A TYPE REDUCTION THEORY FOR SYSTEMS WITH REPLICATED COMPONENTS

TOMASZ MAZUR AND GAVIN LOWE

Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom.
e-mail address: tomasz.mazur@gmail.com, gavin.lowe@cs.ox.ac.uk

Abstract. The Parameterised Model Checking Problem asks whether an implementation $Impl(t)$ satisfies a specification $Spec(t)$ for all instantiations of parameter t. In general, t can determine numerous entities: the number of processes used in a network, the type of data, the capacities of buffers, etc. The main theme of this paper is automation of uniform verification of a subclass of PMCP with the parameter of the first kind, i.e. the number of processes in the network. We use CSP as our formalism.

We present a type reduction theory, which, for a given verification problem, establishes a function ϕ that maps all (sufficiently large) instantiations T of the parameter to some fixed type $\hat{T}$ and allows us to deduce that if $Spec(\hat{T})$ is refined by $\phi(Impl(T))$, then (subject to certain assumptions) $Spec(T)$ is refined by $Impl(T)$. The theory can be used in practice by combining it with a suitable abstraction method that produces a t-independent process $Abstr$ that is refined by $\phi(Impl(T))$ for all sufficiently large T. Then, by testing (with a model checker) if the abstract model $Abstr$ refines $Spec(\hat{T})$, we can deduce a positive answer to the original uniform verification problem.

The type reduction theory relies on symbolic representation of process behaviour. 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. Based on this, we prove results that allow us to infer behaviours of a process instantiated with uncollapsed types from known behaviours of the same process instantiated with a reduced type.

One of the main advantages of our symbolic operational semantics and the type reduction theory is their generality, which makes them applicable in a wide range of settings.

1. Introduction

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.

1998 ACM Subject Classification: D.1.3, D.2.4, D.3.2.
Key words and phrases: model checking, PMCP, type reduction, CSP, counter abstraction.

LOGICAL METHODS
IN COMPUTER SCIENCE
DOI:10.2168/LMCS-???

In contrast to the above, *formal verification* methods concentrate on *proving* the correctness of a given system. One approach to formal verification is *model checking*. Given a model *Impl* of an implementation and a specification *Spec* that the model should satisfy, verification via model checking occurs by exploring (explicitly or symbolically) all states of *Impl* and checking if they satisfy *Spec*. The greatest advantage of model checking is a large scope for automation, at the cost of being applicable only to finite-state systems and a few families of infinite systems. In addition, if the implementation fails to satisfy the specification, then model checking can produce a counterexample (a behaviour of the implementation that is not allowed by the specification) that can be used for debugging purposes. On the other hand, this approach to formal verification suffers from the *state explosion problem*: the time complexity of verification algorithms depends on the size of the implementation, which is typically exponential in the size of its description. This means that standard model checking algorithms can only work in cases where the system to be verified is of finite and (relatively) small size.

One approach to model checking, highly popularised by Clarke, Emerson and Grumberg [CE81, CES86, CGL94, CGP99], is based on temporal logics, where specifications are formulated as expressions in a linear time logic (e.g. LTL [Pnu77]) or a branching time logic (e.g. CTL [EC80, BAMP81]). Another approach defines a partial order $\sqsubseteq$ on the set of all expressible systems. The intuitive meaning of $P \sqsubseteq Q$ (pronounced "P refined by Q") 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 (see Section 2.2 for the formal definition). In this approach *Spec* and *Impl* are modelled using the same formalism and *Impl* is said 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 paper we use the refinement-based approach to model checking, where all implementations and specifications are modelled using the CSP process algebra [Hoa85, Ros97, Ros10] (see Section 2) and refinement checks are performed automatically using the FDR model checker [For09].

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. One is often interested in the *uniform verification* of a given parameterised pair of a specification *Spec* and an implementation *Impl*, i.e. in checking 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 PVP, where we insist on the verification occurring via model checking.

In this paper we concentrate on a subclass of PMCP, where specifications and implementations contains a single parameter t, called the *distinguished type*, which denotes the type of identities of node processes running concurrently to form a network, possibly within some larger system. More precisely, every family of implementations that we consider is of

the form[1]

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

where:

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

In fact, the results of this paper apply to more general implementation processes that the above $Impl(t)$, namely all that are fully symmetric in t (informally, that renaming the elements of t under an arbitrary bijection gives an equivalent process; see Definition 3.6, below); however, $Impl(t)$ captures those processes that we are particularly interested in.

Our overall aim, then, is to verify that for all sufficiently large instantiations T of t:

$$Spec(T) \quad \sqsubseteq \quad Impl(T), \tag{1.1}$$

where $Spec(t)$ is a suitable specification process.

Throughout this paper we assume that every instantiation T of type parameter t is non-empty and finite. In addition, without loss of generality, we assume that every instantiation T of t is an initial segment of the natural numbers, i.e. T is of the form $\{0\,..\,n-1\}$ for some n. Our results and techniques extend to other discrete and finite types T of size n via simple bijections from $\{0\,..\,n-1\}$ to T. We allow processes to contain other parameters in their syntax, but their values must be known and fixed at the time of writing the process definition, or an additional technique for handling parameters (e.g. data independence [Laz99, Ros97]) must be used for complete correctness analysis.

PMCP is, in general, undecidable [AK86], as the Halting Problem [Dav58] can be shown to reduce to it. Therefore, we focus on sound (but incomplete) verification methods.

One general approach is to build a t-independent abstraction process $Abstr$ that captures the behaviours of all the $Impl(T)$ processes, in a sense that we now explain. The alphabets of $Impl(T)$ are (in general) unbounded as a function of T; however, the alphabet of $Abstr$ needs to be fixed. Therefore, the construction of $Abstr$ collapses T to some fixed type $\hat{T} = \{0\,..\,B\}$ for some non-negative integer B, treating all identities in $\{0\,..\,B-1\}$ faithfully, but mapping all other identities onto B. More precisely, for all sufficiently large instantiations T of type t, $Abstr$ is such that[2]

$$Abstr \quad \sqsubseteq \quad \phi(Impl(T)) \tag{1.2}$$

holds by construction, where ϕ is a B-collapsing function:

Definition 1.1. A *B-collapsing function* is a function $\phi : T \to \{0\,..\,B\}$ such that

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

[1]The process $\left|\left| \, i \in I \bullet [A(i)]P(i) \right.\right.$ denotes the parallel composition of the processes $P(i)$ for $i \in I$, where $A(i)$ is the alphabet of $P(i)$, and where nodes synchronise on all events in common between their alphabets; see Section 2.

[2]The process $f(P)$ is a process that acts like P, except every event a is renamed to $f(a)$; see Section 2.

Having constructed such an $Abstr$, we can use a CSP model checker, such as FDR, to verify that

$$Spec(\hat{T}) \quad \sqsubseteq \quad Abstr.$$

Transitivity of refinement then allows us to deduce that

$$Spec(\hat{T}) \quad \sqsubseteq \quad \phi(Impl(T)) \tag{1.3}$$

for all sufficiently large T. An example of such an abstraction method (based on counter abstraction techniques [Lub84, PXZ02, ML09]) can be found in [Maz10, ML11].

The aim of this paper is to bridge the gap between equations (1.3) and (1.1). We present a theory that, under suitable assumptions on the specification and implementation processes, allows us to calculate a suitable value for B such that if equation (1.3) holds (for the values of ϕ and $\hat{T}$ corresponding to B), then equation (1.1) holds for all T such that $\#T > B$ (smaller values of T can be tested directly). In particular, the value of B turns out to depend only on the syntax of the specification, and is independent of the implementation.

Our theory is general, allowing us to combine it with an arbitrary abstraction method that can produce an abstraction $Abstr$ such that (1.2) holds.

The rest of this paper is structured as follows. In Section 2 we introduce the syntax of the CSP process algebra, describe two of its denotational semantics models (traces and stable failures) and briefly talk about FDR, a model checker for CSP. We also give an example to illustrate the goals of this paper. In Section 3 we define the conditions we will require the specification process to satisfy, and also the condition of symmetry in t that we will require the implementation to satisfy.

Proving the main theorems will require us to develop quite a lot of supporting machinery, in order to relate behaviours of the specification process for different values of the parameter t. To this end, Section 4 is devoted to developing a suitable operational semantics for CSP. The main part of this section presents a symbolic operational semantics that allows us to reason about behaviour of processes without the need for instantiating parameters. We also provide a set of translation rules for instantiating symbolic transition graphs into concrete ones, and we prove that this results in an operational semantics congruent to a fairly standard one.

Being able to reason about process behaviour in a symbolic way is a prerequisite for our main theory. We present a number of regularity results for specifications in Section 5, which show that specifications exhibit certain clarity in their behaviour. Our main type reduction theory is in Section 6, where we provide type reduction theorems for the traces and stable failures models. Finally, we conclude in Section 7. In the interests of readability, we relegate most proofs to appendices.

2. Introduction to CSP

CSP [Hoa85, Ros97, Ros10] is a process algebra used for modelling and verification of concurrent reactive systems with communication based on synchronous message passing.

CSP processes interact with each other and the environment within which they operate by communicating *events*. Events occur on *channels*; for example, $c.a.3$ is an event over channel c, passing data a and 3. We assume that each channel has a fixed type (i.e. can pass a fixed number of pieces of data, and the type of the data passed in each position is fixed). The notation $\{|\ c\ |\}$ represents the set of events passed over channel c.

We let Σ be the set of all visible events. We let τ denote a special, internal event (not in Σ). We write Σ^τ to mean $\Sigma \cup \{\tau\}$. We also write Σ^* to mean the set of all finite sequences of events from Σ.

2.1. **Syntax.** In this paper we use the fragment of CSP with the following syntax.

$$\begin{aligned} P ::= {} & STOP \mid \alpha \rightarrow P \mid P \,\Box\, P \mid \Box\, i \in \mathcal{I} \bullet P(i) \mid P \sqcap P \mid \sqcap\, i \in \mathcal{I} \bullet P(i) \mid P \triangleright P \\ & \mid \text{if } b \text{ then } P \text{ else } P \mid b \;\&\; P \mid P \setminus X \mid P[\![\mathcal{R}]\!] \mid P \;{}_X\|_Y\; P \\ & \mid \Big\|\, i \in \mathcal{I} \bullet [A(i)]\; P(i) \mid P \underset{X}{\|} P \mid P \;|||\; P \mid \Big|||\, i \in \mathcal{I} \bullet P(i) \mid X \end{aligned}$$

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

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

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

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

For two processes P and Q, the *external* (or *deterministic*) choice $P \,\Box\, Q$ is a process that offers the environment the choice of performing any initial event of P or Q; if an initial event of P is performed, then the choice is resolved to P, and if an initial event of Q is performed, then the choice is resolved to Q. We can define a replicated version of the operator: $\Box\, i \in \mathcal{I} \bullet P(i)$ is an external choice between processes $P(i)$ for each i in some finite indexing set $\mathcal{I}$; we consider this as syntactic sugar for repeated use of the binary operator.

$P \sqcap Q$ represents an *internal* (or *nondeterministic*) choice, where the process behaves either like P or like Q, where the choice is made by some mechanism that we do not model and which cannot be influenced by the environment. We define a replicated version: $\sqcap\, i \in \mathcal{I} \bullet P(i)$ is an internal choice between processes $P(i)$ for each i in some finite, nonempty indexing set $\mathcal{I}$.

The *sliding* choice (or *timeout*) $P \triangleright Q$ is a process that behaves like P for a nondeterministically long period of time, but if the environment does not engage in any activity with P within this time, it switches to behaving like Q.

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

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

For any set $X \subseteq \Sigma$, $P \setminus X$ is a process which behaves like P except that whenever P would normally communicate an event from set X, $P \setminus X$ performs the internal action, τ, instead.

The process $P[\![\mathcal{R}]\!]$, where $\mathcal{R}$ is a relation over Σ, is a process that behaves like P except that whenever P would perform an event a, the renamed process performs an event b such that $a \mathcal{R} b$ instead. We sometimes define the renaming relation using notation similar to substitution: $P[\![{}^{b}/_{a}]\!]$ is a process that behaves like P except that whenever P would normally perform a, the renamed process performs b instead. If $\mathcal{R}$ is a function, we sometimes write the renaming using functional notation, $\mathcal{R}(P)$.

The notion of parallel composition of processes is key to CSP, allowing one to model concurrency. The process $P \ {}_X\|_Y \ Q$ is a parallel composition of P and Q, where P is allowed to communicate only members of the set of visible events X, Q is allowed to communicate only members of the set of visible events Y, and synchronisation occurs on all common events (i.e. those in $X \cap Y$). We can define its replicated version: $\Big\|\ i \in \mathcal{I} \bullet [A(i)]\ P(i)$ is the parallel composition of processes $P(i)$ indexed over a finite, non-empty set $\mathcal{I}$, where each $P(i)$ is allowed to perform only events from $A(i)$, and synchronises on event $e \in A(i)$ with each process $P(j)$ such that $e \in A(j)$. The process $P \underset{X}{\|} Q$ is the parallel composition of P and Q with handshaken synchronisation on all the members of the set of visible events X. Finally, $P \ ||| \ Q$ is the interleaving of P and Q: the processes run in parallel, but do not synchronise on any event (note that this is equivalent to $P \underset{\{\}}{\|} Q$). We write $\Big|||\ i \in \mathcal{I} \bullet P(i)$ for the replicated version.

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.

So far we have used the term "process" loosely. We now make 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 and it represents a family of processes $Proc(T)$, one for each concrete instantiation T.

2.2. **Denotational models and refinement.** A *trace* of a process is a sequence of visible events that it can perform. We write $traces(P)$ for the traces of P.

Given a process P, we let $initials(P)$ be the set of all the initially available visible events of P, i.e.

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

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

$$initials(P/tr) \quad = \quad \{a \mid tr^\frown\langle a\rangle \in traces(P)\}.$$

CSP specifications are expressed in the same formalism as implementations, i.e. as processes. An implementation *Impl* is said to satisfy 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. In the traces model refinement is defined by:

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

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

In the *stable failures model*, a process P is identified by the set of its traces (as above) together with the set of its *failures* (written $failures(P)$). A failure is a pair (tr, X), where $tr \in traces(P)$ and $X \subseteq \Sigma$, and represents the behaviour where P performs trace tr to reach a stable state P' (i.e. τ is not available in P'), in which it refuses the whole of X (i.e. none of the events in X is available), denoted $P'\ ref\ X$. When refinement is interpreted over the stable failures model, we get the notion of *stable failures refinement*:

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

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

All denotational representations of a process P (including $traces(P)$ and $failures(P)$) can be obtained using the rules of denotational semantics, which can be found, for example, in [Ros97, Chapter 8]. An alternative approach (and the one we take most of the time in this paper) 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 4.2.2.

The FDR (Failures/Divergences Refinement) model checker [For09] allows one to automatically perform refinement checks. When a CSP script with process definitions, say P and Q, is loaded, FDR can automatically test for refinement $P \sqsubseteq_{\mathrm{M}} Q$ in a given denotational model M.

2.3. **Example.** We give here a simple example, to illustrate the problem we are addressing in this paper.

Consider a very simple token-based mutual exclusion protocol for a collection of nodes. Each node i obtains the token (event $getToken.i$), enters the critical section (event $enterCS.i$), leaves the critical section (event $leaveCS.i$), and returns the token (event $returnToken.i$):

$$\begin{aligned} Node(i) &= getToken.i \rightarrow Entering(i), \\ Entering(i) &= enterCS.i \rightarrow CS(i), \\ CS(i) &= leaveCS.i \rightarrow Leaving(i), \\ Leaving(i) &= returnToken.i \rightarrow Node(i). \end{aligned}$$

The nodes are interleaved; recall that we use the variable t to denote the type of all node identities:

$$Nodes(t) \quad = \quad |||\, i : t \bullet Node(i).$$

The nodes are combined with a controller that controls the token, repeatedly giving it to a node and receiving it back. The communications corresponding to passing the token are considered internal so are hidden.

$$\begin{aligned} Controller(t) &= getToken?i{:}t \rightarrow returnToken?j{:}t \rightarrow Controller(t), \\ Impl(t) &= (Nodes(t) \underset{\{|getToken, returnToken|\}}{\parallel} Controller(t)) \\ &\qquad \setminus \{|\ getToken, returnToken\ |\}. \end{aligned}$$

We would like to verify that at most a single node is in the critical section at a time. We can capture this using the specification process

$$Spec(t) \quad = \quad enterCS\$i{:}t \rightarrow leaveCS!i \rightarrow Spec(t).$$

Our requirement, then, is

$$Spec(T) \quad \sqsubseteq_{\mathrm{T}} \quad Impl(T), \qquad \text{for all instantiations } T \text{ of } t. \tag{2.2}$$

The approach we describe in [Maz10, ML11] is to form an abstraction of $Nodes(t)$ based on counter abstraction [PXZ02]. In the process $NodesAbst(n, e, c, l)$, below, the four counter parameters n, e, c and l represent the number of nodes in the *Node*, *Entering*, *CS* and *Leaving* states, respectively; however the counting is capped at some value z, where we take $z = 2$ in this case; hence a counter value of z represents that there are z *or more* processes in the corresponding state. The definition of *NodesAbst* is based on the transitions within a single *Node* process. For most transitions, the counter for the prior *Node* state is decremented, and the counter for the new state is incremented, but not beyond z; we define the following function to perform this:

$$inc(x) \quad = \quad \min(x + 1, z).$$

However, if the counter for the prior state was at the cap z, then there might have been strictly more than z processes in this state before the transition, so the counter should

(nondeterministically) be able to stay at z.

$$
\begin{array}{l}
NodesAbst(n,e,c,l)(t) = \\
\quad (n>0 \,\&\, getToken\$i{:}t \rightarrow \\
\qquad \text{if } n<z \text{ then } NodesAbst(n-1,inc(e),c,l)(t) \\
\qquad \text{else } NodesAbst(n-1,inc(e),c,l)(t) \sqcap NodesAbst(n,inc(e),c,l)(t)) \\
\quad \Box \\
\quad (e>0 \,\&\, enterCS\$i{:}t \rightarrow \\
\qquad \text{if } e<z \text{ then } NodesAbst(n,e-1,inc(c),l)(t) \\
\qquad \text{else } NodesAbst(n,e-1,inc(c),l)(t) \sqcap NodesAbst(n,e,inc(c),l)(t)) \\
\quad \Box \\
\quad (c>0 \,\&\, leaveCS\$i{:}t \rightarrow \\
\qquad \text{if } c<z \text{ then } NodesAbst(n,e,c-1,inc(l))(t) \\
\qquad \text{else } NodesAbst(n,e,c-1,inc(l))(t) \sqcap NodesAbst(n,e,c,inc(l))(t)) \\
\quad \Box \\
\quad (l>0 \,\&\, returnToken\$i{:}t \rightarrow \\
\qquad \text{if } l<z \text{ then } NodesAbst(inc(n),e,c,l-1)(t) \\
\qquad \text{else } NodesAbst(inc(n),e,c,l-1)(t) \sqcap NodesAbst(inc(n),e,c,l)(t)).
\end{array}
$$

We can then build $Abst$ from $NodesAbst(z,0,0,0)$ in the same way that we built $Impl$ from $Nodes$:

$$
\begin{array}{rcl}
Abst(t) & = & (NodesAbst(z,0,0,0)(t) \underset{\{|getToken,returnToken|\}}{\parallel} Controller(t)) \\
& & \qquad \setminus \{|\ getToken, returnToken\ |\}.
\end{array}
$$

In [Maz10, ML11], we show that the process built in this way is an abstraction of the $Impl$ process in the following sense: for every non-negative integer B:

$$Abst(\hat{T}) \quad \sqsubseteq_{\mathrm{T}} \quad \phi(Impl(T)), \tag{2.3}$$

for all instantiations T of t with $\#T \geq B+z$, where $\hat{T} = \{0\,..\,B\}$, and ϕ is a B-collapsing function (see Definition 1.1). We pick $B=1$ in this case. We can then use FDR to verify that

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

and so deduce

$$Spec(\hat{T}) \quad \sqsubseteq_{\mathrm{T}} \quad \phi(Impl(T)), \qquad \text{for all instantiations } T \text{ of } t \text{ with } \#T \geq B+z = 3,$$

by transitivity of refinement. The results in this paper will allow us to deduce our requirement (2.2) from this. We stress, though, that the results in this paper can be used with any abstraction method that produces a process $Abst$ such that (2.3) holds for all sufficiently large instantiations T of t.

It is worth noting that the technique in [Maz10, ML11] is rather more general than the above example illustrates. It allows node processes to store the identities of other nodes, and to pass them on in subsequent events; much of the difficulty of the theory concerns treating these identities correctly.

3. Conditions on processes

In this section we define various conditions on processes that we will use later. In Section 6.1 we will mention tool support, which is able to test for most of the conditions in this section.

We mentioned above 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 Section 6) useful results about specifications that satisfy a certain normality condition, which we define in Section 3.3. Earlier, in Section 3.1 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 Section 6.

In Section 3.4 we define the notion of type symmetry in the type t; our main theorems will require the implementation process to satisfy this property. Then in Section 3.5 we define a property concerning the use of equality tests; our main theorems will require the specification process to satisfy this property.

3.1. **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 3.1. We say that a CSP process syntax is data independent with respect to type t if it does not contain:

(i) replicated constructs indexed over any set depending on t, except for replicated nondeterministic choice ($\sqcap$) indexed over the whole of t; however, we allow the use of deterministic and nondeterministic input selections, ? and \$;
(ii) conditional choices on t, except for equality and inequality tests;
(iii) constants of type t;
(iv) functions whose domains or co-domains involve type t;
(v) operations on t, including polymorphic operations (e.g. tupling or lists);
(vi) selections from sets involving t, unless the selection is over the whole of t; and
(vii) any operations that would extract information about t, e.g. $card(t)$.

Example 3.2. The $Node(i)(T)$ processes from Section 2.3 are data independent in t. However, $Nodes(t)$ is not data independent because it uses an indexed interleaving over t.

Remark 3.3. Clauses (v) and (vi) of Definition 3.1 together imply that, for all constructs $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ of a given data independent process syntax, each X_i is either a type not related to t or precisely the type parameter t, unless $\S_i = !$, in which case $X_i = null$.

3.2. **The Seq condition.**

In order to produce our Semi-Symbolic Operational Semantics, it is useful to restrict the scope of processes considered.

Definition 3.4. A process syntax $Proc(t)$ satisfies **Seq** 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) all guards of conditional choices within $Proc(t)$ contain either only variables of type t, or only variables and values of types other that t;
(v) in external and sliding choices, $Proc(t)$ contains no name clashes between type t nondeterministic-selection variables of one argument and free variables of another argument; e.g. $c\$x{:}t \rightarrow STOP \,\Box\, d.x \rightarrow STOP$ is not allowed;
(vi) constructs of $Proc(t)$ do not contain multiple occurrences of the same input variable of type t; e.g. $c!x!x$, and $c?y{:}Y!y$ for Y not related to t are allowed, but $c?x{:}t!x$ is not.

Seq may be seen as a rather strong condition. However, in practice, almost all useful specification processes can be easily re-written to meet its requirements; we justify this below. However, this condition does place restrictions on the way the specifications are *expressed*. These restrictions will make the production of the semi-symbolic operational semantics easier, and also simplify subsequent proofs.

