$ 

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

$$\text{Threshold}_{\sigma}(\sigma, \Gamma) = \sum \{\#x_i \mid Spec(t) \xrightarrow{\sigma \rightarrow s} \xrightarrow{\epsilon} s \land \rho \in (\text{Cond} \cup \{\tau\})^* \land \epsilon \in \text{Visible} \land i \in !'(\epsilon)\}.$$ 

132
Summarising the last two paragraphs: all elements of $S$ either match deterministic inputs in the events in $S_2$ (at most $\text{Thresh}_{\Sigma}(\sigma, \Gamma)$ such values), or match outputs in either $S_1$ or $S_2$ (at most $\text{Thresh}_{\Pi}(\sigma, \Gamma)$ such values). Therefore,

\[
\#S \leq \text{Thresh}_{\Pi}(\sigma, \Gamma) + \text{Thresh}_{\Pi}(\sigma, \Gamma) \\
\leq \max\{\text{Thresh}_{\Pi}(\sigma', \Gamma') + \text{Thresh}_{\Pi}(\sigma', \Gamma') | \sigma' \in \text{SymbolicTraces}(\text{Proc}(t)) \land (\sigma' = \emptyset) \lor \text{last}(\sigma') \in \text{Visible} \land \Gamma' \in \text{Env}(T)\} \\
\leq \text{Thresh}_{\Pi} \\
\leq B.
\]

Let $\pi : T \rightarrow T$ 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

\[
(\pi(tr), \pi(X)) \in \text{failures}(\text{Impl}(T)) \tag{5.30}
\]

and

\[
(\pi(tr), \pi(X)) \notin \text{failures}(\text{Spec}(T)). \tag{5.31}
\]

Now, by the denotational semantics of renaming [Ros97, Chapter 8],

\[
\text{failures}(\phi(\text{Impl}(T))) = \{(\phi(tr'), Y) | (tr', \phi^{-1}(Y)) \in \text{failures}(\text{Impl}(T))\}. \tag{5.32}
\]

Let $c.v_1 .. v_k$ 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 $T$, $c.v_1 .. v_{i-1}.v'.v_{i+1} .. v_k$ is in $X$. Therefore, if $c.w_1 .. w_k$ 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 $c.w_1 .. w_{i-1}.w'.w_{i+1} .. w_k$ is in $\pi(X)$ for all values $w'$ in $T$. This, thanks to the definition of $\phi$, 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

\[
(\phi(\pi(tr)), \phi(\pi(X))) \in \text{failures}(\phi(\text{Impl}(T))).
\]

However, $\text{Spec}(\phi(T)) \sqsubseteq F \phi(\text{Impl}(T))$, so

\[
(\phi(\pi(tr)), \phi(\pi(X))) \in \text{failures}(\text{Spec}(\phi(T))). \tag{5.33}
\]

We now show that

\[
(\phi(\pi(tr)), \pi(X)) \in \text{failures}(\text{Spec}(T)). \tag{5.34}
\]
Firstly, (5.33) implies that there exists a configuration \((P(t), \Gamma, \phi(T))\) such that
\[
\text{Spec}(\phi(T)) = (\text{Spec}(t), \{\}, \phi(T)) \xrightarrow{\phi(\pi(t))} (P(t), \Gamma, \phi(T)) \text{ ref } \phi(\pi(X)).
\] (5.35)
Let \(e = c.v_1 \ldots v_k\) be in \(\phi(\text{initials}(P(t), \Gamma, T))\) and let \(\alpha\) be the unique construct of \(\text{Spec}(t)\) that \(e\) matches (with uniqueness following from Corollary 5.1.15). Then there exists a function \(v': \{1 \ldots k\} \rightarrow \text{Value}\) such that
\[
e' = c.v'_1 \ldots v'_k \in \text{initials}(P(t), \Gamma, T)
\] (5.36)
and \(\phi(e') = e\), i.e.
\[
\forall i \in \{1 \ldots k\} \bullet \phi(v'_i) = v_i.
\] (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 \(\text{initials}(P(t), \Gamma, \phi(T))\) had been chosen before this state was reached (and are necessarily in \(\phi(T)\)). Hence and since \(\Gamma \in \text{Env}(\phi(T))\) implies \(\Gamma \in \text{Env}(T)\), we have that \(\text{initials}(P(t), \Gamma, \phi(T))\) and \(\text{initials}(P(t), \Gamma, T)\) are the same, except for values of deterministic inputs of type \(t\). Formally,
\[
\text{initials}(P(t), \Gamma, \phi(T)) = \{c.w_1 \ldots w_k \mid c.w'_1 \ldots w'_k \in \text{initials}(P(t), \Gamma, T) \text{ matches construct } \alpha' \wedge w \in \{1 \ldots k\} \rightarrow \text{Value} \wedge (\forall i \in \{1 \ldots k\} \setminus ?^t(\alpha') \bullet w_i = w'_i) \wedge \forall i \in ?^t(\alpha') \bullet w_i \in \phi(T)\}.
\] (5.38)
Also, since \(\Gamma \in \text{Env}(\phi(T))\), all values of type \(t\) used in the events of \(\text{initials}(P(t), \Gamma, T)\) and that match nondeterministic selections or outputs, are in \(\phi(T) = \{0 \ldots B\}\):
\[
\forall i \in \#^t(\alpha) \cup !^t(\alpha) \bullet v'_i \in \{0 \ldots B\},
\] which, thanks to the properties of \(\phi\), and combined with the fact that for all non-\(t\) values \(\text{val}, \phi(\text{val}) = \text{val}\), gives us that
\[
\forall i \in \{1 \ldots k\} \setminus ?^t(\alpha) \bullet \phi(v'_i) = v'_i.
\] Hence and by the definition of \(v'\) (5.37),
\[
\forall i \in \{1 \ldots k\} \setminus ?^t(\alpha) \bullet v_i = v'_i.
\] (5.39)
In addition, since \(c.v_1 \ldots v_k\) is in \(\phi(\text{initials}(P(t), \Gamma, T))\), it must be that
\[
\forall i \in ?^t(\alpha) \bullet v_i \in \phi(T).
\] (5.40)
Combining (5.36), (5.38), (5.39) and (5.40) we get that \(e\) is in \(\text{initials}(P(t), \Gamma, \phi(T))\). Hence
\[
\phi(\text{initials}(P(t), \Gamma, T)) \subseteq \text{initials}(P(t), \Gamma, \phi(T)).
\] (5.41)
Conversely, let \(e = c.v_1 \ldots v_k\) be in \(\text{initials}(P(t), \Gamma, \phi(T))\). From Proposition 5.1.18 we can infer that
\[
\text{initials}(P(t), \Gamma, \phi(T)) \subseteq \text{initials}(P(t), \Gamma, T),
\]
so \( e \) is in \( \text{initials}(P(t), \Gamma, T) \). Hence, \( \phi(e) \) is in \( \phi(\text{initials}(P(t), \Gamma, T)) \). However, for all \( i \in \{1 \ldots k\} \), \( v_i \) is either a value of a non-\( t \) type or it is a value in \( \phi(T) \). Hence,

\[
\forall i \in \{1 \ldots k\} \cdot \phi(v_i) = v_i,
\]

so \( \phi(e) = e \) and therefore \( e \in \phi(\text{initials}(P(t), \Gamma, T)) \). Hence,

\[
\text{initials}(P(t), \Gamma, \phi(T)) \subseteq \phi(\text{initials}(P(t), \Gamma, T)).
\]

Combining the above with (5.41) we have that

\[
\phi(\text{initials}(P(t), \Gamma, T)) = \text{initials}(P(t), \Gamma, \phi(T)). \tag{5.42}
\]

We now aim to show that

\[
(P(t), \Gamma, T) \text{ ref } \pi(X). \tag{5.43}
\]

Suppose for a contradiction that there exists an event \( x \) in \( \pi(X) \cap \text{initials}(P(t), \Gamma, T) \). Then (5.42) implies that \( \phi(x) \in \phi(\pi(X)) \cap \text{initials}(P(t), \Gamma, \phi(T)) \). This means that \( \phi(\pi(X)) \cap \text{initials}(P(t), \Gamma, \phi(T)) \) is non-empty, which contradicts (5.35). Hence, (5.43) holds. Finally, applying Corollary 5.1.19 to (5.35) we have that

\[
(\text{Spec}(t), \{\}, T) \xrightarrow{\phi(\pi(tr))} (P(t), \Gamma, T).
\]

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), \( \pi(tr) \in \text{traces}(\text{Impl}(T)) \). However, we have already shown that \( \text{Spec}(T) \subseteq_T \text{Impl}(T) \), so

\[
\pi(tr) \in \text{traces}(\text{Spec}(T)) = \text{traces}(\text{Spec}(t), \{\}, T).
\]

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 \( \pi \), implies that every output value of type \( t \) of every event in \( \text{initials}(\text{Spec}(T)/\pi(tr)) \) is in \( \{0 \ldots B - 1\} \), which gives us condition (iii). Hence, we can infer from it that

\[
(\pi(tr), \pi(X)) \in \text{failures}(\text{Spec}(t), \{\}, T) = \text{failures}(\text{Spec}(T)).
\]

This is a contradiction to (5.31), which completes our proof. \( \square \)

Some observations related to Theorem 5.2.24 are now in order.

**Remark 5.2.25.** For every verification problem, the value of \( \text{Thresh}_{F} \) in Theorem 5.2.15 depends only on the specification.

**Remark 5.2.26.** For all specifications with a finite SSLTS, the value of \( \text{Thresh}_{F} \) in Theorem 5.2.15 can be calculated in a finite amount of time. The term \( \text{Thresh}_{T} \) can be calculated as in Remark 5.2.19. The other term can be obtained by calculating \( \text{Thresh}_{\kappa_1}(\sigma, \Gamma) + \text{Thresh}_{\kappa_2}(\sigma, \Gamma) \) for each symbolic state that is either the initial state or that has an incoming visible transition.
Remark 5.2.27. The values of $\text{Thresh}_{\mathcal{U}}(\sigma, \Gamma)$ and $\text{Thresh}_{\mathcal{I}}(\sigma, \Gamma)$, used in Theorem 5.2.24, have simple upper bounds: $W^\text{Spec}$ and $I^\text{Spec}$, respectively, defined as in [Ros97, Chapter 15]:

- $W^\text{Spec}$ is the maximum number of type $t$ values that the specification needs to store for use within subsequent events,
- $I^\text{Spec}$ 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, $\max\{ \text{Thresh}_{\mathcal{T}}, W^\text{Spec} + I^\text{Spec} \}$ is an upper bound for $\text{Thresh}_{\mathcal{F}}$. In practice this approximation may be improved by counting the number of values that were double-counted in obtaining $W^\text{Spec} + I^\text{Spec}$. This can result in up to halving the calculated upper bound for $\text{Thresh}_{\mathcal{F}}$ and often gives precisely the value of $\text{Thresh}_{\mathcal{F}}$.

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 $Spec(\phi(T)) \subseteq_F \phi(Impl(T))$ and yet $Spec(T) \not\subseteq_F Impl(T)$. 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

$Spec(t) = c\$x:t?y:t → STOP$

and

$Impl(t) = \Box y:t • c?x:(t \setminus \{y\})!y → STOP.$

Note, in particular that $Impl(t)$ satisfies $\text{TypeSym}$.

Let $B$ be an arbitrary positive number, and let $\phi$ 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 $\text{traces}(\phi(Impl(T))) \subseteq \text{traces}(Spec(\phi(T)))$. Further, whatever value $Impl(T)$ chooses for $y$, $(T \setminus \{y\}) \cap \{B..N\} \neq \{\}$; hence $(\langle \rangle, \{c.B.B\}) \not\in \text{failures}(\phi(Impl(T)))$. This helps to see that

\[
\text{failures}(\phi(Impl(T))) = \{((\langle \rangle, X) \mid X \subseteq \{c.x.x \mid x \in \{0..B-1\}\})
\cup \{((c.x.y), X) \mid y \in \{0..B\} \land x \in \{0..B\} \setminus \{y\} \land X \subseteq \Sigma \}
\cup \{((\langle \rangle, X) \mid X \subseteq \{c.x.y \mid x \in \{0..B\} \setminus \{p\} \land y \in \{0..B\}\} \land p \in \{0..B\})
\cup \{((c.x.y), X) \mid x, y \in \{0..B\} \land X \subseteq \Sigma \}
\} = \text{failures}(Spec(\phi(T))).
\]

Hence,

$Spec(\phi(T)) \subseteq_F \phi(Impl(T))$. 

136
However,

\((\langle \rangle, \{ c.x.x \mid x \in T \}) \in \text{failures} (\text{Impl}(T)) \setminus \text{failures} (\text{Spec}(T))\),

so \(\text{Spec}(T) \not\subseteq_F \text{Impl}(T)\).

**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

\[ \text{Spec}(t) = c\$x:t?y:Y \rightarrow \text{STOP} \]

and

\[ \text{Impl}_B(t) = \square y:Y \bullet (\forall X \subseteq t \land \#X = B + 1 \bullet c?x:X!y \rightarrow \text{STOP}) \, . \]

Note, in particular, that \(\text{Impl}(t)\) satisfies \textbf{TypeSym}.

Let \(\phi\) be as in the statement of Theorem 5.2.24 and let \(T = \{0 \ldots N\}\), where \(N \geq 2B + 1\). It is easy to see that \(\text{traces}(\phi(\text{Impl}(T))) \subseteq \text{traces}(\text{Spec}(\phi(T)))\). Further, whatever value \(\text{Impl}_B(T)\) chooses for \(X, X \cap \{B \ldots N\} \neq \{\}\); hence \((\langle \rangle, \{ c.B.y \}) \notin \text{failures}(\phi(\text{Impl}_B(T)))\) for every \(y \in Y\). This helps to see that

\[ \begin{align*}
\text{failures}(\phi(\text{Impl}_B(T))) & \subseteq \{(\langle \rangle, R) \mid R \subseteq \{ c.x.y \mid x \in \{0 \ldots B - 1\} \land y \in Y\}\} \\
& \cup \{(\{c.x.y\}, R) \mid x \in \{0 \ldots B\} \land y \in Y \land R \subseteq \Sigma\} \\
& \subseteq \{(\langle \rangle, R) \mid R \subseteq \{c.x.y \mid x \in \{0 \ldots B\} \setminus \{p\} \land y \in Y \land p \in \{0 \ldots B\}\} \\
& \cup \{(\{c.x.y\}, R) \mid x \in \{0 \ldots B\} \land y \in Y \land R \subseteq \Sigma\} \\
& = \text{failures}(\text{Spec}(\phi(T))).
\end{align*} \]

Hence,

\[ \text{Spec}(\phi(T)) \subseteq_F \phi(\text{Impl}_B(T)). \]

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, \(\text{Impl}_B(T)\) 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 \(\text{Spec}(T)\), so \(\text{Spec}(T) \not\subseteq_F \text{Impl}_B(T)\).

**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 \textit{CLIENT}' to be the following (and we add suitable events using the agree channel to the alphabets of clients).
CLIENT′(i, j, principal, t) =
  if principal = yes
    then session_end.i.j.f \rightarrow CLIENT(i, t)
      □
      communicate.i.j \rightarrow session_end.i.j.s \rightarrow CLIENT(i, t)
      □
      agree!lj'?k:t \rightarrow session_end.i.j.s \rightarrow CLIENT(i, t)
  else session_end.j.i.f \rightarrow CLIENT(i, t)
    □
    communicate.j.i \rightarrow session_end.j.i.s \rightarrow CLIENT(i, t)
    □
    agree!i!l?k:t \rightarrow session_end.j.i.s \rightarrow CLIENT(i, t)

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

\[ Impl(t) = (SERVER(t) \parallel CLIENTS(t)) \]

and

\[ Spec(t) = session_start$t\rightarrow (session_end.i.j.f \rightarrow Spec(t)
  □
  communicate.i.j \rightarrow session_end.i.j.s \rightarrow Spec(t)
  □
  agree!lj'?k:t \rightarrow session_end.i.j.s \rightarrow Spec(t)) \]

Firstly, we have that \( \text{Thresh}_T = 2 \). In order to establish \( \text{Thresh}_F \), 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 $\Gamma$:
  $\text{Thresh}_{\text{!}}(\sigma, \Gamma) = 0, \text{Thresh}_{?}(\sigma, \Gamma) = 0$,

• state $s_1, \sigma = \langle \text{session\_start}$i$:$t$s\_j$:$t$\rangle$, any $\Gamma$:
  $\text{Thresh}_{\text{!}}(\sigma, \Gamma) = 2, \text{Thresh}_{?}(\sigma, \Gamma) = 1$,

• state $s_2, \sigma = \langle \text{session\_start}$i$:$t$s\_j$:$t$, communicate.$i$.$j$\rangle$, any $\Gamma$:
  $\text{Thresh}_{\text{!}}(\sigma, \Gamma) = 2, \text{Thresh}_{?}(\sigma, \Gamma) = 0$

• state $s_2, \sigma = \langle \text{session\_start}$i$:$t$s\_j$:$t$, agree!$i$!$j$?$k$:$t$\rangle$, any $\Gamma$:
  $\text{Thresh}_{\text{!}}(\sigma, \Gamma) = 2, \text{Thresh}_{?}(\sigma, \Gamma) = 0$

Any other symbolic trace $\sigma$ ends in one of the symbolic states listed above, so there cannot be any other values of $\text{Thresh}_{\text{!}}(\sigma, \Gamma), \text{Thresh}_{?}(\sigma, \Gamma)$. Hence, $\text{Thresh}_{F} = \max\{\text{Thresh}_{\text{T}}, 3\} = 3$ in this case. Then, if we can find a process $\text{Abstr}$ such that $Spec(\{0 \ldots 3\}) \sqsubseteq_{T} \text{Abstr}$ and $\text{Abstr} \sqsubseteq_{T} \phi(\text{Impl}(0 \ldots n))$ for all $n \geq 3$, then $Spec(T) \sqsubseteq_{T} \text{Impl}(T)$ 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

$$Spec(T) \sqsubseteq \text{Impl}(T)$$

for all sufficiently large $T$. However, the main difference is that while data independence allows one to deduce the above given that

$$Spec(\phi(T)) \sqsubseteq \text{Impl}(\phi(T)),$$  \hspace{1cm} (5.44)

where $\phi$ is a $B$-collapsing function (see Definition 5.0.1) for some sufficiently large $B$, our theory requires that

$$Spec(\phi(T)) \sqsubseteq \phi(\text{Impl}(T)).$$  \hspace{1cm} (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 $\text{Abstr}$ that are independent of the instantiation of type $t$ and such that $\text{Abstr} \sqsubseteq \phi(\text{Impl}(T))$ for all sufficiently large $T$ by construction. Then, thanks to transitivity of refinement, verifying that $\text{Abstr}$ 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 \textbf{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 \textbf{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:

- \textbf{PosConjEqT} (to be satisfied in Chapter 6 by controller processes) and \textbf{RevPosConjEqT} (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,

- \textbf{NoEqT}, which bans conditionals on $t$, and

- \textbf{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 \textbf{TypeSym} was defined semantically, we also provided a set of sufficient syntactic conditions that imply \textbf{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 \textbf{SeqNorm} (Definition 4.1.4). Each such check is guaranteed to terminate. Checking for the semantic requirements of the \textbf{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 \textbf{TypeSym}. When doing so, one needs to remember the possibility of false negatives. Such syntactic checks always terminate.

Checking for \textbf{RevPosConjEqT} 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 $\text{cond}$ then $P(x_1,\ldots,x_k)$ else $Q(x_1,\ldots,x_k)$” in a given specification, where $\text{cond} \in \text{Cond}$, $\text{cond}$ is a positive conjunction of equality tests on $t$ and $Q(v_1,\ldots,v_k) \subseteq P(v_1,\ldots,v_k)$ for all values $v_1,\ldots,v_k$. In general, the problem of \textbf{RevPosConjEqT} satisfiability is undecidable, since a general (undecidable) PMCP problem of the form $Spec(x) \subseteq Impl(x)$, where $x$ is a parameter, can be reduced to checking whether

$$in\ ?\ i\ ?\ j\ :\ x \rightarrow \text{if } i = j \text{ then } Impl(x) \text{ else } Spec(x)$$
satisfies \textbf{RevPosConjEqT}''. 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 \( \text{Spec}(t_1, t_2, \ldots, t_n) \) must now be data independent in all of the \( n \) types and each implementation \( \text{Impl}(t_1, \ldots, t_n) \) must satisfy \textbf{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 \((\text{Thresh}_1, \text{Thresh}_2, \ldots, \text{Thresh}_n)\), where each \( \text{Thresh}_i \) is a threshold for the collapsing of the values of type \( t_i \).
In this chapter we return to the following problem, which we attempted to solve in Chapter 3.

Given a concurrent system $\text{Impl}(T)$, consisting of $\#T$ similar processes, and a specification $\text{Spec}(T)$, is it true that $\text{Impl}(T)$ satisfies $\text{Spec}(T)$ 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, \( \phi \), 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 \( \phi \). 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 \( \phi \)-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 \( \{1 \ldots n\} \), 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

\[
Impl(t) = \left(\left(\left(\text{Nodes}(t) \setminus X(t)\right) \parallel \text{Ctrl}(t)\right) \setminus Y(t)\right) \setminus Z(t),
\]

(6.1)
where

- $Nodes(t)$ is of the form $\| \ myId \in t \bullet [A(myId, t)] N_{myId}(t)$,
- $N_{myId}(t)$ models a single, finite-state node with identity $myId$ and with awareness of all node identities in $t$; we assume that $N_{myId}(t)$ 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 $N_{otherId}(t)$, 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 $PosConjEqT_T$ (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 $N_{myId}(t)$ 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 RevPosConjEqT_T (Definition 5.2.8) is undecidable. This follows from the following, stronger result.

Theorem 6.1.3. The problem of uniform verification of $\| i \in t \bullet N_{myId}(t)$ with a specification satisfying SeqNorm and RevPosConjEqT_T, where $N_{myId}(t)$ is defined as above, is undecidable.

Proof sketch: We can reduce the Halting Problem [Sip96] to the problem of uniform verification of $\| i \in t \bullet N_{myId}(t)$ as follows. We use $\| i \in t \bullet N_{myId}(t)$ 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.

\[
STOP \subseteq_T Nodes(t) \setminus (\Sigma \setminus \{\| halt \|})
\]

\(^1\)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, \( N_{\text{myId}}(t) \), combined with data independence of nodes, implies that for all instantiations \( T \) of type \( t \) and all identities \( me \) in \( T \) the process \( N_{me}(T) \) can be represented using the configuration \( (N_{\text{myId}}(t), \{\text{myId} \mapsto me\}, T) \) (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 \( (N_{\text{myId}}(t), \{\text{myId} \mapsto me\}, T) \) and \( N_{\text{myId}}(T)[\{\text{myId} \mapsto me\}] = N_{me}(T) \) are related under the bisimulation relation \( B \) used in the proof of Theorem 4.5.1.

**Example 6.1.5.** Let
\[
N_{\text{myId}}(t) = \text{in}?i:t \rightarrow \text{out.myId}.i \rightarrow \text{STOP}
\]
be a template for a node process with identity \( \text{myId} \) and awareness of identities of type \( t \). Suppose further that \( T \) is an instantiation of type \( t \) and \( me \) is in \( T \). Then
\[
N_{me}(T) = (\text{in}?i:t \rightarrow \text{out.myId}.i \rightarrow \text{STOP}, \{\text{myId} \mapsto me\}, T).
\]

End of example.

We now define a useful quantity that we will repeatedly use within this chapter.

**Definition 6.1.6.** We let \( W_N \) denote the maximum number of node identities, other than \( \text{myId} \), that \( N_{\text{myId}}(t) \) can store for future use.

\( W_N \) 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 \( \tau \) 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 \( W_N \) 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.** \( W_N \) is 2 for both of the following two processes.
\[
N^1_{\text{myId}}(t) = \text{in}?i:t \rightarrow (\text{out}_1.\text{myId}.i \rightarrow \text{STOP})
\]
\[
\quad \text{in}?j:t \rightarrow \text{out}_2.\text{myId}.i.j \rightarrow \text{STOP}
\]
\[
N^2_{\text{myId}}(t) = \text{in}_1\$i:t \rightarrow \text{STOP} \quad \text{\&} \quad \text{in}_2\$j:t \rightarrow \text{STOP}
\]

End of example.
In addition to storing up to $W_N$ identities of other nodes at any given time, every node process stores its own identity at all times. Therefore, each state of $N_{me}(T)$ 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 $0 \leq n \leq W_N$, 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 $N^1_{myId}(T)$ and $N^2_{myId}(T)$ be as in Example 6.1.7. Then the states of $N^1_0((0,1))$ are

$$
\begin{align*}
(N^1_{myId}(t), \{myId \mapsto 0\}, \{0,1\}), \\
(cst, \{myId \mapsto 0, i \mapsto 0\}, \{0,1\}), \\
(cst, \{myId \mapsto 0, i \mapsto 1\}, \{0,1\}), \\
(STOP, \{myId \mapsto 0\}, \{0,1\}), \\
(out_2.myId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 0, j \mapsto 0\}, \{0,1\}), \\
(out_2.myId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 0, j \mapsto 1\}, \{0,1\}), \\
(out_2.myId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 1, j \mapsto 0\}, \{0,1\}), \\
(out_2.myId.i.j \rightarrow STOP, \{myId \mapsto 0, i \mapsto 1, j \mapsto 1\}, \{0,1\}),
\end{align*}
$$

where

$$
\begin{align*}
cst &= out_1.myId.i \rightarrow STOP \\
\square_{in?}:t &\rightarrow out_2.myId.i.j \rightarrow STOP
\end{align*}
$$

and the states of $N^2_0(\{0,1\})$ are

$$
\begin{align*}
(N^2_{myId}(t), \{myId \mapsto 1\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 0\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 1\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, j \mapsto 0\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, j \mapsto 1\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 0, j \mapsto 0\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 0, j \mapsto 1\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 1, j \mapsto 0\}, \{0,1\}), \\
(in_1.i \rightarrow STOP \sqcap in_2$:j$:t \rightarrow STOP, \{myId \mapsto 1, i \mapsto 1, j \mapsto 1\}, \{0,1\}), \\
(STOP, \{myId \mapsto 1\}, \{0,1\}).
\end{align*}
$$

*End of example.*
We make the notational convention that if \((cst, \Gamma, T)\) and \((cst', \Gamma', T)\) 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 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 \(I \subseteq \{1 \ldots k\}\) such that for all \(i\) in \(I\), \(v_i = me\); and \(payloadIDs = \langle v_j \mid j \in \{1 \ldots k\} \setminus I \land v_j \in T \rangle\). 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 \(I_{me} \subseteq \{1 \ldots k\}\) and \(I_{other} \subseteq \{1 \ldots k\}\) such that for all \(i\) in \(I_{me}\), \(v_i = me\); for all \(i\) in \(I_{other}\), \(v_i = other\); and \(payloadIDs = \langle v_j \mid j \in \{1 \ldots k\} \setminus \{i, j\} \land v_j \in T \rangle\). 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 \(payloadIDs = \langle v_i \mid i \in \{1 \ldots k\} \land v_j \in T \rangle\). 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 \(A, B\) and \(C\) such
that for each \( me \) and \( T \),
\[
A(me, T) = \{ c_\alpha.f_\alpha(me, \text{payloadIDs}) \mid \alpha \in A \land \text{payloadIDs} \in T^{n_\alpha} \} \cup \{ c_\beta.f_\beta(me, \text{other}, \text{payloadIDs}), c_\beta.f_\beta(\text{other}, me, \text{payloadIDs}) \mid \beta \in B, \text{other} \in T \setminus \{ me \} \land \text{payloadIDs} \in T^{n_\beta} \} \cup \{ c_\gamma.f_\gamma(\text{payloadIDs}) \mid \gamma \in C \land \text{payloadIDs} \in T^{n_\gamma} \},
\]
(6.2)
where \( c_\alpha, c_\beta, c_\gamma \) are channel names, \( f_\alpha, f_\beta, f_\gamma \) are function, \( n_\alpha, n_\beta, n_\gamma \) are natural number, and the events in the three sets are, respectively, \( \alpha \)-, \( \beta \)- and \( \gamma \)-events.