We assume sequentiality (assumption (ii)). When a process is not sequential, it can be rewritten into a sequential form using algebraic equivalences [Ros97]. Further, we forbid indexed choice operators, since (for finite choices) such indexed operators can always be replaced by binary ones. Note that this means that **Seq** processes are taken from processes with the following syntax:

$$P ::= STOP \mid \alpha \rightarrow P \mid P \,\Box\, P \mid P \sqcap P \mid P \triangleright P \mid \text{if } b \text{ then } P \text{ else } P \mid X.$$

Assumptions (iv)–(vi) have been introduced for technical reasons, to simplify the production of the semi-symbolic operational semantics. With the exception of assumption (vi), they do not reduce expressiveness.

Assumption (iv) 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 the distinguished one, and predicates that involve only the distinguished type, combined together using conjunction and disjunction. The conjunctions and disjunctions can be eliminated using the laws:

$$\begin{aligned} \text{if } P \vee P' \text{ then } Q \text{ else } R &\equiv \text{if } P \text{ then } Q \text{ else (if } P' \text{ then } Q \text{ else } R), \\ \text{if } P \wedge P' \text{ then } Q \text{ else } R &\equiv \text{if } P \text{ then (if } P' \text{ then } Q \text{ else } R) \text{ else } R. \end{aligned}$$

Hence any process can be rewritten to satisfy assumption (iv).

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

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

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

Assumption (vi) ensures that values of all outputs of type t have to be previously stored within a process's memory. This simplifies the semantics, and does not greatly reduce expressiveness.

Thus, most processes can be rewritten into a form that satisfies **Seq**.

3.3. **The SeqNorm condition.** When working with specification processes, it is desirable to ensure their clarity and conformance to a certain standard (normality) to make analyses of their behaviours easier. The **SeqNorm** condition, defined below, achieves this without a major expressiveness reduction. Its effect is to remove all nondeterminism whose effect is not immediately observable. In particular this condition will allow us to deduce that a process reaches a unique state after a particular trace (Proposition 5.3), and that a unique construct gives rise to each event following a given trace (Proposition 5.5).

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

$$\begin{aligned}
\mathit{Channels}(STOP) &\mathrel{\widehat{=}} \{\},\\
\mathit{Channels}(c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \to P) &\mathrel{\widehat{=}} \{c\},\\
\mathit{Channels}(P \mathbin{\Box} Q) &\mathrel{\widehat{=}} \mathit{Channels}(P) \cup \mathit{Channels}(Q),\\
\mathit{Channels}(P \sqcap Q) &\mathrel{\widehat{=}} \mathit{Channels}(P) \cup \mathit{Channels}(Q),\\
\mathit{Channels}(P \rhd Q) &\mathrel{\widehat{=}} \mathit{Channels}(P) \cup \mathit{Channels}(Q),\\
\mathit{Channels}(X) &\mathrel{\widehat{=}} \mathit{Channels}(P), \qquad \text{if } E(X) = P.
\end{aligned}$$

Definition 3.5. A process syntax $\mathit{Proc}(t)$ satisfies **SeqNorm** if it satisfies **Seq**, and in addition for all external choices $P(t) \mathbin{\Box} Q(t)$, internal choices $P(t) \sqcap Q(t)$ and sliding choices $P(t) \rhd Q(t)$ within $\mathit{Proc}(t)$ we have that

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

Our definition of **SeqNorm** is similar to definitions of **Norm** used in the CSP literature [Ros97, Laz99], except that it includes **Seq**, since we will always use **SeqNorm** with processes that satisfy **Seq**.

The first clause does restrict expressiveness. It bans processes such as $c!x \to P \sqcap c!y \to Q$. This is necessary to ensure that a unique construct gives rise to each event (after a given trace), and that a process reaches a unique state after a particular trace; for example, without this condition, the above process could perform the event $c.0$ resulting from either construct (assuming x and y have value 0), and could reach either state P or Q after this event.

If a particular process syntax fails **SeqNorm** because of the second subclause of clause (iv), then the following algebraic laws can be used to convert it to an equivalent process definition, satisfying this subclause:

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

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

Thus, most processes can be rewritten into a form that satisfies **SeqNorm**. (A similar observation about the related **Norm** condition is made in [Ros97, Section 15.2].) Indeed, we are not aware of any specification used in practice that cannot.

3.4. **Type symmetry.** In Section 3.1 we defined the concept of data independence which, undoubtedly, is a very useful property for studying parameterised systems [CR98, RB99, CR99, Low04, RLN04]. 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. A process syntax satisfies the **TypeSym** condition if the behaviours of all its concretisations are invariant under permutations of values of parameter instantiations. Given such a permutation π, we write $[\![\pi]\!]$ for the renaming $[\![{}^{\pi(e)}/_{e} \mid e \in \Sigma]\!]$.

Definition 3.6. A process syntax $Proc(t)$ satisfies the condition **TypeSym** if $Proc(T)$ and $Proc(T)[\![\pi]\!]$ are bisimilar for every T and every bijection $\pi : T \to T$.

Example 3.7. Consider the process

$$COPY(t) \quad = \quad in?x{:}t \to out!x \to COPY(t).$$

This satisfies **TypeSym**, informally because it treats all elements of t the same. More formally, given an instantiation T of t, and a bijection $\pi : T \to T$, the relevant bisimulation is

$$\{(COPY(t), COPY(t)[\![\pi]\!])\} \cup$$
$$\{(out!v \to COPY(t),\ (out!\pi^{-1}(v) \to COPY(t))[\![\pi]\!]) \mid v \in T\}.$$

Example 3.8. Consider a system of N (where $N = \#T$) nodes that communicate using a ring topology, where each node i can send messages only to the node $(i + 1)$ mod N. For example (rather trivially):

$$\begin{aligned} \mathcal{N}_i(t) \quad &= \quad send!i!i \oplus 1 \to \mathcal{N}_i(t) \ \Box\ send!i \ominus 1!i \to \mathcal{N}_i(t), \\ Nodes(t) \quad &= \quad \Big\|\ i \in t \bullet [\{send.i.i \oplus 1, send.i \ominus 1.i\})]\ \mathcal{N}_i(t), \end{aligned}$$

where $\oplus$ and $\ominus$ represent addition and subtraction mod N. This does *not* satisfy **TypeSym**, which insists that the process is *fully* symmetric. For example, if $T = \{0..3\}$ then $Nodes(T)$ has trace $\langle send.1.2\rangle$, but does not have the trace $\langle send.1.3\rangle$, so **TypeSym** does not hold for $\pi = \{0 \mapsto 2, 1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 0\}$.

Semantic definitions, like Definition 3.6, tend to be hard to check efficiently, so we note here sufficient syntactic conditions for **TypeSym**.

Proposition 3.9. *A process syntax $Proc(t)$ satisfies the condition **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 a parallel composition indexed over t; and*
(v) *conditional choices on t, except for equality and inequality tests.*

Note that the process of Example 3.7 satisfies the conditions of this proposition, but the process of Example 3.8 does not, because arithmetic operations are applied to type t.

Proof sketch. 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 the conditions of the proposition.

We use $[\Gamma]$ to denote the syntactic substitution $[\Gamma(x)/x \mid x \in \mathrm{dom}(\Gamma)]$ and $FV(P(t))$ to denote the free variables of $P(t)$. Then

$$\mathcal{B} \;=\; \left\{ \left(P(T)[\Gamma], (P(T)[\pi^{-1}(\Gamma)])[\![\pi]\!] \right) \mid P(t) \in CSP_t, \Gamma \in FV(P(t)) \to T \right\}$$

is the required strong bisimulation relation. The proof is a structural induction on $P(t)$. □

In practice, most systems where the nodes communicate using a fully connected topology satisfy these conditions. In [Mof10], Moffat proves a very similar result, for a larger fragment of the machine-readable CSP language, including the underlying functional language.

The syntactic definition of data independence (Definition 3.1) comprises a superset of the requirements of Proposition 3.9, so we immediately have the following result.

Corollary 3.10. *Every data independent process satisfies* ***TypeSym****.*

Example 3.11. Consider a system built as the parallel composition of node processes $\mathcal{N}_i(t)$ for each $i \in t$:

$$Nodes(t) \;=\; \|\, i \in t \bullet [A(i,t)]\, \mathcal{N}_i(t).$$

This process syntax satisfies **TypeSym** provided:

- the node process $\mathcal{N}_i(t)$ satisfies the conditions of Proposition 3.9, so in particular it treats its "identity" parameter i polymorphically; informally, different nodes need to be identical up to renaming of the identities;
- the alphabet $A(i,t)$ satisfies the conditions of Proposition 3.9, so in particular no operations on type t are applied; informally, the different alphabets depend only on the identities i, polymorphically.

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 $C_t[.]$ that composes its argument with a controller process $Ctrl_t$ and hides some events:

$$C_t[X] \;=\; (X \underset{A_t}{\parallel} Ctrl_t) \setminus B_t$$

then $C_t[Nodes(t)]$ satisfies **TypeSym** provided:

- the controller process $Ctrl_t$ satisfies the conditions of Proposition 3.9; informally, it needs to treat different nodes in the same way;
- the sets A_t and B_t satisfy the conditions of Proposition 3.9.

Recall that this is the type of implementation process that we considered in the Introduction. In particular, the example from Section 2.3 meets this pattern.

Example 3.12. The process $\Box\, y{:}t \bullet c?x{:}(t \setminus \{y\})!y \to STOP$ satisfies **TypeSym**. However, it does not satisfy the conditions of Proposition 3.9, in particular because x is selected from a proper subset of t.

The following remark is a direct consequence of the **TypeSym** condition.

Remark 3.13. Suppose that $Proc(t)$ satisfies **TypeSym**. Then, for all T:

- If $tr \in traces(Proc(T))$ then for all bijections $\pi : T \to T$, $\pi(tr) \in traces(Proc(T))$;
- If $(tr, X) \in failures(Proc(T))$ then for all bijections $\pi : T \to T$, $(\pi(tr), \pi(X)) \in failures(Proc(T))$.

3.5. **Equality tests.** The syntactic condition **PosConjEqT**, formulated by Lazić in [LR98, Chapter 3], specifies that for a conditional choice with an equality test on t, the positive branch is a prefix and the negative branch is simply $STOP$. In [Ros98, RB99], a weaker version of **PosConjEqT** is discussed, where no restriction on the process in the positive branch is in place. Both of these definitions talk about the condition only in relation to the traces model, but it is easy to extend it to other models of CSP as the following definition shows.

Definition 3.14. Given a CSP model $\mathcal{M}$ we say that a process syntax $Proc(t)$ satisfies $\mathbf{PosConjEqT}_{\mathcal{M}}$ if for every conditional choice on t of the form

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

within $Proc(t)$, we have that

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

For technical reasons, it will be desirable in our work to assume the opposite condition for specifications: that every positive branch of a conditional choice is a refinement of the negative branch. This can be viewed as a reversed version of $\mathbf{PosConjEqT}_{\mathcal{M}}$. Hence, we have the following definition.

Definition 3.15. Given a CSP model $\mathcal{M}$ we say that a process syntax $Proc(t)$ satisfies $\mathbf{RevPosConjEqT}_{\mathcal{M}}$ if for every conditional choice on t of the form

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

within $Proc(t)$, we have that

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

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

Example 3.16. The process syntax

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

satisfies $\mathbf{RevPosConjEqT}_{\mathrm{F}}$. However, the process syntax

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

does not satisfy $\mathbf{RevPosConjEqT}_{\mathrm{T}}$, because if x and y are two distinct values, then $out.y \to STOP \not\sqsubseteq_{\mathrm{T}} out.x \to STOP$.

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

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. In this section we look into the operational semantics for CSP. An operational semantics provides a precise step-by-step description of how processes execute. It describes state changes as effects of events being performed by representing processes using *labelled transition systems*, defined as follows.

Definition 4.1. A *labelled transition system (LTS)* is a tuple $\mathcal{L} = (S, s_0, L, \longrightarrow)$, where S is a set of states, $s_0 \in S$ is an initial state, L is a set of labels, and $\longrightarrow \subseteq S \times L \times S$ is a transition relation. We let $\hat{\mathcal{L}} = S$ denote the set of states of $\mathcal{L}$.

In Section 4.1 we give some useful notation and definitions that we will repeatedly use throughout the rest of this paper.

The various operational semantics we present in this paper do not aim to be complete. Their main purpose is to formalise the foundations for the results regarding specifications, presented in Section 6. 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 section. We therefore restrict ourselves to processes that satisfy **Seq** throughout this section. As noted above, 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.

Operational semantics can be defined at different levels of abstraction. In Section 4.2 we present a fairly standard operational semantics at the lowest, implementation level. It generates LTSs from process syntax with no free variables. This means that all parameters must be substituted with concrete values before the transition rules can be used. When variables become bound as the result of inputs or nondeterministic selections, the binding is reflected by syntactic substitution.

We introduce a running example, which we use to illustrate the different styles of operational semantics.

Example 4.2. Let

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

In Figure 1 we represent the standard operational semantics for $P(T)$ where $T = \{0, 1\}$. 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 $d!x \rightarrow \mathit{STOP}$ else STOP. For compatibility with the later semantics, we choose to resolve all non-type-t nondeterministic selections before the type-t nondeterministic selections: hence the τ transitions from the initial states correspond to resolving the "$\$x{:}\{a, b\}$" selection, and the τ transitions from the subsequent states correspond to resolving the "$\$y{:}t$" selection. The transitions labelled with events on channel c also have the effect of resolving the subsequent conditional ("if $y = z \ldots$").

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: this is reflected in Figure 1, where the number of transitions depends on the size of T. 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,

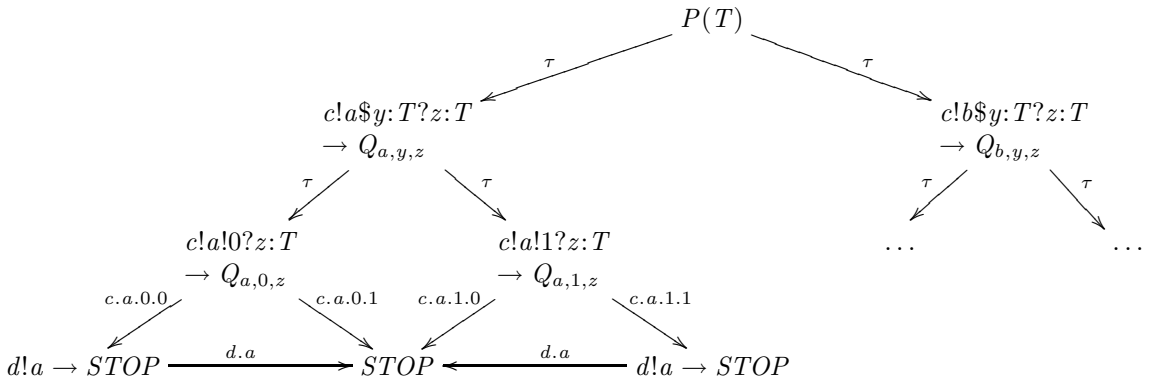

Figure 1: Operational semantics for $P(T)$ from Example 4.2 with $T = \{0, 1\}$.

given a parameterised process syntax, a single symbolic LTS is generated and each of the concrete LTSs can be easily obtained from it by an assignment of values[4]. 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 Section 6. 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.

In Section 4.3 we define *Semi-Symbolic Operational Semantics* (SSOS), a symbolic operational semantics similar to the one from [Laz99]. We explain the idea of SSOS via our running example.

Example 4.3. Recall the following process from Example 4.2:

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

In Figure 2 we represent the semi-symbolic operational semantics for $P(t)$. We again write $Q_{x,y,z}$ as a shorthand for if $y = z$ then $d!x \rightarrow STOP$ else $STOP$. Note that transitions are *symbolic* in that they contain variables corresponding to type-t selections; however, non-type-t values are treated concretely. Further, we include transitions corresponding to the conditional, labelled with the condition ("$y = z$") and its negation ("$\neg y = z$") respectively.

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 giving values to the type-t free variables, 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 illustrate the idea via our running example.

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

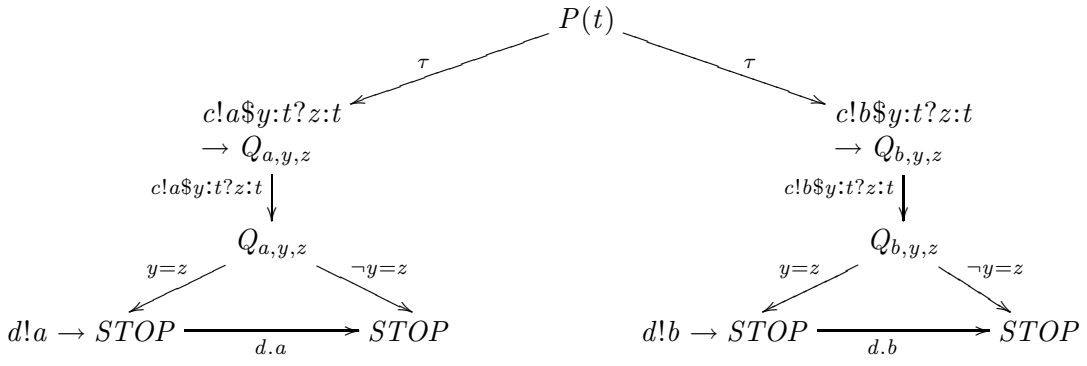

Figure 2: Semi-symbolic operational semantics for $P(t)$ from Example 4.3.

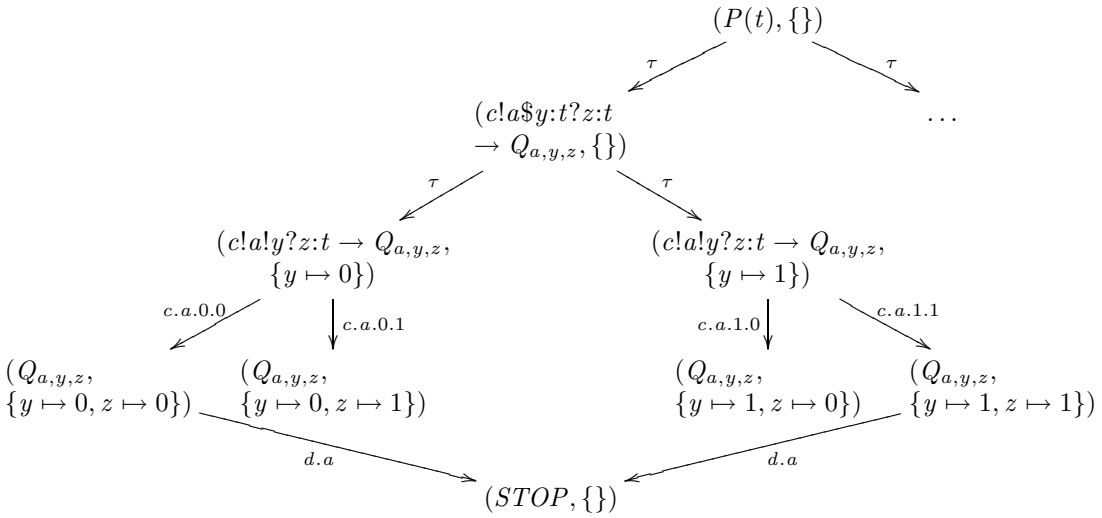

Figure 3: Concrete operational semantics with environments for $P(T)$ with $T = \{0, 1\}$.

Example 4.4. Recall the process $P(t)$ from above. Figure 3 gives the COSE semantics for $P(T)$ with $T = \{0, 1\}$ (strictly speaking, each state should also include T as a third term; we omit this due to lack of space). The τ transitions from the initial states correspond to resolving the "$\$x{:}\{a, b\}$" selection; since x is not of type t, the choice of x is reflected by syntactic substitution. The τ transitions from the subsequent states correspond to resolving the "$\$y{:}t$" selection; the environment stores the resulting value for y. The subsequent transitions with events on channel c resolve the "$?z{:}t$" choices; the environment stores the resulting value for z. Note also that the operational semantics is strongly bisimilar to the semantics in Figure 1.

We show that the combination of SSOS and the translation rules of COSE is always bisimilar to the standard one in Section 4.5. Finally, we define the relationship between symbolic traces and concrete traces in Section 4.6.

4.1. **General definitions and notation.** We 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 section.

We define $Value$ to be the set of all values, and Var to be the set of all variable names; we assume $Var \cap Value = \{\}$.

Prefix constructs may depend on the distinguished type, so, in theory, they should be decorated with parameter t, e.g. $Proc(t) = \alpha(t) \rightarrow Proc'(t)$. However, for brevity, we omit the parameter (or its instantiation) where it is clear from the context or indifferent.

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

$$\begin{aligned}
\$^t(\alpha) &\mathrel{\widehat{=}} \{i \in \{1\,..\,k\} \mid \S_i = \$ \wedge X_i = t\},\\
\$^{non\text{-}t}(\alpha) &\mathrel{\widehat{=}} \{i \in \{1\,..\,k\} \mid \S_i = \$ \wedge X_i \neq t\},\\
\$(\alpha) &\mathrel{\widehat{=}} \$^t(\alpha) \cup \$^{non\text{-}t}(\alpha),\\
?^t(\alpha) &\mathrel{\widehat{=}} \{i \in \{1\,..\,k\} \mid \S_i = ? \wedge X_i = t\},\\
?^{non\text{-}t}(\alpha) &\mathrel{\widehat{=}} \{i \in \{1\,..\,k\} \mid \S_i = ? \wedge X_i \neq t\},\\
?(\alpha) &\mathrel{\widehat{=}} ?^t(\alpha) \cup ?^{non\text{-}t}(\alpha),\\
!^t(\alpha) &\mathrel{\widehat{=}} \{i \in \{1\,..\,k\} \mid \S_i = ! \wedge x_i \text{ is of type } t\},\\
!^{non\text{-}t}(\alpha) &\mathrel{\widehat{=}} \{i \in \{1\,..\,k\} \mid \S_i = ! \wedge x_i \text{ is not of type } t\},\\
!(\alpha) &\mathrel{\widehat{=}} !^t(\alpha) \cup !^{non\text{-}t}(\alpha).
\end{aligned}$$

The following functions allow us to modify constructs. Let $\alpha = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and let $\dagger$ be either t or *non-t*.

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

Example 4.5. Let $\epsilon = c\$x_1{:}t?x_2{:}t\$x_3{:}X!x_4$, where X it a type not related to t and x_4 is some output variable. Then,

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

Substitution will play an important role in defining the operational semantics in the following sections. 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.

4.2. **Standard CSP operational semantics.** In this section we present a standard operational semantics for the fragment of CSP corresponding to **Seq**, i.e. excluding parallel operators, hiding and renaming. (Rules for the remainder of the syntax can be found in, e.g., [Ros97], but we do not need them in this paper.) The operational semantics 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. We let T be a fixed instantiation of type t.

We distinguish two types of transitions: *visible* and *internal*. An internal transition, labelled with τ, represents an event that can be performed by a process without any interaction from the environment and which is not observable by the environment. A visible transition, on the other hand, is labelled with an event that is observable by the environment, and requires its synchronisation in order to be performed. We write $P \xrightarrow{a} Q$ to mean that there is an a-labelled transition from state P to state Q.