**Remark 6.1.9.** We note that:

(i) our construction allows values of types different from \( t \) to be present in \( \alpha \)-, \( \beta \)- and \( \gamma \)-events,

(ii) given \( T \) and an identity \( me \) in \( T \), the alphabet \( A(me, T) \) is partitioned by the sets of \( \alpha \)-, \( \beta \)- and \( \gamma \)-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
\[
N_{\text{myId}}(t) = a?x:X \rightarrow b?i:t \rightarrow c!\text{myId}?i:t \rightarrow \text{STOP}
\]
\[
\quad \Box
\]
\[
c?i:t!\text{myId} \rightarrow d?i:t?j:t \rightarrow \text{STOP}
\]
\[
\quad \Box
\]
\[
c!\text{myId}?i:t \rightarrow \text{STOP}
\]

for some type \( X \) not related to \( t \). Also, let
\[
f_{(a,x)}(\langle \rangle) = x, \quad \text{for } x \in X
\]
\[
f_b(\langle i \rangle) = i
\]
\[
f_c(me, other, \langle \rangle) = me.\text{other}
\]
\[
f_d((i,j)) = i.j
\]
\[
f_e(me, \langle i \rangle) = me.i.
\]

Then
\[
A(me, T) = \{ e.f_e(me, \langle i \rangle) \mid i \in T \} \cup \{ c.f_c(me, other, \langle \rangle), c.f_c(other, me, \langle \rangle) \mid other \in T \setminus \{ me \} \} \cup \{ a.f_{(a,x)}(\langle \rangle), b.f_b(\langle i \rangle), d.f_d((i,j)) \mid x \in X \land i,j \in T \}
\]
is the alphabet of \( N_{me}(T) \), where the three sets give the \( \alpha \)-, \( \beta \)- and \( \gamma \)-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 \( \phi \) 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 \( (cst, \Gamma, T) \) and \( (cst', \Gamma', T') \) are equivalent if

- their control parts are related under \( \sim (cst \sim cst') \),
- their environments are equal under \( \phi (\phi(\Gamma) = \phi(\Gamma')) \), and
- their underlying types are equal under \( \phi (\phi(T) = \phi(T')) \).

Whenever \( \phi \) is clear from the context, we simply write \( \approx \). The equivalence class of state \( (cst, \Gamma, T) \) under \( \approx \) is \n
\[
[(cst, \Gamma, T)] = \{(cst', \Gamma', T') \mid cst \sim cst' \land \phi(\Gamma) = \phi(\Gamma') \land \phi(T) = \phi(T')\}.
\]

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 \( \phi \), since their \textit{myId} variables will always be mapped to values that are different under \( \phi \). 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 \( m = W_N \) (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 \( ABS^{T,B,n}_\infty \) such that for sufficiently large \( T \) and \( n \),

\[
ABS^{T,B,n}_\infty \subseteq T \phi(Nodes(T)). \tag{6.3}
\]

As mentioned before, the abstract model consists of two parts. The first is the parallel composition of the nodes \( N_0(\{0..B+n\}), \ldots, N_{B-1}(\{0..B+n\}) \). The second part is
a counter state machine modelling all the nodes with identities in \{B \ldots \#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 \ldots \#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 \( \zeta_T^T(\mathcal{N}_B(\{0 \ldots B + m\})) \) that abstracts the nodes \( \mathcal{N}_B(\{0 \ldots B + m\}), \ldots, \mathcal{N}_{\#T-1}(\{0 \ldots B + m\}) \), i.e. where those nodes have awareness of the node identities in \{0 \ldots B + m\}. We present this construction in Section 6.2.1. The counter state machine is constructed from the state machine as a function of just \( \mathcal{N}_B(\{0 \ldots B + m\}) \); however \( \forall i, j \in \{B \ldots B + m\} \cdot \mathcal{N}_i(\phi(T)) = \mathcal{N}_j(\phi(T))[\cdot \cdot \cdot j/i] \), so we write the counter state machine as a function of just \( \mathcal{N}_B(\{0 \ldots B + m\}) \).

- 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 \( \mathcal{N}_0(\{0 \ldots B + n\}), \ldots, \mathcal{N}_{B-1}(\{0 \ldots B + n\}) \), and then apply \( \phi \) renaming.

This two-stage process is necessary to ensure synchronisations on \( \beta \)-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:

\[
ABS_T^{T,B,0} \triangleq \left( \begin{array}{c}
\big[ \big[ \big[ i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] \mathcal{N}_i(\{0 \ldots B\}) \big] \big] \big] \\
\bigcup_{i=0}^{B-1} A(i, \{0 \ldots B\}) \bigg| \bigg| A_i^B \\
\zeta_T^T(\mathcal{N}_B(\{0 \ldots B + 1\})) \bigg| \bigg| \mathcal{R} \bigg)
\right),
\]

where

\[
A_i^B \triangleq \mathcal{R}(A(B, \{0 \ldots B + 1\}) \cup A(B + 1, \{0 \ldots B + 1\}))
\]

and

\[
ABS_T^{T,B,m} \triangleq \phi \left( \begin{array}{c}
\big[ \big[ \big[ i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B + m\})] \mathcal{N}_i(\{0 \ldots B + m\}) \big] \big] \big] \\
\bigcup_{i=0}^{B-1} A(i, \{0 \ldots B+m\}) \bigg| \bigg| \bigcup_{i=B}^{B+m} A(i, \{0 \ldots B+m\}) \\
\zeta_T^T(\mathcal{N}_B(\{0 \ldots B + m\}))
\end{array} \right).
\]

The main characteristic of \( ABS_T^{T,B,0} \) and \( ABS_T^{T,B,m} \) 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

1. $(S(i), s_0(i), \Sigma(i, \{0 \ldots B + m\}) \cup \{\tau\}, \rightarrow_i)$ be the state machine of node $\mathcal{N}_i(\{0 \ldots B + m\})$, for $i \in \{B \ldots B + m\}$,

2. $S = \bigcup\{S(i) \mid i \in \{B \ldots B + m\}\}$ (i.e. $S$ is the set of state of nodes $\mathcal{N}_B(\{0 \ldots B + m\}), \ldots, \mathcal{N}_{B + m}(\{0 \ldots B + m\}$, and

3. $E = \{[st\_i], \ldots, [st\_k]\} = S/\approx$ (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

$$\forall i \in \{B \ldots B + m\} \bullet s_0(i) = (\mathcal{N}_{myId}(t), \{myId \mapsto i\}, \{0 \ldots B + m\}).$$

Hence,

$$\forall i, j \in \{B \ldots B + m\} \bullet s_0(i) \approx s_0(j).$$

since $\phi(B) = \phi(B + 1) = \ldots \phi(B + m) = B$. Without loss of generality, we assume that $[st\_i]$ is the equivalence class within $E$ that contains all initial states $s_0(i)$ for $i$ in $\{B \ldots B + m\}$.

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

$$c_{\infty}^{T}(\mathcal{N}_B(\{0 \ldots B + m\})) = (A_\infty(T), a_\infty(T), \Sigma^r_\infty(T), \rightarrow_\infty(T))$$

that abstracts the nodes $\mathcal{N}_B(\{0 \ldots B + m\}), \ldots, \mathcal{N}_{#T-1}(\{0 \ldots B + m\})$. Whenever $T$ is clear from context, we omit it from the right-hand side and simply write $(A_\infty, a_\infty, \Sigma^r_\infty, \rightarrow_\infty)$. Let $T$ be of size at least $B + 1$. Recall that $k$ is the number of equivalence classes of node states.

The states of $A_\infty$ 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 $i \in \{B \ldots #T - 1\}$, $s_i$ is a state of $\mathcal{N}_i(\{0 \ldots B + m\})$, i.e. it records the local state of node $i$. Then the corresponding abstract state (in $A_\infty$) 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.

$$\forall j \in \{1 \ldots k\} \bullet c_j = \#\{i \in \{B \ldots #T - 1\} \mid s_i \approx st\_j\}.$$
Example 6.2.1. Let
\[ N_{\text{myId}}(t) = \text{init!myId?other:} t \rightarrow \text{send1.myId.other} \]
\[ \rightarrow \text{send2.myId.other} \rightarrow N_{\text{myId}}(t) \]
and let
\[ \text{cst}_0 = \text{init!myId?other:} t \rightarrow \text{send1.myId.other} \rightarrow \text{send2.myId.other} \rightarrow N_{\text{myId}}(t) \]
\[ \text{cst}_1 = \text{send1.myId.other} \rightarrow \text{send2.myId.other} \rightarrow N_{\text{myId}}(t) \]
\[ \text{cst}_1' = \text{send2.myId.other} \rightarrow N_{\text{myId}}(t) \]

Let \( \sim \) be an equivalence relation on \( \{\text{cst}_0, \text{cst}_1, \text{cst}_1'\} \) such that \( \text{cst}_1 \sim \text{cst}_1' \). Let \( B = 1, m = 1 \) and let \( \phi \) be a \( B \)-collapsing function (i.e. \( \phi \) maps \( \{B..B+m\} = \{1, 2\} \) onto \( \{B\} = \{1\} \)). Then there are three equivalence classes of node states under \( \sim \) (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 \( N_B(T),...,N_{B+m}(T) \)), where \( T = \{0..B+m\} = \{0, 1, 2\} \):

\[ [(\text{cst}_0, \{\text{myId} \mapsto 1\}, T)] = \{(\text{cst}_0, \{\text{myId} \mapsto 1\}, T), (\text{cst}_0, \{\text{myId} \mapsto 2\}, T), ... \} \]
\[ [(\text{cst}_1, \{\text{myId} \mapsto 1\}, T)] = \{(\text{cst}_1, \{\text{myId} \mapsto 1\}, T), (\text{cst}_1, \{\text{myId} \mapsto 1\}, T), (\text{cst}_1, \{\text{myId} \mapsto 2\}, T), ... \} \]
\[ [(\text{cst}_1, \{\text{myId} \mapsto 1\}, T)] = \{(\text{cst}_1, \{\text{myId} \mapsto 1\}, T), (\text{cst}_1, \{\text{myId} \mapsto 1\}, T), (\text{cst}_1, \{\text{myId} \mapsto 2\}, T), ... \} \]

Let \( T \) be some type. Since there are 3 equivalence classes in \( E \), each state of \( \zeta^2_T (N_B(T)) \) 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 \( N_B(T),...,N_{#T-1}(T) \) 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 \( \text{count} \) be a state in \( A_\infty \). Given a state of a node process, say \( st \), we let \( \text{count}[st] \) denote the coordinate of \( \text{count} \) corresponding to equivalence class \([st]\). Then, we let \( \text{count}[st \leftarrow x] \) denote a tuple of counters like \( \text{count} \), but with \( x \) substituted for the value of the counter corresponding to the equivalence class \([st]\). Also, we let \( \text{count}[st++] \) and \( \text{count}[st--] \) denote the tuple \( \text{count} \) with the counter corresponding to the equivalence class \([st]\) incremented and decremented, respectively, by 1, i.e. \( \text{count}[st++] = \text{count}[st \leftarrow \text{count}[st]+1] \) and \( \text{count}[st--] = \text{count}[st \leftarrow \text{count}[st]-1] \).

As a shorthand notation we will also write, for example, \( \text{count}[st--, st'++] \) to mean the tuple \( \text{count} \) with the counter corresponding to \([st']\) incremented and the counter corresponding to \([st]\) decremented, where both operations happen atomically; observe that this leaves \( \text{count} \) unchanged if \( st \approx st' \).

Since we assumed that \([st]\) is the equivalence class of the initial states of nodes with identities in \( \{B..B+m\} \), the initial state, \( a_\infty \), is the tuple \((#T - B, 0,..., 0)\).
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, $\Sigma^\infty$, it is enough to observe that the transformed system can only perform events that one of $N_B(\{0 \ldots B + m\}), \ldots, N_{B+m}(\{0 \ldots B + m\})$ can perform. Hence, we define

$$\Sigma^\infty = \bigcup \{ \Sigma(i, \{0 \ldots B + m\}) \mid i \in \{B \ldots B + m\} \}$$

and

$$\Sigma^\tau = \Sigma^\infty \cup \{ \tau \}.$$  

Finally, we define the transition relation, $\xrightarrow{-\to}^\infty$. Let $count$ and $count'$ be two states in $A^\infty$. We define the transitions from $count$ to $count'$ by a case analysis on the type of the events.

1. There is a $\tau$-transition between $count$ and $count'$ if and only if
   - there is a concrete state $st$ in $S$ in which $\tau$ 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,

$$count \xrightarrow{\tau}^\infty count' \iff \exists me \in \{B \ldots B + m\} \bullet \exists st, st' \in S(me) \bullet st \xrightarrow{me} st' \land count([st]) \geq 1 \land count' = count([st]-, [st']++).$$  

2. Suppose now that $me$ is a fixed identity in $\{B \ldots B + m\}$. Let $e$ in $\Sigma^\infty$ be either
   - an $\alpha$-event of the form $c.f_\alpha(me, payloadIDs)$, or
   - a $\beta$-event of the form $c.f_\beta(me, other, payloadIDs)$, where $other \in \{0 \ldots B - 1\}$.

This case is very similar to the case for $\tau$, above. Formally, for every $e$ as above,

$$count \xrightarrow{e}^\infty count' \iff \exists st, st' \in S(me) \bullet st \xrightarrow{me} st' \land count([st]) \geq 1 \land count' = count([st]-, [st']++).$$  

3. Now, let $me$ and $other$ be two distinct identities in $\{B \ldots B + m\}$. Let $e$ in $\Sigma^\infty$ be a $\beta$-event of the form $c.f_\beta(me, other, payloadIDs)$. 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,

\[
\begin{align*}
\text{count} & \xrightarrow{e} \text{count}' \\
\iff & \exists st_{me}, st'_{me} \in S(me), st_{other}, st'_{other} \in S(other) \bullet \\
& st_{me} \xrightarrow{e}_{me} st'_{me} \land st_{other} \xrightarrow{e}_{other} st'_{other} \\
& \land count[st_{me}] \geq p \land count[st_{other}] \geq p \\
& \land count' = count[st_{me} \rightarrow, st_{other} \rightarrow, st'_{me} \rightarrow, st'_{other} \rightarrow], \\
& \text{where } p = 2 \text{ if } [st_{me}] = [st_{other}] \text{ and } p = 1 \text{ otherwise.}
\end{align*}
\]

(4) Finally, let \( e \in \Sigma_{\infty} \) be a \( \gamma \)-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 \( \{B \ldots \#T - 1\} \) between states of \( S \), such that for every \( i \) in \( \{1 \ldots k\} \), 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 \( \{1 \ldots k\} \), there are \( count[st_{i}] \) many (not necessarily all distinct) concrete states \( src_{j}^{i} \) in \( S \) for \( 1 \leq j \leq count[st_{i}] \), in all of which \( e \) can be executed to reach some state \( tgt_{j}^{i} \), and
- every counter of \( count' \) is equal to the total number of the target states \( tgt_{j}^{i} \) that are in the equivalence class to which the counter corresponds.

Formally, for every \( e \) as above,

\[
\begin{align*}
\text{count} & \xrightarrow{e} \text{count}' \\
\iff & \exists src, tgt : \mathbb{N} \times \mathbb{N} \rightarrow S \bullet \\
& \forall i \in \{1 \ldots k\}, j \in \{1 \ldots count[st_{i}]\} \bullet \\
& src_{j}^{i} \approx st_{i} \land \exists me \in \{B \ldots B + m\} \bullet src_{j}^{i} \xrightarrow{e}_{me} tgt_{j}^{i} \\
& \land \forall st \in S \bullet count'[st] = \#\{(i, j) \mid i \in \{1 \ldots k\} \land j \in \{1 \ldots count[st_{i}]\} \\
& \land tgt_{j}^{i} \approx st\}.
\end{align*}
\]

By clause (iii) of Remark 6.1.9, the four parts of the definition of \( \xrightarrow{}_{\infty} \) are pairwise disjoint, so together they combine into a well-formed definition of the abstract transition relation.
Example 6.2.2. Recall the $N_{myld}(t)$ 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 $\zeta^T(\mathcal{N}_1(\{0, 1, 2\}))$. Figure 6.1 shows the state machine of $\zeta^T(\mathcal{N}_1(\{0, 1, 2\}))$. 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 $B + 1$,

$\text{ABS}_T^{B,0} \subseteq T \phi(\text{Nodes}(T))$.

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 $\phi$). We therefore ban
conditional choice (i.e. we assume that nodes satisfy \( \text{NoEqT}^t \); 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 \( N \) satisfies \( \text{NoEqT}^t \). Let \( T \) and \( \hat{T} \) be two instantiations of type \( t \) and let \( \psi \) be a function from \( T \) to \( \hat{T} \). Then if \((\text{cst}, \Gamma, T) \xrightarrow{a} \text{me}(\text{cst}', \Gamma', T)\) for some \( \text{me} \in T \), then \((\text{cst}, \psi(\Gamma), \hat{T}) \xrightarrow{\psi(a)} \text{me}(\psi(\text{cst'}), \psi(\Gamma'), \hat{T})\).

**Proof:** If \( \text{cst} \) is a control state of \( N \), then it must also be a control state of \( N \) (by data independence). Suppose that \((\text{cst}, \Gamma, T) \xrightarrow{a} \text{me}(\text{cst'}, \Gamma', T)\). Since \( N \) 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 \( \text{dom}(\Gamma) \); hence, from the state \((\text{cst}, \psi(\Gamma), \hat{T})\), 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 \((\text{cst}, \psi(\Gamma), \hat{T})\) the value \( \psi(v) \) could be input. This implies that \((\text{cst}, \psi(\Gamma), T)\) 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 \( \psi(\Gamma') \). \( \square \)

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 \( \psi \) is a function from \( T \) to \( \{0 \ldots K\} \) such that

- \( \psi(v) = v \) for all \( v \) in \( \{0 \ldots B - 1\} \), and
- \( \psi(v) \in \{B \ldots K\} \) for all \( v \) in \( \{B \ldots \# T - 1\} \).

**Remark 6.3.3.** Let \( T, B \) and \( K \geq B \) be given. Then if \( \phi \) is a \( B \)-collapsing function and \( \psi \) is a \( K \)-precollapsing function, then \( \phi \circ \psi = \phi \).

We now aim for a result that says that if:

- the nodes satisfy \( \text{NoEqT}^t \),
- 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 $\text{ABS}_\infty^\text{T,B,0}$. 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 $a \in A(i,T) \iff \phi(a) \in A(i,\{0..B\})$.

**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 $\mathcal{I}(T)$, $(\text{cst}_i, \Gamma_i, T)$ is equal to some state $N_i'(T)$ of node $N_i(T)$, then we define the semantics of

$$\parallel i \in \mathcal{I}(T) \bullet [A(i,T)] (\text{cst}_i, \Gamma_i, T)$$

to be the same as the semantics of

$$\parallel i \in \mathcal{I}(T) \bullet [A(i,T)] N_i'(T).$$

**Lemma 6.3.5.** Suppose that $N_{\text{myId}}(t)$ satisfies $\text{NoEqT}^t$. Let $T$ be an instantiation of type $t$ such that $\#T \geq B+1$. For all $i$ in $T$, let $(\text{cst}_i, \Gamma_i, T)$ be a state of $N_i(T)$ and suppose that

$$\parallel i \in \{0..B-1\} \bullet [A(i,T)] (\text{cst}_i, \Gamma_i, T)$$

for some control states $\text{cst}_i'$ and environments $\Gamma_i'$. Then

$$\phi(a) \parallel i \in \{0..B-1\} \bullet [A(i,\{0..B\})] (\text{cst}_i, \phi(\Gamma_i), \{0..B\})$$

for some control states $\text{cst}_i'$ and environments $\Gamma_i'$. Then

$$\phi(a) \parallel i \in \{0..B-1\} \bullet [A(i,\{0..B\})] (\text{cst}_i', \phi(\Gamma_i'), \{0..B\}).$$

**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 $\alpha$-event of the form $c.f_a(me,payloadIDs)$ (i.e. a private event of $N_{\text{me}}(T)$) for some identity $me$ in $\{0..B-1\}$ and some list of identities $payloadIDs$,

(ii) a $\beta$-event of the form $c.f_\beta(me,other,payloadIDs)$ (i.e. an event shared by $N_{\text{me}}(T)$ and $N_{\text{other}}(T)$) for some identities $me$ in $\{0..B-1\}$ and $other$ in $\{B..\#T-1\}$ and some list of identities $payloadIDs$, or
(iii) the internal $\tau$ event.

Then
\[
\begin{align*}
(cst_{me}, \Gamma_{me}, T) & \xrightarrow{a} \phi(a) (cst'_{me}, \Gamma'_{me}, T) \\
\end{align*}
\]
and
\[
\forall i \in \{0..B-1\} \setminus \{me\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma_i, T),
\]
where $me$ is as in (i) or (ii) if $a$ is an $\alpha$- or $\beta$-event, respectively, and is some identity in $\{0..B-1\}$ if $a = \tau$. Hence, 
\[
\forall i \in \{0..B-1\} \setminus \{me\} \bullet (cst_i, \phi(\Gamma_i), \{0..B\}) = (cst'_i, \phi(\Gamma'_i), \{0..B\}).
\]  
(6.10)

Using (6.8), Lemma 6.3.1 implies that
\[
(cst_{me}, \phi(\Gamma_{me}), \{0..B\}) \xrightarrow{\phi(a)} (cst'_{me}, \phi(\Gamma'_{me}), \{0..B\}).
\]
However, $\phi(me) = me$ (since $me \in \{0..B-1\}$), so
\[
(cst_{me}, \phi(\Gamma_{me}), \{0..B\}) \xrightarrow{\phi(a)} (cst'_{me}, \phi(\Gamma'_{me}), \{0..B\}).
\]  
(6.11)

In addition, if $a$ is as in either (i) or (ii), then
\[
\phi(a) \in A(me, \{0..B\}) \setminus \bigcup_{i \in \{0..B\} \setminus \{me\}} A(i, \{0..B\})
\]
using Lemma 6.3.4. Also, if $a = \tau$, then $\phi(a) = \tau$. Hence, combining (6.10) and (6.11), we get that
\[
\begin{align*}
\| i \in \{0..B-1\} \bullet [A(i, \{0..B\})] (cst_i, \phi(\Gamma_i), \{0..B\}) & \xrightarrow{\phi(a)}
\| i \in \{0..B-1\} \bullet [A(i, \{0..B\})] (cst'_i, \phi(\Gamma'_i), \{0..B\}),
\end{align*}
\]
as required.

**Case 2.** Suppose that $a$ is a $\beta$-event of the form $c_{f\beta}(x, y, \text{payloadIDs})$ (i.e. an event shared by $N_x(T)$ and $N_y(T)$) for some distinct identities $x$ and $y$ in $\{0..B-1\}$ and some list of identities payloadIDs.

Then
\[
\begin{align*}
(cst_x, \Gamma_x, T) & \xrightarrow{a} (cst'_x, \Gamma'_x, T), \\
(cst_y, \Gamma_y, T) & \xrightarrow{a} (cst'_y, \Gamma'_y, T),
\end{align*}
\]  
(6.12)  
(6.13)

and
\[
\forall i \in \{0..B-1\} \setminus \{x, y\} \bullet (cst'_i, \Gamma'_i, T) = (cst_i, \Gamma, T).
\]  
(6.14)

From (6.12) and (6.13) we can infer using Lemma 6.3.1 that
\[
(cst_x, \phi(\Gamma_x), \{0..B\}) \xrightarrow{\phi(a)} (cst'_x, \phi(\Gamma'_x), \{0..B\})
\]
158
and
\[(cst_y, \phi(\Gamma_y), \{0 \ldots B\}) \xrightarrow{\phi(y)} (cst'_y, \phi(\Gamma'_y), \{0 \ldots B\}).\]

Since \(x\) and \(y\) are both in \(\{0 \ldots B - 1\}\), we have that \(\phi(x) = x\) and \(\phi(y) = y\). Therefore,
\[(cst_x, \phi(\Gamma_x), \{0 \ldots B\}) \xrightarrow{\phi(x)} (cst'_x, \phi(\Gamma'_x), \{0 \ldots B\}) \quad (6.15)\]
and
\[(cst_y, \phi(\Gamma_y), \{0 \ldots B\}) \xrightarrow{\phi(y)} (cst'_y, \phi(\Gamma'_y), \{0 \ldots B\}). \quad (6.16)\]

In addition, from (6.14) we have that
\[
\forall i \in \{0 \ldots B - 1\} \setminus \{x, y\} \bullet (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}) = (cst_i, \phi(\Gamma_i), \{0 \ldots B\}). \quad (6.17)\]

Finally,
\[
\phi(a) \in (A(x, \{0 \ldots B\}) \cap A(y, \{0 \ldots B\}) \setminus \bigcup_{i \in \{0 \ldots B\} \setminus \{x, y\}} A(i, \{0 \ldots B\})
\]
by Lemma 6.3.4. Hence, and from (6.15), (6.16) and (6.17),
\[
\| i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) \xrightarrow{\phi(a)} \| i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}),
\]
as required.

**Case 3.** Suppose that \(a\) is a \(\gamma\)-event of the form \(c.f.(payloadIDs)\) (i.e. an event shared by all nodes) for some list of identities \(payloadIDs\).

Then
\[
\forall i \in \{0 \ldots B - 1\} \bullet (cst_i, \Gamma_i, T) \xrightarrow{a} (cst'_i, \Gamma'_i, T). \quad (6.18)
\]

Hence, using Lemma 6.3.1 we can infer that
\[
\forall i \in \{0 \ldots B - 1\} \bullet (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) \xrightarrow{\phi(a)} (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}).\]

However, for all \(i\) in \(\{0 \ldots B - 1\}\), \(\phi(i) = i\), so
\[
\forall i \in \{0 \ldots B - 1\} \bullet (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) \xrightarrow{\phi(a)} (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}). \quad (6.19)
\]
Further,
\[
\forall i \in \{0 \ldots B - 1\} \bullet \phi(a) \in A(i, \{0 \ldots B\})
\]
by Lemma 6.3.4. Hence, and from (6.19), we have that
\[
\| i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) \xrightarrow{\phi(a)} \| i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}),
\]
as required. This completes our proof. ■

Lemma 6.3.6. Suppose that \( N_{\text{msgId}}(t) \) satisfies \( \text{NoEqT}^t \). Let \( T \) be an instantiation of type \( t \) such that \( \#T \geq B + 1 \). For all \( i \) in \( T \), let \((\text{cst}_i, \Gamma_i, T)\) be a state of \( N_i(T) \) and suppose that

\[
\begin{cases}
i \in \{B..\#T - 1\} \rightarrow \text{cst}(A(i, T)) (\text{cst}_i, \Gamma_i, T) \\
v \in \{B..\#T - 1\} \rightarrow \text{cst}(A(i, T)) (\text{cst}_i', \Gamma_i', T)
\end{cases}
\]

for some control states \( \text{cst}'_i \) and environments \( \Gamma'_i \). Let \( \text{count} \) and \( \text{count}' \) be states of \( \zeta^T (N_B(\{0..B+1\})) \) such that for every state \( st \) we have that

\[
\text{count}[st] = \# \{ i \in \{B..\#T - 1\} \mid (\text{cst}_i, \Gamma_i, T) \approx st \}
\]

and

\[
\text{count}'[st] = \# \{ i \in \{B..\#T - 1\} \mid (\text{cst}'_i, \Gamma'_i, T) \approx st \}.
\]

Then

\[
\text{count}[R] \xrightarrow{\delta(a)} \text{count}'[R],
\]

where \( R \) 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 \( \alpha \)-event of the form \( c.f_a(me, \text{payloadIDs}) \) (i.e. a private event of \( N_{me}(T) \)) for some identity \( me \) in \( \{B..\#T - 1\} \) and some list of identities \( \text{payloadIDs} \),

(ii) a \( \beta \)-event of the form \( c.f_{\beta}(me, other, \text{payloadIDs}) \) (i.e. an event shared by \( N_{me}(T) \) and \( N_{other}(T) \)) for some identities \( me \) in \( \{B..\#T - 1\} \) and \( other \) in \( \{0..B - 1\} \) and some list of identities \( \text{payloadIDs} \), or

(iii) the internal \( \tau \) event.

Then

\[
(\text{cst}_{me}, \Gamma_{me}, T) \xrightarrow{a} (\text{cst}'_{me}, \Gamma'_{me}, T)
\]

and

\[
\forall i \in \{B..\#T - 1\} \setminus \{me\} \rightarrow (\text{cst}'_i, \Gamma'_i, T) = (\text{cst}_i, \Gamma_i, T),
\]

where \( me \) is as in (i) or (ii) if \( a \) is an \( \alpha \)- or \( \beta \)-event, respectively, and is some identity in \( \{B..\#T - 1\} \) if \( a = \tau \). Hence, using the definitions of \( \text{count} \) and \( \text{count}' \),

\[
\text{count}' = \text{count}[(\text{cst}_{me}, \Gamma_{me}, T)++, (\text{cst}'_{me}, \Gamma'_{me}, T)++]
\]

160
Therefore,

\[\text{count}' = \text{count}[(\text{cst}_{me}, \phi(\Gamma_{me}), \{0 \ldots B + 1\})] \rightarrow_{\sim} (\text{cst}_{me}', \phi(\Gamma_{me}'), \{0 \ldots B + 1\})].\] (6.22)

In addition, using Lemma 6.3.1 we can infer from (6.20) that

\[(\text{cst}_{me}, \phi(\Gamma_{me}), \{0 \ldots B + 1\}) \rightarrow_{B} (\text{cst}_{me}', \phi(\Gamma_{me}'), \{0 \ldots B + 1\}).\]

Further, since \(\text{count}[(\text{cst}_{me}, \phi(\Gamma_{me}), T)] \geq 1\), \(\text{count}[(\text{cst}_{me}, \phi(\Gamma_{me}), \{0 \ldots B + 1\})] \geq 1\). Hence, we can infer that

\[\text{count} \rightarrow_{\sim} \text{count}'\]

using the definition of \( \rightarrow_{\sim} \) (6.5) if \(a\) is an \(\alpha\)- or \(\beta\)-event or (6.4) if \(a = \tau\). Hence, and thanks to the fact that \(R \circ \phi = \phi\), we have that

\[\text{count}[[R]] \rightarrow_{\sim} \text{count}'[[R]],\]

as required.