4.2.1. *Transition rules.* Most of the transition rules are given in Figure 4, and are standard. We concentrate our discussion on the semantics for prefixing; a slightly non-standard treatment is required due to the addition of nondeterministic selections.

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

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

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

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

$$\frac{P(T) \xrightarrow{a} P'(T)}{P(T) \triangleright Q(T) \xrightarrow{a} P'(T)} \big[\, a \neq \tau \,\big] \qquad \frac{E(X) = P}{X(T) \xrightarrow{\tau} P(T)}$$

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

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

Figure 4: Operational semantic rules for the choice operators and binding

Let α be a construct of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$. 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.

Prefix Rule 1. (Prefixes with no nondeterministic selections)

$$\frac{}{\alpha \to P(T) \xrightarrow{c.v_1 \ldots v_k} P(T)[v_i/x_i \mid i \in ?(\alpha)]} \big[\, c.v_1 \ldots v_k \in Comms(\alpha) \wedge \#\$(\alpha) = 0 \,\big]$$

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

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

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. For reasons that will become clear later, we simultaneously resolve all nondeterministic selections over types other than t before simultaneously resolving all nondeterministic selections over type t.

The following rule resolve all nondeterministic selections over types other than t, replacing each variable $x_i \in \$^{non\text{-}t}(\alpha)$ with an appropriate value $v_i \in X_i$. (Here and subsequently we treat subscripting as functional application, i.e. v_i is the result of applying function v to index i.)

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

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

The following rule then resolve all nondeterministic selections over t, replacing each variable $x_i \in \$^t(\alpha)$ with an appropriate value $v_i \in T$.

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

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

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

$$\bigsqcap \langle x_i{:}X_i \mid i \in \$^{non\text{-}t}(\alpha)\rangle \bullet \left(\bigsqcap \langle x_i{:}T \mid i \in \$^t(\alpha)\rangle \bullet Replace_{\$\mapsto !}(\alpha) \to P(T)\right),$$

where we use $\bigsqcap \langle x_i{:}X_i \mid i \in \mathcal{I}\rangle \bullet P(x_{i_1}, \ldots, x_{i_n})$ as shorthand for $\bigsqcap (x_{i_1}, \ldots, x_{i_n}) \in X_{i_1} \times \cdots \times X_{i_n} \bullet P(x_{i_1}, \ldots, x_{i_n})$.

Example 4.6. Recall our earlier running example:

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

Figure 1 represents the operational semantics for $P(T)$ where $T = \{0, 1\}$. The first τ transitions correspond to Prefix Rule 2a; the second τ transitions correspond to Prefix Rule 2b; the visible transitions correspond to Prefix Rule 1.

4.2.2. *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 three definitions (from [Ros97, Chapter 7]).

- Given two states $P(T)$ and $Q(T)$, and a sequence of events (visible or invisible) $s = \langle a_i \mid i \in \{1 \mathinner{..} n\}\rangle$ for some $n \geq 0$, we write $P(T) \stackrel{s}{\longmapsto} Q(T)$ if there exist states $P_0(T) = P(T), P_1(T), \ldots, P_n(T) = Q(T)$ such that for all i in $\{0 \mathinner{..} n-1\}$ we have that $P_i(T) \stackrel{a_{i+1}}{\longrightarrow} P_{i+1}(T)$.
- We write $P(T) \stackrel{tr}{\Longrightarrow} Q(T)$ if there is s such that $P(T) \stackrel{s}{\longmapsto} Q(T)$ and tr is the restriction of s to visible events.
- We say that $Q(T)$ refuses X, written $Q(T)\ ref\ X$, if $Q(T)$ cannot perform τ (i.e. it's stable) and cannot perform any event from X: $\forall\, x \in X \cup \{\tau\} \bullet Q(T) \stackrel{x}{\not\longrightarrow}$.

Using the above, we have

$$\begin{aligned} traces(P(T)) &= \{tr \in \Sigma^* \mid \exists\, Q(T) \bullet P(T) \stackrel{tr}{\Longrightarrow} Q(T)\}, \\ failures(P(T)) &= \{(tr, X) \in \Sigma^* \times \Sigma \mid \exists\, Q(T) \bullet P(T) \stackrel{tr}{\Longrightarrow} Q(T) \wedge Q(T)\ ref\ X\}. \end{aligned}$$

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. 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. Its aim, however, is not to be used to perform abstract refinement checking of processes. Given a process syntax $Proc(t)$, it generates a single structure, which acts as a bridge between the different processes obtained from $Proc(t)$ by substituting different concrete values for the parameter t. This will allow us, in Section 6, to use known behaviours of a given instance of the process to deduce facts about the behaviours of other instances of the same process definition. In our work we will apply this operational semantics only to specification processes.

One of the main characteristics of the symbolic operational semantics defined in this section is that only the parts of systems that involve type t are left in their symbolic form. All other components are instantiated in a way similar to that used in the standard operational semantics (Section 4.2). Therefore, the labels of transitions may contain some symbolic parts and some concrete parts. This is why we call any resulting transition graph a *semi-symbolic labelled transition system* (*SSLTS*), and call this operational semantics *Semi-Symbolic Operational Semantics* (*SSOS*).

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

4.3.1. *Symbolic transitions.* In order to be able to tell symbolic and standard transitions apart, the symbolic transition relation is denoted by $\longrightarrow_s$, i.e. $P(t) \stackrel{\alpha}{\longrightarrow}_s Q(t)$ denotes that there is an α-labelled transition from symbolic state $P(t)$ to symbolic state $Q(t)$.

We distinguish the following three types of symbolic transitions.

Internal: The internal symbolic transitions, labelled τ, are in a direct correspondence with the standard internal transitions.

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 (e.g. $\$x{:}t$), deterministic inputs of type t (e.g. $?x{:}t$), outputs of type t (e.g. $!x$, where x is a variable of type t), or outputs of non-t parts (e.g. $!v$ where v is a value not of type t).

Formally, each visible symbolic transition is labelled with a *visible symbolic event*, a construct of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$, where

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

For example, the process $c!a?x{:}t\$y{:}t \rightarrow STOP$ has an initial symbolic transition with label $c!a?x{:}t\$y{:}t$. We let $Visible$ denote the set of all visible symbolic events.

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

Remark 4.7. If ϵ is a visible symbolic event, then $\$^{non\text{-}t}(\epsilon) = ?^{non\text{-}t}(\epsilon) = \{\}$.

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

4.3.2. *Transitions rules.* We define the Semi-Symbolic Operational Semantics using the inference rules below. Recall that we are considering only processes that satisfy **Seq**; therefore, we only provide transition rules for operators that the condition allows.

We begin with prefixing. Let α be a construct of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$. There are two transition rules for prefix. The first one defines the initial symbolic events of $\alpha \rightarrow P(t)$ in the case when α contains no nondeterministic selections over types other than t. It is similar to Prefix Rule 1 from the standard operational semantics (see Section 4.2.1), except that variables of type t are left in their symbolic form when a transition label is obtained from α, so only deterministic selections over types other than t are resolved.

Symbolic Prefix Rule 1.

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

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

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

The second transition rule of prefix deals with prefixes that contain at least one nondeterministic selection over a type other than t. It is similar to Prefix Rule 2a from the standard operational semantics (see Section 4.2.1). All the nondeterministic selections over types other than t are resolved simultaneously, the act of which generates a single τ transition. The values v_i chosen are substituted for the variables x_i of the selections.

Symbolic Prefix Rule 2.

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

The transition rules for external, internal and sliding choice and for binding are very similar to the standard rules, and are given in Figure 5. One exception is the presence of conditional symbolic transitions, which need to be taken into considerations here. Conditional choices must be resolved without any other influence on the overall state of the system, which means that the members of $Cond$ must be promoted by the $\Box$ and $\triangleright$ operators in the same way τ's are.

Clause (iv) of the definition of **Seq** (Definition 3.4) implies that guards of conditional choices may not contain both variables of type t and variables of non-t types. The truth of any boolean condition that contains no variables of type t (i.e. every conditional not in $Cond$) can be fully evaluated at the time of SSLTS generation. Hence we have the following rules, similar to the standard rules.

$$\frac{P(t) \xrightarrow{\alpha}_s P'(t)}{\mathrm{if}\, True \,\mathrm{then}\, P(t) \,\mathrm{else}\, Q(t) \xrightarrow{\alpha}_s P'(t)} \qquad \frac{Q(t) \xrightarrow{\alpha}_s Q'(t)}{\mathrm{if}\, False \,\mathrm{then}\, P(t) \,\mathrm{else}\, Q(t) \xrightarrow{\alpha}_s Q'(t)}$$

Every conditional choice with a boolean condition *cond* that involves type t (i.e. every conditional in $Cond$) may either evolve to the positive branch by following a conditional transition labelled with *cond* or it may evolve to the negative branch by following a conditional transition labelled with the negation of *cond*.

$$\frac{}{\mathrm{if}\, cond \,\mathrm{then}\, P(t) \,\mathrm{else}\, Q(t) \xrightarrow{cond}_s P(t)} \left[\, cond \in Cond \,\right]$$

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

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

$$\frac{P(t) \xrightarrow{\epsilon}_s P'(t)}{P(t) \,\Box\, Q(t) \xrightarrow{\epsilon}_s P'(t)} \big[\, \epsilon \in Visible \,\big] \qquad \frac{Q(t) \xrightarrow{\epsilon}_s Q'(t)}{P(t) \,\Box\, Q(t) \xrightarrow{\epsilon}_s Q'(t)} \big[\, \epsilon \in Visible \,\big]$$

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

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

$$\frac{P(t) \xrightarrow{\epsilon}_s P'(t)}{P(t) \triangleright Q(t) \xrightarrow{\epsilon}_s P'(t)} \big[\, \epsilon \in Visible \,\big] \qquad \frac{E(X) = P}{X(t) \xrightarrow{\tau}_s P(t)}$$

Figure 5: Semi-symbolic operational semantics rules for external, internal and sliding choice, and for binding

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

Example 4.8. Recall our running example:

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

Figure 2 represents the semi-symbolic operational semantics for $P(t)$. The τ transitions correspond to Symbolic Prefix Rule 2; the visible transitions correspond to Symbolic Prefix Rule 1.

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 2.2), except they contain visible symbolic events instead of ordinary visible events, and may contain both conditional and τ symbolic events.

Formally, we define a symbolic trace as follows. Let $\mathcal{S} = (S, s_0, L, \longrightarrow_s)$ be the SSLTS obtained by applying the SSOS to process syntax $Proc(t)$. Given two symbolic states $P(t)$ and $Q(t)$ in S and a sequence of symbolic events $\sigma = \langle \alpha_i \mid i \in \{1 \,..\, n\} \rangle$, we write $P(t) \overset{\sigma}{\longmapsto}_s Q(t)$ to mean that there exist symbolic states $P_0(t) = P(t), P_1(t), \ldots, P_n(t) = Q(t)$ such that for all i in $\{0 \,..\, n-1\}$ $P_i(t) \xrightarrow{\alpha_{i+1}}_s P_{i+1}(t)$; σ is called a *symbolic trace* of $P(t)$. Therefore, a symbolic trace of $Proc(t)$ is a sequence of labels of symbolic events that form a path, starting at s_0, through $\mathcal{S}$. We let $SymbolicTraces(Proc(t))$ denote the set of all symbolic traces of $Proc(t)$. Observe that symbolic traces are quite different from standard traces as they may contain symbolic τ events and conditional symbolic events, while ordinary

traces contain only visible events. In Section 4.6 we will study the relationship between symbolic and concrete traces in more detail. We will usually use σ, ρ and their derivatives (σ', ρ_1, etc.) to denote symbolic traces.

In Section 6 we will work with symbolic traces that are "similar" in the sense that their restrictions to conditional and visible symbolic events are identical.

Definition 4.9. Let σ and σ' be two symbolic traces. Then σ and σ are *non-τ equivalent*, written $\sigma \equiv_{non\text{-}\tau} \sigma'$, if $\sigma \setminus \{\tau\} = \sigma' \setminus \{\tau\}$.

4.4. **Concrete Operational Semantics with Environments.** So far we have presented 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 different from the environment with which processes communicate, or the global map E of identifiers to process definitions that we introduced in Section 2.1. Intuitively, environments map free variables within process syntaxes to concrete values that were previously bound to such variables through inputs.

Definition 4.10. Let $Env(t) \mathrel{\widehat{=}} Var \nrightarrow t$. Then an *environment* is a partial function $\Gamma \in Env(T)$ for some instantiation T of type t.

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

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

Note that for all environments Γ, all symbolic events α, and $\dagger \in \{t, \textit{non-}t\}$, we have that $\$^\dagger(\Gamma(\alpha)) = \$^\dagger(\alpha)$ and $?^\dagger(\Gamma(\alpha)) = ?^\dagger(\alpha)$, which means that $\$(\Gamma(\alpha)) = \$(\alpha)$ and $?(\Gamma(\alpha)) = ?(\alpha)$.

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

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

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

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

- $P(t)$ is a symbolic state, equal to or slightly modified from a state of the SSLTS $\mathcal{S}$ of $Proc(t)$,
- Γ is an environment in $Env(T)$, 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. Whenever the concrete type T is clear from the context or indifferent, we omit it from the configurations and use pairs $(P(t), \Gamma)$.

The initial state of $\mathcal{C}$ is defined to be the configuration $(P_0(t), \{\}, T)$, where $P_0(t)$ is the initial state of $\mathcal{S}$; we sometimes abbreviate this as $P_0(T)$. To emphasise the fact that COSE is a concrete operational semantics we denote the transition relation using the same symbol ($\longrightarrow$) that we used in Section 4.2.

We treat two configurations as identical if they describe exactly the same process. Formally, $(P(t), \Gamma, T) = (P'(t), \Gamma', T')$ if and only if $P(t)[\Gamma] \equiv_\alpha P'(t)[\Gamma']$ and $T = T'$, where $\equiv_\alpha$ denotes operational semantics alpha-equivalence, i.e. equality of operational semantics modulo renaming of bound variables.

Remark 4.11. Observe that[5] $P(t)[\Gamma] = P(t)[FV(P(t)) \lhd \Gamma]$ for all symbolic states $P(t)$ and all environments Γ, where $FV(P(t))$ denotes the free variables of $P(t)$. Therefore, configurations $(P(t), \Gamma, T)$ and $(P(t), FV(P(t)) \lhd \Gamma, T)$ are identical. From now on we always assume environment minimality within configurations, which we achieve by restricting the environment Γ of every configuration $(P(t), \Gamma, T)$ to the free variables of $P(t)$.

4.4.1. *Translation rules.* We present the specification of COSE using translation rules that translate transitions within SSLTSs into corresponding COSE transitions. Let T be a fixed instantiation of the distinguished type parameter t.

Given a symbolic state $P(t)$, we let $Q(t) = \mathit{Replace}^t_{\$\mapsto!}(c, P(t))$ be a symbolic state like $P(t)$, except every transition from $P(t)$ labelled with a visible symbolic transition ϵ on channel c is replaced with an identical transition in $Q(t)$, but labelled with $\mathit{Replace}^t_{\$\mapsto!}(\epsilon)$ instead, i.e. if $P(t) \xrightarrow{\epsilon}_s P'(t)$ then $Q(t) \xrightarrow{\mathit{Replace}^t_{\$\mapsto!}(\epsilon)} P'(t)$. We will see later (Proposition 5.5) that all such transitions over c result from the same construct.

Visible symbolic events that contain a nondeterministic selection over type t are translated into two concrete events: a τ that resolves the nondeterminism; and a subsequent visible event. The first translation rule shows how the τ is produced; for each nondeterministically chosen variable x_i in the symbolic event, the environment is updated to map x_i to some value v_i, and the nondeterministic choice in the subsequent symbolic event is replaced by an output (to be dealt with later).

Translation Rule 1.

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

Clause (v) of the definition of **Seq** (Definition 3.4) implies that there 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.

[5] $S \lhd \Gamma$ denotes Γ restricted to domain S, i.e. $\{x \mapsto y \mid (x \mapsto y) \in \Gamma \wedge x \in S\}$.

Example 4.12. Let $Proc(t) = c_1?x{:}t \to (c_2\$x{:}t?y{:}t \to STOP \;\Box\; c_1!x \to STOP)$ and $T = \{0, 1\}$. Then, after performing $c_1.0$ and a τ resolving the nondeterministic selection by choosing $x = 1$, the configuration $(Proc(t), \{\}, T)$ evolves to $(c_2!x?y{:}t \to STOP \;\Box\; c_1!x \to STOP, \{x \mapsto 1\}, T)$. Then, by Translation Rule 2 (see below), the event $c_1.1$ is available, which clearly should not be the case.

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

Translation Rule 2.

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

Example 4.13. Recall our running example

$$P(t) \;=\; c\$x{:}\{a, b\}\$y{:}t?z{:}t \to \text{if } y = z \text{ then } d!x \to STOP \text{ else } STOP$$

whose SSOS semantics appear in Figure 2. In particular, consider the transition

$$c!a\$y{:}t?z{:}t \to Q_{a,y,z} \quad \xrightarrow{c!a\$y{:}t?z{:}t}_s \quad Q_{a,y,z}. \tag{4.1}$$

Translation Rule 1 implies that configuration $(c!a\$y{:}t?z{:}t \to Q_{a,y,z}, \{\}, T)$, with $T = \{0, 1\}$, can do a τ and become either of

$$\begin{array}{rcl} conf_0 & = & (Replace^t_{\$\mapsto!}(c, c!a\$y{:}t?z{:}t \to Q_{a,y,z}), \{y \mapsto 0\}, T), \\ conf_1 & = & (Replace^t_{\$\mapsto!}(c, c!a\$y{:}t?z{:}t \to Q_{a,y,z}), \{y \mapsto 1\}, T). \end{array}$$

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

$$Replace^t_{\$\mapsto!}(c, c!a\$y{:}t?z{:}t \to Q_{a,y,z}) \quad \xrightarrow{c!a!y?z:t}_s \quad Q_{a,y,z}.$$

Hence, using Translation Rule 2, we can deduce

$$\begin{array}{rcl} conf_0 & \xrightarrow{c.a.0.0} & (Q_{a,y,z}, \{y \mapsto 0, z \mapsto 0\}, T), \\ conf_0 & \xrightarrow{c.a.0.1} & (Q_{a,y,z}, \{y \mapsto 0, z \mapsto 1\}, T); \end{array}$$

and similarly for $conf_1$. (In Figure 3, the process $Replace^t_{\$\mapsto!}(c, c!a\$y{:}t?z{:}t \to Q_{a,y,z})$ is written as $c!a!y{:}t?z{:}t \to Q_{a,y,z}$, for convenience.)

Remark 4.14. We can combine Translation Rules 1 and 2 to deduce that if

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

then

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

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

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

Translation Rule 3.

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

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

Translation Rule 4.

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

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

Example 4.15. Recall our running example

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

whose SSOS semantics appear in Figure 2. The COSE semantics is given in Figure 3. The initial τ-transitions follow from Translation Rule 3. The subsequent τ-transitions and transitions with events on c were explained in Example 4.13. The left-hand final transition with event $d.a$ follows from Translation Rule 4, noting that $[\![y = z]\!]_{\{y \mapsto 0, z \mapsto 0\}}$, and using the fact that $(d!a \rightarrow STOP, \{y \mapsto 0, z \mapsto 0\}, T) \xrightarrow{d.a} STOP$, by Translation Rule 2; other transitions on d follow similarly.

4.5. **Congruence of 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 4.16. (Congruence of COSE to the standard operational semantics.) *Suppose that Proc(t) is some process syntax that satisfies **Seq**. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the LTSs generated from Proc(t), for some fixed instantiation T of type t, using COSE and the standard operational semantics, respectively. Then $\mathcal{L}_1$ and $\mathcal{L}_2$ are strongly bisimilar.*

Proof sketch. By showing that

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

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

One implication of Theorem 4.16 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; so:

$$\begin{aligned} traces(Proc(t), \Gamma, T) &= traces(Proc(T)[\Gamma]), \\ failures(Proc(t), \Gamma, T) &= failures(Proc(T)[\Gamma]), \end{aligned}$$

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

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

Given a process syntax $Proc(t)$ and an instantiation T of type t, we define a relation *generates* written using infix notation: $\sigma\ generates_\Gamma\ tr$, for a symbolic trace σ, an environment Γ and a concrete trace tr:

(i) $\langle\rangle\ generates_\Gamma\ \langle\rangle$,
(ii) $\sigma\ generates_\Gamma\ tr \Leftrightarrow \langle\tau\rangle\hat{}\sigma\ generates_\Gamma\ tr$,
(iii) $\sigma\ generates_\Gamma\ tr \wedge [\![cond]\!]_\Gamma \Leftrightarrow \langle cond\rangle\hat{}\sigma\ generates_\Gamma\ tr$,
(iv) $e \in Insts_\Gamma(\epsilon) \wedge \sigma\ generates_{\Gamma \oplus Match(\epsilon,e)}\ tr \Leftrightarrow \langle\epsilon\rangle\hat{}\sigma\ generates_\Gamma\ \langle e\rangle\hat{}tr$,

where $Insts_\Gamma(\epsilon)$ gives all instantiations of ϵ consistent with Γ, and $Match(\epsilon, e)$ gives the extension to the environment caused by instantiating ϵ with e; let ϵ be of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$, and recall that, by Remark 4.7, $\$^{non\text{-}t}(\epsilon) = ?^{non\text{-}t}(\epsilon) = \{\}$:

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

Observe that rule (ii) indicates that we essentially ignore any τ'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 4.17. A visible event e that is available in $Proc(T)$ immediately after a trace tr *matches* a visible symbolic event ϵ if there exists a symbolic trace σ such that $\sigma\hat{}\langle\epsilon\rangle$ is in $SymbolicTraces(Proc(t))$ and $\sigma\hat{}\langle\epsilon\rangle\ generates_{\{\}}\ tr\hat{}\langle e\rangle$.

5. Regularity results

In this section we present a series of regularity results that are consequences of the **SeqNorm** condition. These results show that specifications exhibit certain clarity in their behaviour. Our main findings can be summarised as follows:

(1) 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 (Proposition 5.3);

(2) There is no ambiguity which construct gives rise to a given concrete event that is available after a given trace (Proposition 5.5); and
(3) 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 5.7).

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

Example 5.2. Let

$$P(t) \quad = \quad c?x{:}t \rightarrow STOP \,\Box\, c\$x{:}t \rightarrow c.x \rightarrow STOP.$$

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).

Note that the process in the above example does not satisfy **SeqNorm**. We will show (in Proposition 5.5) that for processes that do satisfy **SeqNorm**, each event (after a given trace) matches a *unique* construct.

An important property of normality is the lack of ambiguity about what state a process reaches after performing a sequence of visible concrete events not followed by a τ. The following proposition establishes this formally.

Proposition 5.3. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Suppose further that*

$$(Proc(t), \Gamma_{init}) \stackrel{s}{\longmapsto} (P(t), \Gamma) \quad \textit{and} \quad (Proc(t), \Gamma_{init}) \stackrel{s'}{\longmapsto} (Q(t), \Gamma'),$$

where s, s' do not end with a τ, and $s \setminus \tau = s' \setminus \tau$. Then $P(t) = Q(t)$ and $\Gamma = \Gamma'$.

Further, **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 5.4. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Then, if σ, σ' are symbolic traces of $Proc(t)$ such that σ generates$_\Gamma$ tr and σ' generates$_\Gamma$ tr, and either $\sigma = \sigma' = \langle\rangle$ or both σ and σ' end in a visible symbolic event, then $\sigma \equiv_{non\text{-}\tau} \sigma'$.*

For a process that satisfies **SeqNorm**, for each visible event that is performed after a given trace, there is never any ambiguity what construct this event matches.

Proposition 5.5. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Then, if $tr\,\hat{}\,\langle e\rangle$ is a trace of $(Proc(t), \Gamma, T)$ for some type T and some environment Γ, and e matches constructs α and α', then $\alpha = \alpha'$.*

The following lemma compares corresponding constructs in two processes, one of which refines the other.

Lemma 5.6. *Suppose that $P(t)$ and $Q(t)$ satisfy **SeqNorm**. Let T be an instantiation of type t. Suppose that $(Q(t), \Gamma_{init}, T) \sqsubseteq_{\mathrm{T}} (P(t), \Gamma_{init}, T)$. Then for all visible symbolic events ϵ, ϵ', symbolic traces σ, σ' and traces tr such that:*

(i) $tr\,\hat{}\,\langle e\rangle \in traces(P(t), \Gamma_{init}, T)$;
(ii) $\sigma\,\hat{}\,\langle\epsilon\rangle \in SymbolicTraces(P(t))$ $generates_{\Gamma_{init}}$ $tr\,\hat{}\,\langle e\rangle$; *and*
(iii) $\sigma'\,\hat{}\,\langle\epsilon'\rangle \in SymbolicTraces(Q(t))$ $generates_{\Gamma_{init}}$ $tr\,\hat{}\,\langle e\rangle$;

we have that $!(\epsilon') \subseteq !(\epsilon)$.

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 5.8 then extends this observation to traces.

Proposition 5.7. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Let T and $\hat{T}$ be instantiations of type t such that $\hat{T} \subseteq T$ and let Γ be an environment in $Env(T)$. Then, if*

$$(Proc(t), \Gamma, \hat{T}) \xrightarrow{a} (Proc'(t), \Gamma', \hat{T}) \tag{5.1}$$

then

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

Corollary 5.8. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Let T and $\hat{T}$ be instantiations of type T such that $\hat{T} \subseteq T$. Then, if*

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

then

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

6. Type reduction theory

Recall that given an instantiation T of type t and a non-negative integer B, we defined (Definition 1.1) a *B-collapsing function* ϕ to be a function from T to $\{0\,..\,B\}$ such that

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

In other words, ϕ replaces all but a fixed finite number of members of t by a single value. Whenever B is clear from context, we call ϕ a *collapsing function*.

As described in the Introduction, our aim is develop a *type reduction theory*, to show that

$$\begin{aligned} &Spec(\hat{T}) \sqsubseteq \phi(Impl(T)) \text{ implies that } Spec(T) \sqsubseteq Impl(T), \\ &\text{for all } T \text{ such that } T \supseteq \hat{T}, \text{ where } \phi : T \to \hat{T} \text{ is a collapsing function} \end{aligned} \tag{6.1}$$

Object	Application	Meaning
event	$\phi(c.v_1 \ldots v_k)$	$c.\phi(v_1) \ldots \phi(v_k)$
set/type	$\phi(S)$	$\{\phi(x) \mid x \in S\}$
trace	$\phi(tr)$	$\langle \phi(e) \mid e \leftarrow tr \rangle$
environment	$\phi(\Gamma)$	$\{x \mapsto \phi(v) \mid \Gamma(x) = v\}$
process	$\phi(P)$	$P[\![{}^{\phi(e)}/_{e} \mid e \in \Sigma]\!]$

Table 1: Lifting the definition of ϕ.

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 6.5 and Theorem 6.13, prove (6.1) in the traces and stable failures models, respectively. They require suitable assumptions on *Spec* (including **SeqNorm**) and *Impl* (that it is symmetric in t); they give a lower bound on the size of $\hat{T}$ based on the syntax of *Spec*.

A significant part of this section 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).

We present and prove type reduction theorems for both the traces and the stable failures models in Sections 6.1 and 6.2, respectively. Proofs of some subsidiary results are in Appendix B.

First, we lift ϕ to various settings. Given a boolean condition *cond*, we define $\phi(cond)$ to be like *cond*, except that every value or variable x of type t is replaced by $\phi(x)$. We adopt the notational convention that if x is a value or a variable of a type other than t or it is a type other than t, then $\phi(x) = x$.

We lift the application of ϕ to other common objects used in this paper in the natural way (see Table 1).

Finally, given an instantiation T of type t, a B-collapsing function ϕ, and a value v in $\{0 \,..\, B\}$, we define

$$\phi^{-1}(v) \quad = \quad \begin{cases} \{v' \in T \mid \phi(v') = v\}, & \text{if } v \in \{0 \,..\, B\}, \\ \{v\}, & \text{otherwise.} \end{cases}$$

Also, given T, we lift the definition of ϕ^{-1} to events:

$$\phi^{-1}(c.v_1 \ldots v_k) \quad = \quad \{c.v'_1 \ldots v'_k \mid \forall\, i \in \{1 \,..\, k\} \bullet v'_i \in \phi^{-1}(v_i)\},$$

and to sets of events:

$$\phi^{-1}(S) \quad = \quad \bigcup \{\phi^{-1}(e) \mid e \in S\}.$$

6.1. **Threshold results for the traces model.** In this section we present the main results of our type reduction theory for use within the traces model.

We begin with a proposition that establishes that, provided $Proc(t)$ satisfies **SeqNorm** and $\mathbf{RevPosConjEqT}_{\mathrm{T}}$ and given a collapsing function ϕ, if

- tr is a trace of $(Proc(t), \Gamma_{init}, T)$ (for some sufficiently large T),
- $\phi(tr)^\frown\langle e\rangle$ is a trace of $(Proc(t), \phi(\Gamma_{init}), T)$, and
- e does not have outputs of type t from outside of $\{0\,..\,B-1\}$,

then every event that is like e, except with arbitrary values of inputs of type t, is in $initials(Proc(t), \Gamma_{init}, T)/tr)$. In both the statement and the proof of this proposition we take the underlying type of all configurations to be the fixed type T.

Proposition 6.1. *Let B be some natural number. Suppose that*

- *$Proc(t)$ satisfies **SeqNorm** and **RevPosConjEqT**$_{\mathrm{T}}$;*
- *ϕ is a B-collapsing function; and*
- *T is an instantiation of type t of size at least $B+1$.*

Suppose further that

(i) $tr \in traces(Proc(t), \Gamma_{init})$;
(ii) $\phi(tr)^\frown\langle e\rangle \in traces(Proc(t), \phi(\Gamma_{init}))$ *with* $e = c.v_1 \ldots v_k$;
(iii) $\sigma^\frown\langle\epsilon\rangle \in SymbolicTraces(Proc(t))$ *and* $\sigma^\frown\langle\epsilon\rangle$ $generates_{\phi(\Gamma_{init})}$ $\phi(tr)^\frown\langle e\rangle$, *where ϵ is a visible symbolic event of the form $c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$; and*
(iv) $\forall\, i \in !^t(\epsilon) \bullet v_i \in \{0\,..\,B-1\}$.

Then[6]

$$\forall\, v' \in \{1\,..\,k\} \to Value \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i) \bullet tr^\frown\langle c.v'_1 \ldots v'_k\rangle \in traces(Proc(t), \Gamma_{init}).$$

Proof sketch. By a structural induction on $Proc(t)$. The details are in Appendix B.1. □

The following example illustrates some aspects of Proposition 6.1.

Example 6.2. Let

$$Proc(t) \;=\; c!x\$y{:}t?z{:}t \to \text{if } y = z \text{ then } d!x \to STOP \text{ else } d\$w{:}t \to STOP.$$

Note in particular that $Proc(t)$ satisfies **RevPosConjEqT**$_{\mathrm{T}}$. Let $T = \{0,1,2\}$, $B = 1$ and let ϕ be the appropriate 1-collapsing function. We consider four instances.

(1) Let $\Gamma_{init}(x) = 0$, $tr = \langle\rangle$, $e = c.0.1.2$, $\sigma = \langle\rangle$ and $\epsilon = c!x\$y{:}t?z{:}t$. It is easy to check that conditions (i)–(iv) of the proposition hold. The proposition then implies

$$\forall\, v'_2, v'_3 \in T \bullet \langle c.0.v'_2.v'_3\rangle \in traces(Proc(t), \Gamma_{init}),$$

which is clearly true.

(2) Now suppose $\Gamma_{init}(x) = 2$ and again $tr = \langle\rangle$ and $\epsilon = c!x\$y{:}t?z{:}t$. Then condition (ii) implies e is of the form $c.1.v_2.v_3$ for some v_2, v_3. But now condition (iv) does not hold, so no conclusion can be reached from the proposition; and indeed $traces(Proc(t), \Gamma_{init})$ does not include traces of the form $\langle c.1.v'_2.v'_3\rangle$. Condition (iv) ensures that all the values v_i for $i \in !(\epsilon)$ are not collapsed within $\phi(tr)^\frown\langle e\rangle$.

(3) Now consider $\Gamma_{init}(x) = 0$, $tr = \langle c.0.0.2\rangle$, so $\phi(tr) = \langle c.0.0.1\rangle$, and $e = d.2$, $\sigma = \langle c!x\$y{:}t?z{:}t, \neg y = z\rangle$ and $\epsilon = d\$w{:}t$. It is easy to check that conditions (i)–(iv) of the proposition hold. The proposition then implies

$$\forall\, v'_1 \in T \bullet \langle c.0.0.2,\ d.v'_1\rangle \in traces(Proc(t), \Gamma_{init}),$$

which is clearly true, since the process reaches the "else" branch after $\langle c.0.0.2\rangle$.

[6]The notation $\forall\, x \in X \mid P(x) \bullet Q(x)$ is equivalent to $\forall\, x \in X \bullet P(x) \Rightarrow Q(x)$.

(4) Now suppose $tr = \langle c.0.1.2\rangle$, so $\phi(tr) = \langle c.0.1.1\rangle$, and $e = d.0$, $\sigma = \langle c!x\$y{:}t?z{:}t, y = z\rangle$ and $\epsilon = d!x$. It is easy to check that conditions (i)–(iv) of the proposition hold. The proposition then implies

$$\langle c.0.1.2,\ d.0\rangle \in \mathit{traces}(\mathit{Proc}(t), \Gamma_{\mathit{init}}),$$

which is clearly true. This case shows the importance of $\mathbf{RevPosConjEqT}_{\mathrm{T}}$: the "else" branch must be able to perform (at least) the same events as the "then" branch.

We will need the following definition in order to define our threshold.

Definition 6.3. We say that visible symbolic events ϵ and ϵ' are non-t equivalent, written $\epsilon \equiv_{\mathit{non\text{-}t}} \epsilon'$, if they agree on all the fields not of type t. For example, $c!a?t{:}T \equiv_{\mathit{non\text{-}t}} c!a\$t{:}T$, but $c!a?t{:}T \not\equiv_{\mathit{non\text{-}t}} c!b?t{:}T$ where a and b are not of type t.

We lift the relation to sequences of visible symbolic events by pointwise application.

Finally we say that symbolic traces σ and σ' are non-t equivalent, written $\sigma \equiv_{\mathit{non\text{-}t}} \sigma'$, if their restrictions to visible symbolic events are non-t equivalent.

The following function returns the indices of all output variables of type t in all constructs of $P(t)$ corresponding to a symbolic trace that is non-t equivalent to $\sigma\hat{}\langle\epsilon\rangle$.

$$!^t(\sigma,\epsilon)(P(t)) \quad = \quad \bigcup\{!^t(\epsilon') \mid \sigma'\hat{}\langle\epsilon'\rangle \in \mathit{SymbolicTraces}(P(t)) \wedge \sigma\hat{}\langle\epsilon\rangle \equiv_{\mathit{non\text{-}t}} \sigma'\hat{}\langle\epsilon'\rangle\}.$$

Example 6.4. Consider

$$\begin{aligned} P(t) \quad = \quad & \mathit{in}?x{:}t?y{:}t?z{:}t \rightarrow \\ & \quad \text{if } x = y \text{ then } (\text{if } x = z \text{ then } \mathit{STOP} \text{ else } \mathit{out}!x\$w{:}t \rightarrow \mathit{STOP}) \\ & \quad \text{else } (\text{if } x = z \text{ then } \mathit{out}\$w{:}t!y \rightarrow \mathit{STOP} \text{ else } \mathit{out}\$v{:}t\$w{:}t \rightarrow \mathit{STOP}). \end{aligned}$$

Then $!^t(\langle \mathit{in}?x{:}t?y{:}t?z{:}t, \neg x = y, \neg x = z\rangle, \mathit{out}\$v{:}t\$w{:}t)(P(t)) = \{1,2\}$, since all the constructs using *out* can be reached on a trace that is non-t equivalent to $\langle \mathit{in}?x{:}t?y{:}t?z{:}t, \neg x = y, \neg x = z, \mathit{out}\$v{:}t\$w{:}t\rangle$.

Now consider

$$Q(t) \quad = \quad \mathit{in}?x{:}\{a,b\}?i{:}t \rightarrow \text{if } x = a \text{ then } c?j{:}t!i \rightarrow \mathit{STOP} \text{ else } c!i?j{:}t \rightarrow \mathit{STOP},$$

where a and b are not of type t. Then $!^t(\langle \mathit{in}!a?i{:}t\rangle, c?j{:}t!i)(Q(t)) = \{2\}$, since the construct in the else branch cannot be reached after a symbolic trace that is non-t equivalent to $\langle \mathit{in}!a?i{:}t\rangle$.

We now present the first of our two main results of this paper. The following theorem establishes a threshold $\mathit{Thresh}_{\mathrm{T}}$ such that if $\mathit{Spec}(t)$ and $\mathit{Impl}(t)$ fulfil certain requirements, then, for all $B \geq \mathit{Thresh}_{\mathrm{T}}$, if ϕ is a B-collapsing function, then for all $n \geq B$

$$\text{if } \mathit{Spec}(\{0\,..\,B\}) \sqsubseteq_{\mathrm{T}} \phi(\mathit{Impl}(\{0\,..\,n\})), \text{ then } \mathit{Spec}(\{0\,..\,n\}) \sqsubseteq_{\mathrm{T}} \mathit{Impl}(\{0\,..\,n\}).$$

In Section 6.2 we will present an analogous result for the stable failures model.

Theorem 6.5 (Extendibility of traces refinement of systems with replicated components)**.** *Suppose that*

(i) $\mathit{Spec}(t)$ *satisfies* $\boldsymbol{\mathit{SeqNorm}}$ *and* $\boldsymbol{\mathit{RevPosConjEqT}}_{\mathrm{T}}$*;*
(ii) $\mathit{Impl}(t)$ *satisfies* $\boldsymbol{\mathit{TypeSym}}$*;*

(iii) *Thresh*$_{\mathrm{T}}$ *is the maximum number of output positions reachable on non-t equivalent symbolic traces of* $Spec(t)$*, i.e.*

$$Thresh_{\mathrm{T}} \quad = \quad \max\{\#!^t(\sigma,\epsilon)(Spec(t)) \mid \sigma\hat{}\langle\epsilon\rangle \in SymbolicTraces(Spec(t))\};$$

(iv) $B \geq Thresh_{\mathrm{T}}$*;*

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

(vi) ϕ *is a* B*-collapsing function.*

Then if $Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(Impl(T))$*, then* $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$*.*

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

$$\begin{array}{rcl} tr\hat{}\langle e\rangle & \in & traces(Impl(T)), \\ tr & \in & traces(Spec(T)), \\ tr\hat{}\langle e\rangle & \notin & traces(Spec(T)). \end{array}$$

Suppose that $e = c.v_1 \ldots v_k$. Suppose tr is generated by symbolic trace σ_1 of $Spec(t)$. We can construct a symbolic event $\epsilon_1 = c\S x_1{:}X_1 \ldots \S x_k{:}X_k$ to generate e (although $\sigma_1\hat{}\langle\epsilon_1\rangle$ might not be a symbolic trace of $Spec(t)$): if v_i is of type t we set $\S x_i{:}X_i$ to $\$x_i{:}t$; otherwise we set $\S x_i{:}X_i$ to $!v_i{:}null$.

By assumptions (iii) and (iv), $\#!^t(\sigma_1,\epsilon_1)(Spec(t)) \leq B$, so let $\pi : T \to T$ be a bijection that maps $\{v_i \mid i \in !^t(\sigma_1,\epsilon_1)(Spec(t))\}$ into $\{0\,..\,B-1\}$. By Corollary 3.10 $Spec(t)$ satisfies **TypeSym**, and by assumption $Impl(t)$ satisfies **TypeSym**, so by Remark 3.13 we have that

$$\begin{array}{rcl} \pi(tr)\hat{}\langle\pi(e)\rangle & \in & traces(Impl(T)), \\ \pi(tr) & \in & traces(Spec(T)), \\ \pi(tr)\hat{}\langle\pi(e)\rangle & \notin & traces(Spec(T)). \end{array} \tag{6.2}$$

Hence,

$$\phi(\pi(tr))\hat{}\langle\phi(\pi(e))\rangle \quad \in \quad traces(\phi(Impl(T))).$$

But $Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(Impl(T))$, so

$$\phi(\pi(tr))\hat{}\langle\phi(\pi(e))\rangle \quad \in \quad traces(Spec(\phi(T))).$$

However, by Corollary 5.8, $Spec(T) \sqsubseteq_{\mathrm{T}} Spec(\phi(T))$, so

$$\phi(\pi(tr))\hat{}\langle\phi(\pi(e))\rangle \quad \in \quad traces(Spec(T)). \tag{6.3}$$

We can now apply Proposition 6.1 to $Spec(T)$, with $\pi(tr)$ in place of tr, and $\phi(\pi(e))$ in place of e: condition (i) is satisfied, by (6.2); condition (ii) is satisfied, by (6.3); condition (iii) is satisfied by taking a suitable choice of σ to generate $\phi(\pi(tr))$, and taking ϵ to be the symbolic event that generates $\phi(\pi(e))$; condition (iv) is satisfied since $!^t(\epsilon) \subseteq !^t(\sigma_1,\epsilon_1)(Spec(t))$ (since $\sigma\hat{}\langle\epsilon\rangle \equiv_{non\text{-}t} \sigma_1\hat{}\langle\epsilon_1\rangle$), and by construction, all the corresponding fields of $\phi(\pi(e))$ are in $\{0\,..\,B-1\}$. Considering the valuation v' such that $\pi(e) = c.v'_1 \ldots v'_k$ then allows us to deduce that

$$\pi(tr)\hat{}\langle\pi(e)\rangle \quad \in \quad traces(Spec(T)).$$

This is a contradiction, which completes our proof. □

Remark 6.6. For every verification problem, the value of $\mathit{Thresh}_{\mathrm{T}}$ in Theorem 6.5 depends only on the specification.

Example 6.7. Recall the process syntax $P(t)$ from Example 6.4. We argued there that $!^t(\langle \mathit{in}?x{:}t?y{:}t?z{:}t, \neg x = y, \neg x = z\rangle, \mathit{out}\$v{:}t\$w{:}t)(P(t)) = \{1, 2\}$. It is clear that $!^t(\sigma, \epsilon)(P(t))$ has fewer elements for other values of σ and ϵ. Hence $\mathit{Thresh}_{\mathrm{T}} = 2$ in this case.

If $\mathit{Spec}(t)$ contains no conditional choices, then we can obtain a simpler expression for the threshold.

Proposition 6.8. *If $\mathit{Spec}(t)$ contains no conditional choices then*

$$\mathit{Thresh}_{\mathrm{T}} \;\leq\; \max\{\#!^t(\alpha) \mid \alpha \textit{ is a construct of } \mathit{Spec}(t)\}.$$

The proof is in Appendix B, and shows that in this case there is a *unique* construct that contributes towards the calculation of each $!^t(\sigma, \epsilon)(\mathit{Spec}(t))$.

If $\mathit{Spec}(t)$ uses a conditional, then there may be two such constructs, but by Lemma 5.6, $!^t(\alpha_{\mathit{then}}) \supseteq !^t(\alpha_{\mathit{else}})$, where α_{then} and α_{else} are the constructs in the "then" and "else" branches, respectively; hence the above equality still holds. It's only when $\mathit{Spec}(t)$ contains nested conditionals, as in Example 6.4, that one needs to consider multiple constructs together.

Remark 6.9. For all specifications with a finite SSLTS, the value of $\mathit{Thresh}_{\mathrm{T}}$ in Theorem 6.13 can be calculated in a finite amount of time. All states that can be reached by non-t equivalent traces need to be considered together; this can be performed by a process similar to normalisation [Ros97, Appendix C].

Example 6.10. Recall the example from Section 2.3. Earlier, we explained how to use counter abstraction techniques from [Maz10, ML11] to show

$$\mathit{Spec}(\phi(T)) \;\sqsubseteq_{\mathrm{T}}\; \phi(\mathit{Impl}(T)), \qquad \text{for all instantiations } T \text{ of } t \text{ with } \#T \geq 3,$$

where

$$\mathit{Spec}(t) \;=\; \mathit{enterCS}\$i{:}t \rightarrow \mathit{leaveCS}!i \rightarrow \mathit{Spec}(t),$$

and where we took $B = 1$. We can now apply Theorem 6.5. It's clear that condition (i) holds. Condition (ii) holds from the discussion in Example 3.11. From condition (iii) we obtain $\mathit{Thresh}_{\mathrm{T}} = 1$, essentially because $\mathit{Spec}(t)$ contains a single "!"; hence condition (iv) holds. Condition (v) gives a lower bound of 2 on the size of T, which is a weaker condition than we have already imposed. Finally condition (vi) holds by construction. Hence we can apply the theorem to deduce

$$\mathit{Spec}(T) \;\sqsubseteq_{\mathrm{T}}\; \mathit{Impl}(T), \qquad \text{for all instantiations } T \text{ of } t \text{ with } \#T \geq 3.$$

Smaller values of T can be verified directly.

In [ML11], we describe tool support, called TomCAT, for our counter abstraction techniques. In particular, the tool checks the conditions of Theorem 6.5, and calculates the threshold $\mathit{Thresh}_{\mathrm{T}}$. This part of the tool could easily be adapted to other abstraction techniques that build on the type reduction theory of this paper.

6.2. **Threshold results for the stable failures model.** In this section we present type reduction theory results analogous to those in Section 6.1, but extended to the stable failures model.

We begin with a proposition that establishes that, provided $Proc(t)$ satisfies **SeqNorm** and $\mathbf{RevPosConjEqT}_{\mathrm{F}}$ and given a collapsing function ϕ, if tr is a trace of $(Proc(t), \Gamma_{init}, T)$ (for some sufficiently large T), $(\phi(tr), X)$ is a failure of $(Proc(t), \phi(\Gamma_{init}), T)$ and events in $initials(Proc(t), \Gamma_{init}, T)/tr$ do not have outputs of type t from outside $\{0\,..\,B-1\}$, then (tr, X) is a failure of $(Proc(t), \Gamma_{init}, T)$. In this proposition we assume that the underlying type of all configurations is the fixed type T.