**Case 2.** Suppose that \(a\) is a \(\beta\)-event of the form \(c.f_{\beta}(x, y, \text{payloadIDs})\) (i.e. an event shared by \(N_{x}(T)\) and \(N_{y}(T)\)) for some distinct identities \(x\) and \(y\) in \(\{B \ldots \#T - 1\}\) and some list of identities \(\text{payloadIDs}\).

Then

\[(\text{cst}_{x}, \Gamma_{x}, T) \rightarrow_{\sim} (\text{cst}_{x}', \Gamma_{x}', T),\] (6.23)

\[(\text{cst}_{y}, \Gamma_{y}, T) \rightarrow_{\sim} (\text{cst}_{y}', \Gamma_{y}', T),\] (6.24)

and

\[\forall i \in \{B \ldots \#T - 1\} \setminus \{x, y\} \bullet (\text{cst}_{i}', \Gamma_{i}', T) = (\text{cst}_{i}, \Gamma, T).\] (6.25)

Hence, using the definitions of \(\text{count}\) and \(\text{count}'\),

\[\text{count}' = \text{count}[(\text{cst}_{x}, \Gamma_{x}, T), \ldots, (\text{cst}_{y}, \Gamma_{y}, T) \rightarrow_{\sim}, (\text{cst}_{x}', \Gamma_{x}', T) \rightarrow_{\sim}, (\text{cst}_{y}', \Gamma_{y}', T) \rightarrow_{\sim}]\] (6.26)

Let \(\psi\) 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 \(\{B \ldots \#T - 1\}\), implying \(\#T \geq B + 2\)). We have that \(\phi(T) = \phi([0 \ldots B + 1])\) and, by Remark 6.3.3,
\(\phi(\psi(\Gamma)) = \phi(\Gamma)\) for every environment \(\Gamma\). Hence, by the definition of the \(\approx\) equivalence relation (Definition 6.1.11),

\[
[(\text{cst}_x, \Gamma_x, T)] = [(\text{cst}_x, \psi(\Gamma_x), \{0 \ldots B + 1\})]
\]

\[
[(\text{cst}'_x, \Gamma'_x, T)] = [(\text{cst}'_x, \psi(\Gamma'_x), \{0 \ldots B + 1\})]
\]

and

\[
[(\text{cst}_y, \Gamma_y, T)] = [(\text{cst}_y, \psi(\Gamma_y), \{0 \ldots B + 1\})]
\]

\[
[(\text{cst}'_y, \Gamma'_y, T)] = [(\text{cst}'_y, \psi(\Gamma'_y), \{0 \ldots B + 1\})].
\]

Therefore, (6.26) implies that

\[
\text{count}' = \text{count}[(\text{cst}_x, \psi(\Gamma_x), \{0 \ldots B + 1\})-\rightarrow, (\text{cst}_y, \psi(\Gamma_y), \{0 \ldots B + 1\})-\rightarrow, (\text{cst}'_x, \psi(\Gamma'_x), \{0 \ldots B + 1\})+\rightarrow, (\text{cst}'_y, \psi(\Gamma'_y), \{0 \ldots B + 1\})+\rightarrow]
\]

In addition, using Lemma 6.3.1 we can infer from (6.23) and (6.24) that

\[
(\text{cst}_x, \psi(\Gamma_x), \{0 \ldots B + 1\}) \xrightarrow{\psi(\alpha)} (\text{cst}'_x, \psi(\Gamma'_x), \{0 \ldots B + 1\})
\]

and

\[
(\text{cst}_y, \psi(\Gamma_y), \{0 \ldots B + 1\}) \xrightarrow{\psi(\alpha)} (\text{cst}'_y, \psi(\Gamma'_y), \{0 \ldots B + 1\}).
\]

Let \(p = 2\) if \((\text{cst}_x, \psi(\Gamma_x), T) \approx (\text{cst}_y, \psi(\Gamma_y), \{0 \ldots B + 1\})\) and \(p = 1\) otherwise; then, since \(\text{count}[(\text{cst}_x, \Gamma_x, T)] \geq p\) and \(\text{count}[(\text{cst}_y, \Gamma_y, T)] \geq p\),

\[
\text{count}[(\text{cst}_x, \psi(\Gamma_x), \{0 \ldots B + 1\})] \geq p
\]

\[
\text{count}[(\text{cst}_y, \psi(\Gamma_y), \{0 \ldots B + 1\})] \geq p.
\]

Hence, from (6.27) and the definition of \(\rightarrow_\infty\) (6.6),

\[
\text{count} \xrightarrow{\psi(\alpha)_\infty} \text{count}'.
\]

Hence, and since \(R \circ \psi = \phi\) (as \(\psi\) collapses \(T\) to \(\{0 \ldots B + 1\}\) and \(R\) renames \(B + 1\) to \(B\)),

\[
\text{count}[R] \xrightarrow{\phi(\alpha)_\infty} \text{count}'[R],
\]

as required.

Note that this is the case where it was necessary to counter abstract the nodes with identities in \(\{B \ldots \#T - 1\}\) 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 \(B\).

\textbf{Case 3.} Suppose that \(a\) is a \(\gamma\)-event of the form \(c.f.(\text{payloadIDs})\) (i.e. an event shared by all nodes) for some list of identities \(\text{payloadIDs}\).

Then

\[
\forall i \in \{B \ldots \#T - 1\} \bullet (\text{cst}_i, \Gamma_i, T) \xrightarrow{\alpha_i} (\text{cst}'_i, \Gamma'_i, T).
\]
Hence, using Lemma 6.3.1 we can infer from definition of $\phi$ that

$$\forall i \in \{B \ldots \#T - 1\} \bullet (\text{cst}_B, \phi(\Gamma_B), \{0 \ldots B+1\}) \xrightarrow{\phi(a)} \{\text{cst}_B', \phi(\Gamma_B'), \{0 \ldots B+1\}\}.$$  \hfill (6.28)

The definition of the $\approx$ equivalence relation (Definition 6.1.11) implies that

$$\forall i \in \{B \ldots \#T - 1\} \bullet [(\text{cst}_i, \Gamma_i, T)] = [(\text{cst}_i, \phi(\Gamma_i), \{0 \ldots B + 1\})]$$  \hfill (6.29)

and

$$\forall i \in \{B \ldots \#T - 1\} \bullet [(\text{cst}_i', \Gamma_i', T)] = [(\text{cst}_i', \phi(\Gamma_i'), \{0 \ldots B + 1\})].$$  \hfill (6.30)

From (6.29) and by the definition of $\text{count}$ we have that

$$\forall i \in \{1 \ldots k\} \bullet \text{count}_{\text{st}_i} = \#\{j \in \{B \ldots \#T - 1\} \mid (\text{cst}_j, \Gamma_j, T) \approx \text{st}_i\}$$

$$= \#\{j \in \{B \ldots \#T - 1\} \mid (\text{cst}_j, \phi(\Gamma_j), \{0 \ldots B + 1\}) \approx \text{st}_i\}.$$  \hfill (6.31)

Hence, for every $i$ in $\{1 \ldots k\}$ there are $\text{count}_{\text{st}_i}$ many states

$$\text{src}_i^1 = (\text{cst}_j^1, \phi(\Gamma_j^1), \{0 \ldots B + 1\}),$$

$$\text{src}_i^2 = (\text{cst}_j^2, \phi(\Gamma_j^2), \{0 \ldots B + 1\}),$$

$$\vdots$$

$$\text{src}_{\text{count}_{\text{st}_i}} = (\text{cst}_j^{\text{count}_{\text{st}_i}}, \phi(\Gamma_j^{\text{count}_{\text{st}_i}}), \{0 \ldots B + 1\}),$$

all related to $\text{st}_i$ by $\approx$, where for all $i$ in $\{1 \ldots k\}$ and for all $l$ in $\{1 \ldots \text{count}_{\text{st}_i}\}$, the indices $j_i^l$ are in $\{B \ldots \#T - 1\}$ and are all distinct. From (6.28) we have that

$$\forall i \in \{1 \ldots k\} \bullet \forall l \in \{1 \ldots \text{count}_{\text{st}_i}\} \bullet \text{src}_i^l \xrightarrow{\phi(a)} \text{tgt}_i^l,$$  \hfill (6.32)

where for all $i$ in $\{1 \ldots k\},$

$$\text{tgt}_i^1 = (\text{cst}_j^1, \phi(\Gamma_j^1), \{0 \ldots B + 1\}),$$

$$\text{tgt}_i^2 = (\text{cst}_j^2, \phi(\Gamma_j^2), \{0 \ldots B + 1\}),$$

$$\vdots$$

$$\text{tgt}_{\text{count}_{\text{st}_i}} = (\text{cst}_j^{\text{count}_{\text{st}_i}}, \phi(\Gamma_j^{\text{count}_{\text{st}_i}}), \{0 \ldots B + 1\}),$$

with each $j_i^l$ as above. Further,

$$\{ (\text{cst}_j^l, \phi(\Gamma_j^l), \{0 \ldots B + 1\}) \mid l \in \{1 \ldots \text{count}_{\text{st}_i}\}\}$$

$$= \{ \text{tgt}_i^l \mid i \in \{1 \ldots k\}, l \in \{1 \ldots \text{count}_{\text{st}_i}\}\}$$

and

$$\#\{B \ldots \#T - 1\} = \#\{(i, l) \mid i \in \{1 \ldots k\} \land l \in \{1 \ldots \text{count}_{\text{st}_i}\}\}.$$  \hfill (6.33)

Hence, from (6.30), we have that

$$\forall \text{st} \in S \bullet \text{count}'[\text{st}] = \#{j \in \{B \ldots \#T - 1\} \mid (\text{cst}_j', \Gamma_j', \text{T}) \approx \text{st}}$$

$$= \#{j \in \{B \ldots \#T - 1\} \mid (\text{cst}_j', \phi(\Gamma_j'), \{0 \ldots B + 1\}) \approx \text{st}}$$

$$= \#{(i, l) \mid i \in \{1 \ldots k\} \land l \in \{1 \ldots \text{count}_{\text{st}_i}\} \land \text{tgt}_i^l \approx \text{st}.}$$  \hfill (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
\[
\text{count} \phi(a) \xrightarrow{r}\text{count}'.
\]
Hence and since \( R \circ \phi = \phi \),
\[
\text{count}[R] \phi(a) \xrightarrow{r}\text{count}'[R],
\]
as required. This completes our proof. \( \square \)

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.** \( a \in \bigcup_{i=B}^{#T-1} A(i, T) \Leftrightarrow \phi(a) \in A^B_\zeta. \)

**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 \( \beta \)-events for illustration.

So suppose \( a = c.f_\beta(i, \text{other}, \text{payloadIDs}) \in A(i, T) \) is a \( \beta \)-event with \( i \in \{ B .. \# T - 1 \}, \text{other} \in T \backslash \{ i \} \) and \( \text{payloadIDs} \in T^n. \) (The case of \( c.f_\beta(\text{other}, i, \text{payloadIDs}) \) is very similar.) If \( \text{other} \in \{ 0 .. B - 1 \}, \) then
\[
\phi(a) = c.f_\beta(B, \phi(\text{other}), \phi(\text{payloadIDs})) \in A(B, \{ 0 .. B \}) \subseteq A^B_\zeta,
\]
using (6.2). Alternatively, if \( \text{other} \in \{ B .. \# T - 1 \} \backslash \{ i \} \) then
\[
\phi(a) = c.f_\beta(\phi(i), \phi(\text{other}), \phi(\text{payloadIDs})) = c.f_\beta(B, B, \phi(\text{payloadIDs})).
\]
Now, \( c.f_\beta(B, B + 1, \phi(\text{payloadIDs})) \in A(B, \{ 0 .. B + 1 \}), \) using (6.2). Hence
\[
\phi(a) = R(c.f_\beta(B, B + 1, \phi(\text{payloadIDs}))) \in R(A(B, \{ 0 .. B + 1 \})) \subseteq A^B_\zeta.
\]

For the right-to-left direction, suppose \( \phi(a) \in A^B_\zeta = R(A(B, \{ 0 .. B + 1 \}) \cup A(B + 1, \{ 0 .. B + 1 \})). \) Then suppose \( \phi(a) = R(a') \) 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 \( \beta \)-events for illustration.

So suppose \( a' = c.f_\beta(B, \text{other}, \text{payloadIDs}) \) is a \( \beta \)-event, with \( \text{other} \in \{ 0 .. B + 1 \} \backslash \{ B \} \) and \( \text{payloadIDs} \in \{ 0 .. B + 1 \}^n. \) (The case of \( c.f_\beta(\text{other}, B, \text{payloadIDs}) \) is very similar.) Then
\[
\phi(a) = R(a') = c.f_\beta(B, R(\text{other}), R(\text{payloadIDs})).
\]

Now, \( a \) and \( a' \) agree on non-\( T \) fields, so they must be generated using the same function \( f_\beta \) (see clause (iv) of Remark 6.1.9). So \( a = c.f_\beta(i, j, \text{payloadIDs}') \) for some \( i \in T, j \in T \backslash \{ i \} \) and \( \text{payloadIDs}' \in T^n, \) where \( \phi(i) = B, \phi(j) = R(\text{other}) \) and \( \phi(\text{payloadIDs}') = R(\text{payloadIDs}). \) Hence \( i \geq B, \) and \( a \in A(i, T) \subseteq \bigcup_{k=B}^{#T-1} A(k, T), \) using (6.2). \( \square \)
Proposition 6.3.8. Suppose that $N_{myId}(t)$ satisfies NoEqT'. Let $T$ be an instantiation of type $t$ such that $\#T \geq B + 1$. For all $i$ in $T$, let $(cst_i, \Gamma_i, T)$ be a state of $N_i(T)$ and suppose that

$$\parallel i \in T \bullet [A(i, T)] (cst_i, \Gamma_i, T) \parallel$$

$$\xrightarrow{a} \parallel i \in T \bullet [A(i, T)] (cst'_i, \Gamma'_i, T) \parallel$$

for some control states $cst'_i$ and environments $\Gamma'_i$. Let $\text{count}$ and $\text{count}'$ be states of $\Theta_T^\infty (N_B((0..B + 1)))$ such that for every state $st$ we have that

$$\text{count}[st] = \#\{i \in \{B..#T - 1\} \mid (cst_i, \Gamma_i, T) \approx st\}$$

and

$$\text{count}'[st] = \#\{i \in \{B..#T - 1\} \mid (cst'_i, \Gamma'_i, T) \approx st\}.$$ 

Then

$$\phi(a) \parallel i \in \{0..B - 1\} \bullet [A(i, \{0..B\})] (cst_i, \phi(\Gamma_i), \{0..B\}) \parallel \xrightarrow{\text{count}[\|R\|]} \parallel i \in \{0..B - 1\} \bullet [A(i, \{0..B\})] (cst'_i, \phi(\Gamma'_i), \{0..B\}) \parallel$$

where $R$ 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

$$\parallel i \in \{0..B - 1\} \bullet [A(i, T)] (cst_i, \Gamma_i, T) \parallel$$

$$\xrightarrow{a} \parallel i \in \{0..B - 1\} \bullet [A(i, T)] (cst'_i, \Gamma'_i, T),$$

and

$$\forall i \in \{B..#T - 1\} \bullet (cst_i, \Gamma_i, T) = (cst'_i, \Gamma'_i, T). \ (6.35)$$

Then, Lemma 6.3.5 implies that

$$\phi(a) \parallel i \in \{0..B - 1\} \bullet [A(i, \{0..B\})] (cst_i, \phi(\Gamma_i), \{0..B\}) \parallel$$

$$\xrightarrow{\text{count}[\|R\|]} \parallel i \in \{0..B - 1\} \bullet [A(i, \{0..B\})] (cst'_i, \phi(\Gamma'_i), \{0..B\}). \ (6.36)$$
In addition, \( \text{count}' = \text{count} \) by (6.35), so
\[
\text{count}'[\mathcal{R}] = \text{count}[\mathcal{R}].
\] (6.37)

Finally, if \( a \) is a visible event, then
\[
\phi(a) \in \left( \bigcup_{i=0}^{B-1} A(i, \{0 \ldots B\}) \right) \setminus A^B_{\xi}.
\]
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 \( \{B \ldots \# T - 1\} \):
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
\[
\| i \in \{B \ldots \# T - 1\} \bullet [A(i, T)] \ (cst_i, \Gamma_i, T)
\]
\[
\xrightarrow{a} \| i \in \{B \ldots \# T - 1\} \bullet [A(i, T)] \ (cst'_i, \Gamma'_i, T),
\]
and
\[
\forall i \in \{0 \ldots B - 1\} \bullet (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) = (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}).
\] (6.39)

Then, Lemma 6.3.6 implies that
\[
\text{count}[\mathcal{R}] \xrightarrow{\phi(a)} \text{count}'[\mathcal{R}].
\] (6.38)

Further,
\[
\forall i \in \{0 \ldots B - 1\} \bullet (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) = (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}).
\]

Finally, if \( a \) is a visible event, then
\[
\phi(a) \in A^B_{\xi} \setminus \left( \bigcup_{i=0}^{B-1} A(i, \{0 \ldots B\}) \right).
\]
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 \( \bigcup_{i=\# T-1}^{\# T-1} A(i, T) \). Then,
\[
\| i \in \{0 \ldots B - 1\} \bullet [A(i, T)] \ (cst_i, \Gamma_i, T)
\]
\[
\xrightarrow{a} \| i \in \{0 \ldots B - 1\} \bullet [A(i, T)] \ (cst'_i, \Gamma'_i, T)
\]
and
\[
\| i \in \{B \ldots \# T - 1\} \bullet [A(i, T)] \ (cst_i, \Gamma_i, T)
\]
\[
\xrightarrow{a} \| i \in \{B \ldots \# T - 1\} \bullet [A(i, T)] \ (cst'_i, \Gamma'_i, T).
\]
Then, Lemma 6.3.5 implies that
\[ \left\| \{ i \in \{0 \ldots B - 1\} \cdot [A(i, \{0 \ldots B\})] (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) \right\|_{\phi(a)} \]
\[ \xrightarrow{\phi(a)} \left\| \{ i \in \{0 \ldots B - 1\} \cdot [A(i, \{0 \ldots B\})] (cst'_i, \phi(\Gamma'_i), \{0 \ldots B\}) \right\| \]
and Lemma 6.3.6 implies that
\[ \text{count}[[R]] \xrightarrow{\phi(a)} \text{count}'[[R]]. \]
Finally,
\[ \phi(a) \in \left( \bigcup_{i=0}^{B-1} A(i, \{0 \ldots B\}) \right) \cap A^B. \]
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
\[ P \xrightarrow{s} Q \]
to mean that there exist processes $P_i$ for $i$ in $\{0 \ldots n\}$ such that $P_0 = P$, $P_n = Q$ and
\[ \forall i \in \{0 \ldots n\} \cdot P_i \xrightarrow{a_{i+1}} P_{i+1}. \]
In addition, $P \xrightarrow{s}$ if there exists some process $Q$ such that $P \xrightarrow{s} Q$.

We now present a result that says that, given nodes that contain no equality tests, if $\phi(\text{Nodes}(T))$ can perform a sequence of events $s$ and reach some state $P$, then $\text{ABS}_{T,B,0}^\infty$ can also perform $s$ and reach a state corresponding to $P$.

**Lemma 6.3.9.** Suppose that $\mathcal{N}_{\text{myId}}(t)$ satisfies $\text{NoEqT}^t$. Let $T$ be an instantiation of type $t$ such that $\# T \geq B + 1$. Then if
\[ \phi(\text{Nodes}(T)) \xrightarrow{s} \phi \left( \left\| i \in T \cdot [A(i, T)] (cst_i, \Gamma_i, T) \right\| \right) \]
for some control states $cst_i$ and environments $\Gamma_i$, then
\[ \text{ABS}_{T,B,0}^\infty \xrightarrow{s} \left( \left\| i \in \{0 \ldots B - 1\} \cdot [A(i, \{0 \ldots B\})] (cst_i, \phi(\Gamma_i), \{0 \ldots B\}) \right\|_{\text{count}[[R]]} \right), \]
where $\text{count}$ is a state of $\zeta_{\infty}^T(\mathcal{N}_B(\{0 \ldots B + 1\}))$ such that for all states $st$,
\[ \text{count}[st] = \# \{ i \in \{B \ldots \# T - 1\} \mid (cst_i, \Gamma_i, T) \approx st \} \]
and $R$ is the renaming function that replaces every occurrence of $B + 1$ by $B$. 

167
Proof: We prove the result by an induction on the length of $s$.

**Base Case.** Suppose that $s = \emptyset$.

Then $(\text{cst}_i, \Gamma_i, T) = \mathcal{N}_i(T)$ and $\Gamma_i = \{\text{myId} \mapsto i\}$ for all $i$ in $T$. Therefore, by Remark 6.1.4,

$$\forall i \in \{0 \ldots B - 1\} \bullet (\text{cst}_i, \phi(\Gamma_i), \{0 \ldots B\}) = \mathcal{N}_{\phi(i)}(\{0 \ldots B\}).$$

However, for all $i$ in $\{0 \ldots B - 1\}$, $\phi(i) = i$. Hence,

$$\forall i \in \{0 \ldots B - 1\} \bullet (\text{cst}_i, \phi(\Gamma_i), \{0 \ldots B\}) = \mathcal{N}_i(\{0 \ldots B\}).$$

Also, by the definition of $\text{count}$,

$$\text{count} = (\#T - B, 0, \ldots, 0).$$

Recall that

$$\text{ABST}_{\infty}^{T, B, 0} = \left( \left\lfloor i \in \{0 \ldots B - 1\} \bullet [\text{A}(i, \{0 \ldots B\})] \mathcal{N}_i(\{0 \ldots B\}) \right\rfloor \cup_{i=0}^{B-1} \mathcal{A}_{\xi}^B(\#T - B, 0, \ldots, 0) \mathcal{R} \right).$$

Hence,

$$\text{ABST}_{\infty}^{T, B, 0} \xrightarrow{\emptyset} \left( \left\lfloor i \in \{0 \ldots B - 1\} \bullet [\text{A}(i, \{0 \ldots B\})] (\text{cst}_i, \phi(\Gamma_i), \{0 \ldots B\}) \right\rfloor \cup_{i=0}^{B-1} \mathcal{A}_{\xi}^B(\#T - B, 0, \ldots, 0) \mathcal{R} \right),$$

establishing the result in the base case.

**Inductive Case.** Suppose the result holds for sequence $s$ and let $\text{sa} = s^\ast(a)$ for some event $a$. Suppose that

$$\phi(\text{Nodes}(T)) \xrightarrow{\text{sa}} \phi(\left\lfloor i \in T \bullet [\text{A}(i, T)] (\text{cst}_t, \Gamma_t, T) \right\rfloor)$$

for some control states $\text{cst}_t$ and environments $\Gamma_t$. Then, for every $i$ in $T$ there exists a control state $\text{cst}_t$ and an environment $\Gamma_t$, such that

$$\phi(\text{Nodes}(T)) \xrightarrow{s} \phi(\left\lfloor i \in T \bullet [\text{A}(i, T)] (\text{cst}_t, \Gamma_t, T) \right\rfloor)$$

(6.42)

Then, by the inductive hypothesis,

$$\text{ABST}_{\infty}^{T, B, 0} \xrightarrow{s} \left( \left\lfloor i \in \{0 \ldots B - 1\} \bullet [\text{A}(i, \{0 \ldots B\})] (\text{cst}_i, \phi(\Gamma_i), \{0 \ldots B\}) \right\rfloor \cup_{i=0}^{B-1} \mathcal{A}_{\xi}^B(\#T - B, 0, \ldots, 0) \mathcal{R} \right),$$

(6.43)
where count is a state of $\zeta_\infty^T(N_B(\{0 \ldots B + 1\}))$ such that for all states $st$,

$$\text{count}[st] = \# \{i \in \{B \ldots \# T - 1\} \mid (\text{cst}_i, \Gamma_i, T) \approx st\}.$$ 

From (6.42) we can infer that there exists an event $a'$ such that $\phi(a') = a$ and

$$\| i \in T \bullet [A(i, T)] (\text{cst}_i, \Gamma_i, T) \xrightarrow{a'} \| i \in T \bullet [A(i, T)] (\text{cst}'_i, \Gamma'_i, T).$$

Hence, by Proposition 6.3.8 we have that

$$\frac{\phi(a')}{\text{count}'[st]} \xrightarrow{\text{count}'[\text{st}]} \left( \begin{array}{c}
\| i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] (\text{cst}_i, \phi(\Gamma_i), \{0 \ldots B\}) \\
\cup_{i=0}^{B-1} A(i,\{0..B\})]\big\|_{\text{count}'[\text{st}]}
\end{array} \right),$$

where count' is a state of $\zeta_\infty^T(N_B(\{0 \ldots B + 1\}))$ such that for all states $st$,

$$\text{count}'[st] = \# \{i \in \{B \ldots \# T - 1\} \mid (\text{cst}'_i, \Gamma'_i, T) \approx st\}.$$ 

By combining the above with (6.43) and by recalling that $\phi(a') = a$, we get that

$$ABS_{\infty, B, 0} \xrightarrow{sa} \left( \begin{array}{c}
\| i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] (\text{cst}'_i, \phi(\Gamma'_i), \{0 \ldots B\}) \\
\cup_{i=0}^{B-1} A(i,\{0..B\})]\big\|_{\text{count}'[\text{st}]}
\end{array} \right),$$

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 $N_{\text{myId}}(t)$ satisfies $\text{NoEqT}'$, then

$$ABS_{\infty, B, 0} \equiv_T \phi(\text{Nodes}(T)).$$

**Proof:** Let $tr$ be a trace of $\phi(\text{Nodes}(T))$ and let $s$ be a sequence of events such that $s \setminus \{\tau\} = tr$ and $\phi(\text{Nodes}(T)) \xrightarrow{s} \tau$. Then, Lemma 6.3.9 implies that $ABS_{\infty, B, 0} \xrightarrow{s} \tau$, which means that $tr$ is a trace of $ABS_{\infty, B, 0}$. 

### 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 $W_N$ (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 \textbf{NoEqT}, 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 \textbf{NoEqT}, 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 \( W_N \geq 1 \).

Example 6.4.1. Let

\[ N_{myId}(t) = \text{in}?x:t?y:t \rightarrow N''_{myId}(t) \]

\[ N''_{myId}(t) = \begin{cases} \text{if } x = myId \lor y = myId \lor x = y \text{ then } c_1.myId.x.y \rightarrow \text{STOP} \\ \text{else } c_2.myId.x.y \rightarrow \text{STOP}. \end{cases} \]

Let \( B = 0 \), so \( \{0..B+1\} = \{0, 1\} \). In the nodes \( N_B(\{0..B+1\}) \) and \( N_{B+1}(\{0..B+1\}) \), 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 \( \zeta_T(N_B(\{0..B+1\})) \) is built using nodes \( N_B(\{0..B+1\}) \) and \( N_{B+1}(\{0..B+1\}) \), this counter state machine also cannot perform any communication over channel \( c_2 \), and hence, no communication on channel \( c_2 \) is available within \( ABS_{\infty}^{0,1,2,0,0} \). 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 \( \phi(Nodes(\{0, 1, 2\})) \), but that is contrary to our requirements.

\textit{End of example.}

The root of the problem exhibited by Example 6.4.1 is that in Lemma 6.3.1 \( \psi \) 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 \( \psi \). Hence, there may be some transitions available after an equality test that evaluates to \( False \) (respectively, \( True \)) in \( N_i(T) \) for some \( i \in \{B..#T-1\} \), but that evaluates to \( True \) (respectively, \( False \)) in \( N_i(\{0..B+1\}) \) for \( i \in \{B, B+1\} \).

Recall that \( W_N \) (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 \( m = W_N \) ensures that all transitions of \( \phi(Nodes(T)) \) are available in \( ABS_{\infty}^{0,1,2,0,0} \).

Remark 6.4.2. Throughout this section we assume that all \( \beta \)- and \( \gamma \)-events contain no payloads with node identities, i.e. every \( \beta \)-event is of the form \( c.f_\beta(me, other) \) for some distinct identities \( me \) and \( other \) and every \( \gamma \)-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 $e = c.v_1 \ldots v_n$ be a visible event and let $\epsilon = c_{S_1}x_1:X_1 \ldots c_{S_n}x_n:X_n$ be a (not necessarily unique) construct it matches after resolving all nondeterministic selections (i.e. $S_i \in \{?,!\}$ for all $i \in \{1 \ldots n\}$). Then

\[
\text{inputs}^i(e, \epsilon) = \{ v_i \mid i \in \{1 \ldots n\} \land S_i = ? \land X_i = t \},
\]

\[
\text{outputs}^i(e, \epsilon) = \{ v_i \mid i \in \{1 \ldots n\} \land S_i = ! \land X_i = t \},
\]

and, given an environment $\Gamma$,

\[
\text{remembered}(v_i, e, \epsilon, \Gamma) = x_i \in \text{dom}(\Gamma)
\]

for all $i$ in $\{1 \ldots n\}$.

We make the notational convention that $\tau$ matches every construct and

\[
\text{inputs}^i(\tau) = \text{outputs}^i(\tau) = \{ \}.
\]

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 $\text{inputs}^i(e)$, $\text{outputs}^i(e)$ and $\text{remembered}(v_i, e, \Gamma)$.

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 $(\text{cst}, \Gamma, T) \xrightarrow{a} (\text{cst}', \Gamma', T)$ for some identity $me \in T$. Then, for all injective partial functions $\psi, \psi' : T \rightarrow \hat{T}$ such that

- $\text{dom}(\psi) = \text{Im}(\Gamma)$,
- $\text{dom}(\psi') = \text{Im}(\Gamma')$, and
- $\psi'(v) = \psi(v)$ for $v \in \text{Im}(\Gamma) \cap \text{Im}(\Gamma')$

we have that $(\text{cst}, \psi(\Gamma), \hat{T}) \xrightarrow{a} (\psi'(\Gamma), \hat{T})$ for every event $\hat{a}$ that is like $a$, except that

- every value $v$ in $\text{outputs}^i(a)$ is replaced by $\psi(v)$,
- every value $v$ in $\text{inputs}^i(a)$ such that $\text{remembered}(v, a, \Gamma')$ is replaced by $\psi'(v)$, and
- every value $v$ in $\text{inputs}^i(a)$ such that $\neg\text{remembered}(v, a, \Gamma')$ 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 \( \mathcal{N}_{ψ(me)}(\hat{T}) \), since the choice of an instantiation of type \( t \) cannot influence the control flow of \( \mathcal{N}_{myId}(t) \), by data independence.

Suppose \((cst, Γ, T) \xrightarrow{a} me(cst', Γ', T)\). 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 \( \text{dom}(Γ) \).

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, \((cst, ψ(Γ), \hat{T})\) 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 \( ψ(Γ) \), i.e. \( ψ(v) \),
- every deterministic input value \( v \) of type \( t \) in \( a \) that is stored by the node for future use is replaced by \( ψ'(v) \), 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 \( \text{Im}(Γ') \) to \( \hat{T} \) (which means that any equality or inequality tests evaluates to \( \text{True} \) under \( Γ' \) if and only if it evaluates to \( \text{True} \) under \( ψ'(Γ') \)), i.e. no conditional choice immediately after \( \hat{a} \) can influence the reachability of \((cst', ψ'(Γ'), \hat{T})\), and
- \( ψ'(v) = ψ(v) \) for \( v \in \text{Im}(Γ) \cap \text{Im}(Γ') \) (i.e. the remembered values remain unmodified),

so after performing \( \hat{a} \), \((cst, ψ(Γ), \hat{T})\) behaves like \((cst', ψ'(Γ'), \hat{T})\), as required.

The following corollary applies Lemma 6.4.4 to instantiations \( \hat{T} = \{0..B + W_N \} \) of type \( t \) and functions \( ψ \) and \( ψ' \) that are \((B + W_N)\)-precollapsing.

**Corollary 6.4.5.** Let \( T \) be an instantiation of \( t \) and let \( \hat{T} = \{0..B + W_N \} \). Then, for every injective partial \((B + W_N)\)-precollapsing function \( ψ : T \rightarrow \hat{T} \) such that \( \text{dom}(ψ) = \text{Im}(Γ) \) we have that if \((cst, Γ, T) \xrightarrow{a} me(cst', Γ', T)\) for some \( me \in T \), then there exists an injective partial \((B + W_N)\)-precollapsing function \( ψ' : T \rightarrow \hat{T} \) such that

- \( \text{dom} ψ' = \text{Im}(Γ') \),
- \( ψ'(v) = ψ(v) \) for \( v \in \text{Im}(Γ) \cap \text{Im}(Γ') \),

and \((cst, ψ(Γ), \hat{T}) \xrightarrow{a}_{ψ(me)} (cst', ψ'(Γ'), \hat{T})\) for every event \( \hat{a} \) that is like \( a \), except that

172
• every value \( v \) in outputs\( ^t(a) \) is replaced by \( \psi(v) \),

• every value \( v \) in inputs\( ^t(a) \) such that \( \text{remembered}(v, a, \Gamma') \) is replaced by \( \psi'(v) \), and

• every value \( v \) in inputs\( ^t(a) \cap \{ B \ldots \# T - 1 \} \) such that \( \neg \text{remembered}(v, a, \Gamma') \) is replaced by an arbitrary value in \( \{ B \ldots B + W_N \} \).

**Proof:** Let \( \psi \) be defined as in the statement of the corollary. We show that there exists an injective partial \((B + W_N)\)-precollapsing function \( \psi' : T \to \hat{T} \) such that

\[
\begin{align*}
\text{dom}(\psi') &= \text{Im}(\Gamma'), \\
\psi'(v) &= \psi(v) \quad \text{for} \quad v \in \text{Im}(\Gamma) \cap \text{Im}(\Gamma').
\end{align*}
\]

We construct \( \psi' \) in three stages. Firstly, let \( \psi'_1 : \{ 0 \ldots B - 1 \} \to \{ 0 \ldots B - 1 \} \) be such that

\[
\begin{align*}
\text{dom}(\psi'_1) &= \text{Im}(\Gamma') \cap \{ 0 \ldots B - 1 \}, \quad \text{(6.44)} \\
\psi'_1(v) &= v \quad \text{for} \quad v \in \{ 0 \ldots B - 1 \} \cap \text{dom}(\psi'_1), \quad \text{(6.45)}
\end{align*}
\]

i.e. \( \psi'_1 \) is the identity function on all the values in \( \text{Im}(\Gamma') \) that are less than \( B \). This means that \( \psi'_1 \) is injective. Observe that since \( \psi(v) = v \) for \( v \in \{ 0 \ldots B - 1 \} \cap \text{Im}(\Gamma) \), (6.44) and (6.45) imply that \( \psi'_1(v) = \psi(v) \) for \( v \in \text{Im}(\Gamma) \cap \text{Im}(\Gamma') \cap \{ 0 \ldots B - 1 \} \).

Secondly, let \( \psi'_2 \) be such that \( \psi'_2(\text{Im}(\Gamma) \cap \text{Im}(\Gamma') \cap \{ B \ldots \# T - 1 \} \ldots \psi \). This means that \( \psi'_2 \) is injective, since \( \psi \) is. Now, by the definition of a \((B + W_N)\)-precollapsing function (Definition 6.3.2),

\[
\forall v \in \text{Im}(\Gamma) \cap \{ B \ldots \# T - 1 \} \quad \bullet \quad \psi(v) \in \{ B \ldots B + W_N \},
\]

so

\[
\forall v \in \text{Im}(\Gamma) \cap \text{Im}(\Gamma') \cap \{ B \ldots \# T - 1 \} \quad \bullet \quad \psi'_2(v) \in \{ B \ldots B + W_N \}.
\]

Finally, we construct a function \( \psi'_3 \) from

\[
X = (\text{Im}(\Gamma') \cap \{ B \ldots \# T - 1 \}) \setminus \text{Im}(\Gamma)
\]

to

\[
Y = \{ B \ldots B + W_N \} \setminus V,
\]

where

\[
V = \text{Im}(\psi'_2).
\]

If \( X = \{ \} \), then \( \psi'_3 \) is the empty function, which is vacuously injective. Otherwise we proceed as follows. Since \( \mathcal{N}_{me}(T) \) can remember at most \( W_N \) node identities other than \( \text{me} \), \# \( \text{Im}(\Gamma') \leq W_N + 1 \). In addition,

\[
X = (\text{Im}(\Gamma') \cap \{ B \ldots \# T - 1 \}) \setminus \text{Im}(\Gamma) \\
= (\text{Im}(\Gamma') \cap \{ B \ldots \# T - 1 \}) \setminus (\text{Im}(\Gamma) \cap \text{Im}(\Gamma') \cap \{ B \ldots \# T - 1 \})
\]

173
and
\[ \# V = \#(\text{Im}(\Gamma) \cap \text{Im}(\Gamma') \cap \{B \ldots \# T - 1\}). \]

Hence
\[
\# X = \#(\text{Im}(\Gamma') \cap \{B \ldots \# T - 1\}) - \#(\text{Im}(\Gamma) \cap \text{Im}(\Gamma') \cap \{B \ldots \# T - 1\}) \\
\leq W_N + 1 - \# V \\
= \# Y.
\]

This means that there is at least one injective function from \( X \) to \( Y \). Let \( \psi_3' \) be such an injective function.

We now let \( \psi' = \psi_1' \cup \psi_2' \cup \psi_3' \). Then \( \psi' : \text{Im}(\Gamma') \to \{B \ldots B + W_N\} \) is well-defined, since \( \text{dom}(\psi_1') \cup \text{dom}(\psi_2') \cup \text{dom}(\psi_3') = \text{Im}(\Gamma') \) and the domains of \( \psi_1', \psi_2' \) and \( \psi_3' \) are pairwise disjoint. In addition, \( \psi' \) is injective, since all of \( \psi_1', \psi_2' \) and \( \psi_3' \) are, and the images of \( \psi_1', \psi_2' \) and \( \psi_3' \) are pairwise disjoint. Finally, it is easy to check that the properties of \( \psi_1', \psi_2' \) and \( \psi_3' \) imply that
\begin{itemize}
  \item \( \psi'(v) = v \) for \( v \in \{0 \ldots B - 1\} \cap \text{dom}(\psi') \),
  \item \( \psi'(v) \in \{B \ldots B + W_N\} \) for \( v \in \{B \ldots \# T - 1\} \cap \text{dom}(\psi') \), and
  \item \( \psi'(v) = \psi(v) \) for \( v \in \text{Im}(\Gamma) \cap \text{Im}(\Gamma') \),
\end{itemize}

which proves the existence of an injective function \( \psi' \) as defined in the conclusion of the corollary.

Observe that the statement

“every value \( v \) in \( \text{inputs}^t(a) \cap \{B \ldots \# T - 1\} \) such that \( \neg \text{remembered}(v, a, \Gamma') \) is replaced by an arbitrary value in \( \{B \ldots B + W_N\} \)”

is more specific than

“every value \( v \) in \( \text{inputs}^t(a) \) such that \( \neg \text{remembered}(v, a, \Gamma') \) is replaced by an arbitrary value in \( \{0 \ldots B + W_N\} \)”.

Hence, the result follows from Lemma 6.4.4.

The following corollary applies Lemma 6.4.4 to instantiations \( \hat{T} = \{0 \ldots B + W_N\} \) of type \( t \) and functions \( \psi \) and \( \psi' \) that are bijective on \( \hat{T} \) and are the identity function on \( \{0 \ldots B - 1\} \).

**Corollary 6.4.6.** Let \( \hat{T} = \{0 \ldots B + W_N\} \). Then, for every bijection \( \pi : \hat{T} \to \hat{T} \) such that \( \pi(v) = v \) for all \( v \) in \( \{0 \ldots B - 1\} \), if \( (\text{cst}, \Gamma, \hat{T}) \xrightarrow{a \text{ me}} (\text{cst'}, \Gamma', \hat{T}) \) for some \( m \in T \), then for every bijection \( \pi' : \hat{T} \to \hat{T} \) such that
\begin{itemize}
  \item \( \pi'(v) = v \) for all \( v \) in \( \{0 \ldots B - 1\} \),
  \item \( \pi'(v) = \pi(v) \) for all \( v \) in \( \text{Im}(\Gamma) \cap \text{Im}(\Gamma') \),
\end{itemize}
we have that $\pi, \pi'(\Gamma), \hat{T} \xrightarrow{a \pi(\Gamma), \pi'} (cst', \pi'((\Gamma'), \hat{T})$ for every event $\hat{a}$ that is like $a$, except that

- every value $v$ in $\text{outputs}^T(a)$ is replaced by $\pi(v)$,
- every value $v$ in $\text{inputs}^T(a)$ such that $\text{remembered}(v, a, \Gamma')$ is replaced by $\pi'(v)$, and
- every value $v$ in $\text{inputs}^T(a) \cap \{B \ldots T-1\}$ such that $\neg\text{remembered}(v, a, \Gamma')$ is replaced by an arbitrary value in $\{B \ldots B + W_N\}$.

**Proof:** Let $\pi$ and $\pi'$ be bijections from $\hat{T}$ to itself as in the statement of the corollary. Then, since $\pi$ and $\pi'$ are bijections that are the identity function on $\{0 \ldots B-1\}$, it must be that for any $v$ in $\{B \ldots B + W_N\}$, $\pi(v)$ and $\pi'(v)$ are in $\{B \ldots B + W_N\}$. Let $\psi = \text{Im}(\Gamma)\cup\pi$ and $\psi' = \text{Im}(\Gamma')\cup\pi'$. Since every bijection is injective, so must be every function obtained from a bijection using domain restriction. Therefore $\psi : \hat{T} \rightarrow \hat{T}$ is injective and such that $\text{dom}(\psi) = \text{Im}(\Gamma)$, $\psi : \hat{T} \rightarrow \hat{T}$ is injective and such that $\text{dom}(\psi') = \text{Im}(\Gamma')$, and $\psi'(v) = \psi(v)$ for all $v$ in $\text{Im}(\Gamma) \cap \text{Im}(\Gamma')$. Observe that

- $\pi(\Gamma) = \psi(\Gamma)$,
- $\pi(\text{me}) = \psi(\text{me})$ (since $\text{me} \in \text{Im}(\Gamma)$),
- $\pi'(\Gamma') = \psi'(\Gamma')$,
- $v \in \text{outputs}^T(a) \Rightarrow \pi(v) = \psi(v)$ (since $v \in \text{outputs}^T(a)$ implies $v \in \text{Im}(\Gamma)$),
- $v \in \text{inputs}^T(a) \land \text{remembered}(v, a, \Gamma') \Rightarrow \pi'(v) = \psi'(v)$,

and that the statement

“every value $v$ in $\text{inputs}^T(a) \cap \{B \ldots T-1\}$ such that $\neg\text{remembered}(v, a, \Gamma')$ is replaced by an arbitrary value in $\{B \ldots B + W_N\}$”

is more specific than

“every value $v$ in $\text{inputs}^T(a)$ such that $\neg\text{remembered}(v, a, \Gamma')$ 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 $W_N$ identities in addition to their own, where $W_N \geq 1$,
- all $\beta$- and $\gamma$-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 $ABS_{\infty}^{T'.B,W_N}$. The two main differences between the following proposition and Proposition 6.3.8 are the following:

- Node processes no longer have to satisfy $\text{NoEqT}'$. 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 $\beta$- and $\gamma$-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 $W_N \geq 1$ and that all $\beta$- and $\gamma$-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

$$\| i \in T \cdot [A(i, T)] \ (cst_i, \Gamma_i, T)$$

$$\xrightarrow{a} \| i \in T \cdot [A(i, T)] \ (cst'_i, \Gamma'_i, T)$$

for some control states $cst_i$ and environments $\Gamma_i$. Let count and count' be states of $\zeta^T(N_B(\{0 \ldots B + W_N\}))$ such that for any state $st$ we have that

$$\text{count}[st] = \#\{i \in \{B \ldots \#T - 1\} \mid (cst_i, \Gamma_i, T) \approx st\}$$

and

$$\text{count'}[st] = \#\{i \in \{B \ldots \#T - 1\} \mid (cst'_i, \Gamma'_i, T) \approx st\}.$$ 

Then, for all $i$ in $\{0 \ldots B - 1\}$, given injective partial $(B + W_N)$-precollapsing functions $\psi_i : T \rightarrow \{0 \ldots B + W_N\}$ such that $\text{dom}(\psi_i) = \text{Im}(\Gamma_i)$, there exist injective partial $(B + W_N)$-precollapsing functions $\psi'_i : T \rightarrow \{0 \ldots B + W_N\}$ such that $\text{dom}(\psi'_i) = \text{Im}(\Gamma'_i)$, and $\psi'_i(v) = \psi_i(v)$ for all $v$ in $\text{Im}(\Gamma_i) \cap \text{Im}(\Gamma'_i)$ and, on letting

$$s = \left(\begin{array}{c}
\| i \in \{0 \ldots B - 1\} \cdot \\
[A(i, \{0 \ldots B + W_N\})] \ (cst_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\})
\end{array}\right)$$

$$\bigcup_{m=0}^{B-1} A(i, \{0 \ldots B + W_N\}) \bigcup_{m+B}^{B+W_N} A(i, \{0 \ldots B + W_N\})$$

and

$$s' = \left(\begin{array}{c}
\| i \in \{0 \ldots B - 1\} \cdot \\
[A(i, \{0 \ldots B + W_N\})] \ (cst'_i, \psi'_i(\Gamma'_i), \{0 \ldots B + W_N\})
\end{array}\right)$$

$$\bigcup_{m=0}^{B-1} A(i, \{0 \ldots B + W_N\}) \bigcup_{m+B}^{B+W_N} A(i, \{0 \ldots B + W_N\})$$

we have that $\phi(s) \xrightarrow{\phi(a)} \phi(s')$. 

176
Proof: Suppose that
\[ \left\{ i \in T \mid [A(i, T)](\text{cst}_t, \Gamma_i, T) \right\} \xrightarrow{a} \left\{ i \in T \mid [A(i, T)](\text{cst}_t', \Gamma_i', T) \right\}. \]

Let \( \text{count} \) and \( \text{count}' \) be as in the statement of the proposition. For every \( i \in T \), let \( \psi_i : T \rightarrow \{0 \ldots B + W_N\} \) be an injective partial \((B + W_N)\)-precollapsing function such that \( \text{dom}(\psi_i) = \text{Im}(\Gamma_i) \). We prove the result using a case analysis on the kind of event that \( a \) can be.

**Case 1.** Suppose that \( a \) is an \( \alpha \)-event of the form \( e.f_a(me, payloadIDs) \) for some \( me \) in \( T \) (i.e. \( a \) is a private event of \( N_{me}(T) \)). Then
\[ (\text{cst}_{me}, \Gamma_{me}, T) \xrightarrow{a} (\text{cst}_{me}', \Gamma_{me}', T) \quad (6.46) \]
and
\[ \forall i \in T \setminus \{me\} \bullet (\text{cst}_i', \Gamma_i', T) = (\text{cst}_i, \Gamma_i, T). \quad (6.47) \]

We now consider the various possibilities for the value of \( me \).

**Subcase 1.** Suppose that \( me \) is in \( \{0 \ldots B - 1\} \).

Then (6.47) implies that, on letting \( \psi_i' = \psi_i \) for all \( i \in \{0 \ldots B - 1\} \setminus \{me\} \),
\[ \forall i \in \{0 \ldots B - 1\} \setminus \{me\} \bullet (\text{cst}_i', \psi'_i(\Gamma_i'), \{0 \ldots B + W_N\}) = (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}). \quad (6.48) \]

Using (6.46), Corollary 6.4.5 implies that there exists an injective partial \((B + W_N)\)-precollapsing function \( \psi'_{me} \) such that \( \text{dom}(\psi'_{me}) = \text{Im}(\Gamma'_{me}) \), \( \psi'_{me}(v) = \psi_{me}(v) \) for all \( v \) in \( \text{Im}(\Gamma_{me}) \cap \text{Im}(\Gamma'_{me}) \), and
\[ (\text{cst}_{me}, \psi_{me}(\Gamma_{me}), \{0 \ldots B + W_N\}) \xrightarrow{a} (\text{cst}'_{me}, \psi'_{me}(\Gamma'_{me}), \{0 \ldots B + W_N\}), \]
where \( \hat{a} \) is like \( a \), except that
- every value \( v \) in \( \text{outputs}^t(a) \) is replaced by \( \psi_{me}(v) \),
- every value \( v \) in \( \text{inputs}^t(a) \) such that \( \text{remembered}(v, a, \Gamma'_{me}) \) is replaced by \( \psi'_{me}(v) \), and
- every value \( v \) in \( \text{inputs}^t(a) \cap \{B \ldots \#T - 1\} \) such that \( \neg \text{remembered}(v, a, \Gamma'_{me}) \) is replaced by an arbitrary value in \( \{B \ldots B + W_N\} \).

However, since \( me \) is in \( \{0 \ldots B - 1\} \), \( \psi_{me}(me) = me \), so
\[ (\text{cst}_{me}, \psi_{me}(\Gamma_{me}), \{0 \ldots B + W_N\}) \xrightarrow{a} (\text{cst}'_{me}, \psi'_{me}(\Gamma'_{me}), \{0 \ldots B + W_N\}). \quad (6.49) \]

177
From (6.47) we also have that
\[ \forall i \in \{B \ldots \# T - 1\} \bullet (\text{cst}_i', \Gamma_i', T) = (\text{cst}_i, \Gamma_i, T), \]
which means that
\[ \text{count}' = \text{count}. \quad (6.50) \]
Finally, using ideas similar to those used in Lemma 6.3.4 and Lemma 6.3.7, we can show that
\[ \hat{a} \in A(me, \{0 \ldots B + W_N\}) \setminus \bigcup_{i \in \{0 \ldots B + W_N\} \setminus \{me\}} A(i, \{0 \ldots B + W_N\}). \]
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.

\textit{Subcase 2.} Suppose that \( me \) is in \( \{B \ldots \# T - 1\} \).
Then \( me \) is not in \( \{0 \ldots B - 1\} \), so from (6.47) we have that
\[ \forall i \in \{0 \ldots B - 1\} \bullet (\text{cst}_i', \Gamma_i', T) = (\text{cst}_i, \Gamma_i, T). \]
Hence, on letting \( \psi'_i = \psi_i \) for all \( i \) in \( \{0 \ldots B - 1\} \),
\[ \forall i \in \{0 \ldots B - 1\} \bullet \]
\[ (\text{cst}_i', \psi'_i(\Gamma'_i), \{0 \ldots B + W_N\}) = (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}). \quad (6.51) \]
Let \( \psi_{me} : T \rightarrow \{0 \ldots B + W_N\} \) be an injective partial \((B + W_N)\)-precollapsing function such that \( \text{dom}(\psi_{me}) = \text{Im}(\Gamma_{me}) \). Using (6.46), Corollary 6.4.5 implies that there exists an injective partial \((B + W_N)\)-precollapsing function \( \psi'_{me} \) such that \( \text{dom}(\psi'_{me}) = \text{Im}(\Gamma'_{me}) \), \( \psi'_{me}(v) = \psi_{me}(v) \) for all \( v \) in \( \text{Im}(\Gamma_{me}) \cap \text{Im}(\Gamma'_{me}) \), and
\[ \hat{a} \xrightarrow{\psi_{me}(me)} (\text{cst}_{me}', \psi_{me}(\Gamma_{me}), \{0 \ldots B + W_N\}) \]
\[ \xrightarrow{\psi'_{me}(me)} (\text{cst}'_{me}, \psi'_{me}(\Gamma'_{me}), \{0 \ldots B + W_N\}), \]
where \( \hat{a} \) is like \( a \), except that
- every value \( v \) in \( \text{outputs}^l(a) \) is replaced by \( \psi_{me}(v) \),
- every value \( v \) in \( \text{inputs}^l(a) \) such that \( \text{remembered}(v, a, \Gamma'_{me}) \) is replaced by \( \psi'_{me}(v) \), and
- every value \( v \) in \( \text{inputs}^l(a) \cap \{B \ldots \# T - 1\} \) such that \( \neg \text{remembered}(v, a, \Gamma'_{me}) \) is replaced by an arbitrary value in \( \{B \ldots B + W_N\} \).
However, \( me \) is in \( \{ B \ldots \# T - 1 \} \) and also in \( \text{dom}(\psi_{me}) = \text{Im}(\Gamma_{me}) \) (since \( \Gamma_{me}(\text{myId}) \) must equal \( me \)), so the definition of \( \psi_{me} \) implies that \( \psi_{me}(me) = x \) for some \( x \in \{ B \ldots B + W_N \} \). Hence

\[
\dot{a} \rightarrow_{x} (\text{cst}_{me}, \psi_{me}(\Gamma_{me}), \{ 0 \ldots B + W_N \})
\]

(6.52)

In addition, by (6.47), it must be that

\[
\forall i \in \{ B \ldots \# T - 1 \} \setminus \{ me \} \bullet (\text{cst}_{i}', \Gamma_{i}', T) = (\text{cst}_{i}, \Gamma_{i}, T).
\]

Hence,

\[
\text{count}' = \text{count}[(\text{cst}_{me}, \Gamma_{me}, T) --, (\text{cst}_{me}', \Gamma_{me}', T)++].
\]

However, by Remark 6.3.3 we have that \( \phi(\psi_{me}(\Gamma)) = \phi(\psi_{me}'(\Gamma)) = \phi(\Gamma) \) for every environment \( \Gamma \). In addition, \( \phi(\{ 0 \ldots B + W_N \}) = \phi(T) \). Hence, by the definition of \( \approx \) (Definition 6.1.11),

\[
[(\text{cst}_{me}, \Gamma_{me}, T)] = [(\text{cst}_{me}, \psi_{me}(\Gamma_{me}), \{ 0 \ldots B + W_N \})]
\]

(6.53)

and

\[
[(\text{cst}_{me}', \Gamma_{me}', T)] = [(\text{cst}_{me}', \psi_{me}'(\Gamma_{me}), \{ 0 \ldots B + W_N \})].
\]

(6.54)

Therefore,

\[
\text{count}' = \text{count}[(\text{cst}_{me}, \psi_{me}(\Gamma_{me}), \{ 0 \ldots B + W_N \}) --,
(\text{cst}_{me}', \psi_{me}'(\Gamma_{me}), \{ 0 \ldots B + W_N \})++].
\]

Further, \( \text{count}[(\text{cst}_{me}, \Gamma_{me}, T)] \geq 1 \), which, thanks to (6.53), means that \( \text{count}[(\text{cst}_{me}, \psi_{me}(\Gamma_{me}), \{ 0 \ldots B + W_N \})] \geq 1 \). Hence, from (6.52) we can infer

\[
\text{count} \rightarrow_{\text{count}'} \text{count}.
\]

(6.55)

using the definition of \( \rightarrow_{\text{count}} \) (6.5). Finally, using ideas similar to those used in Lemma 6.3.4 and Lemma 6.3.7, we can show that

\[
\hat{a} \in \left( \bigcup_{i=B}^{B+W_N} A(i, \{ 0 \ldots B + W_N \}) \right) \setminus \bigcup_{i=0}^{B-1} A(i, \{ 0 \ldots B + W_N \})
\]

Therefore, combining (6.51) and (6.55) we have that

\[
s \rightarrow \hat{a} \rightarrow s'.
\]

Hence, and since \( \phi(\hat{a}) = \phi(a) \),

\[
\phi(s) \rightarrow_{\phi(a)} \phi(s'),
\]

as required.
Case 2. Suppose that $a = \tau$.
This case is almost identical to Case 1, above.

Case 3. Suppose that $a = c.f_3(x, y)$ for some distinct identities $x$ and $y$ in $T$ (i.e. $a$ is an event shared by $N_x(T)$ and $N_y(T)$).
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
\[
(cst_x, \Gamma_x, T) \xrightarrow{\alpha} (cst_x', \Gamma_x', T),
\]
(matching a visible symbolic event $\epsilon_x$ (see Definition 4.6.1),
\[
(cst_y, \Gamma_y, T) \xrightarrow{\alpha} (cst_y', \Gamma_y', T),
\]
(matching a visible symbolic event $\epsilon_y$, and
\[
\forall i \in T \setminus \{x, y\} \bullet (cst_i', \Gamma_i', T) = (cst_i, \Gamma_i, T).
\]
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\},

\[
\forall i \in \{0..B − 1\} \setminus \{x\} \bullet (cst_i', \psi_i'(\Gamma_i'), \{0..B + W_N\}) = (cst_i, \psi_i(\Gamma_i), \{0..B + W_N\}).
\]

Using Corollary 6.4.5 we can infer from (6.56) that there exists an injective partial $(B + W_N)$-precollapsing function $\psi_x'$ such that $\text{dom}(\psi_x') = \text{Im}(\Gamma_x')$, $\psi_x'(v) = \psi_x(v)$ for all $v$ in $\text{Im}(\Gamma_x) \cap \text{Im}(\Gamma_x')$, and
\[
(cst_x, \psi_x(\Gamma_x), \{0..B + W_N\}) \xrightarrow{\alpha_{\psi_x(x)}} (cst_x', \psi_x'(\Gamma_x'), \{0..B + W_N\}),
\]
where $\alpha_{\psi_x(x)}$ is like $\alpha$, except that

- every value $v$ in $\text{outputs}^l(a, \epsilon_x)$ is replaced by $\psi_x(v)$,
- every value $v$ in $\text{inputs}^l(a, \epsilon_x)$ such that $\text{remembered}(v, a, \epsilon_x, \Gamma_x')$ is replaced by $\psi_x'(v)$, and
- every value $v$ in $\text{inputs}^l(a, \epsilon_x) \cap \{B..#T − 1\}$ such that $\neg\text{remembered}(v, a, \epsilon_x, \Gamma_x')$ is replaced by an arbitrary value in $\{B..B + W_N\}$.