Proposition 6.11. *Let B be some natural number. Suppose that*

- *$Proc(t)$ satisfies **SeqNorm** and $\mathbf{RevPosConjEqT}_{\mathrm{F}}$;*
- *ϕ is a B-collapsing function; and*
- *T is an instantiation of type t of size at least $B+1$.*

Suppose further that

(i) $tr \in traces(Proc(t), \Gamma_{init})$;
(ii) $(\phi(tr), X) \in failures(Proc(t), \phi(\Gamma_{init}))$; *and*
(iii) *if P is a configuration such that $(Proc(t), \Gamma_{init}) \stackrel{tr}{\Longrightarrow} P$, then every output value of type t of every event in $initials(P)$ is in $\{0\,..\,B-1\}$.*

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

Proof sketch. By a structural induction on $Proc(t)$. The details are in Appendix B.2.

The following example illustrates some aspects of Proposition 6.11.

Example 6.12. Recall the following process from Example 6.2:

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

Note in particular that $Proc(t)$ satisfies $\mathbf{RevPosConjEqT}_{\mathrm{F}}$. Let $T = \{0, 1, 2\}$, $B = 1$ and let ϕ be the appropriate 1-collapsing function. We consider four instances.

(1) Let $\Gamma_{init}(x) = 0$, $tr = \langle\rangle$ and $X = \{|c|\} - \{|c.0.1|\}$. Condition (i) of the proposition clearly holds. Condition (ii) holds, considering the case that the nondeterministic selection picks $y = 1$. Condition (iii) holds since the only output value in an event after tr is the value 0 for x. The proposition then implies $(tr, X) \in failures(Proc(t), \Gamma_{init})$, which is clearly true, considering the case that the nondeterministic selection again picks $y = 1$.
(2) Now suppose $\Gamma_{init}(x) = 2$, $tr = \langle\rangle$ and $X = \{|c.2|\}$. Condition (i) clearly holds; condition (ii) holds since the environment $\phi(\Gamma_{init})$ maps x to 1. However, condition (iii) does not hold, since the initial configuration can output 2 for x. And indeed $(tr, X) \notin failures(Proc(t), \Gamma_{init})$, since for some $y \in T$ the event $c.2.y.0$ will be available. Condition (iii) ensures that the output values in initial events after tr are not collapsed.
(3) Now consider $\Gamma_{init}(x) = 0$, $tr = \langle c.0.0.2\rangle$, so $\phi(tr) = \langle c.0.0.1\rangle$, and $X = \{|c|\} \cup \{d.v \mid v \neq 2\}$. Conditions (i) and (iii) clearly hold. Condition (ii) holds, since after $\phi(tr)$ the process takes the "else" branch and can select $w = 2$. The proposition then implies $(tr, X) \in failures(Proc(t), \Gamma_{init})$, which is clearly true, since after tr the process again takes the "else" branch and can select $w = 2$.

(4) Now suppose $tr = \langle c.0.1.2 \rangle$, so $\phi(tr) = \langle c.0.1.1 \rangle$, and $X = \{\!|c|\!\} \cup \{d.v \mid v \neq 0\}$. It is easy to check the three conditions; note that for condition (ii), the failure corresponds to the "then" branch. The proposition then implies $(tr, X) \in failures(Proc(t), \Gamma_{init})$, which is clearly true, since after tr the process takes the "else" branch and can select $w = 0$. This case shows the importance of **RevPosConjEqT**$_{\mathrm{F}}$: the "else" branch must have (at least) all the failures of the "then" branch.

The following theorem is our second key result of this paper. It extends Theorem 6.5 to the stable failures model by establishing a threshold *Thresh* such that if $Spec(t)$ and $Impl(t)$ fulfil certain requirements, then, for all $B \geq Thresh$, if $Spec(\{0 \,..\, B\}) \sqsubseteq_{\mathrm{F}} \phi(Impl(\{0 \,..\, n\})$ then $Spec(\{0 \,..\, n\}) \sqsubseteq_{\mathrm{F}} Impl(\{0 \,..\, n\})$ for all $n \geq B$.

Recall that, given a symbolic conditional event *cond* and an environment Γ, $[\![cond]\!]_\Gamma$ denotes the truth value of the proposition obtained from *cond* by substituting all free variables of type t with their corresponding values contained within Γ. We lift the definition of $[\![\cdot]\!]_\Gamma$ to symbolic traces without visible symbolic events in the following way. Given a symbolic trace σ in $(Cond \cup \{\tau\})^*$ we let $[\![\sigma]\!]_\Gamma$ be equal to $\bigwedge\{[\![cond]\!]_\Gamma \mid cond \text{ in } \sigma, cond \in Cond\}$.

Theorem 6.13 (Extendibility of stable failures refinement of systems with replicated components)**.** *Suppose that*

(i) *$Spec(t)$ satisfies **SeqNorm** and **RevPosConjEqT**$_{\mathrm{F}}$, and is divergence-free and has a finite alphabet for every finite instantiation of type t;*
(ii) *$Impl(t)$ satisfies **TypeSym**;*
(iii) *no construct α in $Spec(t)$ combines nondeterministic inputs of type t and deterministic input of any type, i.e. if $\#\$^t(\alpha) > 0$, then $\#?(\epsilon) = 0$;*
(iv) *T is an instantiation of type t such that $\#T \geq B + 1$, where B is as below;*
(v) *$Thresh = \max\{ Thresh_\Gamma,$*

$$\max\{\, Thresh_{!^t}(\sigma, \Gamma) + Thresh_{?^t}(\sigma, \Gamma) \mid \sigma \in SymbolicTraces(Spec(t)) \wedge (\sigma = \langle\rangle \vee last(\sigma) \in Visible) \wedge \Gamma \in Env(T)\}\},$$

where

- *$Thresh_{!^t}(\sigma, \Gamma)$ counts the number of unique output variables of type t in all the visible symbolic events ϵ available in $Spec(t)$ immediately after σ such that all conditionals between the last symbolic event of σ and ϵ evaluate to $True$, i.e.*

$$Thresh_{!^t}(\sigma, \Gamma) = \#\{x_i \mid Spec(t) \xrightarrow{\sigma}_s \xrightarrow{\rho}_s \xrightarrow{\epsilon}_s \wedge \rho \in (Cond \cup \{\tau\})^* \wedge [\![\rho]\!]_\Gamma = True \wedge \epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \in Visible \wedge i \in !^t(\epsilon)\};$$

- *$Thresh_{?^t}(\sigma, \Gamma)$ counts the number of (not necessarily unique) input variables of type t in all the visible symbolic events ϵ available in $Spec(t)$ immediately after σ such that all conditionals between the last symbolic event of σ and ϵ evaluate to $True$, i.e.*

$$Thresh_{?^t}(\sigma, \Gamma) = \sum\{\#?^t(\epsilon) \mid Spec(t) \xrightarrow{\sigma}_s \xrightarrow{\rho}_s \xrightarrow{\epsilon}_s \wedge \rho \in (Cond \cup \{\tau\})^* \wedge \epsilon \in Visible \wedge [\![\rho]\!]_\Gamma = True\};$$

- *$Thresh_{\mathrm{T}}$ is as in Theorem 6.5;*

(vi) *$B \geq Thresh$; and*

(vii) *ϕ is a B-collapsing function.*

Then, if $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$*, then* $Spec(T) \sqsubseteq_{\mathrm{F}} Impl(T)$.

Proof. Suppose that the refinement $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$ holds and assume for a contradiction that $Spec(T) \not\sqsubseteq_{\mathrm{F}} Impl(T)$. Refinement in the stable failures model implies refinement in the traces model, so $Spec(\phi(T)) \sqsubseteq_{\mathrm{T}} \phi(Impl(T))$. Then, by Theorem 6.5 (which is applicable since its assumptions are weaker than those of this theorem), $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$.

Consider a minimal counterexample (tr, X) to the refinement $Spec(T) \sqsubseteq_{\mathrm{F}} Impl(T)$, i.e.

$$(tr, X) \quad \in \quad failures(Impl(T)), \tag{6.4}$$

$$(tr, X) \quad \notin \quad failures(Spec(T)), \tag{6.5}$$

$$\forall\, e \in X \bullet (tr, X \setminus \{e\}) \quad \in \quad failures(Spec(T)). \tag{6.6}$$

Observe that there is such a minimal counterexample since we have assumed that specifications are divergence-freedom and have finite alphabets.

Combining (6.5) and (6.6) we obtain that for all events e in X there exists a state $P_e(T)$ such that

$$Spec(T) \stackrel{tr}{\Longrightarrow} P_e(T) \ \wedge \ P_e(T)\ ref\ (X) \setminus \{e\} \ \wedge \ P_e(T) \stackrel{e}{\longrightarrow} . \tag{6.7}$$

This also means that every event in X is accepted in some stable state of $Spec(T)$ after tr. Hence,

$$X \quad \subseteq \quad initials(Spec(T)/tr). \tag{6.8}$$

We now aim to show that X is dependent upon at most $Thresh$ 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 construct α of $Spec(t)$ (uniqueness follows from Proposition 5.5) with $\#\$^t(\alpha) > 0$ (which, by assumption (iii), implies that $\#?(\alpha) = 0$) and show that

$$\begin{array}{l} \forall\, v' : \{1 \,..\, k\} \to Value \mid \\ \quad (\forall\, i \in \$^t(\alpha) \bullet v'_i \in T) \wedge (\forall\, i \in \{1 \,..\, k\} \setminus \$^t(\alpha) \bullet v'_i = v_i) \bullet \\ \qquad c.v'_1 \ldots v'_k \in X. \end{array} \tag{6.9}$$

Let v' be as in (6.9) 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 (6.7) where $X \setminus \{e\}$ is refused and e is accepted, except that the nondeterministic selections of α are resolved in a way such that the values v'_i are chosen instead of v_i for all $i \in \$^t(\alpha)$; call this stable state $P_{e'}(T)$ (see Figure 6 for an example). The initials of $P_{e'}(T)$ are the same[7] as those of $P_e(T)$, except they contain e' instead of e. Therefore, since $X \setminus \{e\}$ is refused in $P_e(T)$, $X \setminus \{e'\}$ must be refused in $P_{e'}(T)$. However, $e' \notin X$ by assumption, so X is refused in $P_{e'}(T)$, which contradicts (6.5).

(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 suppose that $e = c.v_1 \ldots v_k \in X$ is an event

[7]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 6.17 and Example 6.18 below.

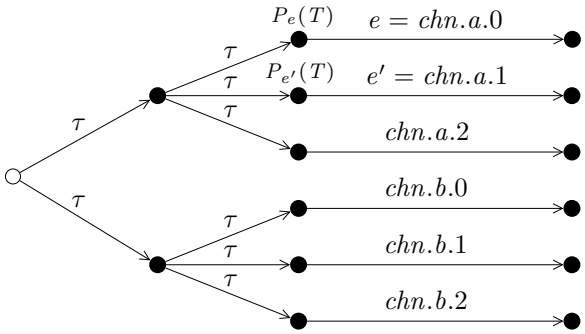

Figure 6: The LTS of $Proc(\{0,1,2\})$, where $Proc(t) = chn\$x{:}\{a,b\}\$y{:}t \to STOP$.

matching a construct α of $Spec(t)$ (uniqueness guaranteed by Proposition 5.5) with $\#?(\alpha) > 0$ (which, by assumption (iii), implies that $\#\$^t(\alpha) = 0$) and show that

$$\begin{array}{l} \forall\, v' : \{1\,..\,k\} \to Value \mid \\ \quad (\forall\, i \in \{1\,..\,k\} \setminus ?(\alpha) \bullet v'_i = v_i) \wedge (\exists\, i \in ?(\alpha) \bullet v'_i \neq v_i) \bullet \\ \quad\quad c.v'_1 \ldots v'_k \notin X. \end{array} \tag{6.10}$$

Let v' be as in (6.10) 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 (6.7) 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 Proposition 5.3); i.e.

$$(Spec(t), \{\}, T) \quad \stackrel{s}{\longmapsto} \quad (Spec'(t), \Gamma, T), \tag{6.11}$$

for some sequence of concrete events s such that s does not end with a τ and $s \setminus \{\tau\} = tr$. Observe that

$$(Spec(t), \{\}, T) \quad \stackrel{tr}{\Longrightarrow} \quad (Spec'(t), \Gamma, T).$$

So, thanks to the uniqueness of $Spec'(t)$ and Γ, and since s does not end with a τ,

$$initials(Spec(T)/tr) \quad = \quad initials(Spec'(t), \Gamma, T).$$

Let $S = S_1 \cup S_2$, where

$$\begin{array}{lcll} S_1 & = & \{v_i \mid e = c.v_1 \ldots v_k \in initials(Spec(T)/tr) & (6.12) \\ & & \quad \wedge\ e \text{ matches construct } \alpha \text{ of } Spec(t) \wedge i \in !^t(\alpha)\}, & \\ S_2 & = & \{v_i \mid c.v_1 \ldots v_k \in X \wedge i \in \{1\,..\,k\} \wedge v_i \in T & (6.13) \\ & & \quad \wedge \exists\, v' \in T \bullet c.v_1 \ldots v_{i-1}.v'.v_{i+1} \ldots v_k \notin X\}. & \end{array}$$

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 (6.8), 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 appropriate environments. Also, all conditional symbolic events occurring between any two visible symbolic events are always evaluated within the same environment. Therefore, when working out $initials(Spec'(t), \Gamma, T)$, we can ignore those initial visible symbolic events of $Spec'(t)$ that are preceded by a conditional symbolic event that evaluates to $False$ in Γ. Finally, from (6.11) and the translation rules of COSE (see Section 4.4.1) we have that there exists $\sigma \in SymbolicTraces(Spec(t))$ such that either $\sigma = \langle\rangle$ or $last(\sigma) \in Visible$ and $Spec(t) \stackrel{\sigma}{\longmapsto}_s Spec'(t)$ (so σ $generates_{\{\}}$ s). Hence, the number of type t values matched by deterministic inputs in the events in S_2 is at most

$$\begin{array}{rcl} Thresh_{?^t}(\sigma, \Gamma) & = & \sum\{\#?^t(\epsilon) \mid Spec(t) \stackrel{\sigma}{\longmapsto}_s \stackrel{\rho}{\longmapsto}_s \stackrel{\epsilon}{\longrightarrow}_s \wedge \rho \in (Cond \cup \{\tau\})^* \\ & & \qquad \wedge\, \epsilon \in Visible \wedge [\![\rho]\!]_\Gamma = True\}. \end{array}$$

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

$$\begin{array}{rcl} Thresh_{!^t}(\sigma, \Gamma) & = & \#\{x_i \mid Spec(t) \stackrel{\sigma}{\longmapsto}_s \stackrel{\rho}{\longmapsto}_s \stackrel{\epsilon}{\longrightarrow}_s \wedge \rho \in (Cond \cup \{\tau\})^* \wedge [\![\rho]\!]_\Gamma = True \\ & & \qquad \wedge\, \epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k \in Visible \wedge i \in !^t(\epsilon)\}. \end{array}$$

Summarising the last two paragraphs: all elements of S either match deterministic inputs in the events in S_2 (at most $Thresh_{?^t}(\sigma, \Gamma)$ such values), or match outputs in either S_1 or S_2 (at most $Thresh_{!^t}(\sigma, \Gamma)$ such values). Therefore,

$$\begin{array}{ll} & \#S \\ \leq & Thresh_{!^t}(\sigma, \Gamma) + Thresh_{?^t}(\sigma, \Gamma) \\ \leq & \max\{ Thresh_{!^t}(\sigma', \Gamma') + Thresh_{?^t}(\sigma', \Gamma') \mid \sigma' \in SymbolicTraces(Proc(t)) \\ & \qquad\qquad \wedge\, (\sigma' = \langle\rangle \vee last(\sigma') \in Visible) \\ & \qquad\qquad \wedge\, \Gamma' \in Env(T)\} \\ \leq & Thresh \\ \leq & B. \end{array}$$

Let $\pi : T \to T$ be a bijection that maps S into $\{0 \,..\, B-1\}$. Then, using Remark 3.13, we can infer from (6.4) and (6.5) that

$$(\pi(tr), \pi(X)) \in failures(Impl(T)), \tag{6.14}$$

$$(\pi(tr), \pi(X)) \notin failures(Spec(T)). \tag{6.15}$$

Now, by the denotational semantics of renaming [Ros97],

$$failures(\phi(Impl(T))) = \{(\phi(tr'), Y) \mid (tr', \phi^{-1}(Y)) \in failures(Impl(T))\}. \tag{6.16}$$

Let $c.v_1 \ldots v_k$ be an event in X. Let i be in $\{1 \,..\, k\}$ such that v_i is of type t. Our construction of S implies that either (1) v_i is in S, in which case $\pi(v_i)$ is in $\{0 \,..\, B-1\}$, or (2) v_i matches a nondeterministic input of type t, in which case the closure of X under nondeterministic inputs of the specification (clause 1 on p. 40) implies that for all values v' in T, $c.v_1 \ldots v_{i-1}.v'.v_{i+1} \ldots v_k$ is in X. Therefore, if $c.w_1 \ldots w_k$ is an event in $\pi(X)$, then for every i in $\{1 \,..\, k\}$ such that w_i is of type t, we have that either (1) w_i is in $\{0 \,..\, B-1\}$, or (2) $c.w_1 \ldots w_{i-1}.w'.w_{i+1} \ldots w_k$ is in $\pi(X)$ for all values w' in T. This,

thanks to the definition of ϕ, means that $\forall\, e \in \phi(\pi(X)) \bullet \phi^{-1}(e) \subseteq \pi(X)$. This implies that $\phi^{-1}(\phi(\pi(X))) \subseteq \pi(X)$, which trivially implies that

$$\phi^{-1}(\phi(\pi(X))) = \pi(X).$$

Combining with (6.14) and (6.16), we get that

$$(\phi(\pi(tr)), \phi(\pi(X))) \quad \in \quad failures(\phi(Impl(T))).$$

However, $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$, so

$$(\phi(\pi(tr)), \phi(\pi(X))) \quad \in \quad failures(Spec(\phi(T))). \tag{6.17}$$

We now show that

$$(\phi(\pi(tr)), \pi(X)) \quad \in \quad failures(Spec(T)). \tag{6.18}$$

Firstly, (6.17) implies that there exists a configuration $(P(t), \Gamma, \phi(T))$ such that

$$Spec(\phi(T)) \;=\; (Spec(t), \{\}, \phi(T)) \stackrel{\phi(\pi(tr))}{\Longrightarrow} (P(t), \Gamma, \phi(T)) \;\; ref \;\; \phi(\pi(X)). \tag{6.19}$$

Let $e = c.v_1 \ldots v_k$ be in $\phi(initials(P(t), \Gamma, T))$ and let α be the unique construct of $Spec(t)$ that e matches (with uniqueness following from Proposition 5.5). Then there exists a function $v' : \{1 \,..\, k\} \to Value$ such that

$$e' \;=\; c.v'_1 \ldots v'_k \;\in\; initials(P(t), \Gamma, T) \tag{6.20}$$

and $\phi(e') = e$, i.e.

$$\forall\, i \in \{1 \,..\, k\} \bullet \phi(v'_i) = v_i. \tag{6.21}$$

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 $initials(P(t), \Gamma, \phi(T))$ had been chosen before this state was reached (and are necessarily in $\phi(T)$). Hence, and since $\Gamma \in Env(\phi(T))$ implies $\Gamma \in Env(T)$, we have that $initials(P(t), \Gamma, \phi(T))$ and $initials(P(t), \Gamma, T)$ are the same, except for values of deterministic inputs of type t. Formally,

$$\begin{array}{l} initials(P(t), \Gamma, \phi(T)) = \\ \quad \{c.w_1 \ldots w_k \mid c.w'_1 \ldots w'_k \in initials(P(t), \Gamma, T) \text{ matches construct } \alpha' \wedge \\ \qquad\qquad w \in \{1 \,..\, k\} \to Value \wedge (\forall\, i \in \{1 \,..\, k\} \setminus ?^t(\alpha') \bullet w_i = w'_i) \wedge \\ \qquad\qquad \forall\, i \in ?^t(\alpha') \bullet w_i \in \phi(T)\}. \end{array} \tag{6.22}$$

Also, since $\Gamma \in Env(\phi(T))$, all values of type t used in the events of $initials(P(t), \Gamma, T)$ and that match nondeterministic selections or outputs, are in $\phi(T) = \{0 \,..\, B\}$:

$$\forall\, i \in \$^t(\alpha) \cup !^t(\alpha) \bullet v'_i \in \{0 \,..\, B\},$$

which, thanks to the properties of ϕ, and combined with the fact that for all non-t values val, $\phi(val) = val$, gives us that

$$\forall\, i \in \{1 \,..\, k\} \setminus ?^t(\alpha) \bullet \phi(v'_i) = v'_i.$$

Hence and by the definition of v' (6.21),

$$\forall\, i \in \{1 \,..\, k\} \setminus ?^t(\alpha) \bullet v_i = v'_i. \tag{6.23}$$

In addition, since $c.v_1 \ldots v_k$ is in $\phi(initials(P(t), \Gamma, T))$, it must be that

$$\forall\, i \in ?^t(\alpha) \bullet v_i \in \phi(T). \tag{6.24}$$

Combining (6.20), (6.22), (6.23) and (6.24) we get that e is in $initials(P(t), \Gamma, \phi(T))$. Hence

$$\phi(initials(P(t), \Gamma, T)) \quad \subseteq \quad initials(P(t), \Gamma, \phi(T)). \tag{6.25}$$

Conversely, let $e = c.v_1 \ldots v_k$ be in $initials(P(t), \Gamma, \phi(T))$. From Proposition 5.7 we can infer that

$$initials(P(t), \Gamma, \phi(T)) \quad \subseteq \quad initials(P(t), \Gamma, T),$$

so e is in $initials(P(t), \Gamma, T)$. Hence, $\phi(e)$ is in $\phi(initials(P(t), \Gamma, T))$. However, for all i in $\{1 \,..\, k\}$, v_i is either a value of a non-t type or it is a value in $\phi(T)$. Hence,

$$\forall\, i \in \{1 \,..\, k\} \bullet \phi(v_i) = v_i,$$

so $\phi(e) = e$ and therefore $e \in \phi(initials(P(t), \Gamma), T)$. Hence,

$$initials(P(t), \Gamma, \phi(T)) \quad \subseteq \quad \phi(initials(P(t), \Gamma, T)).$$

Combining the above with (6.25) we have that

$$\phi(initials(P(t), \Gamma, T)) \quad = \quad initials(P(t), \Gamma, \phi(T)). \tag{6.26}$$

We now aim to show that

$$(P(t), \Gamma, T) \quad ref \quad \pi(X). \tag{6.27}$$

Suppose for a contradiction that there exists an event x in $\pi(X) \cap initials(P(t), \Gamma, T)$. Then (6.26) implies that $\phi(x) \in \phi(\pi(X)) \cap initials(P(t), \Gamma, \phi(T))$. This means that $\phi(\pi(X)) \cap initials(P(t), \Gamma, \phi(T))$ is non-empty, which contradicts (6.19). Hence, (6.27) holds. Finally, applying Corollary 5.8 to (6.19), we have that

$$(Spec(t), \{\}, T) \quad \stackrel{\phi(\pi(tr))}{\Longrightarrow} \quad (P(t), \Gamma, T).$$

This, combined with (6.27) implies that (6.18) holds.