However, $\psi_x(x) = x$ (since $x$ is in \{0..B − 1\}), so
\[
(cst_x, \psi_x(\Gamma_x), \{0..B + W_N\}) \xrightarrow{\alpha_{\psi_x(x)}} (cst_x', \psi_x'(\Gamma_x'), \{0..B + W_N\}).
\]

Also, since $y$ is in \{B..#T − 1\}, the properties of $(B + W_N)$-precollapsing functions imply that $\psi_y(y)$ and $\psi_x'(y)$ are in $\{B..B + W_N\}$. Therefore $\alpha_x = c.f_3(x, \hat{y})$, where $\hat{y}$ equals
• \( \psi_x(y) \) if \( y \in \text{outputs}^I(a, \epsilon_x) \),

• \( \psi'_x(y) \) if \( y \in \text{inputs}^I(a, \epsilon_x) \) and \( \text{remembered}(y, a, \epsilon_x, \Gamma'_y) \), or

• some value in \( \{ B \cdot B + W_N \} \) if \( y \in \text{inputs}^I(a, \epsilon_x) \) and \( \neg \text{remembered}(y, a, \epsilon_x, \Gamma'_y) \).

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 \( \hat{y} \) is well-defined. Note that in all cases we have that \( \hat{y} \in \{ B \cdot B + W_N \} \).

Let \( \psi_y : T \rightarrow \{ 0 \ldots B + W_N \} \) be an injective partial \((B + W_N)\)-precollapsing function such that \( \text{dom}(\psi_y) = \text{Im}(\gamma_y) \). Using Corollary 6.4.5 we can infer from (6.57) that there exists an injective partial \((B + W_N)\)-precollapsing function \( \psi'_y \) such that \( \text{dom}(\psi'_y) = \text{Im}(\gamma'_y), \psi'_y(v) = \psi_y(v) \) for all \( v \) in \( \text{Im}(\gamma_y) \cap \text{Im}(\gamma'_y) \) and

\[
\frac{\hat{y}}{\psi_y(y)} \quad \frac{\hat{y}}{\psi'_y(y)} \quad \quad (6.61)
\]

where \( \hat{y} \) is like \( a \), except that

• every value \( v \) in \( \text{outputs}^I(a, \epsilon_y) \) is replaced by \( \psi_y(v) \),

• every value \( v \) in \( \text{inputs}^I(a, \epsilon_y) \) such that \( \text{remembered}(v, a, \epsilon_y, \Gamma'_y) \) is replaced by \( \psi'_y(v) \), and

• every value \( v \) in \( \text{inputs}^I(a, \epsilon_y) \setminus \{ B \cdot \# T - 1 \} \) such that \( \neg \text{remembered}(v, a, \epsilon_y, \Gamma'_y) \) is replaced by an arbitrary value in \( \{ B \cdot B + W_N \} \).

We have that \( \psi_y(x) = \psi'_y(x) = x \), since \( x \) is in \( \{ 0 \ldots B - 1 \} \). In addition, clearly \( y \) must be in \( \text{outputs}^I(a, \epsilon_y) \). Therefore,

\[
\hat{y} = c.f_3(x, \psi_y(y)).
\]

Let \( \pi : \{ 0 \ldots B + W_N \} \rightarrow \{ 0 \ldots B + W_N \} \) be a bijection such that \( \pi(v) = v \) for all \( v \) in \( \{ 0 \ldots B - 1 \} \) and \( \pi(\psi_y(y)) = \hat{y} \). Since \( y \) is in \( \{ B \cdot \# T - 1 \} \) and \( y \) is in \( \text{dom}(\psi_y) \) (as \( \gamma_y(\text{myld}) \) must equal \( y \)), the definition of \( \psi_y \) implies that \( \psi_y(y) \) is in \( \{ B \cdot B + W_N \} \). In addition, \( \hat{y} \) is in \( \{ B \cdot B + W_N \} \). So \( \pi \) is well-defined. Then

\[
\pi(\hat{y}) = c.f_3(\pi(x), \pi(\psi_y(y))) = c.f_3(x, \hat{y}) = \hat{a}_x.
\]

Therefore, using Corollary 6.4.6 (with \( \pi' = \pi \)), we can infer from (6.61) that

\[
\frac{\hat{a}_x}{\pi(\psi_y(y))} \quad \frac{\hat{a}_x}{\pi(\psi'_y(y))} \quad (6.62)
\]

In addition, since \( x \) is not in \( \{ B \cdot \# T - 1 \} \), (6.58) implies that

\[
\forall i \in \{ B \cdot \# T - 1 \} \setminus \{ y \} \quad (\text{cst}^i, \Gamma_i', T) = (\text{cst}^i, \Gamma_i, T).
\]

Hence,

\[
\text{count}' = \text{count}[(\text{cst}^y, \Gamma_y, T) \ldots (\text{cst}^i, \Gamma_i', T)^{++}].
\]
Therefore, by Remark 6.3.3 and thanks to the properties of \( \pi \) we have that
\[
\phi(\pi(\psi_y(\Gamma))) = \phi(\pi(\psi_y(\Gamma))) = \phi(\Gamma)
\]
for every environment \( \Gamma \). In addition, \( \phi(\{0 .. B + W_N\}) = \phi(T) \). Hence, by the definition of the \( \approx \) equivalence relation (Definition 6.1.11)
\[
[(\text{cst}_y, \Gamma_y, T)] = [(\text{cst}_y, \pi(\psi_y(\Gamma_y)), \{0 .. B + W_N\})]
\]
(6.64)
and
\[
[(\text{cst}_y', \Gamma_y', T)] = [(\text{cst}_y', \pi(\psi_y'(\Gamma_y')), \{0 .. B + W_N\})].
\]
(6.65)
Therefore, from (6.63) we have that
\[
count' = count([\text{cst}_y, \pi(\psi_y(\Gamma_y)), \{0 .. B + W_N\}],
\]
(6.66)
\[
= count([\text{cst}_y', \pi(\psi_y'(\Gamma_y')), \{0 .. B + W_N\}]).
\]
Further, \( count([\text{cst}_y, \pi(\psi_y(\Gamma_y)), \{0 .. B + W_N\}]) \geq 1 \), which, thanks to (6.64), means that \( count([\text{cst}_y, \pi(\psi_y(\Gamma_y)), \{0 .. B + W_N\}]) \geq 1 \). Hence, from (6.62), combined with the fact that \( \pi(\psi_y(y)) \) is in \( \{B .. B + W_N\} \), we can infer that
\[
\text{count} \xrightarrow{\hat{a}_x} \infty \text{count}'.
\]
(6.66)
using the definition of \( \xrightarrow{\infty} \) (6.5).

Finally, using ideas similar to those used in Lemma 6.3.4 and Lemma 6.3.7, we can show that
\[
\hat{a}_x \in \left(A(x, \{0 .. B + W_N\}) \cap \bigcup_{i=B}^{B+W_N} A(i, \{0 .. B + W_N\})\right)
\]
\[
\setminus \bigcup_{i\notin\{0..B-1\}\setminus\{x\}} A(i, \{0 .. B + W_N\}),
\]
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
\[
\forall i \in T \bullet (\text{cst}_i, \Gamma_i, T) \xrightarrow{a} (\text{cst}_i', \Gamma_i', T).
\]
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 \( (B+W_N)\)-precollapsing function \( \psi_i' \) such that \( \text{dom}(\psi_i') = \text{Im}(\Gamma_i') \), \( \psi_i'(v) = \psi_i(v) \) for all \( i \) in \( \text{Im}(\Gamma) \cap \text{Im}(\Gamma') \) and
\[
\forall i \in T \bullet (\text{cst}_i, \psi_i(\Gamma_i), \{0 .. B + W_N\}) \xrightarrow{a} (\text{cst}_i', \psi_i'(\Gamma_i'), \{0 .. B + W_N\}).
\]

Therefore, since for every \(i\) in \(\{0 \ldots B - 1\}\) we have that \(\psi_i(i) = i\),

\[
\forall i \in \{0 \ldots B - 1\} \quad (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) \xrightarrow{a} (\text{cst}_i', \psi_i'(\Gamma_i'), \{0 \ldots B + W_N\}),
\]

(6.67)

and, since for every \(i\) in \(\{B \ldots \# T - 1\}\) we have that \(\psi_i(i)\) is in \(\{B \ldots B + W_N\}\),

\[
\forall i \in \{B \ldots \# T - 1\} \quad (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) \xrightarrow{a} (\text{cst}_i', \psi_i'(\Gamma_i'), \{0 \ldots B + W_N\}),
\]

(6.68)

for some \(x_i\) in \(\{B \ldots B + W_N\}\).

For every \(i\) in \(\{B \ldots \# T - 1\}\) and every environment \(\Gamma\), Remark 6.3.3 implies that \(\phi(\psi_i'(\Gamma)) = \phi(\psi_i(\Gamma)) = \phi(\Gamma)\). In addition, \(\phi(\{0 \ldots B + W_N\}) = \phi(T)\). Hence,

\[
\forall i \in \{B \ldots \# T - 1\} \quad [(\text{cst}_i, \Gamma_i, T)] = [(\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\})]
\]

(6.69)

and

\[
\forall i \in \{B \ldots \# T - 1\} \quad [(\text{cst}_i', \Gamma_i', T)] = [(\text{cst}_i', \psi_i'(\Gamma_i'), \{0 \ldots B + W_N\})].
\]

(6.70)

From (6.69) and by the definition of \(\text{count}\) we have that

\[
\forall i \in \{1 \ldots k\} \quad \text{count}[\text{src}_i] = \#\{j \in \{B \ldots \# T - 1\} \mid (\text{cst}_j, \Gamma_j, T) \approx \text{src}_i\}
\]

\[= \#\{j \in \{B \ldots \# T - 1\} \mid (\text{cst}_j, \psi_j(\Gamma_j), \{0 \ldots B + W_N\}) \approx \text{src}_i\}.
\]

Hence, for every \(i\) in \(\{1 \ldots k\}\) there are \(\text{count}[\text{src}_i]\) many states

\[
\text{src}_1^i = (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}),
\]

\[
\text{src}_2^i = (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}),
\]

\[
\vdots
\]

\[
\text{src}_\text{count}[\text{src}_i]^i = (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}),
\]

(6.71)

all related to \([\text{src}_i]\) by \(\approx\), where for all \(i\) in \(\{1 \ldots k\}\) and for all \(l\) in \(\{1 \ldots \text{count}[\text{src}_i]\}\), the indices \(j_l^i\) are in \(\{B \ldots \# T - 1\}\) and are all distinct. From (6.68) we have that

\[
\forall i \in \{1 \ldots k\} \forall l \in \{1 \ldots \text{count}[\text{src}_i]\} \quad \text{src}_l^i \xrightarrow{a} \text{tgt}_l^i,
\]

(6.72)

where each \(x_l^i\) is in \(\{B \ldots B + W_N\}\) and where, for all \(i\) in \(\{1 \ldots k\}\),

\[
\text{tgt}_1^i = (\text{cst}_i, \psi_i'(\Gamma_i'), \{0 \ldots B + W_N\}),
\]

\[
\text{tgt}_2^i = (\text{cst}_i, \psi_i'(\Gamma_i'), \{0 \ldots B + W_N\}),
\]

\[
\vdots
\]

\[
\text{tgt}_\text{count}[\text{src}_i]^i = (\text{cst}_i, \psi_i'(\Gamma_i'), \{0 \ldots B + W_N\}),
\]

(6.73)

with each \(j_l^i\) as above. Further,

\[
\{\text{cst}_1^i, \psi_B(\Gamma_B'), \{0 \ldots B + W_N\}\},
\]

\[
\text{cst}_2^i, \psi_B(\Gamma_B'), \{0 \ldots B + W_N\},
\]

\[
\vdots
\]

\[
\text{cst}_{\# T - 1}^i, \psi_B(\Gamma_{\# T - 1}', \{0 \ldots B + 1\})\}
\]

\[= \{\text{tgt}_1^i \mid i \in \{1 \ldots k\}, l \in \{1 \ldots \text{count}[\text{src}_i]\}\},
\]

183
and
\[
\#\{B \ldots \# T - 1\} = \#\{(i, l) \mid i \in \{1 \ldots k\} \land l \in \{1 \ldots \text{count}[st_i]\}\}.
\]

Hence, from (6.70), we have that
\[
\forall st \in S \bullet \quad \text{count}'[st] = \#\{j \in \{B \ldots \# T - 1\} \mid (cst'_j, \Gamma'_j, T) \approx st\}
\]
\[
= \#\{j \in \{B \ldots \# T - 1\} \mid (\text{cst}'_j, \psi(\Gamma'_j), \psi(T)) \approx st\}
\]
\[
= \#\{(i, l) \mid i \in \{1 \ldots k\} \land l \in \{1 \ldots \text{count}[st_i]\} \land \text{tgt}_i \approx st\}. 
\]

Combining (6.71)–(6.74) and the assumption of the proposition that \(\# T \geq B + 1\), we use the definition of \(\rightarrow_{\infty}\) (6.7) to infer that
\[
\text{count} \xrightarrow{a}_{\infty} \text{count}'. 
\]

Finally, the definition of a \(\gamma\)-event implies that
\[
\forall i \in \{0 \ldots B + W_N\} \bullet a \in A(i, \{0 \ldots B + W_N\}).
\]

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 \(\beta\)- and \(\gamma\)-events contain no payload identities, i.e. that every \(\beta\)-event is of the form \(c.f_\beta(me, other)\) for some distinct identities \(me\) and \(other\) in \(T\) and every \(\gamma\)-event is of the form \(c.f_\gamma\). The following example shows that node identities present in payloads of \(\beta\)-events break the proposition. An example showing the same for \(\gamma\)-events is similar.

**Example 6.4.8.** Let
\[
N_{\text{myId}}(t) = \text{env!myId?x:x} \rightarrow (\text{sync!myId?other:t!x} \rightarrow \text{STOP}) \\
\quad a \rightarrow \text{sync?other:t!myId!x} \rightarrow \text{STOP}).
\]

Observe that constructs \(\text{sync!myId?other:t!x}\) and \(\text{sync?other:myId!x}\) can generate \(\beta\)-events that contain payload of type \(t\) (i.e. the identity \(x\)). Since \(N_{\text{myId}}(t)\) remembers only at most one node identity (\(x\)) in addition to its own, we have that \(W_N = 1\). Let \(B = 2\) and \(T = \{0 \ldots 2\}\). Also, let
\[
cst_0 = \text{sync!myId?other:t!x} \rightarrow \text{STOP} \\
\quad a \rightarrow \text{sync?other:t!myId!x} \rightarrow \text{STOP},
\]

184
\[\text{cst}_1 = \text{sync?other!:myId!x} \rightarrow \text{STOP},\]
\[\forall i \in \{0, 1\} \bullet \text{cst}_i' = \text{STOP}\]

and
\[
\Gamma_0 = \{\text{myId} \mapsto 0, x \mapsto 2\}, \quad \Gamma'_0 = \{\text{myId} \mapsto 0\},
\Gamma_1 = \{\text{myId} \mapsto 1, x \mapsto 2\}, \quad \Gamma'_1 = \{\text{myId} \mapsto 1\}.
\]

Then, for all environments \(\Gamma_2\) and \(\Gamma'_2\) such that \(\Gamma_2 = \Gamma'_2\) and all control states \(\text{cst}_2\) and \(\text{cst}'_2\) such that \(\text{cst}_2 = \text{cst}'_2\), we have that
\[
\parallel i \in T \bullet [A(i, T)] (\text{cst}_i, \Gamma_i, T)
\]
\[
\xrightarrow{\text{sync.}0.1.2}\]
\[
\parallel i \in T \bullet [A(i, T)] (\text{cst}'_i, \Gamma'_i, T).
\]

Now, let \(\psi_0, \psi_1 : T \rightarrow \{0 \ldots B + W_N\}\) be such that
\[
\psi_0(0) = 0, \quad \psi_0(2) = B,
\psi_1(1) = 1, \quad \psi_1(2) = B + 1.
\]

Then for both \(i = 0\) and \(i = 1\), \(\psi_i\) is an injective partial \((B + W_N)\)-precollapsing function such that \(\text{dom}(\psi_i) = \text{Im}(\Gamma_i)\). However,
\[
\psi_0(\Gamma_0) = \{\text{myId} \mapsto 0, x \mapsto B\},
\psi_1(\Gamma_1) = \{\text{myId} \mapsto 1, x \mapsto B + 1\},
\]

so
\[
\parallel i \in \{0, 1\} \bullet [A(i, \{0 \ldots B + W_N\})] (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\})
\]
cannot do \(\phi(\text{sync.}0.1.2) = \text{sync.}0.1.2\). Hence, if \(s\) is defined as in Proposition 6.4.7, then \(\phi(s) \xrightarrow{\phi(\text{sync.}0.1.2)} \).

End of example.

Our next result says that given nodes that can remember at least one identity (in addition to their own), if \(\phi(\text{Nodes}(T))\) can perform a sequence of events \(s\) and reach some state \(P\), then \(\text{ABS}_{\infty}^{T,B,W_N}\) can also perform \(s\) and reach a state corresponding to \(P\).

**Lemma 6.4.9.** Suppose that \(W_N \geq 1\) and suppose that all \(\beta\)- and \(\gamma\)-events contain no payloads with node identities. Let \(T\) be an instantiation of type \(t\) such that \(\#T \geq B + 1\). Then if
\[
\phi(\text{Nodes}(T)) \xrightarrow{s} \phi(\parallel i \in T \bullet [A(i, T)] (\text{cst}_i, \Gamma_i, T))
\]
for some control states \(\text{cst}_i\) and environments \(\Gamma_i\), then for all \(i\) in \(\{0 \ldots B - 1\}\), there exist an injective partial \((B + W_N)\)-precollapsing function \(\psi_i : T \rightarrow \{0 \ldots B + W_N\}\)

185
such that \( \text{dom}(\psi_i) = \text{Im}(\Gamma_i) \) and

\[
\text{ABS}^T_{\infty, B, W_N} \mapsto \phi \left( \left( \begin{array}{c}
\top \quad i \in \{0 \ldots B - 1\} \bullet \\
\downarrow \quad [A(i, \{0 \ldots B + W_N\})] \quad (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) \\
\cup_{i=0}^{\infty-1} A(i, \{0 \ldots B + W_N\}) \cup_{i=0}^{\infty} A(i, \{0 \ldots B + W_N\}) \\
\#T - B, 0, \ldots, 0
\end{array} \right) \right),
\]

where \( \text{count} \) is a state of \( \zeta^T(\mathcal{N}_B(\{0 \ldots B + W_N\})) \) such that for all states \( s \),

\[
\text{count}[s] = \#\{i \in \{B \ldots \#T - 1\} \mid (\text{cst}_i, \Gamma_i, T) \approx s\}.
\]

**Proof:** We prove the result by an induction on the length of \( s \).

**Base Case.** Suppose that \( s = () \).

Then, for all \( i \) in \( \{0 \ldots B - 1\} \), \( \Gamma_i = \{\text{myId} \mapsto i\} \). Now, for every \( i \) in \( \{0 \ldots B - 1\} \) let \( \psi_i : T \rightarrow \{0 \ldots B + W_N\} \) be such that \( \text{dom}(\psi_i) = \{i\} \) and \( \psi_i(i) = i \). Then, for every \( i \) in \( \{0 \ldots B - 1\} \), \( \psi_i \) is an injective partial \((B + W_N)\)-precollapsing function from \( T \) to \( \{0 \ldots B + W_N\} \) such that \( \text{dom}(\psi_i) = \text{Im}(\Gamma_i) \).

Observe that \((\text{cst}_i, \Gamma_i, T) = \mathcal{N}_i(T) \) for all \( i \) in \( T \). Therefore, by Remark 6.1.4 we have that

\[
\forall i \in \{0 \ldots B - 1\} \bullet (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) = \mathcal{N}_{\psi_i(i)}(\{0 \ldots B + W_N\}).
\]

However, for all \( i \) in \( \{0 \ldots B - 1\} \), \( \psi_i(i) = i \). Hence,

\[
\forall i \in \{0 \ldots B - 1\} \bullet (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) = \mathcal{N}_i(\{0 \ldots B + W_N\})
\]

Also, by the definition of \( \text{count} \),

\[
\text{count} = (\#T - B, 0, \ldots, 0).
\]

Therefore, since

\[
\text{ABS}^T_{\infty, B, W_N} = \phi \left( \left( \begin{array}{c}
\top \quad i \in \{0 \ldots B - 1\} \bullet \\
\downarrow \quad [A(i, \{0 \ldots B + W_N\})] \quad (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) \\
\cup_{i=0}^{\infty-1} A(i, \{0 \ldots B + W_N\}) \cup_{i=0}^{\infty} A(i, \{0 \ldots B + W_N\}) \\
\#T - B, 0, \ldots, 0
\end{array} \right) \right),
\]

we have that

\[
\text{ABS}^T_{\infty, B, W_N} \mapsto \phi \left( \left( \begin{array}{c}
\top \quad i \in \{0 \ldots B - 1\} \bullet \\
\downarrow \quad [A(i, \{0 \ldots B + W_N\})] \quad (\text{cst}_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) \\
\cup_{i=0}^{\infty-1} A(i, \{0 \ldots B + W_N\}) \cup_{i=0}^{\infty} A(i, \{0 \ldots B + W_N\}) \\
\#T - B, 0, \ldots, 0
\end{array} \right) \right),
\]

\[
186
\]
establishing the result in the base case.

**Inductive Case.** Suppose the result holds for sequence $s$.

Let $sa = s^*\langle a \rangle$ for some event $a$. Suppose that

$$\phi(Nodes(T)) \rightarrow^a \phi(\{ i \in T \cdot [A(i, T)] (cst'_i, \Gamma'_i, T) \})$$

for some control states $cst'_i$ and environments $\Gamma'_i$. Then, for every $i$ in $T$ there exists a control state $cst_i$ and an environment $\Gamma_i$ such that

$$\phi(Nodes(T)) \rightarrow^a \phi(\{ i \in T \cdot [A(i, T)] (cst_i, \Gamma_i, T) \}) \quad (6.76)$$

Then, by the inductive hypothesis, for $i$ in $\{0 \ldots B - 1\}$, there exist injective partial $(B + W_N)$-precollapsing functions $\psi_i : T \rightarrow \{0 .. B + W_N\}$ such that $\text{dom}(\psi_i) = \text{Im}(\Gamma_i)$ and

$$\text{ABS}^T_{\infty} (B, W_N) \rightarrow^s \phi \left( \bigg\{ i \in \{0 \ldots B - 1\} \cdot \left[ A(i, \{0 \ldots B + W_N\})\right] (cst_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) \right)$$

$$\bigcup_{i=0}^{B-1} A(i,\{0..B+W_N\})\bigcup_{\text{count}}^{B+W_N} A(i,\{0..B+W_N\})$$

where $\text{count}$ is a state of $\xi^T_{\infty} (N_B(\{0 \ldots B + W_N\}))$ such that for all states $st$,

$$\text{count}[st] = \# \{ i \in \{B \ldots \# T - 1\} \mid (cst_i, \Gamma_i, T) \approx st \}.$$

From (6.76) we can infer that there exists an event $a'$ such that $\phi(a') = a$ and

$$\bigg\{ i \in T \cdot [A(i, T)] (cst'_i, \Gamma'_i, T) \rightarrow' a' \bigg\{ i \in T \cdot [A(i, T)] (cst'_i, \Gamma'_i, T) \}.$$

Hence, by Proposition 6.4.7 we have that for $i$ in $\{0 \ldots B - 1\}$, there exist injective partial $(B + W_N)$-precollapsing functions $\psi'_i : T \rightarrow \{0 .. B + W_N\}$ such that $\text{dom}(\psi'_i) = \text{Im}(\Gamma'_i)$ and

$$\phi\left( \bigg\{ i \in \{0 \ldots B - 1\} \cdot \left[ A(i, \{0 \ldots B + W_N\})\right] (cst_i, \psi_i(\Gamma_i), \{0 \ldots B + W_N\}) \right)$$

$$\bigcup_{i=0}^{B-1} A(i,\{0..B+W_N\})\bigcup_{\text{count}}^{B+W_N} A(i,\{0..B+W_N\})$$

$$\phi(\phi(a')) \rightarrow' \phi\left( \bigg\{ i \in \{0 \ldots B - 1\} \cdot \left[ A(i, \{0 \ldots B + W_N\})\right] (cst'_i, \psi'_i(\Gamma'_i), \{0 \ldots B + W_N\}) \right)$$

$$\bigcup_{i=0}^{B-1} A(i,\{0..B+W_N\})\bigcup_{\text{count'}}^{B+W_N} A(i,\{0..B+W_N\})$$

187
where \(\text{count}'\) is a state of \(\zeta_T^\infty(\mathcal{N}_B(\{0 \ldots B + W_N\}))\) such that for all states \(st\),

\[
\text{count}'[st] = \#\{i \in \{B \ldots T\} \mid (\text{cst}'_i, \Gamma'_i, T) \approx st\}.
\]

By combining the above with (6.77) and by recalling that \(\phi(a') = a\), we get that

\[
\text{ABS}_{T,B,N}^\infty \trans{sa} \phi \left( \left\{ \begin{array}{l}
i \in \{0 \ldots B - 1\} \bullet \\
[A(i, \{0 \ldots B + W_N\})] (\text{cst}'_i, \psi'_i(\Gamma'_i), \{0 \ldots B + W_N\})\\n\bigcup_{j=0}^{B-1} A(i,\{0..B+WN\}) |\bigcup_{j=0}^{B-1} WN A(i,\{0..B+WN\}) \end{array} \right\} \right),
\]

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 \(W_N \geq 1\) and suppose that all \(\beta\)- and \(\gamma\)-events contain no payloads with node identities. Let \(T\) be an instantiation of type \(t\) such that \(\#T \geq B + 1\). Then

\[
\text{ABS}_{T,B,N}^\infty \subseteq_T \phi(\text{Nodes}(T)).
\]

**Proof:** Let \(tr\) be a trace of \(\phi(\text{Nodes}(T))\) and let \(s\) be a sequence of events such that \(s \setminus \{\tau\} = tr\) and

\[
\phi(\text{Nodes}(T)) \trans{s} .
\]

Then, Lemma 6.4.9 implies that

\[
\text{ABS}_{T,B,N}^\infty \trans{s},
\]

which means that \(tr\) is a trace of \(\text{ABS}_{T,B,N}^\infty\).

### 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 \(\phi\). 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 \(\text{ABS}_{z,n}^B\) such that, for certain values of \(n\) (to be established later) and \(T\) large enough,

\[
\text{ABS}_{z,n}^B \subseteq \phi(\text{Nodes}(T)). \tag{6.78}
\]

\(\text{ABS}_{z,n}^B\) is very similar to \(\text{ABS}_{T,B,n}^\infty\), 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
The techniques described in this section are similar to those presented in Section 3.3.1.

In Section 6.5.1 we will define a counter state machine $\zeta_z(\mathcal{N}_B(\{0 \ldots B + m\}))$ that abstracts the nodes $\mathcal{N}_B(\{0 \ldots B + m\}), \ldots, \mathcal{N}_{#T-1}(\{0 \ldots B + m\})$. The full abstract model, for $n = 0$, and $n = m$ for $m > 0$, respectively, is:

$$ABS^{B,0}_z \doteq \begin{cases} \ | i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B\})] \mathcal{N}_i(\{0 \ldots B\}) \\ \cup_{r=0}^{B-1} A(i, \{0 \ldots B\}) \| A_r^B \\ \zeta_z(\mathcal{N}_B(\{0 \ldots B + 1\}))\|\mathcal{R}\end{cases}$$

and

$$ABS^{B,m}_z \doteq \phi \begin{cases} \ | i \in \{0 \ldots B - 1\} \bullet [A(i, \{0 \ldots B + m\})] \mathcal{N}_i(\{0 \ldots B + m\}) \\ \cup_{r=0}^{B-1} A(i, \{0 \ldots B + m\}) \| \cup_{r=B}^{B+m} A(i, \{0 \ldots B + m\}) \\ \zeta_z(\mathcal{N}_B(\{0 \ldots B + m\}))\end{cases},$$

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 $T$, $ABS^{B,0}_z$ and $ABS^{B,W_N}_z$ form traces anti-refinements of $ABS^{T,B,0}_z$ and $ABS^{T,B,W_N}_z$, respectively. Hence, using the results from Section 6.3 and Section 6.4, they form anti-refinements of $\phi(\text{Nodes}(T))$ 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

- $(S(i), s_0(i), \Sigma(i, \{0 \ldots B + m\}) \cup \{\tau\}, \rightarrow_i)$ is the state machine of node $\mathcal{N}_i(\{0 \ldots B + m\})$ for $i$ in $\{B \ldots B + m\}$,
- $S = \bigcup \{ S(i) \mid i \in \{B \ldots B + m\} \}$ is the set of states of nodes $\mathcal{N}_B(\{0 \ldots B + m\}), \ldots, \mathcal{N}_{B+m}(\{0 \ldots B + m\}),$
- $E = \{[s_{t_1}], \ldots, [s_{t_k}]\} = S/\approx$, and
- $[s_{t_i}]$ is the equivalence class within $E$ that contains all the initial states $s_0(i)$ for $i$ in $\{B \ldots B + m\}$.

Let $z$ be a function from $\{1 \ldots k\}$ to $\mathbb{N}$. For every $i$ in $\{1 \ldots k\}$, $z_i$ is a threshold for the counter corresponding to $[s_{t_i}]$. Given a state $st$, we make the notational convention that $z[st] = z_i$, where $i$ is the unique element in $\{1 \ldots k\}$ such that $[st] = [s_{t_i}]$.

Our aim is to create an abstract state machine

$$\zeta_z(\mathcal{N}_B(\{0 \ldots B + m\})) = (A_z, a_z, \Sigma^\tau_z, \rightarrow_z),$$

which is $t$-independent and which, for all sufficiently large $T$, forms an abstraction of $\zeta^\tau_z(\mathcal{N}_B(\{0 \ldots B + m\}))$.

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 \( \{1 \ldots k\} \), \( c_i \) is an integer between 0 and \( z_i \), and

(ii) either there exists \( i \) in \( \{1 \ldots k\} \) 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 \( \{B \ldots B+m\} \). 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.

\[
\Sigma_z = \bigcup \{ \Sigma(i, \{0 \ldots B + m\}) \mid i \in \{B \ldots B+m\} \}
\]

and

\[
\Sigma^t_z = \Sigma_z \cup \{\tau\}.
\]

Finally, we define the transition relation, \( \rightarrow_z \). Let \( \text{count} \) and \( \text{count}' \) be two states in \( A_z \). Intuitively, there is an \( a \)-transition available between \( \text{count} \) and \( \text{count}' \) if there are some local states where \( a \) is available, the values of the counters of \( \text{count} \) indicate there are enough processes in those states, and the counters of \( \text{count}' \) 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 \( a \rightarrow_z \) by considering the type of event that \( a \) can be.

Let \( a = \tau \). Then, there is a \( \tau \)-transition between \( \text{count} \) and \( \text{count}' \) if the following hold.

- There is a local state \( st \) in \( S(me) \), for some \( me \) in \( \{B \ldots B+m\} \), in which \( \tau \) can be executed to reach some state \( st' \).

- Let \( i \) in \( \{1 \ldots k\} \) 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 \( \text{count}' \) 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

\[
\text{count}[st_{++z}] = \text{count}[st] \leftarrow \min\{\text{count}[st] + 1, z[st]\},
\]

and

\[
\text{count}[st_{--z}] = \text{count}[st] \leftarrow \max\{\text{count}[st] - 1, 0\}.
\]

Then, formally:

\[
\begin{align*}
\text{count} & \xrightarrow{\tau_z} \text{count}' \\
\iff & \exists me \in \{B \ldots B + m\} \bullet \\
& \exists st, st' \in S(me) \bullet \\
& st \xrightarrow{\alpha_{me}} st' \land \text{count}[st] \geq \min\{z[st], 1\} \\
& \land (\text{count}' = \text{count}[st_{--z}, st'_{++z}]) \\
& \lor \text{count}[st] = z[st] \land \text{count}' = \text{count}[st'_{++z}]).
\end{align*}
\]

(6.79)

Next, let \( me \) be some identity in \( \{B \ldots B + m\} \). Let \( a \) be either

- an \( \alpha \)-event of the form \( c.f_\alpha(me, payloadIDs) \), or
- a \( \beta \)-event of the form \( c.f_\beta(me, other, payloadIDs) \) or \( c.f_\beta(other, me, payloadIDs) \), where \( other \in \{0 \ldots B - 1\} \).

This case is almost identical to the case above. Formally, for every \( a \) as above,

\[
\begin{align*}
\text{count} & \xrightarrow{a_z} \text{count}' \\
\iff & \exists st, st' \in S(me) \bullet \\
& st \xrightarrow{\alpha_{me}} st' \land \text{count}[st] \geq \min\{z[st], 1\} \\
& \land (\text{count}' = \text{count}[st_{--z}, st'_{++z}]) \\
& \lor \text{count}[st] = z[st] \land \text{count}' = \text{count}[st'_{++z}]).
\end{align*}
\]

(6.80)

Now, let \( me \) and \( other \) be two distinct identities in \( \{B \ldots B + m\} \). Let \( a \) be a \( \beta \)-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,

\[
\begin{align*}
count & \xrightarrow{a} z \ count' \\
\iff & \exists \ st_{me}, \ st'_{me} \in S(\text{me}), \ st_{other}, \ st'_{other} \in S(\text{other}) \bullet \\
& st_{me} \xrightarrow{a} st'_{me} \land st_{other} \xrightarrow{a} st'_{other} \\
& \land count[st_{me}] \geq \min\{z[st_{me}], p\} \\
& \land count[st_{other}] \geq \min\{z[st_{other}], p\} \\
& \land (count' = count[st_{me} - z, st_{other} - z, st'_{me} + z, st'_{other} + z] \\
& \quad \lor count[st_{me}] = z[st_{me}] \\
& \quad \land count'[st_{me}] = z[st_{me}] \\
& \quad \land count' = count[st_{other} - z, st'_{me} + z, st'_{other} + z] \\
& \quad \lor count[st_{other}] = z[st_{other}] \\
& \quad \land count'[st_{other}] = z[st_{other}] \\
& \quad \land count' = count[st'_{me} + z, st'_{other} + z]),
\end{align*}
\]

\((6.81)\)

where \( p = 2 \) if \([st_{me}] = [st_{other}]\) and \( p = 1 \) otherwise.

Finally, let \( a \) be a \( \gamma \)-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 \( \{1 \ldots k\} \), 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 \( \{1 \ldots k\} \), 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 \( v : \{1 \ldots k\} \rightarrow \mathbb{N} \) such that:

1. \( \sum_{i=1}^{k} v_i \geq \max\{z_1, 1\} \),
2. for \( i \) in \( \{1 \ldots k\} \), 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
3. there exists a concrete model such that for every \( i \) in \( \{1 \ldots k\} \), 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 $\{1..k\}$, $\text{count}[st_i]$ processes can move from states in $[st_i]$, and $\text{count}'[st_i]$ 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 $v_i = \text{count}[st_i] < z_i$ or $v_i \geq \text{count}[st_i] = z_i$, and

- for every $i$ in $\{1..k\}$ there are $v'_i$ transitions entering equivalence class $[st_i]$, where either $v'_i = \text{count}'[st_i] < z_i$ or $v_i \geq \text{count}'[st_i] = z_i$.

Formally, for every $\gamma$-event $a$ in $\Sigma_z$,

\[
\text{count} \xrightarrow{a} \text{count}'
\]

\[\Leftrightarrow\]

\[
\exists v : \{1..k\} \rightarrow \mathbb{N} \quad \exists \text{src, tgt} : \mathbb{N} \times \mathbb{N} \rightarrow S \quad \sum_{i=1}^{k} v_i \geq \max\{z_1, 1\} \quad \text{and} \quad (\forall i \in \{1..k\}, j \in \{1..v_i\} \quad \text{src}_i^j \approx st_i \wedge \exists \text{me} \in \{B..B+m\} \quad \text{src}_i^j \xrightarrow{a} \text{me} \quad \text{tgt}_i^j \approx st_i)
\]

\[
\wedge \forall st \in S \quad \text{count}'[st] = \min\{z[st], \#\{(i,j) \mid i \in \{1..k\} \wedge j \in \{1..v_i\} \wedge \text{tgt}_i^j \approx st\}\}
\]

By clause (iii) of Remark 6.1.9, the four parts of the definition of $\xrightarrow{a}$ 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.

193
Proposition 6.5.1. For every $i$ in $\{1..k\}$, \(\max\{\sum_{j \in \text{targets}(i)} \text{count}'[st_j], z_i, 1\}\) is an upper bound for the values of $v_i$ that need to be considered in the body of (6.82), where \(\text{targets}(i) = \{j \in \{1..k\} \mid \exists st, st' \in S, me \in \{B..B+m\} \bullet st \approx st_i \land st \overset{e}{\rightarrow}_{me} st' \land st' \approx st_j\}\).

Proof: Similar to the proof of Proposition 6.5.1.

Example 6.5.2. Recall the $N_{myId}(t)$ 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 $\zeta(z(N_1(\{0,1,2\})))$, as shown in Figure 6.3.

End of example.

Figure 6.3: Counter state machine $\zeta(z(N_1(\{0,1,2\})))$ for $z_1 = z_2 = z_3 = 1$ and $N_B(\{0..B+m\})$ 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 $\phi$-renamed parallel compositions of nodes.
Recall Definition 3.3.5 that says that, given type \( T \) and function \( z : \{1 \ldots k\} \to \mathbb{N} \), 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 \( \text{cor}_z(s_1, s_2) \), if \( d_i = c_i \cap z_i \) for all \( i \in \{1 \ldots k\} \).

The following lemma says that, given a threshold function \( z \), two states in \( A_z \) are related by \( \frac{a}{a}_z \) if and only if there exist two \( z \)-corresponding states in \( A_\infty(T) \), related by \( \frac{a}{a}_\infty(T) \), for some \( T \) of size at least \( B + \max\{z_1, 1\} \).

**Lemma 6.5.3.** Let \( z \) be a function from \( \{1 \ldots k\} \) to \( \mathbb{N} \). Let \( d, d' \) be two states in \( A_z \). Then for all events \( a \) in \( \Sigma_z^* \) we have that

\[
d \xrightarrow{a}_z d' \iff \exists T \cdot \#T \geq B + \max\{z_1, 1\} \land \exists c, c' \in A_\infty(T) \cdot \text{cor}_z(c, d) \land \text{cor}_z(c', d') \land c \xrightarrow{a}_\infty(T) c'.
\]

**Proof:** Similar to the proof of Lemma 3.3.6.  

**Corollary 6.5.4.** Let \( z \) be a function from \( \{1 \ldots k\} \) 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 \( \Sigma_z^* = \Sigma_\infty^*(T) \), if

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

then

\[
(c_1 \cap z_1, \ldots, c_k \cap z_k) \xrightarrow{a}_z (c'_1 \cap z_1, \ldots, c'_k \cap z_k).
\]

**Proposition 6.5.5.** Let \( z \) be a function from \( \{1 \ldots k\} \) to \( \mathbb{N} \). Let \( T \) be an instantiation of type \( t \) of size at least \( B + \max\{z_1, 1\} \). Then

\[
\zeta_z(N_B(\{0 \ldots B + m\})) \subseteq_T \zeta^T_{\infty}(N_B(\{0 \ldots B + m\})).
\]

**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

\[
(#T - B, 0, \ldots, 0) \xrightarrow{tr} (c'_1, \ldots, c'_k),
\]

then

\[
(#T - B) \cap z_1, 0, \ldots, 0 \xrightarrow{tr} (c'_1 \cap z_1, \ldots, c'_k \cap z_k).
\]

Observe that \( (#T - B, 0, \ldots, 0) \) is the initial state of \( \zeta^T_{\infty}(N_B(\{0 \ldots B + m\})) \). In addition, we assumed that \( #T \geq B + \max\{z_1, 1\} \), so \( (#T - B) \cap z_1 = z_1 \). However, \( (z_1, 0, \ldots, 0) \) is the initial state of \( \zeta_z(N_B(\{0 \ldots B + m\})) \). Therefore, \( tr \) is in traces(\( \zeta_z(N_B(\{0 \ldots B + m\})) \)), which implies the result.  

**Corollary 6.5.6.** Let \( z \) be a function from \( \{1 \ldots k\} \) 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

\[
ABS^B_n \subseteq_T ABS^T_{\infty,B,n}.
\]
Proof: Using the definitions of $\text{ABS}^T_{∞, B, 0}$ (p. 150) and $\text{ABS}^B_{∞, 0}$ (p. 189), as well as $\text{ABS}^T_{∞, B, m}$ (p. 150) and $\text{ABS}^B_{m}$ (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 $\{1 \ldots k\}$ to $\mathbb{N}$. Let $T$ be an instantiation of type $t$ of size at least $B + \max\{z_1, 1\}$. Then

(i) if $N_{myId}(t)$ satisfies $\text{NoEq}^T_t$, then

$$\text{ABS}^B_{z} \subseteq_T \phi(\text{Nodes}(T))$$

(ii) if $W_N \geq 1$, and $\beta$- and $\gamma$-events contain no payloads with node identities, then

$$\text{ABS}^B_{z, W_N} \subseteq_T \phi(\text{Nodes}(T)).$$

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 $\text{Spec}(\phi(T)) \subseteq_T \phi(\text{Nodes}(T))$. However, in Section 6.1 we allowed the implementations under our consideration to have node processes running in contexts of the form

$$C_X(t), Y(t), Z(t)[\cdot] = \left( (\cdot \setminus X(t)) \parallel Y(t) \right) \setminus Z(t)$$

for some polymorphic definitions of sets of events $X(t)$, $Y(t)$ and $Z(t)$ and where $Ctrl(t)$ is data independent and satisfies $\text{Pos ConjEq}^T_T$. Let

$$\text{Impl}(T) = C_X(t), Y(t), Z(t)[\text{Nodes}(T)].$$

Hence, we would like to be able to prove problems of the form

$$\text{Spec}(T) \subseteq_T \text{Impl}(T)$$

(6.83)

for all sufficiently large $T$.

In order to make the refinement check independent of $T$, we abstract the context and look at

$$C_X(\phi(T)), Y(\phi(T)), Z(\phi(T))[\text{ABS}^B_{z,n}].$$

We can then use a model checker to verify that the above traces-refines $\text{Spec}(\phi(T))$, which, thanks to Theorem 6.5.7, tells us that

$$\text{Spec}(\phi(T)) \subseteq_T C_X(\phi(T)), Y(\phi(T)), Z(\phi(T))[\phi(\text{Nodes}(T))].$$
To complete the proof we will need that
\[ C_{X(\phi(T)),Y(\phi(T)),Z(\phi(T))}[\phi(P(T))] \sqsubseteq_T \phi(C_{X(T),Y(T),Z(T)}[P(T)]) \quad (6.84) \]
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
\( \text{dom}(f) = T \),
\[ C_{X(f(T)),Y(f(T)),Z(f(T))}[f(P(T))] \sqsubseteq_T f(C_{X(T),Y(T),Z(T)}[P(T)]) \quad (6.85) \]
We proceed by showing corresponding results for parallel composition and hiding.

Firstly, given types \( X_1(t), \ldots, X_n(t) \), for each \( i \in \{1 \ldots n\} \) 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 \( \{1 \ldots n\} \), \( x_i \) 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 \( \text{dom}(f) = T \).

**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 \( \{1 \ldots n\} \), either \( X_i(t) = t \) or \( X_i(t) \) is not related to \( t \) in any way:
\[
X(f(T)) = \{c.y_1 \ldots y_n \mid y_1 \in f_1(X_1(T)), \ldots, y_n \in f_n(X_n(T))\} = \{c.f_1(x_1) \ldots f_n(x_n) \mid x_1 \in X_1(T), \ldots, x_n \in X_n(T)\} = \{f(c.x_1 \ldots x_n) \mid x_1 \in X_1(T), \ldots, x_n \in X_n(T)\} = f(X(T)).
\]
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 \( \text{PosConjEqT}^C_1 \).
Let \( T \) be an instantiation of type \( t \) and let \( f \) be a function such that \( \text{dom}(f) = T \). Then
\[ P(f(T)) \equiv_T f(P(T)). \]
We would like to show that distributing the application of every renaming function \( f \) with \( \text{dom}(f) = T \) over generalised parallel results in a traces anti-refinement, i.e. that
\[
\begin{align*}
\text{for all processes } P \text{ and } Q. \text{ Unfortunately, this is not true in general. To demonstrate this, let } P = Q = c?x.t \rightarrow \text{STOP}, X = \{c.1\}, T = \{1,2\} \text{ and } f(1) = f(2) = 1. \text{ Then } \langle c.1, c.1 \rangle \text{traces}(f(P ∥ X Q)), \text{ but } \langle c.1, c.1 \rangle \notin \text{traces}(f(P ∥ f(X) Q)). \text{ The reason why the refinement above does not hold is that the application of } f \text{ created new synchronisations: the processes } P \text{ and } Q \text{ do not synchronise on } c.2, \text{ but } f(P) \text{ and } f(Q) \text{ synchronise on } f(c.2) = c.1.
\end{align*}
\]
We therefore introduce a condition, which we call \text{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 \text{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 \text{NoNewSynx}_X is satisfied.

**Lemma 6.6.4.** Suppose that \( f \) satisfies \text{NoNewSync}_X. Then
\[
\begin{align*}
f(P) \parallel_{f(X)} f(Q) \subseteq_T f(P ∥ Q) \quad (6.86)
\end{align*}
\]
Proof: We can show that if \( f(P ∥ Q) \xrightarrow{a} f(P' ∥ Q') \), then \( f(P) \parallel_{f(X)} f(Q) \xrightarrow{a} f(P') \parallel_{f(X)} f(Q') \); this is proved by a simple case analysis on the type of event that \( a \) can be and its membership in \( \text{initials}(P) \) and \( \text{initials}(Q) \). The result then follows by an induction on the length of an arbitrary trace of \( f(P ∥ X Q) \).

A similar but stronger result (with equality instead of refinement) is true for hiding.

**Lemma 6.6.5.** Suppose \( f \) satisfies \text{NoNewSync}_X. Then
\[
\begin{align*}
f(P) \setminus f(X) \equiv_T f(P \setminus X)
\end{align*}
\]
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 \( \text{dom}(f) = T \) satisfies \text{NoNewSync}_{X(T)}.
The result for arbitrary polymorphic set definitions then follows from distributivity of 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 \( \text{dom}(f) = T \). Then

\[
C_{X(f(T)), Y(f(T)), Z(f(T))}[f(P(T))] \sqsubseteq_T f(C_{X(T), Y(T), Z(T)}[P(T)])
\]

for all process syntaxes \( P(t) \).

**Proof:** Let \( P(t) \) be given. From Lemma 6.6.6 we have that \( f \) satisfies \( \text{NoNewSync}_{X(T)} \), \( \text{NoNewSync}_{Y(T)} \) and \( \text{NoNewSync}_{Z(T)} \). Then

\[
C_{X(f(T)), Y(f(T)), Z(f(T))}[f(P(T))] = \begin{cases}
\text{by the definition of } C_{X(f(T)), Y(f(T)), Z(f(T))}[:]
\left( (f(P(T)) \setminus X(f(T))) \parallel_{Y(f(T))} \text{Ctrl}(f(T)) \right) \setminus f(Z(T)) \\
\text{by Lemma 6.6.1}
\left( (f(P(T)) \setminus f(X(T))) \parallel_{f(Y(T))} \text{Ctrl}(f(T)) \right) \setminus f(Z(T)) \\
\text{by Lemma 6.6.2, since } \text{Ctrl}(t) \text{ satisfies } \text{PosConjEqT}_T \text{ (see Section 6.1)}
\left( (f(P(T)) \setminus f(X(T))) \parallel_{f(Y(T))} f(\text{Ctrl}(T)) \right) \setminus f(Z(T)) \\
\text{by Lemma 6.6.5}
\left( f(P(T) \setminus X(T)) \parallel_{f(Y(T))} f(\text{Ctrl}(T)) \right) \setminus f(Z(T)) \\
\text{by Lemma 6.6.4}
\left( f(P(T) \setminus X(T)) \parallel_{Y(T)} \text{Ctrl}(T) \right) \setminus f(Z(T))
\end{cases}
\]
\( \sqsubseteq_T \{ \text{by Lemma 6.6.5} \} \)
\[
f\left( \left( (P(T) \setminus X(T)) \parallel Ctrl(T) \right) \setminus Z(T) \right)_{Y(T)}
= \{ \text{by the definition of } C_{X(t), Y(t), Z(t)}[\cdot] \}
\]
\[
f(C_{X(T), Y(T), Z}(T)[P(T)])
\]

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 \( Spec(\phi(T)), C_{X(\phi(T)), Y(\phi(T)), Z(\phi(T))}[ABS_{T,0}] \) and \( C_{X(\phi(T)), Y(\phi(T)), Z(\phi(T))}[ABS_{T,WN}] \) are all independent of \( T \), it allows us to perform a single refinement check in order to deduce the truth of the refinement \( Spec(T) \sqsubseteq_T Impl(T) \) for infinitely many types \( T \).

**Theorem 6.6.8.** Suppose that

(i) \( Spec(t) \) satisfies \( SeqNorm \) and \( RevPosConjEqT^t \),

(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) \( B \geq \max\{ \#^t(\alpha) \mid \alpha \text{ is a construct of } Spec(t) \} \), if \( Spec(t) \) satisfies \( NoEqT^t \)
\[
\max\{ \#^t(\sigma, \epsilon)(Spec(t)) \mid \sigma \hat{\epsilon} \in SymbolicTraces(Spec(t)) \}, \text{ otherwise},
\]

(iv) \( z \) is a function from \( \{1 \ldots k\} \) to \( N \),

(v) \( T \) is an instantiation of type \( t \) of size at least \( B + \max\{z_1, 1\} \), and

(vi) \( \phi \) is a \( B \)-collapsing function.

Then

(a) if \( N_{myId}(t) \) satisfies \( NoEqT^t \) and

\[
Spec(\{0 \ldots B\}) \sqsubseteq_T C_{X(\{0 \ldots B\}), Y(\{0 \ldots B\}), Z(\{0 \ldots B\})} [ABS_{T,0}^B],
\]
then

\[
Spec(T) \sqsubseteq_T Impl(T),
\]

(b) if \( N_{myId}(t) \) can remember at least one identity (in addition to its own), \( \beta \)- and \( \gamma \)-events contain no payloads with node identities, and

\[
Spec(\{0 \ldots B\}) \sqsubseteq_T C_{X(\{0 \ldots B\}), Y(\{0 \ldots B\}), Z(\{0 \ldots B\})} [ABS_{T,WN}^B],
\]
then

\[
Spec(T) \sqsubseteq_T Impl(T).
\]
**Proof:** Let $n$ be one of $0$ (for case (a)) or $W_N$ (for case (b)). Since $\# T \geq B + 1$, we have that $\phi(T) = \{0 \ldots B\}$. So

\[
\begin{align*}
\text{Spec}(\phi(T)) & \sqsubseteq_T \{\text{by assumption}\} \ C_X(\phi(T), Y(\phi(T)), Z(\phi(T))) [ABS^B,n] \\
\text{Spec}(\phi(T)) & \sqsubseteq_T \{\text{by Theorem 6.5.7 using conditions (iv), (v) and (vi)}\} \ C_X(\phi(T), Y(\phi(T)), Z(\phi(T))) [\phi(\text{Nodes}(T))] \\
& = \{\text{by Proposition 6.6.7}\} \ \phi(C_X(T), Y(T), Z(T) [\text{Nodes}(T)]) \\
& = \phi(\text{Impl}(T)).
\end{align*}
\]

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 Spec(t) does not satisfy NoEqT) and Corollary 5.2.16 (if it does) that

\[
\text{Spec}(T) \sqsubseteq_T \text{Impl}(T),
\]

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 $ABS_T^{B,n}$ 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 NoEqT), $ABS_T^{B,0}$ forms traces anti-refinements of $\phi(\text{Nodes}(T))$ for sufficiently large $T$. An analogous result was proved for $ABS_T^{B,0}$ when dealing with processes that might contain such tests. We noted that even though the alphabets of $ABS_T^{B,0}$ and $ABS_T^{B,W_N}$ 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 $ABS^B,0$ (for processes that do not contain equality or inequality tests on $t$) and $ABS^B,W_N$ (for processes that might do), independent of $T$. We proved that, for sufficiently large $T$, the abstract models form traces anti-refinements of $\phi(\text{Nodes}(T))$.  

201
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 $N_{myId}(t)$, 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 $W_N$ for processes that contain such tests), the LTSs of nodes $N_B(\{0 \ldots B + m\}), \ldots, N_{B+m}(\{0 \ldots B + m\})$ need to be obtained.
- A threshold function $z$ is chosen.
- The techniques from Section 6.5 are used to construct the counter state machine $\zeta_z(N_B(\{0 \ldots B + m\}))$.
- The counter state machine is defined as a CSP process and combined with a parallel composition of explicitly modelled nodes with identities in $\{0 \ldots B - 1\}$ (with awareness of node identities from a collapsed type) to give a CSP definition of the abstract model $ABS_z^{B,0}$ or $ABS_z^{B,W_N}$.
- 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 $Spec(\{0 \ldots B\})$.
- 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).
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 5. 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\(^{1}\)Available from [http://www.comlab.ox.ac.uk/tomasz.mazur/tools/TomCAT](http://www.comlab.ox.ac.uk/tomasz.mazur/tools/TomCAT).

2\(^{2}\)For a given specification/implemention pair, the calculation of \(B\) and \(m\) is fully automatable, but this has not been implemented in the current version of TomCAT.

3\(^{3}\)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 be 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 \textit{input} and deposits it into the slot using channel \textit{put} (see Figure 7.1(a)). Every node can retrieve data from the slot using channel \textit{get} and pass it to the environment using channel \textit{output}.

In our system we use three binary semaphores:

- \textit{EmptySlotSemaphore} is used to wake up the producer whenever the slot is ready to accept data.
- \textit{FullSlotSemaphore} is used to wake up a node whenever the slot contains a piece of data.
- \textit{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{equation}
\text{EmptySlotSemaphore}(\text{state}) = \\
state = \text{up} & \& \text{emptySlotDown} \rightarrow \text{EmptySlotSemaphore}(\text{down}) \\
\square \text{emptySlotUp} \rightarrow \text{EmptySlotSemaphore}(\text{up})
\end{equation}

$^4$Our use of the \textit{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.

204
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.

\[
\text{FullSlotSemaphore}(state, t) = \\
state = \text{up} \land \text{fullSlotDown} ? i : t \rightarrow \text{FullSlotSemaphore}(\text{down}, t) \\
\text{fullSlotUp} \rightarrow \text{FullSlotSemaphore}(\text{up}, t)
\]

\[
\text{Mutex}(state, t) = \\
state = \text{up} \land \text{mutexDown} ? i : t \rightarrow \text{Mutex}(\text{down}, t) \\
\text{mutexUp} ? i : t \rightarrow \text{Mutex}(\text{up}, t)
\]

The FullSlotSemaphore and Mutex semaphores use events with parameters in order to allow single nodes to synchronise on them.