We now seek to apply Proposition 6.11 to $\pi(tr)$ and $\pi(X)$. From equation (6.14), $\pi(tr) \in traces(Impl(T))$. However, we have already shown that $Spec(T) \sqsubseteq_{\mathrm{T}} Impl(T)$, so

$$\pi(tr) \;\in\; traces(Spec(T)) \;=\; traces(Spec(t), \{\}, T).$$

This gives us condition (i) of Proposition 6.11. Equation (6.18) (and the fact that $\phi(\{\}) = \{\}$) gives us condition (ii). In addition, our definition of S_1 (6.12), combined with the definition of π, implies that every output value of type t of every event in $initials(Spec(T)/\pi(tr))$ is in $\{0 \,..\, B-1\}$, which gives us condition (iii). Hence, we can infer that

$$(\pi(tr), \pi(X)) \;\in\; failures(Spec(t), \{\}, T) \;=\; failures(Spec(T)).$$

This is a contradiction to (6.15), which completes our proof. □

Some observations related to Theorem 6.13 are now in order.

Remark 6.14. For every verification problem, the value of *Thresh* in Theorem 6.13 depends only on the specification.

Remark 6.15. For all specifications with a finite SSLTS, the value of *Thresh* in Theorem 6.13 can be calculated in a finite amount of time. The term $Thresh_{\Gamma}$ can be calculated as in Remark 6.9. The other term can be obtained by calculating $Thresh_{!^t}(\sigma,\Gamma)+Thresh_{?^t}(\sigma,\Gamma)$ for each symbolic state that is either the initial state or that has an incoming visible transition.

Example 6.16. Recall the example from Section 2.3. In Example 6.10 we showed how to apply Theorem 6.5 to deduce results in the traces model. We now, similarly, show how to apply Theorem 6.13 to deduce results in the stable failures model. We can use the counter abstraction techniques to verify

$$Spec(\phi(T)) \quad \sqsubseteq_{\mathrm{F}} \quad \phi(Impl(T)), \qquad \text{for all instantiations } T \text{ of } t \text{ with } \#T \geq 3,$$

where again

$$Spec(t) \quad = \quad enterCS\$i{:}t \rightarrow leaveCS!i \rightarrow Spec(t),$$

and where we took $B = 1$. It is clear that conditions (i), (ii) and (iii) of Theorem 6.13 hold. From condition (v) we obtain $Thresh = 1$, essentially because $Spec(t)$ contains a single "!" and no "?"; hence condition (vi) holds. Condition (iv) gives a lower bound of 2 on the size of T, which is a weaker condition than we have already imposed. Finally condition (vii) holds by construction. Hence we can apply the theorem to deduce

$$Spec(T) \quad \sqsubseteq_{\mathrm{F}} \quad Impl(T), \qquad \text{for all instantiations } T \text{ of } t \text{ with } \#T \geq 3.$$

Smaller values of T can be verified directly.

As with the theorem for the traces model, the tool TomCAT can be used to verify the conditions of Theorem 6.13 and to calculate the threshold.

At first, condition (iii) of Theorem 6.13 —that no construct of the specification combines nondeterministic inputs of type t and deterministic inputs of any type— may seem somewhat arbitrary. However, without it, there are specifications $Spec(t)$ and implementations $Impl(t)$ such that no threshold exists: for all values of B, there exists an instantiation T of type t such that $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$ and yet $Spec(T) \not\sqsubseteq_{\mathrm{F}} Impl(T)$. The following examples illustrate such pairs of processes, where a nondeterministic input of type t is combined with a deterministic input of type t (Example 6.17) and with a deterministic input of a non-t type (Example 6.18).

Example 6.17. Let

$$\begin{aligned} Spec(t) &= c\$x{:}t?y{:}t \rightarrow STOP, \\ Impl(t) &= \Box\, y{:}t \bullet c?x{:}(t \setminus \{y\})!y \rightarrow STOP. \end{aligned}$$

Note, in particular, that $Impl(t)$ satisfies **TypeSym**.

Let B be an arbitrary positive number, and let ϕ be as in the statement of Theorem 6.13. Let $T = \{0\,..\,N\}$ where $N \geq B+1$. It is easy to see that $traces(\phi(Impl(T))) \subseteq traces(Spec(\phi(T)))$. Further, whatever value $Impl(T)$ chooses for y, $(T \setminus \{y\}) \cap \{B\,..\,N\} \neq$

$\{\}$; hence $(\langle\rangle, \{c.B.B\}) \notin failures(\phi(Impl(T)))$. This helps to see that

$$\begin{array}{rl} & failures(\phi(Impl(T))) \\ = & \{(\langle\rangle, X) \mid X \subseteq \{c.x.x \mid x \in \{0\,..\,B-1\}\}\} \\ & \cup\{(\langle c.x.y\rangle, X) \mid y \in \{0\,..\,B\} \wedge x \in \{0\,..\,B\} \setminus \{y\} \wedge X \subseteq \Sigma\} \\ \subseteq & \{(\langle\rangle, X) \mid X \subseteq \{c.x.y \mid x \in \{0\,..\,B\} \setminus \{p\} \wedge y \in \{0\,..\,B\}\} \wedge p \in \{0\,..\,B\}\} \\ & \cup\{(\langle c.x.y\rangle, X) \mid x, y \in \{0\,..\,B\} \wedge X \subseteq \Sigma\} \\ = & failures(Spec(\phi(T))). \end{array}$$

Hence $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl(T))$.

However,

$$(\langle\rangle, \{c.x.x \mid x \in T\}) \;\in\; failures(Impl(T)) \setminus failures(Spec(T)),$$

so $Spec(T) \not\sqsubseteq_{\mathrm{F}} Impl(T)$.

Example 6.18. Let B be an arbitrary positive integer, and $Y = \{y_1, y_2\}$ a type other than t of size 2. Let

$$\begin{array}{rcl} Spec(t) & = & c\$x{:}t?y{:}Y \rightarrow STOP, \\ Impl_B(t) & = & \Box\, y{:}Y \bullet \left(\sqcap X \subseteq t \wedge \#X = B+1 \bullet c?x{:}X!y \rightarrow STOP\right). \end{array}$$

Note, in particular, that $Impl(t)$ satisfies **TypeSym**.

Let ϕ be as in the statement of Theorem 6.13, and let $T = \{0\,..\,N\}$ where $N \geq 2B+1$. It is easy to see that $traces(\phi(Impl_B(T))) \subseteq traces(Spec(\phi(T)))$. Further, whatever value $Impl_B(T)$ chooses for X, $X \cap \{B\,..\,N\} \neq \{\}$; hence $(\langle\rangle, \{c.B.y\}) \notin failures(\phi(Impl_B(T)))$ for every $y \in Y$. This helps to see that

$$\begin{array}{rl} & failures(\phi(Impl_B(T))) \\ \subseteq & \{(\langle\rangle, R) \mid R \subseteq \{c.x.y \mid x \in \{0\,..\,B-1\} \wedge y \in Y\}\} \\ & \cup\{(\langle c.x.y\rangle, R) \mid x \in \{0\,..\,B\} \wedge y \in Y \wedge R \subseteq \Sigma\} \\ \subseteq & \{(\langle\rangle, R) \mid R \subseteq \{c.x.y \mid x \in \{0\,..\,B\} \setminus \{p\} \wedge y \in Y\} \wedge p \in \{0\,..\,B\}\} \\ & \cup\{(\langle c.x.y\rangle, R) \mid x \in \{0\,..\,B\} \wedge y \in Y \wedge R \subseteq \Sigma\} \\ = & failures(Spec(\phi(T))). \end{array}$$

Hence $Spec(\phi(T)) \sqsubseteq_{\mathrm{F}} \phi(Impl_B(T))$.

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 $Impl_B(T)$ 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 $Spec(T)$, so $Spec(T) \not\sqsubseteq_{\mathrm{F}} Impl_B(T)$.

7. Conclusions

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 3.1) 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 (with the key results captured by Theorem 6.5 and Theorem 6.13), which establishes the

size of a fixed type $\hat{T}$ and a collapsing function ϕ that maps all types T larger than $\hat{T}$ to $\hat{T}$, and such that for all T such that $\hat{T} \subseteq T$,

$$Spec(\hat{T}) \sqsubseteq \phi(Impl(T)) \quad \text{implies that} \quad Spec(T) \sqsubseteq Impl(T) \tag{7.1}$$

with both refinements in either the traces or the stable failures model. In order for the above to hold, the processes have to satisfy certain conditions, the most important of which include a normality condition, **SeqNorm** (see Definition 3.5), for specifications and a type symmetry condition, **TypeSym** (see Definition 3.6), for implementations.

Our type reduction theory makes 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 Section 4 we presented a symbolic operational semantics for CSP processes that satisfy **Seq**, 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 $\phi(Impl(T))$ used in (7.1) still depends on T, the type reduction theory, on its own, does not resolve the problem of an infinite number of refinement checks needed to solve a given verification problem. However, the usefulness of the theory comes from the fact that it can be combined with an abstraction method that produces models $Abstr$ such that for all sufficiently large T,

$$Abstr \quad \sqsubseteq \quad \phi(Impl(T)). \tag{7.2}$$

We can then test, using a model checker, that $Spec(\hat{T}) \sqsubseteq Abstr$. This allows us to deduce, from transitivity of refinement and (7.1), that $Spec(T) \sqsubseteq Impl(T)$ holds for all sufficiently large T (and the verification problem can be solved directly for all smaller T). One suitable abstraction technique (based on ideas of counter abstraction) can be found in [Maz10, ML11].

7.1. **Automation.** We can automatically check process syntaxes for the syntactic requirements of data independence (Definition 3.1) and **SeqNorm** (Definition 3.5). Checking for the semantic requirements of the **TypeSym** condition (Definition 3.6) 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 syntactic conditions of Proposition 3.9 and infer **TypeSym**.

Checking **RevPosConjEqT** (Definition 3.15) is the most problematic when it comes to automation. The problem lies in the universal quantification over all instantiations of the parameter variables of the arguments of conditional choices. Currently, it is left to the user to provide a proof that for every conditional choice of the form "if $cond$ then $P(x_1, \ldots, x_k)$ else $Q(x_1, \ldots, x_k)$" in a given specification, where $cond \in Cond$, $cond$ is a positive conjunction of equality tests on t and $Q(v_1, \ldots, v_k) \sqsubseteq P(v_1, \ldots, v_k)$ for all values $v_1, \ldots, v_k$. In general, the problem of **RevPosConjEqT** satisfiability is undecidable, since a general (undecidable) PMCP problem of the form $Spec(x) \stackrel{?}{\sqsubseteq} Impl(x)$, where x is a parameter, can be reduced to checking whether $in?i{:}x?j{:}x \rightarrow$ if $i = j$ then $Impl(x)$ else $Spec(x)$ satisfies **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 Remarks 6.9 and 6.15, the calculation of the thresholds in Theorems 6.5 and 6.13 can be fully automated.

7.2. **Multiple distinguished types.** Throughout this paper we assumed the presence of a single distinguished type t. It is easy to extend our techniques to any finite number of distinguished types, say $t_1, t_2, \ldots, t_n$, provided all of them are pairwise independent. All requirements are extended in the natural way, e.g. each specification $Spec(t_1, t_2, \ldots, t_n)$ must now be data independent in each of the n types, and each implementation $Impl(t_1, \ldots t_n)$ must satisfy **TypeSym** with respect to each of t_1, $t_2, \ldots, t_n$. The threshold in each of Theorems 6.5 and 6.13 is then replaced by a tuple of values $(Thresh_1, Thresh_2, \ldots, Thresh_n)$, where each $Thresh_i$ is a threshold for the collapsing of the values of type t_i.

7.3. **Related work.** We are not aware of any other, similar type reduction theory for parameterised systems with the parameter describing the number of node processes forming a network. Ideas closest to ours are those of data independence. In [Laz99] Lazić provides results similar to ours, except that allows us to deduce that

$$Spec(\hat{T}) \sqsubseteq Impl(\hat{T}) \quad \text{implies that} \quad Spec(T) \sqsubseteq Impl(T)$$

instead of (7.1). This makes data independence theory applicable without the need for abstraction techniques, but, since both *Spec* and *Impl* are assumed to be data independent, it does not allow the use of replicated operators indexed over the distinguished type, which is a key part of all the implementations we consider.

7.4. **Future work.** The operational semantics that we presented in Section 4.3 served an important purpose in proving the results of our type reduction theory. However, our type reduction theory assumes that processes satisfy the **Seq**condition. Therefore, for brevity, we provided operational semantics rules only for those operators that **Seq**allows. To increase the generality, it would be desirable to formalise symbolic transition rules for parallel compositions, renaming, hiding and replicated choices.

We would like to extend our type reduction theory to the failures/divergences model of CSP (see e.g. [Ros97]). However, 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.

Finally, we presented our type reduction techniques for processes modelled using the CSP process algebra. It would be desirable to research how well these ideas map across to other formalisms.

Acknowledgements. We would like to thank Bill Roscoe, Marta Kwiatkowska, Ranko Lazić and the anonymous referees for very useful comments. This work was partially funded by EPSRC and ONR.

References

[AK86] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *POPL'81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 164–176, New York, NY, USA, 1981. ACM.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs: Workshop*, pages 52–71, London, UK, 1981. Springer-Verlag.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[CR98] S. Creese and A. W. Roscoe. TTP: A case study in combining induction and data independence. Technical report, University of Oxford, 1998.

[CR99] S. Creese and A. W. Roscoe. Formal verification of arbitrary network topologies. In *PDPTA'99: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 1999.

[Dav58] M. Davis. *Computability and Unsolvability*. McGraw-Hill, 1958.

[EC80] E. A. Emerson and E. M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of ICALP*, pages 169-181, 1980.

[For09] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement — FDR 2 user manual,* `http://www.fsel.com/fdr2_manual.html`, 2009.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall Europe, 1985.

[Laz99] R. S. Lazić. *A semantic study of data independence with applications to model checking*. DPhil thesis, University of Oxford, 1999.

[Low04] G. Lowe. On the application of counterexample-guided abstraction refinement and data independence to the parameterised model checking problem. In *AVIS'04: Proceedings of the 3rd International Workshop on Automatic Verification of Infinite-State Systems*, 2004.

[LR98] R. S. Lazić and A. W. Roscoe. Verifying determinism of concurrent systems which use unbounded arrays. Technical report, University of Oxford, 1998.

[Lub84] B. D. Lubachevsky. An Approach to Automating the Verification of Compact Parallel Coordination Programs I. *Acta Informatica* 21:12-169, 1984.

[Maz10] T. Mazur. Model Checking Systems with Replicated Components using CSP. DPhil thesis, University of Oxford, 2010.

[ML09] T. Mazur and G. Lowe. Counter abstraction in the CSP/FDR setting. *Electronic Notes in Theoretical Computer Science*, 250(1):171–186, 2009.

[ML11] T. Mazur and G. Lowe. CSP-based counter abstraction for systems with node identifiers. Submitted for publication.

[McM92] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[Mof10] N. Moffat. Identifying and Exploiting Symmetry for CSP Refinement Checking. DPhil thesis, University of Oxford, submitted 2010.

[Pnu77] A. Pnueli. The temporal logic of programs. In *SFCS'77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PXZ02] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 107–122, London, UK, 2002. Springer-Verlag.

[RB99] A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2-3):147–190, 1999.

[RLN04] A. W. Roscoe, R. S. Lazić, and T. C. Newcomb. On model checking data-independent systems with arrays without reset. *Theory and Practice of Logic Programming*, 4(5):659–693, 2004.
[Ros97] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall PTR, 1997.
[Ros98] A. W. Roscoe. Proving security protocols with model checkers by data independence techniques. In *CSFW'98: Proceedings of the 11th IEEE workshop on Computer Security Foundations*, page 84, Washington, DC, USA, 1998. IEEE Computer Society.
[Ros08] A. W. Roscoe. The three platonic models of divergence-strict CSP. In *ICTAC'08: Proceedings of the 5th International Colloquium on Theoretical Aspects of Computing*, pages 23–49, Berlin, Heidelberg, 2008. Springer-Verlag.
[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

Appendix A. Proofs for Section 5

In this appendix, we prove the results from Section 5. We start with a few lemmas that are used in the proofs of those results.

In some proofs by structural induction in this and subsequent appendices, some of the cases are straightforward and are omitted; they can be found in [Maz10].

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

Lemma A.1. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Let ϵ be a visible or conditional symbolic event and suppose that*

$$Proc(t) \stackrel{\sigma_1}{\longmapsto}_s Proc_1'(t) \quad \text{and} \quad Proc(t) \stackrel{\sigma_2}{\longmapsto}_s Proc_2'(t),$$

where $\sigma_1 = \tau^a \mathbin{^\frown} \langle \epsilon \rangle$ for $a \geq 0$, and $\sigma_2 = \tau^b \mathbin{^\frown} \langle \epsilon \rangle$ for $b \geq 0$. Then $Proc_1'(t) = Proc_2'(t)$.

Proof. We prove the result by a structural induction on $Proc(t)$. We give just the cases for prefix and external choice.

Prefix. Suppose $Proc(t) = \alpha \rightarrow Proc'(t)$ for some construct $\alpha = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$. Clearly ϵ must be a visible symbolic event matching α, say $c\S_1' x_1'{:}X_1' \ldots \S_k' x_k'{:}X_k'$. We consider two different cases, corresponding to the number of nondeterministic selections over non-t types of α.

Case 1. Suppose that $\#\$^{non\text{-}t}(\alpha) = 0$. Then, by Symbolic Prefix Rule 1 (p. 24), it must be that $\sigma_1 = \sigma_2 = \langle \epsilon \rangle$ with ϵ in $Comms^{non\text{-}t}(\alpha)$ and

$$Proc_1'(t) \;=\; Proc_2'(t) \;=\; Proc'(t)[x_i'/x_i \mid i \in ?^{non\text{-}t}(\alpha)].$$

Case 2. Suppose that $\#\$^{non\text{-}t}(\alpha) > 0$. Then, Symbolic Prefix Rule 2 (p. 24) implies that the only symbolic transitions in $Proc(t)$ are

$$Proc(t) \quad \stackrel{\tau}{\longrightarrow}_s \quad \left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha) \rightarrow Proc'(t)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)],$$

for each function v with $\mathrm{dom}(v) = \{1\,..\,k\}$ and such that if i is in $\$^{non\text{-}t}(\alpha)$, then v_i is in X_i. We are guaranteed that $\#\$^{non\text{-}t}(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha)) = 0$, so, Symbolic Prefix Rule 1 (p. 24) implies that there are two functions v as above, say v^1 and v^2, such that for $j \in \{1, 2\}$:

$$\begin{aligned} Proc_j'(t) \quad &= \quad (Proc'(t)[v_i^j/x_i \mid i \in \$^{non\text{-}t}(\alpha)])[x_i'/x_i \mid i \in ?^{non\text{-}t}(\alpha)], \\ \epsilon \quad &\in \quad Comms^{non\text{-}t}((Replace^{non\text{-}t}_{\$\mapsto !}(\alpha))[v_i^j/x_i \mid i \in \$^{non\text{-}t}(\alpha)]). \end{aligned}$$

Then, thanks to the definition of $Comms^{non\text{-}t}$, v^1 and v^2 are equal under domain restriction to $\$^{non\text{-}t}(\alpha)$. Therefore, $Proc'_1(t) = Proc'_2(t)$.

External choice. Suppose that $Proc(t) = P(t) \,\Box\, Q(t)$ for some process syntaxes $P(t)$ and $Q(t)$. Since $Proc(t)$ satisfies **SeqNorm**, we know that neither $P(t)$ nor $Q(t)$ contains a conditional choice on t before a prefix. Therefore ϵ cannot be a conditional symbolic event, so must be a visible symbolic event. **SeqNorm** implies that the channels of the initial visible symbolic events of $P(t)$ and $Q(t)$ are disjoint, so we have that either $P(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s$ or $Q(t) \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s$, but not both. Without loss of generality we assume the former. Then the inductive hypothesis implies that there is a unique symbolic state $P'(t)$ such that

$$P(t) \quad \xrightarrow{\tau}{}^*_s \xrightarrow{\epsilon}_s \quad P'(t).$$

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

The following corollary lifts the previous lemma to traces.

Corollary A.2. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Suppose further that*

$$Proc(t) \overset{\sigma}{\longmapsto}_s P(t) \quad \text{and} \quad Proc(t) \overset{\sigma'}{\longmapsto}_s Q(t).$$

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

Proof. By induction on the number of visible and conditional symbolic events of σ and σ' (which must be equal, since $\sigma \equiv_{non\text{-}\tau} \sigma'$), and using Lemma A.1. □

The following lemma relates two initial visible symbolic events on the same channel.

Lemma A.3. *Suppose that $Proc(t)$ satisfies **SeqNorm**, and $\sigma, \sigma' \in (Cond \cup \{\tau\})^*$ are such that $\sigma \equiv_{non\text{-}\tau} \sigma'$. Then, if $Proc(t) \overset{\sigma}{\longmapsto}_s \xrightarrow{\epsilon}_s$ and $Proc(t) \overset{\sigma'}{\longmapsto}_s \xrightarrow{\epsilon'}_s$, where $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and $\epsilon' = c'\S'_1 x'_1{:}X'_1 \ldots \S'_l x'_l{:}X'_l$ are visible symbolic events, then:*

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

$$\$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon) = \$^t(\epsilon') \cup ?^t(\epsilon') \cup !^t(\epsilon') \wedge$$
$$\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon) \bullet \S_i = \S'_i \wedge x_i = x'_i \wedge X_i = X'_i;$$

(ii) *if $Insts_\Gamma(\epsilon) \cap Insts_\Gamma(\epsilon') \neq \{\}$ for some environment Γ, then $\epsilon = \epsilon'$.*

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

Suppose that $Proc(t) = \alpha \rightarrow Proc'(t)$ for some construct $\alpha = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and some process syntax $Proc'(t)$. We perform a case analysis on the number of nondeterministic selections over non-t types of α.

Case 1. Suppose that $\#\$^{non\text{-}t}(\alpha) = 0$. Then, by Symbolic Prefix Rule 1 (p. 24) (observe that the other symbolic firing rules are not applicable in this case), it must be that $\sigma = \sigma' = \langle\rangle$, and that both ϵ and ϵ' are in $Comms^{non\text{-}t}(\alpha)$. By the definition of $Comms^{non\text{-}t}$ (p. 24), ϵ and ϵ' may differ only in the values of deterministic inputs of non-t types of α. Hence, clause (i) of the lemma holds. To prove clause (ii), we let $c.v_1 \ldots v_k$ be a common member of $Insts_\Gamma(\epsilon)$ and $Insts_\Gamma(\epsilon')$. Then the definition of $Insts_\Gamma$ (p. 30) implies that

$$\forall\, i \in !(\epsilon) \bullet \S_i = \S'_i = ! \wedge x_i = x'_i \wedge v_i = \Gamma(x_i) \wedge X_i = X'_i = null. \tag{A.1}$$

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

$$\forall\, i \in ?^{non\text{-}t}(\alpha) \bullet \S_i = \S'_i = ! \wedge x_i = x'_i \wedge X_i = X'_i = null.$$

This, combined with clause (i) of the lemma, (A.1) and the fact that $\$^{non\text{-}t}(\alpha) = 0$, implies that $\epsilon = \epsilon'$.