Let \( Data = \{0, 1\} \) 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.

\[
Producer = \text{input} ? d : Data \rightarrow \text{emptySlotDown} \rightarrow \text{put}.d \rightarrow Producer
\]
We model a node process in a slightly contrived way in order to obtain alphabets that contain \(\alpha\)-, \(\beta\)- and \(\gamma\)-events. All nodes (consumers) are first simultaneously initialised (this introduces a \(\gamma\)-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 \(\beta\)-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.

\[
\begin{align*}
\text{Node}(i, t) &= \text{init} \rightarrow \text{Node}'(i, t) \\
\text{Node}'(i, t) &= \text{mutexDown}.i \rightarrow \text{Node}''(i, t) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \text{exchangeMutex}?.j!:i \rightarrow \text{Node}''(i, t) \\
\text{Node}''(i, t) &= \text{fullSlotDown}.i \rightarrow \text{get}.?d:\text{Data} \rightarrow \text{output}.i.d \rightarrow \text{Node}'''(i, t) \\
\text{Node}'''(i, t) &= \text{exchangeMutex}!.i?.j:t \rightarrow \text{Node}'(i, t) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{mutexUp}.i \rightarrow \text{Node}'(i, t)
\end{align*}
\]

We let \(\text{Nodes}(t)\) model the parallel composition of all node processes.

\[
\text{Nodes}(t) = \bigparallel i \in t \mid A(i, t) \bigparallel \text{Node}(i, t)
\]

\[
A(i, t) = \{\text{init, mutexUp}.i, \text{mutexDown}.i, \text{fullSlotDown}.i\} \\
\quad \cup \{\text{get}.i.d, \text{output}.i.d \mid d \in \text{Data}\} \\
\quad \cup \{\text{exchangeMutex}.i.j, \text{exchangeMutex}.j.i \mid j \in t \setminus \{i\}\}
\]

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.

\[
\text{EmptySlot}(t) = \text{emptySlotUp} \rightarrow \text{put}?d:\text{Data} \rightarrow \text{FullSlot}(d, t)
\]
\[
FullSlot(d, t) = fullSlotUp \rightarrow \text{get?i:t!d} \rightarrow EmptySlot(t)
\]

To respect the nomenclature we used in Section 6.6, we let \( \text{Ctrl}(t) \) model the part of our implementation not including the node processes.

\[
\text{Ctrl}(t) = \text{Mutex}(up, t) \parallel \\
\left( \text{Producer} \parallel \text{EmptySlot}(t) \parallel \alpha_{\text{Sem}} \right) \\
\left( \text{EmptySlotSemaphore(down)} \parallel \text{FullSlotSemaphore(down, t)} \right)
\]

\( \alpha_{\text{Sem}} = \{\text{emptySlotDown}, \text{emptySlotUp}, \text{fullSlotUp}\} \)

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

\[
\text{Impl}(t) = C_{X(t), Y(t), Z(t)}[\text{Nodes}(T)]
\]

where

\[
C_{X(t), Y(t), Z(t)}[\cdot] = \left( \left( \cdot \setminus X(t) \right) \parallel \text{Ctrl}(t) \right) \setminus Z(t)
\]

\[
X(t) = \{\text{init}\}
\]

\[
Y(t) = \{\floater{mutexUp, mutexDown, fullSlotDown, get}\}
\]

\[
Z(t) = \{\floater{mutexUp, mutexDown, exchangeMutex, fullSlotUp, fullSlotDown, emptySlotUp, emptySlotDown, put, get}\}.
\]

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.

\[
\text{Buff}(\langle \rangle, t) = \text{input?d:Data} \rightarrow \text{Buff}(\langle d \rangle, t)
\]

\[
\text{Buff}(\langle d \rangle^* \text{seq}, t) = \begin{cases} \\
\text{output?i:t!d} \rightarrow \text{Buff}(\text{seq}, t) \\
\text{length(seq) < 2} & \text{input?d:Data} \rightarrow \text{Buff}(\langle d \rangle^* \text{seq}^* \langle d' \rangle, t)
\end{cases}
\]

Using the above, we let our specification be the following.

\[
\text{Spec}(t) = \text{Buff}(\langle \rangle, t)
\]

Then, our verification becomes: given type \( \text{Data} \), is it true that

\[
\text{Spec}(T) \sqsubseteq_T \text{Impl}(T)
\]

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

$$\max\{\#!^i(\alpha) \mid \alpha \text{ is a construct of } Spec(t)\} = 0,$$

so we let $B = 0$ and we let $\phi$ 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 $\phi$ and a definition of a threshold function.

The output script contains the following definition, where $x$ is a node identity:

$$Transitions(x) =$$

if $x = 0$ then

$$\{(0, \text{init}, 1), (1, \text{exchangeMutex}.0.0.2), (1, \text{exchangeMutex}.1.0.2), (1, \text{mutexDown}.0.2), (2, \text{fullSlotDown}.0.3), (3, \text{get}.0.1.4), (3, \text{get}.0.0.5), (4, \text{output}.0.1.6), (5, \text{output}.0.0.6), (6, \text{exchangeMutex}.0.0.6), (6, \text{mutexUp}.0.1), (6, \text{exchangeMutex}.1.0.1)\}$$

else if $x = 1$ then

$$\{(7, \text{init}, 8), (8, \text{exchangeMutex}.1.1.9), (8, \text{exchangeMutex}.0.1.9), (8, \text{mutexDown}.1.9), (9, \text{fullSlotDown}.1.10), (10, \text{get}.1.1.11), (10, \text{get}.1.0.12), (11, \text{output}.1.1.13), (12, \text{output}.1.0.13), (13, \text{exchangeMutex}.1.1.13), (13, \text{exchangeMutex}.1.0.8), (13, \text{mutexUp}.1.8)\}$$

else if $x = 2$ then ... else {}.

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\(^5\):

$$E = \{\{0, 7\}, \{1, 8\}, \{2, 9\}, \{3, 10\}, \{4, 11\}, \{5, 12\}, \{6, 13\}\}.$$

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:

$$Threshold = (1, 1, 1, 1, 0, 1, 1),$$

\(^5\)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.
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 \( \text{Spec}(\{0..B\}) \subseteq_T C_{\{0..B\},\{0..B\},\{0..B\}} [ABS_z^{B,0}] \), with the following counterexample returned by FDR:

\[
tr = \langle \text{output}.0.1 \rangle.
\]

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 \( \text{output}.0.1 \) is available. So \( \text{output}.0.1 \) 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

\[
\text{Threshold} = \langle 1, 1, 1, 1, 1, 1, 1 \rangle,
\]

then we still have that \( \text{Spec}(\{0..B\}) \not\subseteq_T C_{\{0..B\},\{0..B\},\{0..B\}} [ABS_z^{B,0}] \), with the following counterexample:

\[
tr = \langle \text{input}.1, \text{output}.1.1, \text{output}.1.1 \rangle.
\]

This corresponds to the following trace that is performed by the abstract model without the hiding of events.

\[
tr' = \langle \text{init}, \text{emptySlotUp}, \text{input}.1, \text{emptySlotDown}, \text{put}.1, \text{fullSlotUp}, \text{mutexDown}.1, \text{fullSlotDown}.1, \text{get}.1.1, \text{output}.1.1, \text{output}.1.1 \rangle
\]

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 \( \text{Node} \) or \( \text{Node}' \)), 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:

$$Threshold = (0,2,2,2,2,2,2),$$

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 or more\} domain. This time FDR confirms that the refinement

$$Spec(\{0..B\}) \subseteq_T C_{\{0..B\},\{0..B\}} [ABS_{z}^{B,0}]$$

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 $Spec(T) \subseteq_T Impl(T)$ 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 $Data = \{0, 1\}$ 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 $Spec(t)$ and $Impl(t)$ satisfy $\text{NoEqT}^t$ (with respect to the $Data$ type) and $Spec(t)$ satisfies the $\text{Norm}$ 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].
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 $\phi$ 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 $\hat{T} \subseteq T$,

$$Spec(\hat{T}) \sqsubseteq \phi(Impl(T)) \text{ implies that } Spec(T) \sqsubseteq Impl(T) \quad (8.1)$$

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 $\phi(\text{Impl}(T))$ 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 $\text{Abstr}$ such that for all sufficiently large $T$,

$$Abstr \subseteq \phi(\text{Impl}(T)).$$

Then, by testing that $\text{Spec}(\hat{T}) \subseteq \text{Abstr}$ we can deduce from transitivity of refinement and (8.1) that $\text{Spec}(T) \subseteq \text{Impl}(T)$ 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 $\text{Abstr}_z$, based on a threshold function $z$, satisfy

$$\text{Abstr}_z \subseteq_\text{T} \text{Impl}(T) \quad \text{and} \quad \text{Abstr}_z \subseteq_\text{F} \text{Impl}(T)$$

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 $\text{Spec} \subseteq \text{Abstr}_z$ is enough to deduce that $\text{Spec}(T) \subseteq \text{Impl}(T)$ 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 $\phi$. We showed how to build abstract models, $\text{Abstr}_z$, such that (8.2) holds. This, combined with our type reduction theory allows us to deduce that if a single refinement check (namely, $\text{Spec}(\hat{T}) \subseteq \text{Abstr}_z$) 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 $Spec \sqsubseteq_F Abstr$ and deduces that $Spec \setminus X \sqsubseteq_{FD} Abstr \setminus X$ (where $\sqsubseteq_{FD}$ 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 $\zeta(z(N))$. 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:

\[
\text{Counter}_i(\text{value}) = \langle \text{counter definition as before} \rangle \\
\text{counter}.i.v \rightarrow \text{Counter}_i(\text{value}),
\]

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

\[
\text{Spec} = \text{Chaos}(\{| \text{counter}.1, \text{counter}.3 \} \cup \{ \text{counter}.2.i \mid i \in \{0..2\}\}),
\]

if used in a traces refinement check, tests if always at most two node processes are in the second state. Similarly,

\[
\text{Spec} = \text{Chaos}(\{| \text{counter}.1, \text{counter}.3 \} \cup \{ a, \text{counter}.2.0, \text{counter}.2.1 \})
\]

\[
\text{c2.2} \rightarrow a \rightarrow \text{Spec},
\]

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.

\[
\text{Spec}' = \text{Chaos}(\Sigma \setminus (\text{Moveto}_2 \cup \text{Movefrom}_2)) \Box ?a: \text{Moveto}_2 \rightarrow \text{Spec}'
\]

\[
\text{Spec}' = \text{Chaos}(\Sigma \setminus (\text{Moveto}_2 \cup \text{Movefrom}_2))
\]

\[
\Box ?a: \text{Moveto}_2 \rightarrow \text{Spec} \Box ?a: \text{Moveto}_2 \rightarrow \text{Spec}''
\]

\[
\text{Spec}'' = \text{Chaos}(\Sigma \setminus (\text{Moveto}_2 \cup \text{Movefrom}_2)) \Box ?a: \text{Movefrom}_2 \rightarrow \text{Spec}'
\]

where Moveto2 and Movefrom2 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, $\triangledown$.

The process $P \triangledown Q$ 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]):

$$P \triangledown Q = \left( (P[\mathcal{R}_P] \parallel Q[\mathcal{R}_Q]) \parallel \mathcal{R} \parallel \mathcal{R} \right)_{\Sigma(P) \cup \Sigma(Q)}, \quad (8.4)$$

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

$$\mathcal{R} = p?x: \Sigma(P) \to \mathcal{R} \parallel q?x: \Sigma(Q) \to \text{Run}(|\{q\}|).$$

Here, $P$ is allowed to perform all of its communication until $Q$ performs one of its initial events, at which point $\mathcal{R}$ 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 $\triangledown$.

The throws operator, $\Theta_A$.

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 $\Theta_A$ using $\triangledown$ (see above) and operators of the CSP language we used in this thesis (this construction is based on a similar one in [Ros08c]):

$$P \Theta_A Q = \left( (P[\mathcal{R}_P] \triangledown c \to Q[\mathcal{R}_Q]) \parallel \mathcal{R} \parallel \mathcal{R} \right)_{\Sigma(P)} \setminus \{c\},$$

where $\mathcal{R}_P$, $\mathcal{R}_Q$ and $\mathcal{R}$ are as in the construction for $\triangledown$, above, and

$$\mathcal{R} = p?x:(\Sigma(P) \setminus A) \to \mathcal{R} \parallel p?x:A \to c \to \text{STOP}.$$

\(^1\)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 $\Theta_A$ 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, $\checkmark$. The process $P; Q$ behaves like $P$ until it successfully terminates (i.e. performs $\checkmark$), 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 $\checkmark$ 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 $\checkmark$ 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 $\tau$’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

$$N_{myId}(t) = in!myId?i.t \rightarrow out.i \rightarrow STOP.$$  

Suppose that $B$ is some fixed non-negative integer. Let $\phi$ 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 $\gamma$-event (see Section 6.1). We then have that

$$(\langle in.i.B \mid i \in \{0..B-1\}\rangle \hat{\langle in.i.i \mid i \in \{B..\#T-1\}\rangle}, \Sigma) \in failures(Nodes(T)),$$

so

$$(\langle in.i.B \mid i \in \{0..B-1\}\rangle \hat{\langle in.B.B \rangle}^{T-B}, \phi(\Sigma)) \in failures(\phi(Nodes(T))),$$

216
where \( \text{Nodes}(t) \) is defined as in Section 6.1. However, after performing the trace
\[
\text{tr} = \langle \text{in}.i.B \mid i \in \{0 \ldots B-1\} \rangle \cdot (\text{in}.B.B)^{\#T-B},
\]
the counter that corresponds to the equivalence class \([\langle \text{out}.i \to \text{STOP}, \{\text{myId} \to B, i \to B\}, T \rangle] \) within the counter state machine \( \zeta_T^T(N_B(\{0 \ldots B+1\})) \) must be equal to \( \#T-B \) and all other counters must be 0. Hence, by the definition of the abstract transition relation, (6.7), \( \zeta_T^T(N_B(\{0 \ldots B+1\})) \) cannot refuse \text{out}.B after performing \text{tr}. Hence,
\[
(\text{tr}, \phi(\Sigma)) \notin \text{failures}(\text{ABS}_T^T,B,0).
\]

End of example.

The above example implies that our abstraction techniques do not, in general, guarantee that \( \text{ABS}_T^T,B,n \subseteq_F \phi(\text{Impl}(T)) \) for \( n = 0 \) and \( n = W_N \) (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 \( \phi \) to be a \((B + \text{max}\{n,1\})\)-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, \( \text{Nodes}(T) \), and their counter abstracted models without thresholds, \( \zeta_T^T(N) \) (Proposition 3.2.2). This implies that \( \zeta_T^T(N) \equiv_{FD} \text{Nodes}(T) \) (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, \( \zeta_z(N) \subseteq_{FD} \zeta_T^T(N) \). Since addition of thresholds cannot remove any divergences, it must be that \( \zeta_z(N) \subseteq_{FD} \zeta_T^T(N) \) and hence \( \zeta_z(N) \subseteq_{FD} \text{Nodes}(T) \) 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 \( \tau \) 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 \( \tau \)'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 interpret 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 $\zeta_z(N)$ is an exact abstraction, i.e. such that

$$\forall T \cdot Spec \subseteq C \left[ \bigparallel_{A_{Sync}} i \in T \cdot N \right] \tag{8.5}$$

if and only if

$$Spec \subseteq \zeta_z(N). \tag{8.6}$$

Theorem 3.1.7 showed that verifying if (8.5) holds is undecidable. However, if we assume that $Spec$ and $N$ 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 $\left( (\cdot \setminus X(t)) \parallel Ctrl(t) \right) \setminus Z(t)$. 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 \texttt{SeqNorm} condition (see Section 4.1.2). Therefore, for brevity, we provided inference rules only for those operators that \texttt{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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{N} )</td>
<td>natural numbers</td>
</tr>
<tr>
<td>( \mathbb{Z} )</td>
<td>integers</td>
</tr>
<tr>
<td>( { } )</td>
<td>empty set</td>
</tr>
<tr>
<td>#( s )</td>
<td>length of ( s ) (for sequences) or size of ( s ) (for sets)</td>
</tr>
<tr>
<td>( s \circ s' )</td>
<td>concatenation of sequences ( s ) and ( s' )</td>
</tr>
<tr>
<td>( \text{dom}(f) )</td>
<td>domain of ( f )</td>
</tr>
<tr>
<td>( \text{Im}(f) )</td>
<td>image of ( f )</td>
</tr>
<tr>
<td>( { } )</td>
<td>bag</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>partial function</td>
</tr>
<tr>
<td>( \triangleleft )</td>
<td>domain restriction</td>
</tr>
<tr>
<td>( \psi )</td>
<td>( B )-collapsing function</td>
</tr>
<tr>
<td>( \phi )</td>
<td>( K )-precollapsing function</td>
</tr>
<tr>
<td>$!^\epsilon, ?^{non-\epsilon}, !^\epsilon, \ldots $</td>
<td>index sets of inputs and outputs of construct ( \epsilon )</td>
</tr>
<tr>
<td>( \sqcap )</td>
<td>binary minimum operator</td>
</tr>
<tr>
<td>( \equiv_{\alpha} )</td>
<td>alpha equivalence</td>
</tr>
<tr>
<td>( \sim )</td>
<td>node control state equivalence relation</td>
</tr>
<tr>
<td>( \approx_{\phi} )</td>
<td>node state equivalence relation</td>
</tr>
<tr>
<td>( \text{Value} )</td>
<td>set of all values</td>
</tr>
<tr>
<td>( \text{Var} )</td>
<td>set of all variables names</td>
</tr>
<tr>
<td>( FV(P) )</td>
<td>set of free variables of ( P )</td>
</tr>
<tr>
<td>( \text{last}(seq) )</td>
<td>last element of non-empty sequence ( seq )</td>
</tr>
</tbody>
</table>

## Alphabets

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma )</td>
<td>default alphabet</td>
</tr>
<tr>
<td>( \Sigma(P) )</td>
<td>alphabet of process ( P )</td>
</tr>
<tr>
<td>( \Sigma^\tau )</td>
<td>alphabet extended with ( \tau )</td>
</tr>
<tr>
<td>( \Sigma^* )</td>
<td>finite sequences with symbols from ( \Sigma )</td>
</tr>
</tbody>
</table>
Abstract models
\[ \zeta_T^\infty(N), \zeta_T^\infty(N_B(\{0\ldots m\})) \] counter abstraction without thresholds ................. 42
\[ \zeta_\infty(N), \zeta_\infty(N_B(\{0\ldots m\})) \] counter abstraction with thresholds ..................... 45
\[ ABS_T^{\infty, B, n} \] abstraction without thresholds (Impl with node IDs) 150
\[ ABS_B^{\infty, B, n} \] abstraction with thresholds (Impl with node IDs) 189

Operational semantics
- single event transition ........................................ 67
- single event transition of node with identity \( i \) ........ 147
- multiple event transition .................................... 75
- multiple visible event transition .............................. 75
- optional transition ............................................ 86
- set of nodes of LTS \( \mathcal{L} \) ................................. 34

Symbolic operational semantics
- single symbolic event transition ............................. 76
- multiple symbolic event transition .......................... 81
Visible set of visible symbolic events ........................... 77
SymbolicTraces(\( P \)) set of symbolic traces of \( P \) .................... 82
\( \equiv_{non-r} \) non-\( r \) equivalence of symbolic traces .......... 95
\( \equiv_{non-t} \) non-t equivalence of symbolic events and traces ...... 98
generates relation between symbolic and concrete traces ....... 87
Insts(\( \epsilon \)) concrete instantiations of \( \epsilon \) within \( \Gamma \) ............ 88
Match(\( \epsilon, e \)) matching of variables of \( \epsilon \) and values of \( e \) .......... 88

Denotational semantics
- traces(\( P \)) traces of process \( P \) .............................. 9
- failures(\( P \)) stable failures of process \( P \) ...................... 10
- refinement .................................................. 9
- traces refinement .......................................... 9
- stable failures refinement .................................. 10
- traces equivalence .......................................... 10
- stable failures equivalence ................................ 10
- process \( P \) refuses set \( X \) ............................... 10
- initially available events of process \( P \) .................... 10
- process \( P \) after trace \( tr \) .................................. 10
- restriction of \( tr \) to events in set \( A \) ...................... 57
Conditions
SeqNorm sequential normality .................................. 63
TypeSym type symmetry ........................................... 108
PosConjEqT_M positive conjunction of equality tests .......... 110
RevPosConjEqT_M reversed positive conjunction of equality tests .... 110
NoEqT no equality tests ........................................... 111
NoNewSync_X no new synchronisations .............................. 198

Abbreviations
PMCP Parameterised Model Checking Problem ..................... 4
VAS vector addition system ........................................ 35
VASS vector addition system with states .......................... 36
LTS labelled transition system ..................................... 34
SSLTS semi-symbolic labelled transition system .................. 76
SSOS Semi-Symbolic Operational Semantics ....................... 76
COSE Concrete Operational Semantics with Environments .... 82

Miscellaneous
t distinguished type ............................................... 13
∀ x ∈ X | P(x) • Q(x) quantified implication ..................... 112
Replace^t_{s−1} replacement of t type nondet. inputs by outputs ... 66
Replace^{non-t}_{s−1} replacement of non-t type nondet. inputs by outputs . 66
Replace^{all}_{s−1} replacement of all nondeterministic inputs by outputs 66
P[v/x] substitution .................................................. 66
Env(t) all environments of type t .................................. 82
[[cond]]_Γ truth evaluation of condition within environment .... 86
W_N maximum number of remembered node identities ........ 145
type t values of e, matched by det. inputs of ϵ .......... 145
type t values of e, matched by outputs of ϵ ................. 171
test if the variable of ϵ that v_i matches is in dom(Γ) . 171


[cit10] Most cited computer science articles (generated from CiteSeerX), [http://citeseerx.ist.psu.edu/stats/articles](http://citeseerx.ist.psu.edu/stats/articles), February 2010.


M. Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag, 1996.


232


237


Appendix A

CSP definition of $\zeta_z(\mathcal{N})$ for Example 3.3.12

\[ CANodes'_z(c_1, c_2, c_3, c_4, c_5) = \]
\[ c_1 \geq 1 \& load \rightarrow CANodes'_z(0, 1, c_3, c_4, c_5) \]
\[ \square \]
\[ c_2 \geq 1 \& run \rightarrow (CANodes'_z(c_1, 0, (c_3 + 1) \sqcap 2, c_4, c_5)
\square CANodes'_z(c_1, 1, (c_3 + 1) \sqcap 2, c_4, c_5)) \]
\[ \square \]
\[ c_3 \geq 1 \& deschedule \rightarrow
\quad (if \ c_3 = 2 \ then \ CANodes'_z(c_1, (c_2 + 1) \sqcap 1, 1, c_4, c_5)
\quad \square CANodes'_z(c_1, (c_2 + 1) \sqcap 1, 2, c_4, c_5)
\quad else \ CANodes'_z(c_1, (c_2 + 1) \sqcap 1, 0, c_4, c_5)) \]
\[ \square \]
\[ c_3 \geq 1 \& block \rightarrow
\quad (if \ c_3 = 2 \ then \ CANodes'_z(c_1, c_2, 1, (c_4 + 1) \sqcap z_4, c_5)
\quad \square CANodes'_z(c_1, c_2, 2, (c_4 + 1) \sqcap z_4, c_5)
\quad else \ CANodes'_z(c_1, c_2, 0, (c_4 + 1) \sqcap z_4, c_5)) \]
\[ \square \]
\[ c_3 \geq 1 \& terminate \rightarrow
\quad (if \ c_3 = 2 \ then \ CANodes'_z(c_1, c_2, 1, c_4, (c_5 + 1) \sqcap 1)
\quad \square CANodes'_z(c_1, c_2, 2, c_4, (c_5 + 1) \sqcap 1)
\quad else \ CANodes'_z(c_1, c_2, 0, c_4, (c_5 + 1) \sqcap 1)) \]
\[ \square \]
\[ c_4 \geq 1 \& interrupt \rightarrow (CANodes'_z(c_1, (c_2 + 1) \sqcap 1, c_3, 0, c_5)
\quad \square CANodes'_z(c_1, (c_2 + 1) \sqcap 1, c_3, 1, c_5)) \]
Appendix B

The $\langle f[\cdot]\text{-ren-app}\rangle$ law

Given a renaming function $F$ and renaming relation $\mathcal{R}$, we define

$$\mathcal{R}[F] = \{(F(a), F(b)) \mid aRb\}. $$

Then we have the following result.

**Lemma B.0.2 ($\langle f[\cdot]\text{-ren-app}\rangle$).** 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 $b$

$$a(F \circ \mathcal{R})b \iff \{\text{definition of relation composition}\} \iff \exists c \cdot a\mathcal{R}c \land cFb$$

$$\iff \{F \text{ is a function}\} \iff \exists c \cdot a\mathcal{R}c \land b = F(c)$$

$$\iff \{\text{definition of } \mathcal{R}[F]\} \iff F(a)(\mathcal{R}[F])b$$

$$\iff \{F \text{ is a function}\} \iff \exists d \cdot aFd \land d(\mathcal{R}[F])b$$

$$\iff \{\text{definition of relation composition}\} \iff a((\mathcal{R}[F]) \circ F)b.$$  

This means that $F \circ \mathcal{R} = (\mathcal{R}[F]) \circ F$. Hence

$$F(P[\mathcal{R}]) = P[[F \circ \mathcal{R}]] = P[[\mathcal{R}[F]] \circ F] = (F(P))[[\mathcal{R}[F]]],$$

which is what we had to prove. $\blacksquare$
Appendix C

CSP scripts

C.1 Example 3.0.1

C.1.1 Source code

This script models a process scheduling mechanism in an operating system for a multiprocessor machine.

The implementation consists of a number of identical nodes, each representing a single process requesting CPU access, and a scheduler.

The specification says that there should never be more processes with CPU access than there are CPUs. Further, every process with CPU access cannot be denied giving up the access.

T = {0..4} — an example instantiation T of type t

t is not needed for TomCAT, but allows one to directly verify a single size of the implementation by executing the input script in FDR.

cores = 2 — the number of CPUs

datatype NodeStates = new | runnable | running | blocked
datatype CoreStates = idle | busy

channel load, run, block, terminate, deschedule, interrupt, stopRun

A single node process

Node = Node'(new)

Node'(state) = state == new & load -> Node'( runnable)

state == runnable & run -> Node'( running)

state == running & ( block -> Node'( blocked))
36 | terminate -> STOP
37 | deschedule -> Node'(runnable)
38 | state == blocked & interrupt -> Node'(runnable)

— Parallel composition of all node processes

40 Nodes = [||ASync] i:T @ Node

— The set of events on which all nodes have to synchronise

44 ASync = {load}

— A single CPU

48 Core(idle) = run -> Core(busy)
49 Core(busy) = deschedule -> Core(idle)
50 | | block -> Core(idle)
51 | | terminate -> Core(idle)

— The most nondeterministic scheduler

55 Scheduler = |||i:{0..(cores-1)} @ Core(idle)

— Composing the implementation

59 C(P) = (P [||{run,terminate,deschedule,block}||] Scheduler)
60 | | [[deschedule<->stopRun,block<->stopRun,terminate<->stopRun]]
61 Impl = C(Nodes)

— Defining the specification

65 Spec = Spec'(0)
66 Spec'(x) = x < cores & (run -> Spec'(x+1) |~| STOP)
67 x > 0 & stopRun -> Spec'(x-1)
68 | | (load -> Spec'(x) |~| interrupt -> Spec'(x) |~| STOP)