Case 2. Suppose that $\#\$^{non\text{-}t}(\alpha) > 0$. Then, by Symbolic Prefix Rule 2 (p. 24) (observe that the other symbolic firing rules are not applicable in this case), the only transitions in $Proc(t)$ are

$$\begin{array}{rcl} Proc(t) & \xrightarrow{\tau}_s & \left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha) \rightarrow Proc'(t)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)] \\ & = & Replace^{non\text{-}t}_{\$\mapsto !}(\alpha)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)] \rightarrow Proc'(t)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)] \end{array}$$

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

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

for every such function v, so Symbolic Prefix Rule 1 (p. 24) implies that

$$\begin{array}{rcl} \epsilon & \in & Comms^{non\text{-}t}\left(\left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)]\right), \\ \epsilon' & \in & Comms^{non\text{-}t}\left(\left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha)\right)[v'_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)]\right), \end{array}$$

for some functions v and v' such that $\mathrm{dom}(v) = \mathrm{dom}(v') = \$^{non\text{-}t}(\alpha)$ and if i is in $\$^{non\text{-}t}(\alpha)$, then v_i and v'_i are in X_i. The definition of $Comms^{non\text{-}t}$ (p. 24) implies that the parts of ϵ and ϵ' that involve type t are identical, which proves clause (i) of the lemma. To prove clause (ii), we let $c.v_1 \ldots v_k$ be a common member of $Insts_\Gamma(\epsilon)$ and $Insts_\Gamma(\epsilon')$. Then, the definition of $Insts_\Gamma$ (p. 30) implies that

$$\forall\, i \in !(\epsilon) \bullet \S_i = \S'_i = ! \wedge x_i = x'_i \wedge v_i = \Gamma(x_i) \wedge X_i = X'_i = null. \tag{A.2}$$

However, the definition of $Comms^{non\text{-}t}$ implies that

$$\begin{array}{rcl} ?^{non\text{-}t}(\alpha) & = & ?^{non\text{-}t}\left(\left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)]\right) \;\subseteq\; !(\epsilon), \\ \$^{non\text{-}t}(\alpha) & \subseteq & !^{non\text{-}t}\left(\left(Replace^{non\text{-}t}_{\$\mapsto !}(\alpha)\right)[v_i/x_i \mid i \in \$^{non\text{-}t}(\alpha)]\right) \;\subseteq\; !(\epsilon). \end{array}$$

Hence,

$$\forall\, i \in \$^{non\text{-}t}(\alpha) \cup ?^{non\text{-}t}(\alpha) \bullet \S_i = \S'_i \wedge x_i = x'_i \wedge X_i = X'_i.$$

This, combined with clause (i) of the lemma and (A.2), implies that $\epsilon = \epsilon'$. □

The following lemma shows that if a process can perform a conditional event initially (after only τs), then *all* its initial events (after τs) must be that conditional or its negation.

Lemma A.4. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Then, if $Proc(t) \stackrel{\sigma}{\longmapsto}_s \xrightarrow{cond}_s$ and $Proc(t) \stackrel{\sigma'}{\longmapsto}_s \xrightarrow{\alpha}_s$, where $\sigma, \sigma' \in \{\tau\}^*$, $cond \in Cond$ and $\alpha \neq \tau$, then $\alpha \in \{cond, \neg cond\}$.*

Proof. Since $Proc(t)$ satisfies **SeqNorm**, we know that there are no conditionals before prefixes in branches of external, internal and sliding choices. Hence one of the following must hold:

(i) $Proc(t)$ is a conditional choice on t, where the boolean condition is equal to $cond$ or $\neg cond$;
(ii) $Proc(t)$ is a process identifier bound by the global environment E to a conditional choice like that in clause (i) or (iii); or
(iii) $Proc(t)$ is a conditional choice whose boolean condition immediately evaluates to $True$ or $False$, and the appropriate branch is a process syntax as in clause (i) or (ii).

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

The following lemma shows that if two symbolic traces each contain a single visible symbolic event, and each trace can be instantiated in the same environment, then they contain the same conditional events before the visible event, essentially because those conditionals must evaluate to $True$ in the initial environment.

Lemma A.5. *Suppose that $Proc(t)$ satisfies **SeqNorm**. Let $\sigma\,\hat{}\langle\epsilon\rangle, \sigma'\,\hat{}\langle\epsilon'\rangle$ be symbolic traces of $Proc(t)$ such that $\sigma, \sigma' \in (Cond \cup \{\tau\})^*$, $\epsilon, \epsilon' \in Visible$, $\sigma\,\hat{}\langle\epsilon\rangle$ $generates_\Gamma$ $\langle e\rangle$ and $\sigma'\,\hat{}\langle\epsilon'\rangle$ $generates_\Gamma$ $\langle e'\rangle$ for some environment Γ and some visible events e and e'. Then $\sigma \equiv_{non\text{-}\tau} \sigma'$.*

Proof. Let $\kappa : SymbolicTraces(Proc(t)) \to \mathbb{N}$ be a function that returns the number of conditional symbolic events within a given symbolic trace. We prove the result using an induction on $\kappa(\sigma)$.

Base case. Suppose that $\kappa(\sigma) = 0$. Then $\sigma \in \{\tau\}^*$, so $Proc(t) \xrightarrow{\tau}_s^* \xrightarrow{\epsilon}_s$. Suppose, for a contradiction, $\kappa(\sigma') > 0$. Then $\sigma' = \tau^{a}\hat{}\langle cond\rangle\hat{}\rho$ for some $a \geq 0$, some $cond \in Cond$ and some symbolic trace $\rho \in (Cond \cup \{\tau\})^*$. Therefore $Proc(t) \xrightarrow{\tau}_s^* \xrightarrow{cond}_s$. By Lemma A.4, $\epsilon \in \{cond, \neg cond\}$. This is a contradiction as $\epsilon \in Visible$. Therefore $\kappa(\sigma') = 0$, which means that $\sigma' \in \{\tau\}^*$. Hence $\sigma \equiv_{non\text{-}\tau} \sigma'$.

Inductive case. Suppose that the result holds for all process syntaxes and all their symbolic traces with exactly k conditional symbolic events. Consider $\kappa(\sigma) = k + 1$. Then $\sigma = \tau^{a}\hat{}\langle cond\rangle\hat{}\rho$ for some $a \geq 0$, some $cond \in Cond$ and some $\rho \in (Cond \cup \{\tau\})^*$ with $\kappa(\rho) = k$. Arguing similarly in the base case, $\kappa(\sigma') > 0$. Therefore, $\sigma' = \tau^{b}\hat{}\langle cond'\rangle\hat{}\rho'$ for some $b \geq 0$, some $cond' \in Cond$ and some $\rho' \in (Cond \cup \{\tau\})^*$. By Lemma A.4, $cond' \in \{cond, \neg cond\}$. Since $\sigma\hat{}\langle\epsilon\rangle$ $generates_\Gamma$ $\langle e\rangle$ and $\sigma'\hat{}\langle\epsilon'\rangle$ $generates_\Gamma$ $\langle e'\rangle$, $cond$ and $cond'$ must both evaluate to $True$ within Γ, because there are no visible symbolic events within σ and σ' before $cond$ and $cond'$, respectively, that could modify the environment Γ. So it must be that $cond = cond'$. By Lemma A.1, there is a unique state $P(t)$ such that $Proc(t) \xrightarrow{\tau}_s^* \xrightarrow{cond}_s P(t)$. Hence, $\rho\hat{}\langle\epsilon\rangle$ and $\rho'\hat{}\langle\epsilon'\rangle$ are both symbolic traces of $P(t)$ such that $\rho\hat{}\langle\epsilon\rangle$ $generates_\Gamma$ $\langle e\rangle$ and

$\rho'\hat{}\langle\epsilon'\rangle$ $generates_\Gamma$ $\langle e'\rangle$. Therefore, by the inductive hypothesis, $\rho \equiv_{non\text{-}\tau} \rho'$, which implies that $\sigma \equiv_{non\text{-}\tau} \sigma'$. □

A.1. **Proofs of main results.** We can now prove the results stated in Section 5. In order to prove Proposition 5.3. We will need the following lemma.

Lemma A.6. *Suppose that Proc(t) satisfies **SeqNorm**. Suppose further that*

$$\begin{aligned}(Proc(t), \Gamma_{init}) &\xrightarrow{\tau}{}^{*}\xrightarrow{e} (P(t), \Gamma),\\ (Proc(t), \Gamma_{init}) &\xrightarrow{\tau}{}^{*}\xrightarrow{e} (Q(t), \Gamma'),\end{aligned}$$

where e is a visible event. Then $P(t) = Q(t)$ and $\Gamma = \Gamma'$.

Proof. By the translation rules of COSE (see Section 4.4.1), there must exist symbolic traces σ, σ' and visible symbolic events $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and $\epsilon' = c'\S'_1 x'_1{:}X'_1 \ldots \S'_l x'_l{:}X'_l$ such that

- $Proc(t) \xrightarrow{\sigma}_s \xrightarrow{\epsilon}_s P(t)$ and $Proc(t) \xrightarrow{\sigma'}_s \xrightarrow{\epsilon'}_s Q(t)$; and
- $\sigma\hat{}\langle\epsilon\rangle$ $generates_{\Gamma_{init}}$ $\langle e\rangle$ and $\sigma'\hat{}\langle\epsilon'\rangle$ $generates_{\Gamma_{init}}$ $\langle e\rangle$.

Then it must be that $\sigma, \sigma' \in (Cond \cup \{\tau\})^*$. So, by Lemma A.5, $\sigma \equiv_{non\text{-}\tau} \sigma'$. From the definition of the *generates* relation (p. 30) we have that $e \in Insts_{\Gamma_{init}}(\epsilon) \cap Insts_{\Gamma_{init}}(\epsilon')$, so Lemma A.3 implies that $\epsilon = \epsilon'$. Hence, $\sigma\hat{}\langle\epsilon\rangle \equiv_{non\text{-}\tau} \sigma'\hat{}\langle\epsilon'\rangle$ and so we can infer, using Corollary A.2, 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

$$\begin{aligned}\Gamma &= \Gamma_{init} \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon) \cup ?^t(\epsilon)\},\\ \Gamma' &= \Gamma_{init} \oplus \{x'_i \mapsto v_i \mid i \in \$^t(\epsilon') \cup ?^t(\epsilon')\}.\end{aligned}$$

However, $\epsilon = \epsilon'$, so $\Gamma = \Gamma'$, as required. □

Proof of Proposition 5.3. By a straightforward induction on the length of $s \setminus \{\tau\}$ and using Lemma A.6. □

Proof of Proposition 5.4. We prove the result by an induction on the length of *tr*.

Base case. Suppose that $tr = \langle\rangle$. Then σ $generates_\Gamma$ $\langle\rangle$ and σ' $generates_\Gamma$ $\langle\rangle$. Therefore, by the definition of the *generates* relation (p. 30), σ and σ' cannot contain visible symbolic events. Hence, by the assumptions of this proposition, $\sigma = \sigma' = \langle\rangle$, which means that $\sigma \equiv_{non\text{-}\tau} \sigma'$.

Inductive case. Suppose that the result holds for all traces of length k. Consider a trace tr_{k+1} of length $k+1$. Then there exists a trace tr_k of length k and a visible event e such that $tr_{k+1} = tr_k\hat{}\langle e\rangle$. There must also exist symbolic traces $\sigma_1, \sigma'_1, \sigma_2$, and σ'_2 such that

- either $\sigma_1 = \sigma'_1 = \langle\rangle$ or both σ_1 and σ'_1 end in a visible symbolic event;
- $\sigma = \sigma_1\hat{}\sigma_2$ and $\sigma' = \sigma'_1\hat{}\sigma'_2$; and
- σ_1 $generates_\Gamma$ tr_k and σ'_1 $generates_\Gamma$ tr_k.

Then, by the inductive hypothesis, $\sigma_1 \equiv_{non\text{-}\tau} \sigma'_1$. Hence, if $P(t)$ and $Q(t)$ are such that $Proc(t) \xrightarrow{\sigma_1}_s P(t)$ and $Proc(t) \xrightarrow{\sigma'_1}_s Q(t)$, then, by Corollary A.2, $P(t) = Q(t)$.

Let Γ be the environment reached after tr_k, i.e. such that $(Proc(t), \Gamma) \xrightarrow{s} (P(t), \Gamma)$ for some s such that $s \setminus \tau = tr_k$; by Proposition 5.3, Γ is unique. We now have that σ_2 $generates_\Gamma$ $\langle e\rangle$ and σ'_2 $generates_\Gamma$ $\langle e\rangle$. Hence, $\sigma_2, \sigma'_2 \neq \langle\rangle$. Therefore, both σ_2 and σ'_2

must end in a visible symbolic event (since they are suffixes of σ and σ'). So, $\sigma_2 = \rho\hat{}\langle\epsilon\rangle$ and $\sigma_2' = \rho'\hat{}\langle\epsilon'\rangle$ for some symbolic traces $\rho, \rho' \in (Cond \cup \{\tau\})^*$ and some visible symbolic events ϵ and ϵ'. Hence, by Lemma A.5, $\rho \equiv_{non\text{-}\tau} \rho'$. Also from the definition of *generates* we have that $e \in Insts_\Gamma(\epsilon) \cap Insts_\Gamma(\epsilon')$, so, by Lemma A.3, $\epsilon = \epsilon'$. Therefore, $\sigma_2 \equiv_{non\text{-}\tau} \sigma_2'$, and hence $\sigma \equiv_{non\text{-}\tau} \sigma'$. □

Proof of Proposition 5.5. Suppose for contradiction that $\alpha \neq \alpha'$. Definition 5.1 implies that α and α' give rise to e immediately after tr. By Proposition 5.4, if $\sigma\hat{}\langle\epsilon\rangle$ and $\sigma'\hat{}\langle\epsilon'\rangle$ are both symbolic traces of $Proc(t)$ such that $\sigma\hat{}\langle\epsilon\rangle, \sigma'\hat{}\langle\epsilon'\rangle$ $generates_{\{\}}$ $tr\hat{}\langle e\rangle$ and where ϵ and ϵ' are visible, then $\sigma \equiv_{non\text{-}\tau} \sigma'$ and $\epsilon = \epsilon'$. We now have that ϵ matches both α and α'. Then, the firing rules of SSOS (see Section 4.3.2) imply that α and α' must be constructs in different branches of an external, internal or sliding choice. Since both of these constructs give rise to the same concrete event, e, their channels must be identical. Hence, $Proc(t)$ contains a binary choice with branches sharing a common channel name. This contradicts the fact that $Proc(t)$ satisfies **SeqNorm**, so it must be that $\alpha = \alpha'$. □

Proof of Lemma 5.6. Suppose for a contradiction that there is some $j \in !(\epsilon') \setminus !(\epsilon)$. Then $j \in \$^t(\epsilon) \cup ?^t(\epsilon)$ (since $\$^{non\text{-}t}(\epsilon) = ?^{non\text{-}t}(\epsilon) = \{\}$ by Remark 4.7). This means that the j-th variable or value of ϵ' is of type t, so $j \in !^t(\epsilon')$. Suppose $e = c.v_1 \ldots v_k$ and let $e' = c.v_1 \ldots v_{j-1}.v_j'.v_{j+1} \ldots v_k$, where $v_j' \in T \setminus \{v_j\}$. By Remark 3.3, if a process can perform a given event, then it can also perform every other event that differs only in the values of inputs. Therefore, $tr\hat{}\langle e'\rangle \in traces(P(t), \Gamma_{init}, T)$, i.e. e' matches ϵ.

Since $(Q(t), \Gamma_{init}, T) \sqsubseteq_{\mathrm{T}} (P(t), \Gamma_{init}, T)$, we must have $tr\hat{}\langle e'\rangle \in traces(Q(t), \Gamma_{init}, T)$. Clause (iii), combined with the fact that v_j' is an output for $Q(t)$, different from v_j, implies that $\sigma'\hat{}\langle\epsilon'\rangle$ cannot generate $tr\hat{}\langle e'\rangle$ within Γ_{init}. So let $\rho\hat{}\langle\epsilon''\rangle \in SymbolicTraces(Q(t))$ be such that $\rho\hat{}\langle\epsilon''\rangle$ $generates_{\Gamma init}$ $tr\hat{}\langle e'\rangle$. Also, let $\sigma' = \sigma_1\hat{}\sigma_2$ and $\rho = \rho_1\hat{}\rho_2$, where σ_1 and ρ_1 are either both the empty symbolic trace or both end with visible symbolic events, and $\sigma_2, \rho_2 \in (Cond \cup \{\tau\})^*$. Then, clause (iii) implies that σ_1 $generates_{\Gamma_{init}}$ tr and ρ_1 $generates_{\Gamma_{init}}$ tr, so by Proposition 5.4, $\sigma_1 \equiv_{non\text{-}\tau} \rho_1$. Hence, if $Q'(t)$ and $Q''(t)$ are symbolic states such that $Q(t) \overset{\sigma_1}{\longmapsto}_s Q'(t)$ and $Q(t) \overset{\rho_1}{\longmapsto}_s Q''(t)$, then, thanks to Corollary A.2, we have that $Q'(t) = Q''(t)$.

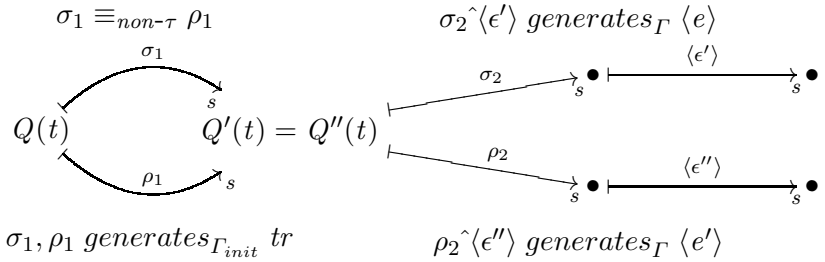

Figure 7: Illustration of the proof of Lemma 5.6

Let Γ be the environment reached after tr, i.e. such that $(Q(t), \Gamma_{init}, T) \overset{s}{\longmapsto} (Q'(t), \Gamma, T)$ for some s such that $s \setminus \tau = tr$; by Proposition 5.3, Γ is unique. Then, we have that

- $\sigma_2\hat{}\langle\epsilon'\rangle, \rho_2\hat{}\langle\epsilon''\rangle \in SymbolicTraces(Q'(t))$;

- $\langle e\rangle, \langle e'\rangle \in traces(Q'(t), \Gamma, T)$;
- $\sigma_2\hat{}\langle\epsilon'\rangle$ $generates_\Gamma$ $\langle e\rangle$; and
- $\rho_2\hat{}\langle\epsilon''\rangle$ $generates_\Gamma$ $\langle e'\rangle$.

Hence, we can deduce from Lemma A.5 that $\sigma_2 \equiv_{non\text{-}\tau} \rho_2$. Let $\epsilon' = c\S'_1 x'_1{:}X'_1 \ldots \S'_k x'_k{:}X'_k$ and $\epsilon'' = c\S''_1 x''_1{:}X''_1 \ldots \S''_k x''_k{:}X''_k$. Then, since the channels of ϵ' and ϵ'' are the same, Lemma A.3 implies that $!^t(\epsilon') = !^t(\epsilon'')$ and $x'_j = x''_j$. Since $\sigma_2\hat{}\langle\epsilon'\rangle$ $generates_\Gamma$ $\langle e\rangle = \langle c.v_1 \ldots v_k\rangle$ and $\rho_2\hat{}\langle\epsilon''\rangle$ $generates_\Gamma$ $\langle e'\rangle = \langle c.v_1 \ldots v_{j-1}.v'_j.v_{j+1} \ldots v_k\rangle$ and $j \in !^t(\epsilon'')$ (as $j \in !^t(\epsilon')$), we have that $x'_j = v_j$ and $x''_j = v'_j$. Hence $v_j = v'_j$. This is a contradiction, so $!(\epsilon') \subseteq !(\epsilon)$. □

Proof of Proposition 5.7. Since $\hat{T}$ is a subset of T, we have that $\Gamma, \Gamma' \in Env(\hat{T})$ implies $\Gamma, \Gamma' \in Env(T)$, since every partial function from Var to $\hat{T}$ is also a partial function from Var to T. We now prove the result using an induction on n, the number of times Translation Rule 4 of COSE (p. 29) had to be applied in order to obtain the transition in (5.1).

Base case. Suppose that $n = 0$. We separately consider the cases for a being τ or visible.

Case 1. Suppose that $a = \tau$. Then, the translation rules of COSE (see Section 4.4.1) imply that the transition in (5.1) can be a result of either Translation Rule 1 (p. 27) or Translation Rule 3 (p. 29).

For Translation Rule 1, it must be that $Proc(t) \xrightarrow{\epsilon}_s P(t)$ for some visible symbolic event $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ such that $\#\$^t(\epsilon) > 0$ and some symbolic state $P(t)$. In addition, $Proc'(t) = Replace^t_{\$\mapsto!}(c, Proc(t))$ and $\Gamma' = \Gamma \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon)\}$, where v is a function in $\$^t(\epsilon) \to \hat{T}$. Then, v is also a function in $\$^t(\epsilon) \to T$, so Translation Rule 1 implies that

$$\begin{array}{rcl}(Proc(t), \Gamma, T) & \xrightarrow{\tau} & (Replace^t_{\$\mapsto!}(c, Proc(t)), \Gamma \oplus \{x_i \mapsto v_i \mid i \in \$^t(\epsilon)\}, T) \\ & = & (Proc'(t), \Gamma', T).\end{array}$$

If the transition in (5.1) is a result of Translation Rule 3, then it must be that $Proc(t) \xrightarrow{\tau}_s Proc'(t)$ and $\Gamma = \Gamma'$. Hence, the same rule implies that $(Proc(t), \Gamma, T) \xrightarrow{\tau} (Proc'(t), \Gamma', T)$.

Case 2. Suppose that $a = c.v_1 \ldots v_k$ is a visible event. Then, the translation rules of COSE imply that the transition in (5.1) must be the result of Translation Rule 2 (p. 28). The rule implies that $Proc(t) \xrightarrow{\epsilon}_s Proc'(t)$ for some visible symbolic event $\epsilon = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ such that $\#\$^t(\epsilon) = 0$. In addition, $\Gamma' = \Gamma \oplus \{x_i \mapsto v_i \mid i \in ?^t(\epsilon)\}$, and for all i in $?^t(\epsilon)$, v_i is in $\hat{T}$. However, since $\hat{T} \subseteq T$, we have that for all i in $?^t(\epsilon)$, v_i is in T. Therefore, Translation Rule 2 implies that

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

This completes the base case.