— An assertion for FDR

72 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 cores = 2
15
datatype NodeStates = new | runnable | running | blocked
16 datatype CoreStates = idle | busy
17 channel load, run, block, terminate, deschedule, interrupt, stopRun
18
— A single node process
19 Node = Node'(new)
20
21 Node'(state) = state == new & load -> Node'(runnable)
22      []
23     state == runnable & run -> Node'(running)
24      []
25     state == running & (block -> Node'(blocked))
26     []
27     terminate -> STOP
28     []
29     deschedule -> Node'(runnable))
30      []
31     state == blocked & interrupt -> Node'(runnable)
32
— Parallel composition of all node processes
33 Nodes = [||ASync||] i:T @ Node
34
— The set of events on which all nodes have to synchronise
35 ASync = {load}
36
— A single CPU
37 Core(idle) = run -> Core(busy)
38 Core(busy) = deschedule -> Core(idle)
```
block \rightarrow \text{Core(idle)}

terminate \rightarrow \text{Core(idle)}

--- The most nondeterministic scheduler

Scheduler = \{i : \{0..(\text{cores} - 1)\} @ \text{Core(idle)}\}

--- Composing the implementation

C(P) = (P \{\{\text{run}, \text{terminate}, \text{deschedule}, \text{block}\}\} \text{Scheduler})

\{\{\text{deschedule} \rightarrow \text{stopRun}, \text{block} \rightarrow \text{stopRun}, \text{terminate} \rightarrow \text{stopRun}\}\}

\text{Impl} = C(\text{Nodes})

--- Defining the specification

\text{Spec} = \text{Spec}'(0)

\text{Spec}'(x) = x < \text{cores} \& (\text{run} \rightarrow \text{Spec}'(x+1) \mid \text{STOP})

x > 0 \& \text{stopRun} \rightarrow \text{Spec}'(x-1)

\text{load} \rightarrow \text{Spec}'(x) \mid \text{interrupt} \rightarrow \text{Spec}'(x) \mid \text{STOP})

--- An assertion for FDR

assert \text{Spec} \ [F=\text{Impl}]

--- ### THRESHOLDS ###

--- Define a sequence of thresholds for the states of Node, e.g.

\text{Thresholds} = <2,0,3,0,1>

--- z function allows us to extract values of thresholds

\text{z(x)} =

let

\text{zsub}(0, <n>\text{`ns}) = n
\text{zsub}(x, <n>\text{`ns}) = \text{zsub}(x-1, ns)

within

\text{zsub}(x, \text{Thresholds})

sumz =

let

\text{sumzsub}(\text{total}, 0) = \text{total} + z(0)
\text{sumzsub}(\text{total}, x) = \text{sumzsub}(\text{total} + z(x), x-1)

within
--- ### STATES AND TRANSITIONS ---

States = \{0..4\}

Transitions = \{(0, load, 1), (1, run, 2), (2, terminate, 3), (2, deschedule, 1),
(2, block, 4), (4, interrupt, 1)\}

initState = 0

--- ### ALPHA TRANSITIONS ---

\(\text{AlphaTransitions} = \{(s0, e, s1) | (s0, e, s1) \leftarrow \text{Transitions},
\text{\(\not\)member}(e, \text{ASync})\}\)

\(\text{IncomingAlpha}(st) = \{(s0, e, s1) | (s0, e, s1) \leftarrow \text{AlphaTransitions},
\text{s0} \neq \text{st} \text{ and } s1 \equiv \text{st}\}\)

\(\text{SelfloopsAlpha}(st) = \{(s0, e, s1) | (s0, e, s1) \leftarrow \text{AlphaTransitions},
\text{s0} \equiv \text{st} \text{ and } s1 \equiv \text{st}\}\)

\(\text{OutgoingAlpha}(st) = \{(s0, e, s1) | (s0, e, s1) \leftarrow \text{AlphaTransitions},
\text{s0} \equiv \text{st} \text{ and } s1 \neq \text{st}\}\)

--- ### GAMMA-EVENTS AND TRANSITIONS ---

\(\text{GammaEvents} = \text{ASync}\)

\(\text{GammaTransitions} = \{(s0, e, s1) | (s0, e, s1) \leftarrow \text{Transitions},
\text{\(\not\)member}(e, \text{GammaEvents})\}\)

\(\text{GenerateDoGammaTransitions} =
\) let
\(\text{GammaSeq} = \text{makeSeq}(\text{GammaTransitions})\)
within
\(\{(e, <(s0, s1) | (s0, e', s1) \leftarrow \text{GammaSeq}, e \equiv e'>) | e \leftarrow \text{GammaEvents}\}\)

\(\text{lex}((s0, e, s1), (s0', e', s1')) = s0' \prec s0 \text{ or } (s0' \equiv s0 \text{ and } s1' \equiv s1)\)

\(\text{stateMin}(S) = \{x | x \leftarrow S, \{y | y \leftarrow S, \text{\(\not\)lex}(x, y)\} \equiv \{x\}\}\)

\(\text{makeSeq}() = <>\)

\(\text{makeSeq}(S) =
\) let
\(\text{first} = \text{pick}(\text{stateMin}(S))\)
within
\(<\text{first} >\text{makeSeq}(\text{diff}(S, \{\text{first}\}))\)

\(\text{getSeq}(e) = \text{pick}(\{\text{seq} | (e', \text{seq}) \leftarrow \text{GenerateDoGammaTransitions},
e \equiv e'\})\)

\(\text{allCounts}(<>) = \{<>\}\)

\(\text{allCounts}(<<s0, s1>> \not\text{ps}) =\)
\{(s0, s1, c) \mid ts \leftarrow \text{allCounts}(ps),
    c \leftarrow \{0..\max(\max(\text{sum}z, z(0)), 1)\}\}

--- ### AUXILIARY FUNCTIONS ###

\[
\begin{align*}
\text{min}(a, b) &= \text{if } a < b \text{ then } a \text{ else } b \\
\text{max}(a, b) &= \text{if } a < b \text{ then } b \text{ else } a \\
pick(\{\}) &= \langle\rangle \\
pick(\{x\}) &= x
\end{align*}
\]

--- ### SINGLE COUNTER ###

\[
\begin{align*}
\text{Counter}(st, c) &= \\
\text{let}
\text{sumActual}(<> &= 0 \\
\text{sumActual}(<(s0, s1, c)>ps) &= c + \text{sumActual}(ps) \\
\text{v}(st, <> &= 0 \quad \text{--- } v \text{ specifies an actual number of processes that a}
\text{counter represents} \\
\text{v}(st, <(s0, s1, c)>ps) &= \text{if st} = s0 \text{ then } c + \text{v}(st, ps) \text{ else } \text{v}(st, ps) \\
\text{firstState}(st, <) &= \text{false} \\
\text{firstState}(st, <(s0, s1, c)>ps) &= \text{if st} = s0 \text{ then } \text{true} \\
\text{else } \text{firstState}(st, ps) \\
\text{sumzTargets}(st, <> &= 0 \quad \text{--- } \text{calculates how many processes move into}
\text{state st} \\
\text{sumzTargets}(st, <(s0, s1, c)>ps) &= \\
\text{if st} = s0 \text{ then } z(s1) + \text{sumzTargets}(st, ps) \text{ else } \text{sumzTargets}(st, ps) \\
\text{sumSecond}(st, <> &= 0 \quad \text{--- } \text{calculates how many processes move into}
\text{state st} \\
\text{sumSecond}(st, <(s0, s1, c)>ps) &= \\
\text{if st} = s1 \text{ then } c + \text{sumSecond}(st, ps) \text{ else } \text{sumSecond}(st, ps) \\
\text{within}
\text{(doAlpha? \langle:IncomingAlpha(st) \rightarrow \text{Counter}(st, min(c+1, z(st)))\rangle )} \\
\text{(c >= min(z(st), 1) \& doAlpha? \langle:SelfloopsAlpha(st) \rightarrow \text{Counter}(st, c)\rangle )} \\
\text{)} \\
\text{(c >= min(z(st), 1) \& doAlpha? \langle:OutgoingAlpha(st) \rightarrow}
\text{(if c == z(st) then Counter(st, c) \mid \text{Counter}(st, c-1) \text{ else Counter}(st, c-1))\rangle )} \\
\text{)} \\
\text{doGamma? \langle:e:GammaEvents?seq : \{seq |}
\text{seq \leftarrow \text{allCounts(getSeq(e)), sumActual(seq) > 0 and}
\text{(not firstState(st, seq) or (firstState(st, seq) and}
\text{v(st, seq) <= sumzTargets(st, seq) and}
\text{if c == z(st) then}\text{v(st, seq) >= c else v(st, seq) == c)\} \rightarrow}
\text{Counter(st, min(sumSecond(st, seq), z(st)))\} \rightarrow}
\text{Counter(st, min(sumSecond(st, seq), z(st)))}\}
\text{AAlpha(st) = \{doAlpha.a |}
\text{a \leftarrow Union(\{IncomingAlpha(st), SelfloopsAlpha(st), OutgoingAlpha(st)\})\} \\
\text{AGamma(st) = \{doGamma.a.seq | a \leftarrow GammaEvents,}
\text{seq \leftarrow \text{allCounts(getSeq(a))}\} \\
\text{AlphabetOfCounter(st) = union(AAlpha(st), AGamma(st))}
\end{align*}
\]
--- ### PARALLEL COMPOSITION OF ALL COUNTERS ###

Counters = | | st:States @ [AlphabetOfCounter(st)]

\[\text{Counter}(\text{st}, \text{if st == initState then } z(\text{st}) \text{ else } 0)\]

CANodes =

\[\{\text{doAlpha.(s0,e1,s1) \leftarrow e1 | (s0,e1,s1) \leftarrow AlphaTransitions} \} \}

\[\{\text{doGamma.e2.seq \leftarrow e2 | e2 \leftarrow GammaEvents, seq \leftarrow allCounts(getSeq(e2))} \} \]

--- ### ASSERTION ###

assert Spec [ F= C(CANodes)]

---

C.2 Case study

C.2.1 Source code

--- This script models a producer–consumer problem.

--- The implementation consists of a single producer,
--- a single slot that can hold at most one item of data and
--- a number of nodes (consumers) with identities of type t.
--- In our system we use three binary semaphores: Mutex,
--- SlotEmptySemaphore and SlotFullSemaphore. The Mutex
--- semaphore is used to achieve mutual exclusion of
--- critical sections of node processes. We use
--- SlotEmptySemaphore to wake up the producer whenever
--- the slot is ready to accept data. Finally,
--- SlotEmptySemaphore is used to wake up nodes whenever
--- the slot contains a piece of data.
--- We want to verify that every implementation
--- traces refines a three-place buffer.

\[T = \{0..2\} \text{ -- an example instantiation } T \text{ of type } t\]

\[\text{must be a superset of } \{0..B+m\}\]

\[\text{so that the types of channels are well-defined}\]

\[\text{Data} = \{0,1\} \text{ -- type of data being input from / output to}\]

\[\text{the environment}\]

\[\text{datatype Semaphore = up | down}\]

\[\text{channel init, fullSlotUp, emptySlotUp, emptySlotDown}\]

\[\text{channel input, put: Data}\]

\[\text{channel mutexDown, mutexUp, fullSlotDown : T}\]

\[\text{channel get : T.Data}\]
32 channel exchangeMutex : T.T
33 channel output : T.Data
34 35 — Defining the three semaphores
36 Mutex(state, t) =
37    state == up & mutexDown?i:t -> Mutex(down, t)
38    []
39    mutexUp?i:t -> Mutex(up, t)
40 41 FullSlotSemaphore(state, t) =
42    state == up & fullSlotDown?i:t -> FullSlotSemaphore(down, t)
43    []
44    fullSlotUp -> FullSlotSemaphore(up, t)
45 46 EmptySlotSemaphore(state) =
47    state == up & emptySlotDown -> EmptySlotSemaphore(down)
48    []
49    emptySlotUp -> EmptySlotSemaphore(up)
50 51 — Defining the producer
52 Producer = input?d:Data -> emptySlotDown -> put.d -> Producer
53 54 — Defining the nodes
55 Node(i, t) = init -> Node'(i, t)
56 57 Node'(i, t) =
58    mutexDown.i -> Node''(i, t)
59    []
60    exchangeMutex?j:t!i -> Node''(i, t)
61 62 Node''(i, t) =
63    fullSlotDown.i -> get.i?d:Data -> output.i.d -> Node''''(i, t)
64 65 Node''''(i, t) =
66    exchangeMutex.i?j:t -> Node'(i, t)
67    []
68    mutexUp.i -> Node'(i, t)
69 70 A(i, t) = {init, mutexDown.i, fullSlotDown.i, get.i.d, output.i.d, mutexUp.i,
71    exchangeMutex.i.j, exchangeMutex.j.i | j <- diff(t, {i}), d <- Data}
72 73 Nodes(t) = || i:t @ [A(i, t)] Node(i, t)
74 75 — Defining the slot
76 EmptySlot(t) = emptySlotUp -> put?d:Data -> FullSlot(d, t)
77 248
86 FullSlot(d, t) = fullSlotUp -> get?i:t!d -> EmptySlot(t)
87
— Defining the controller
88
Controller(t) =
89 Mutex(up, t) ||
90 
91 ((Producer [[|put|]] EmptySlot(t))
92 ||
93 (emptySlotDown, emptySlotUp, fullSlotUp))
94 (EmptySlotSemaphore(down)
95 ||
96 FullSlotSemaphore(down, t))

— Composing all parts together
98
99 X(t) = {init}
100 Y(t) = {mutexUp, mutexDown, fullSlotDown, get[]}
101 Z(t) = {mutexUp, mutexDown, exchangeMutex, fullSlotUp, fullSlotDown, emptySlotUp, emptySlotDown, put, get[]}
102 C(P, t) = ((P \ X(t)) || Y(t) || Controller(t)) \ Z(t)
103 Impl(t) = C(Nodes(t), t)

— Defining the specification
110
111 Buff(<>, t) = input?d:Data -> Buff(<d>, t)
112 Buff(<d>^seq, t) =
113 output?i:t!d -> Buff(seq, t)
114 ||
115 length(seq) < 2 & input?d':Data -> Buff(<d>^seq^d', t)
116 Spec(t) = Buff(<>, t)

— 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 
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, s1 <- States, c <- {0..max{max(sumz.z(1)),1})})
7 channel TAU_

9 T = {0..2} — an example instantiation T of type t
10 — must be a superset of {0..B+m}
11 — so that the types of channels are well-defined
Data = \{0,1\}  --- type of data being input from / output to
--- the environment

datatype Semaphore = up | down

channel init, fullSlotUp, emptySlotUp, emptySlotDown
channel input, put: Data
channel mutexDown, mutexUp, fullSlotDown : T
channel get : T, Data
channel exchangeMutex : T, T
channel output : T, T

--- Defining the three semaphores

Mutex(state, t) =
  state == up & mutexDown?i : t -> Mutex(down, t)
  []
  mutexUp?i : t -> Mutex(up, t)

FullSlotSemaphore(state, t) =
  state == up & fullSlotDown?i : t -> FullSlotSemaphore(down, t)
  []
  fullSlotUp -> FullSlotSemaphore(up, t)

EmptySlotSemaphore(state) =
  state == up & emptySlotDown -> EmptySlotSemaphore(down)
  []
  emptySlotUp -> EmptySlotSemaphore(up)

--- Defining the producer

Producer = input?d : Data -> emptySlotDown -> put . d -> Producer

--- Defining the nodes

Node(i, t) = init -> Node'(i, t)

Node'(i, t) =
  mutexDown . i -> Node''(i, t)
  []
  exchangeMutex?j : t ! i -> Node''(i, t)

Node''(i, t) =
  fullSlotDown . i -> get . i ? d : Data -> output . i . d -> Node'''(i, t)

Node'''(i, t) =
  exchangeMutex . i ? j : t -> Node'(i, t)
  []
  mutexUp . i -> Node'(i, t)

A(i, t) = \{init, mutexDown . i, fullSlotDown . i, get . i . d, output . i . d, mutexUp . i,
exchangeMutex . i . j , exchangeMutex . j . i | j <= diff(t, {i}), d <= Data

Nodes(t) = | | i : t @ [A(i, t)] Node(i, t)

— Defining the slot
EmptySlot(t) = emptySlotUp -> put?d:Data -> FullSlot(d, t)
FullSlot(d, t) = fullSlotUp -> get?i:t!d -> EmptySlot(t)

— Defining the controller
Controller(t) = Mutex(up, t) || (Producer [|| put ||] EmptySlot(t))
            [|| emptySlotDown, emptySlotUp, fullSlotUp ]||
            (EmptySlotSemaphore(down)
            || FullSlotSemaphore(down, t)))

— Composing all parts together
X(t) = { init }
Y(t) = { mutexUp, mutexDown, fullSlotDown, get }
Z(t) = { mutexUp, mutexDown, exchangeMutex, fullSlotUp, fullSlotDown,
          emptySlotUp, emptySlotDown, put, get }
C(P, t) = (P \ X(t)) || Y(t) || Controller(t) \ Z(t)
Impl(t) = C( Nodes(t), t)

— Defining the specification
Buff(<>, t) = input?d:Data -> Buff(<d>, t)
Buff(<d>^seq, t) =
output?i:t!d -> Buff(seq, t)

[] length(seq) < 2 & input?d':Data -> Buff(<d>^seq^'<d'>, t)
Spec(t) = Buff(<>, t)

— an assertion for FDR
assert Spec(T) [ T= Impl(T) ]

### PARAMETERS ###
B = 0
m = 1
### AUXILIARY FUNCTIONS

\[
\phi(x) = \begin{cases} 
  x & \text{if } x < B \\
  B & \text{otherwise} 
\end{cases}
\]

\[
\min(a,b) = \begin{cases} 
  a & \text{if } a < b \\
  b & \text{otherwise} 
\end{cases}
\]

\[
\max(a,b) = \begin{cases} 
  a & \text{if } a < b \\
  b & \text{otherwise} 
\end{cases}
\]

\[
pick(\emptyset) = -1
\]

\[
pick(\{x\}) = x
\]

### THRESHOLDS

Define a sequence of thresholds for all equivalence classes, e.g.

\[
\text{Thresholds} = <2,0,3,0,1>
\]

### STATES AND TRANSITIONS

States = \{0..13\}

### z function allows us to extract values of thresholds

\[
z(x) =
\]

\[
\text{let } z_{\text{sub}}(1,\langle n\rangle^{\text{ns}}) = n
\]

\[
\text{let } z_{\text{sub}}(x,\langle n\rangle^{\text{ns}}) = z_{\text{sub}}(x-1,\text{ns})
\]

\[
\text{within } z_{\text{sub}}(x,\text{Thresholds})
\]

\[
\text{sumz is the sum of all thresholds}
\]

\[
\text{sumz =}
\]

\[
\text{let } \text{sumz}_{\text{sub}}(\text{total},1) = \text{total} + z(1)
\]

\[
\text{let } \text{sumz}_{\text{sub}}(\text{total},x) = \text{sumz}_{\text{sub}}(\text{total} + z(x),x-1)
\]

### STATES AND TRANSITIONS

Transitions\(_x(x) =

if x = = 0 then \{(0,\text{init},1),(1,\text{exchangeMutex},1.0,2),\}

(1,\text{mutexDown},0.2),(1,\text{exchangeMutex},0.0,2),\}

(2,\text{fullSlotDown},0.3),(3,\text{get},0.1,4),\}

(3,\text{get},0.0,5),(4,\text{output},0.1,6),\}

(5,\text{output},0.0,6),(6,\text{exchangeMutex},0.0,0),\}

(6,\text{mutexUp},0.1),(6,\text{exchangeMutex},0.1,1)\} else

if x = = 1 then \{(7,\text{init},8),(8,\text{exchangeMutex},1.1,9),\}

(8,\text{mutexDown},1.9),(8,\text{exchangeMutex},0.1,9),\}

(9,\text{fullSlotDown},1.10),(10,\text{get},1.1,11),\}

(10,\text{get},1.0,12),(11,\text{output},1.1,13)\}
if \( x_0 = 2 \) then
\[
(14, \text{init}, 15), (15, \text{exchangeMutex} \cdot 1.2, 16),
(16, \text{fullSlotDown} \cdot 2, 17), (17, \text{get} \cdot 2.0, 18),
(18, \text{output} \cdot 2.0, 19), (19, \text{output} \cdot 2.1, 20),
(20, \text{mutexUp} \cdot 2, 15), (20, \text{exchangeMutex} \cdot 2.0, 15),
(20, \text{exchangeMutex} \cdot 2.1, 15)
\]
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.

\[
\begin{align*}
\text{initState}(1) &= 0 \quad \text{--- this state needs to be in the first equivalence class} \\
\text{initState}(2) &= 7 \quad \text{--- this state needs to be in the first equivalence class}
\end{align*}
\]

\[
\text{AllTransitions} = \text{Union}(\{ \text{Transitions}(i) \mid i \leftarrow \{ B \ldots (B+m) \} \})
\]

\[
\text{labels( transitions) } = \{ e \mid (s,e,s') \leftarrow \text{transitions} \}
\]

\[
\text{TransitionLabels} = \text{labels(AllTransitions)}
\]

--- ### EQUIVALENCE CLASSES ###

--- Define equivalence classes by providing a sequence of sets of related states, e.g. \(<\{0,1,4,5\},\{2,6\},\{3,7\}>\)

--- Source and target states of Transitions(3) do not have to be included in the equivalence classes.

\[
E = \langle \text{ENTER A SEQUENCE OF SETS OF RELATED STATES HERE} \rangle
\]

\[
k = \text{length}(E)
\]

\[
\text{initEc} = 1
\]

--- ### ALPHABET PARTITIONING ###

--- tests if state st is a member of the ec–th equivalence class

\[
\text{ecmember}(st, ec) =
\]

let
\[
\text{ecmembersub}(st, 1, \langle st \rangle^{stss} = \text{member}(st, \text{sts})
\]

\[
\text{ecmembersub}(st, ec, \langle st \rangle^{stss} = \text{ecmembersub}(st, ec-1, \text{sts})
\]

\[
\text{within}
\]

\[
\text{ecmembersub}(st, ec, E)
\]

--- count the number of alphabets a particular event occurs in to decide if it is a alpha–, beta– or gamma–event

\[
\text{countAs}(e) =
\]

let
countAssub(e, 0) = if member(e, labels(Transitions(B))) then 1 else 0

countAssub(e, i) = if member(e, labels(Transitions(B+i))) then 1 +
    countAssub(e, i-1) else countAssub(e, i-1)

within

countAssub(e, m+1)

--- ### ALPHA-EVENTS AND TRANSITIONS ###

AlphaEvents = \{ e | e \leftarrow TransitionLabels, countAss(e) = 1 or e == TAU \}

AlphaTransitions = \{ (s0, e, s1) | (s0, e, s1) \leftarrow AllTransitions,
    member(e, AlphaEvents) \}

IncomingAlpha(ec) = \{ (s0, e, s1) | (s0, e, s1) \leftarrow AlphaTransitions,
    not ecmember(s0, ec)
    and ecmember(s1, ec) \}

SelfloopsAlpha(ec) = \{ (s0, e, s1) | (s0, e, s1) \leftarrow AlphaTransitions,
    ecmember(s0, ec) and ecmember(s1, ec) \}

OutgoingAlpha(ec) = \{ (s0, e, s1) | (s0, e, s1) \leftarrow AlphaTransitions,
    ecmember(s0, ec)
    and not ecmember(s1, ec) \}

--- ### BETA-EVENTS AND TRANSITIONS ###

BetaEvents = \{ e | e \leftarrow TransitionLabels, countAss(e) == 2 and e != TAU \}

BetaTransitions = \{ (s0, e, s1) | (s0, e, s1) \leftarrow AllTransitions,
    member(e, BetaEvents) \}

BetaEventsOf(ec) = \{ e | (s0, e, s1) \leftarrow BetaTransitions,
    ecmember(s0, ec) or ecmember(s1, ec) \}

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 s1 and s2 and 2 target states s1' and s2' such that s1 and s2 can both perform e to reach s1' and s2', respectively, and at least one of s1, s1', s2, s2' is in equivalence class ec

let

whichLTS(st, l, st') = pick\{ (i | i \leftarrow \{B..(B+m)\},
    member((st, l, st'), Transitions(i)) \})

within

\{ (st1, st1', st2, st2') \ | (st1, e1, st1') \leftarrow BetaTransitions,
(st2, e2, st2') \leftarrow BetaTransitions,
    e1 == e and e2 == e,
    ecmember(st1, ec) or ecmember(st1', ec)
    or ecmember(st2, ec) or ecmember(st2', ec),
    whichLTS(st1, e1, st1') != whichLTS(st2, e2, st2'),
    st2 >= st1 \} — \{ st1, st1', st2, st2' \} is the same as \{ st2, st2', st1, st1' \} for our purposes, so we only take one of these

--- ### GAMMA-EVENTS AND TRANSITIONS ###

254
GammaEvents = \( \text{diff}(\text{TransitionLabels}, \text{union}(\text{AlphaEvents}, \text{BetaEvents})) \)

GammaTransitions = \{ (s0, e, s1) \mid (s0, e, s1) \leftarrow \text{AllTransitions}, \text{member}(e, \text{GammaEvents}) \} 

GenerateDoGammaTransitions =
let
GammaSeq = \text{makeSeq}(\text{GammaTransitions})
within
\{ (e, (s0, s1)) \mid (s0, e', s1) \leftarrow \text{GammaSeq}, e = e' \} | e \leftarrow \text{GammaEvents} \}

\text{lex}((s0, e, s1), (s0', e', s1')) = s0' < s0 \text{ or } (s0' = s0 \text{ and } s1' <= s1)

\text{stateMin}(S) = \{ x \mid x \leftarrow S, \{ y \mid y \leftarrow S, \text{lex}(x, y) \} = \{x\} \}

\text{makeSeq}(\{\}) = <>

\text{makeSeq}(S) =
let
first = \text{pick}(\text{stateMin}(S))
within
<first>^\text{makeSeq}(\text{diff}(S, \{first\}))

\text{getSeq}(e) = \text{pick}(\{\text{seq} \mid (e', \text{seq}) \leftarrow \text{GenerateDoGammaTransitions}, e = e'\})

\text{allCounts}(<>^1) = \{<>^1\}

\text{allCounts}(<>^1, ^1) = \{<s0, s1, c>^1 \mid ts \leftarrow \text{allCounts}(ps), c \leftarrow 0..\max(\max(sumz, z(1)), 1)\}

--- SINGLE COUNTER ---

--- a single counter, corresponding to equivalence class ec and with value c

\text{Counter}(ec, c) =
let
\text{compare}(st) = \text{if ecmember}(st, ec) \text{ then } 1 \text{ else } 0
\text{ups}(st1', st2') = \text{compare}(st1') + \text{compare}(st2')
\text{downs}(st1, st2) = \text{compare}(st1) + \text{compare}(st2)
\text{p}(st1, st2) = \text{compare}(st1) + \text{compare}(st2)
\text{sumActual}(<>^1) = 0
\text{sumActual}(<>^1, ^1) = c + \text{sumActual}(ps)

v(ec, <>^1) = 0 --- v specifies an actual number of processes that a counter represents

\text{firstState}(ec, <>^1) = \text{false}
\text{firstState}(ec, <>^1, ^1) = \text{if ecmember}(s0, ec) \text{ then } true \text{ else } \text{firstState}(ec, ps)
\text{sumSecond}(ec, <>^1) = 0
\text{sumSecond}(ec, <>^1, ^1) = c + \text{sumSecond}(ec, ps)
sumzTargets(ec,<>)=0
sumzTargets(ec,<(s0,s1,c)>^ps)=
    if ecmember(s0,ec) then z(whichEc(s1)) + sumzTargets(ec,ps)
else sumzTargets(ec,ps)

whichEc(st)=pick({i | i <- {1..k}, ecmember(st,i)})

[doAlpha?::IncomingAlpha(ec) -> Counter(ec, min(c+1,z(ec))))]
[|]
(c >= min(z(ec),1) & doAlpha?::SelfloopsAlpha(ec) -> Counter(ec,c))
[|]
(c >= min(z(ec),1) & doAlpha?::OutgoingAlpha(ec) ->
    (if c == z(ec) then Counter(ec,c) | Counter(ec,c-1)
else Counter(ec,c-1)))
[|]
(] e: BetaEventsOf(ec),
(st1,st1',st2,st2'): BetaTransitionPairs(ec,ec) @
c >= min(p(st1,st2),z(ec)) & doBeta.e.st1.st1'.st2.st2' ->
    (if c == z(ec)
    then ¦|x:{min((c+ups(st1',st2'))
    -downs(st1,st2)),z(ec))..z(ec})
    @ Counter(ec,x)
else Counter(ec,min(c+ups(st1',st2'))
    -downs(st1,st2),z(ec)))
[|]
doGamma?e:GammaEvents?seq:{seq |
    seq <- allCounts(getSeq(e)), sumActual(seq) > 0 and
    (not firstState(ec,seq) or (firstState(ec,seq)
and v(ec,seq) <= sumzTargets(ec,seq) and
    if c == z(ec) then v(ec,seq) >= c
else v(ec,seq) == c))} ->
    Counter(ec,min(sumSecond(ec,seq),z(ec))))

— Defining alphabets of counters

ECAAlpha(ec) = {doAlpha.a | a <- Union({IncomingAlpha(ec),
    SelfloopsAlpha(ec),
    OutgoingAlpha(ec)})}

ECABeta(ec) = {doBeta.a.st1.st1'.st2.st2' | a <- BetaEventsOf(ec),
(st1,st1',st2,st2') <= BetaTransitionPairs(a,ec})

ECAGamma(ec) = {doGamma.a.seq | a <- GammaEvents,
    seq <- allCounts(getSeq(a))}

ECA(ec) = Union({ECAAlpha(ec),ECABeta(ec),ECAGamma(ec)})

— ### ABS CONSTRUCTION ###

Counters = || ec:{1..k} @ [ECA(ec)]
    Counter(ec, if ec == initEc then z(ec) else 0)

— Renaming events to their original form and converting all TAU events
to real tau's
Counter Renamed =
  (((Counter
  
  \{TAU\})
  \{doAlpha.(s0,e1,s1) <- e1 | (s0,e1,s1) <- AlphaTransitions \})
  \{doBeta.e2.st1'st1'.st2'.st2' <- e2 | e2 <- BetaEvents, ec <- \{1..k\},
  (st1, st1', st2, st2') <- BetaTransitionPairs(e2, ec)\})
  \{doGamma.e3.seq <- e3 | e3 <- GammaEvents, seq <- allCounts(getSeq(e3)) \})

--- Abstract model of all node processes

ABS =
  (\{\ i: \{0..(B-1)\} @ [A(i,\{0..(B+m)\})] Node(i,\{0..(B+m)\}) \}
  \{Union\{A(i,\{0..(B+m)\}) | i <- \{0..(B-1)\}\}
  \{|\}
  \{Union\{A(i,\{0..(B+m)\}) | i <- \{B..(B+m)\}\}\} Counter Renamed
  \{mutexDown.x1 <- mutexDown.phi(x1) | x1 <- \{0..(B+m)\}\}
  \{mutexUp.x1 <- mutexUp.phi(x1) | x1 <- \{0..(B+m)\}\}
  \{fullSlotDown.x1 <- fullSlotDown.phi(x1) | x1 <- \{0..(B+m)\}\}
  \{get.x2.x1 <- get.phi(x2).x1 | x2 <- \{0..(B+m)\}, x1 <- Data\}
  \{exchangeMutex.x2.x1 <- exchangeMutex.phi(x2).phi(x1) | x2 <- \{0..(B+m)\}, x1 <- Data\}
  \{output.x2.x1 <- output.phi(x2).x1 | x2 <- \{0..(B+m)\}, x1 <- Data\}

--- ASSERTION ---

assert Spec\{0..B\} \(\trianglerighteq\) C(ABS,\{0..B\})

---

257