Inductive case. Suppose the result holds for some $n = k$, where $k \geq 0$. Suppose that the transition in (5.1) requires $k + 1$ applications of Transition Rule 4. Then it must be that $Proc(t) \xrightarrow{cond}_s P(t)$ and $(P(t), \Gamma, \hat{T}) \xrightarrow{a} (Proc'(t), \Gamma', \hat{T})$. The latter transition requires k applications of Transition Rule 4, so the inductive hypothesis implies that $(P(t), \Gamma, T) \xrightarrow{a} (Proc'(t), \Gamma', T)$. Hence, by Translation Rule 4, $(Proc(t), \Gamma, T) \xrightarrow{a} (Proc'(t), \Gamma', T)$, which completes our proof. □

Proof of Corollary 5.8. Let s be a sequence of events such that $(Proc(t), \Gamma, \hat{T}) \stackrel{s}{\longmapsto} (Proc'(t), \Gamma', \hat{T})$ and $s \setminus \{\tau\} = tr$. Then the result follows from a simple induction on the length of s using Proposition 5.7. □

Appendix B. Proofs for Section 6

B.1. **Proofs for Section 6.1.**

Proof of Proposition 6.1. We prove the result using a structural induction on $Proc(t)$. We give just the cases for prefix and conditional choice.

Prefix. Suppose that $Proc(t) = \alpha \to P(t)$ where $\alpha = c'\S'_1 x'_1{:}X'_1 \ldots \S'_k x'_k{:}X'_k$. We consider two cases.

Subcase 1. Suppose that $tr = \langle\rangle$. Then, $\sigma \in \{\tau\}^*$, and $Proc(t) \stackrel{\tau}{\longrightarrow}{}^*_s \stackrel{\epsilon}{\longrightarrow}_s$. Using Translation Rule 3 (p. 29) and Remark 4.14 we get that

$$\begin{array}{l} \forall\, v' \in \{1\,..\,k\} \to Value \mid \\ \quad (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = \Gamma_{init}(x_i)) \bullet \\ \quad\quad \langle c.v'_1 \ldots v'_k \rangle \in traces(Proc(t), \Gamma_{init}) \end{array} \tag{B.1}$$

Since $tr = \langle\rangle$, assumption (iii) of the proposition implies that $\langle\epsilon\rangle\ generates_{\phi(\Gamma_{init})}\ \langle e\rangle$. Therefore, by assumption (iv),

$$\forall\, i \in !^t(\epsilon) \bullet v_i = (\phi(\Gamma_{init}))(x_i) \wedge v_i \in \{0\,..\,B-1\}.$$

Suppose that

$$\begin{array}{l} v' \in \{1\,..\,k\} \to Value \text{ is such that} \\ \quad (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i) \end{array}$$

Then $\forall\, i \in !^t(\epsilon) \bullet v'_i = (\phi(\Gamma_{init}))(x_i) \wedge v'_i \in \{0\,..\,B-1\}$. However, the properties of ϕ imply that for all variables var and all values val, we have that

$$(\phi(\Gamma_{init}))(var) = val \wedge val \in \{0\,..\,B-1\} \Rightarrow \Gamma_{init}(var) = val,$$

so

$$\forall\, i \in !^t(\epsilon) \bullet v'_i = \Gamma_{init}(x_i). \tag{B.2}$$

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

$$\forall\, i \in !^{non\text{-}t}(\epsilon) \bullet v'_i = (\phi(\Gamma_{init}))(x_i) = \Gamma_{init}(x_i)$$

with the last equality following from the fact that for all i in $!^{non\text{-}t}(\epsilon)$, x_i must be of a non-t type. Hence and from (B.2),

$$\forall\, i \in !(\epsilon) \bullet v'_i = \Gamma_{init}(x_i).$$

Therefore, (B.1) implies that $\langle c.v'_1 \ldots v'_k \rangle \in traces(Proc(t), \Gamma_{init})$. We have shown:

$$\begin{array}{l} \forall\, v' \in \{1\,..\,k\} \to Value \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i) \bullet \\ \quad \langle c.v'_1 \ldots v'_k \rangle \in traces(Proc(t), \Gamma_{init}), \end{array}$$

which is what we wanted to show.

Subcase 2. Suppose that $tr = \langle e' \rangle \hat{} tr'$ for some visible event a and some trace tr'. Then $\phi(tr) = \langle \phi(e') \rangle \hat{} \phi(tr')$. So clearly $\phi(tr)$ is non-empty and we know that σ $generates_{\phi(\Gamma_{init})}$ $\phi(tr)$. By the definition of *generates*, it must be that there is at least one visible symbolic event within σ. Hence, $\sigma = \sigma_1 \hat{} \langle \epsilon' \rangle \hat{} \sigma_2$ for some visible symbolic event ϵ' and some symbolic traces σ_1 and σ_2 such that σ_1 is in $\{\tau\}^*$ (σ_1 cannot contain any conditional symbolic events because $Proc(t)$ is a prefix). Then,

$$Proc(t) \xrightarrow{\tau}{}^* \xrightarrow{\epsilon'} P'(t) \qquad \text{and} \qquad \sigma_2 \hat{} \langle \epsilon \rangle \in SymbolicTraces(P'(t)), \tag{B.3}$$

where $P'(t)$ is like $P(t)$, but with some substitutions of concrete values for the non-t type input variables of α, as dictated by the SSOS firing rules for prefix (Section 4.3.2). We aim to apply the inductive hypothesis to $P'(t)$.

We can infer using Translation Rule 3 (p. 29) and Remark 4.14 that

$$(Proc(t), \phi(\Gamma_{init})) \xrightarrow{\tau}{}^* \xrightarrow{\phi(e')} (P'(t), \phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e'))),$$

where, recall from Section 4.6, $Match(\epsilon', \phi(e'))$ is a map from type t input variables of ϵ' to the corresponding concrete values of event $\phi(e')$. We have that configuration $(P'(t), \phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e')))$ is unique (thanks to Proposition 5.3). We know from assumption (ii) that $\langle \phi(e') \rangle \hat{} \phi(tr') \hat{} \langle e \rangle \in traces(Proc(t), \phi(\Gamma_{init}))$. Hence

$$\phi(tr') \hat{} \langle e \rangle \quad \in \quad traces(P'(t), \phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e'))). \tag{B.4}$$

Similarly, since $Proc(t) \xrightarrow{\tau}{}^* \xrightarrow{\epsilon'} P'(t)$ and $tr = \langle e' \rangle \hat{} tr' \in traces(Proc(t), \Gamma_{init})$ (from assumption (i)), using Translation Rule 3 (p. 29), Remark 4.14 and Proposition 5.3 we can infer that

$$(Proc(t), \Gamma_{init}) \xrightarrow{\tau}{}^* \xrightarrow{e'} (P'(t), \Gamma_{init} \oplus Match(\epsilon', e')) \xRightarrow{tr'} . \tag{B.5}$$

Hence,

$$tr' \quad \in \quad traces(P'(t), \Gamma_{init} \oplus Match(\epsilon', e')). \tag{B.6}$$

Finally, assumption (iii) implies that

$$\sigma \hat{} \langle \epsilon \rangle \;=\; \sigma_1 \hat{} \langle \epsilon' \rangle \hat{} \sigma_2 \hat{} \langle \epsilon \rangle \;\; generates_{\phi(\Gamma_{init})} \;\; \phi(tr) \hat{} \langle e \rangle \;=\; \langle \phi(e') \rangle \hat{} \phi(tr') \hat{} \langle e \rangle.$$

So, by the definition of *generates* (p. 30)

$$\sigma_2 \hat{} \langle \epsilon \rangle \quad generates_{\phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e'))} \quad \phi(tr') \hat{} \langle e \rangle. \tag{B.7}$$

We can now deduce the inductive hypothesis for $P'(t)$, with tr' in place of tr, σ_2 in place of σ, and $\Gamma_{init} \oplus Match(\epsilon', e')$ in place of Γ_{init}: (B.6) gives us condition (i); (B.4) gives us condition (ii), observing that $\phi(\Gamma_{init}) \oplus Match(\epsilon', \phi(e')) = \phi(\Gamma_{init} \oplus Match(\epsilon', e'))$; and (B.3) and (B.7) give us condition (iii). Hence

$$\forall\, v' \in \{1 \,..\, k\} \to Value \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v_i = v'_i) \bullet \\ tr'^{\hat{}} \langle c.v'_1 \ldots v'_k \rangle \in traces(P'(t), \Gamma_{init} \oplus Match(\epsilon', e')).$$

This, combined with (B.5), gives us that

$$\forall\, v' \in \{1 \,..\, k\} \to Value \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v_i = v'_i) \bullet \\ \langle e' \rangle \hat{} tr'^{\hat{}} \langle c.v'_1 \ldots v'_k \rangle \in traces(Proc(t), \Gamma_{init}).$$

However, $tr = \langle e' \rangle \hat{} tr'$, so the result holds.

Conditional choice. Suppose that $Proc(t) = \text{if } cond \text{ then } P(t) \text{ else } Q(t)$. If *cond* is not in *Cond*, then *cond* immediately evaluates to $True$ or $False$ and the result is immediately implied by the inductive hypothesis for $P(t)$ or $Q(t)$, respectively. So suppose *cond* is in *Cond*. Then, by the SSOS firing rules for conditional choice (see Section 4.3.2) it must be that

$$\sigma = \langle cond \rangle^\frown \rho \quad \text{or} \quad \sigma = \langle \neg cond \rangle^\frown \rho$$

for some symbolic trace ρ. We now perform a case analysis on the truth value of the evaluation of *cond* within the environments Γ_{init} and $\phi(\Gamma_{init})$.

Case 1. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = [\![cond]\!]_{\Gamma_{init}} = True$. Then it must be that $\rho \in SymbolicTraces(P(t))$. From assumption (iii) we have that

$$\rho^\frown\langle\epsilon\rangle \quad generates_{\phi(\Gamma_{init})} \quad \phi(tr)^\frown\langle e\rangle.$$

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

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

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

Case 2. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = [\![cond]\!]_{\Gamma_{init}} = False$. This case is like Case 1, above, with $Q(t)$ in place of $P(t)$.

Case 3. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = True \wedge [\![cond]\!]_{\Gamma_{init}} = False$. Then, by assumption (i) and (ii),

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

Since $Proc(t)$ satisfies **RevPosConjEqT**, we have that $(Q(t), \phi(\Gamma_{init})) \sqsubseteq_{\mathrm{T}} (P(t), \phi(\Gamma_{init}))$, so

$$\phi(tr)^\frown\langle e\rangle \in traces(Q(t), \phi(\Gamma_{init})).$$

Let $\rho'^\frown\langle\epsilon'\rangle \in SymbolicTraces(Q(t))$ be such that $\rho'^\frown\langle\epsilon'\rangle \; generates_{\phi(\Gamma_{init})} \; \phi(tr)^\frown\langle e\rangle$. Then, by Lemma 5.6, $!(\epsilon') \subseteq !(\epsilon)$. Hence, $!^t(\epsilon') \subseteq !^t(\epsilon)$, so assumption (iv) implies that

$$\forall\, i \in !^t(\epsilon') \bullet v_i \in \{0 \,..\, B-1\}.$$

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

$$\begin{array}{l} \forall\, v' \in \{1 \,..\, k\} \to Value \mid \\ \quad (\forall\, i \in \$^t(\epsilon') \cup ?^t(\epsilon') \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon') \bullet v'_i = v_i) \bullet \\ \quad\quad tr^\frown\langle c.v'_1 \ldots v'_k\rangle \in traces(Q(t), \Gamma_{init}). \end{array} \tag{B.8}$$

Suppose $v' \in \{1 \,..\, k\} \to Value$ is such that

$$(\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i).$$

The fact that $!(\epsilon') \subseteq !(\epsilon)$ implies that

$$\forall\, i \in !(\epsilon') \bullet v'_i = v_i. \tag{B.9}$$

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

$$\$^t(\epsilon') \cup ?^t(\epsilon') \cup !^t(\epsilon') \;=\; \$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon),$$

which means that

$$\$^t(\epsilon') \cup ?^t(\epsilon') \quad \subseteq \quad \$^t(\epsilon) \cup ?^t(\epsilon) \cup !^t(\epsilon).$$

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 (B.9) and (B.8), we get that

$$\forall\, v' \in \{1\,..\,k\} \to \mathit{Value} \mid (\forall\, i \in \$^t(\epsilon) \cup ?^t(\epsilon) \bullet v'_i \in T) \wedge (\forall\, i \in !(\epsilon) \bullet v'_i = v_i) \bullet \\ tr^\frown\langle c.v'_1 \ldots v'_k\rangle \in \mathit{traces}(Q(t), \Gamma_{\mathit{init}}).$$

The result now follows, because $(\mathit{Proc}(t), \Gamma_{\mathit{init}}) \stackrel{\langle\rangle}{\Longrightarrow} (Q(t), \Gamma_{\mathit{init}})$.

Case 4. Suppose that $[\![\mathit{cond}]\!]_{\phi(\Gamma_{\mathit{init}})} = \mathit{False} \wedge [\![\mathit{cond}]\!]_{\Gamma_{\mathit{init}}} = \mathit{True}$. This case is not possible since *cond* is a conjunction of equality tests and for no function ϕ we can ever have $x = y$ and $\phi(x) \neq \phi(y)$. □

We now prove Proposition 6.8. We will need the following lemma which shows that in this case non-t equivalent symbolic traces are in fact non-τ equivalent.

Lemma B.1. *Suppose σ, σ' are symbolic traces that contain no conditional symbolic events, $\sigma \equiv_{\text{non-}t} \sigma'$, neither σ nor σ' ends in τ, and*

$$P(t) \stackrel{\sigma}{\longmapsto}_s Q(t) \qquad \textit{and} \qquad P(t) \stackrel{\sigma'}{\longmapsto}_s Q'(t).$$

Then $\sigma \equiv_{\text{non-}\tau} \sigma'$ and $Q(t) = Q'(t)$.

Proof. We prove the result by induction on the number of visible symbolic events in σ and σ'. The base case of $\sigma = \sigma' = \langle\rangle$ is trivial.

Suppose $\sigma = \sigma_0{}^\frown\tau^{a^\frown}\langle\epsilon\rangle$, $\sigma' = \sigma'_0{}^\frown\tau^{b^\frown}\langle\epsilon'\rangle$, and σ_0, σ'_0 do not end in τ. Then $\sigma_0 \equiv_{\text{non-}t} \sigma'_0$, $\epsilon \equiv_{\text{non-}t} \epsilon'$, and

$$P(t) \stackrel{\sigma_0}{\longmapsto}_s Q_0(t) \stackrel{\tau^{a^\frown}\langle\epsilon\rangle}{\longmapsto}_s Q(t) \qquad \text{and} \qquad P(t) \stackrel{\sigma'_0}{\longmapsto}_s Q'_0(t) \stackrel{\tau^{b^\frown}\langle\epsilon'\rangle}{\longmapsto}_s Q'(t)$$

for some $Q_0(t)$ and $Q'_0(t)$. Then by the inductive hypothesis, $\sigma_0 \equiv_{\text{non-}\tau} \sigma'_0$ and $Q_0(t) = Q'_0(t)$. Then by Lemma A.3, the t parts of ϵ and ϵ' are equal, so $\epsilon = \epsilon'$; hence $\sigma \equiv_{\text{non-}\tau} \sigma'$. Finally, by Lemma A.1, $Q(t) = Q'(t)$. □

Proof of Proposition 6.8. Let $\sigma^\frown\langle\epsilon\rangle$ be a symbolic trace of $\mathit{Spec}(t)$. If $\sigma'^\frown\langle\epsilon'\rangle$ is a symbolic trace of $\mathit{Spec}(t)$ such that $\sigma^\frown\langle\epsilon\rangle \equiv_{\text{non-}t} \sigma'^\frown\langle\epsilon'\rangle$, then by the above lemma, $\epsilon = \epsilon'$. Hence $!^t(\sigma, \epsilon)(\mathit{Spec}(t)) = !^t(\epsilon)$. So in Theorem 6.5,

$$\begin{aligned} \mathit{Thresh}_{\mathrm{T}} &= \max\{\#!^t(\epsilon)(\mathit{Spec}(t)) \mid \sigma^\frown\langle\epsilon\rangle \in \mathit{SymbolicTraces}(P(t))\} \\ &\leq \max\{\#!^t(\alpha) \mid \alpha \text{ is a construct of } \mathit{Spec}(t)\} \end{aligned}$$

with equality in the normal case that every construct is reachable. □

B.2. **Proofs for Section 6.2.**

Proof of Proposition 6.11. We prove the result using a structural induction on $Proc(t)$. We give just the cases for prefix and conditional choice.

Prefix. Suppose that $Proc(t) = \alpha \to Proc'(t)$ for some construct $\alpha = c\S_1 x_1{:}X_1 \ldots \S_k x_k{:}X_k$ and some process syntax $Proc'(t)$. We now consider two cases.

Subcase 1. Suppose that $tr = \langle\rangle$. The fact that $(\phi(tr), X) \in failures(Proc(t), \phi(\Gamma_{init}))$ implies that there exists an environment Γ with $\mathrm{dom}(\Gamma) = \$^t(\alpha)$ such that

$$(Proc(t), \phi(\Gamma_{init})) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \phi(\Gamma_{init}) \oplus \Gamma)\ ref\ X,$$

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

$$(Proc(t), \Gamma_{init}) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \Gamma_{init} \oplus \Gamma)$$

by resolving the nondeterministic selections of α (if any) in the same way. We now show that $(P(t), \Gamma_{init} \oplus \Gamma)\ ref\ X$. Observe that the only difference between the initial events of two configurations (S, Γ_1) and (S, Γ_2) are the output values of type t that come from the environments Γ_1 and Γ_2. Therefore,

$$\begin{array}{l} initials(P(t), \Gamma_{init} \oplus \Gamma) = \\ \quad \{c.v'_1 \ldots v'_k \mid (\forall\, i \in !^t(\alpha) \bullet v'_i = (\Gamma_{init} \oplus \Gamma)\,(x_i)) \\ \qquad\qquad \wedge\, \exists\, c.v_1 \ldots v_k \in initials(P(t), \phi(\Gamma_{init}) \oplus \Gamma) \bullet \\ \qquad\qquad\qquad \forall\, i \in \{1\,..\,k\} \setminus !^t(\alpha) \bullet v'_i = v_i)\}. \end{array}$$

Let v be such that $c.v_1 \ldots v_k$ is in $initials(P(t), \phi(\Gamma_{init}) \oplus \Gamma)$ and let v' be such that $c.v'_1 \ldots v'_k$ is in $initials(P(t), \Gamma_{init} \oplus \Gamma)$ with $\forall\, i \in \{1\,..\,k\} \setminus !^t(\alpha) \bullet v'_i = v_i$. Also, let $i \in !^t(\alpha)$. Hence $v_i = (\phi(\Gamma_{init}) \oplus \Gamma)(x_i)$ and $v'_i = (\Gamma_{init} \oplus \Gamma)(x_i)$. Hence, by assumption (iii) of the proposition, $v'_i \in \{0\,..\,B-1\}$. So, thanks to the properties of ϕ, $\phi(v'_i) = v'_i$. Hence $(\phi(\Gamma_{init}) \oplus \Gamma)(x_i) = v'_i$ since $x_i \notin \mathrm{dom}(\Gamma)$. Therefore

$$\forall\, i \in !^t(\alpha) \bullet v'_i \;=\; (\Gamma_{init} \oplus \Gamma)(x_i) \;=\; (\phi(\Gamma_{init}) \oplus \Gamma)(x_i) \;=\; v_i.$$

Hence,

$$initials(P(t), \Gamma_{init} \oplus \Gamma) \;=\; initials(P(t), \phi(\Gamma_{init}) \oplus \Gamma). \tag{B.10}$$

Since $(P(t), \Gamma_{init} \oplus \Gamma)$ is stable, $(P(t), \Gamma_{init} \oplus \Gamma)\ ref\ Y$ for all $Y \subseteq \Sigma \setminus initials(P(t), \Gamma_{init} \oplus \Gamma)$. However, from the fact that $(P(t), \phi(\Gamma_{init}) \oplus \Gamma)\ ref\ X$ we can infer that $X \subseteq \Sigma \setminus initials(P(t), \phi(\Gamma_{init}) \oplus \Gamma)$, so by (B.10) $(P(t), \Gamma_{init} \oplus \Gamma)\ ref\ X$. This implies that $(\langle\rangle, X) \in failures(Proc(t), \Gamma_{init})$, as required.

Subcase 2. Suppose that $tr \neq \langle\rangle$. Then $tr = \langle e \rangle^\frown tr'$ for some visible event e that matches α and trace tr'. Let $\Gamma = \phi(\Gamma_{init}) \oplus Match(\alpha, \phi(e))$ and $\Gamma' = \Gamma_{init} \oplus Match(\alpha, e)$. From the assumptions of the proposition we can infer that

$$tr' \in traces(P(t), \Gamma') \qquad \text{and} \qquad (\phi(tr'), X) \in failures(P(t), \Gamma),$$

where $P(t)$ is like $Proc'(t)$, but with some substitutions of concrete values for the non-t type input variables of α, as dictated by the SSOS firing rules for prefix (see Section 4.3.2). Assumption (iii), combined with the fact that $(Proc(t), \Gamma_{init}) \stackrel{\langle e \rangle}{\Longrightarrow} (P(t), \Gamma')$, implies that if P is a configuration such that $(P(t), \Gamma') \stackrel{tr'}{\Longrightarrow} P$, then every output of type t of every event in $initials(P)$ is in $\{0 \,..\, B-1\}$. Observe that $\Gamma = \phi(\Gamma')$. So, by the inductive hypothesis for $P(t)$, $(tr', X) \in failures(P(t), \Gamma')$, which implies that $(tr, X) \in failures(Proc(t), \Gamma_{init})$.

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

Case 1. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = [\![cond]\!]_{\Gamma_{init}} = True$. Then

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

In addition, assumption (iii), combined with the fact that $(Proc(t), \Gamma_{init}) \stackrel{\langle\rangle}{\Longrightarrow} (P(t), \Gamma_{init})$, implies that if P is a configuration such that $(P(t), \Gamma_{init}) \stackrel{tr}{\Longrightarrow} P$, then every output of type t of every event in $initials(P)$ is in $\{0 \,..\, B-1\}$. Then, the inductive hypothesis for $P(t)$ implies that $(tr, X) \in failures(P(t), \Gamma_{init})$. Therefore, $(tr, X) \in failures(Proc(t), \Gamma_{init})$.

Case 2. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = [\![cond]\!]_{\Gamma_{init}} = False$. This case is like Case 1, above, with $Q(t)$ in place of $P(t)$.

Case 3. Suppose that $[\![cond]\!]_{\phi(\Gamma_{init})} = True \wedge [\![cond]\!]_{\Gamma_{init}} = False$. 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 $\mathbf{RevPosConjEqT}_{\mathrm{F}}$, so $(Q(t), \phi(\Gamma_{init})) \sqsubseteq_{\mathrm{F}} (P(t), \phi(\Gamma_{init})$. Hence,

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

In addition, assumption (iii), combined with the fact that $(Proc(t), \Gamma_{init}) \stackrel{\langle\rangle}{\Longrightarrow} (Q(t), \Gamma_{init})$, implies that if P is a configuration such that $(Q(t), \Gamma_{init}) \stackrel{tr}{\Longrightarrow} P$, then every output of type t of every event in $initials(P)$ is in $\{0..B-1\}$. 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 $[\![cond]\!]_{\phi(\Gamma_{init})} = False \wedge [\![cond]\!]_{\Gamma_{init}} = True$. This case is not possible, since *cond* is a conjunction of equality tests and for no function ϕ we can ever have $x = y$ and $\phi(x) \neq \phi(y)$. □