IT 11 050
Examensarbete 30 hp
Juni 2011

Exercising Psi-calculi

A Psi-calculi workbench

Ramunas Gutkovas

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Exercising Psi-calculi: a Psi-calculi workbench

Ramunas Gutkovas

This thesis presents an automated tool for manipulation and analysis of mobile concurrent systems described in the Psi-calculi framework. Psi-calculi is a family of process calculi, parameterised on data, conditions and a logic. We provide a general framework for implementing instantiations of these parameters, yielding a Psi-calculus. The tool implements simulation of Psi-calculus processes based on symbolic operational semantics, process constants for providing an environment for processes, and a symbolic bisimulation algorithm for checking bisimilarity. The tool has a command interpreter frontend for interactive use.

Handledare: Johannes Borgström
Ämnesgranskare: Björn Victor
Examinator: Anders Jansson
IT 11 050
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Developing reliable concurrent software and hardware systems is a well known challenge. In order to model such systems, we need to be able to describe them in a formal and precise way.

Process calculi are one of the means for modeling and reasoning about communicating concurrent systems which are systems composed of processes simultaneously exchanging messages between themselves. The pi-calculus [21, 24, 20] is one of the more prominent calculi; its attractiveness is due to well established simple and expressive mathematical theory and its ability to model mobility.

The pi-calculus is a minimalistic process calculus where the only data structure is a communication channel name. Nevertheless, other data structures like integers, booleans, lists can be encoded [20]. But when modeling complex systems in practice, such a minimalistic process calculus soon becomes a disadvantage as the models grow in size and irrelevant details obscure and increase the complexity of the model analysed. By departing from minimalism, a number of process calculi intended for application, building on the pi-calculus foundation, have been developed. For example, the Spi-calculus [1] extends the pi-calculus with cryptographic primitives.

Psi-calculi [4] are a framework of process calculi retaining many aspects of the pi-calculus semantic 'pureness'. Psi-calculi are parameterised with three nominal datatypes for data structures called terms, conditions, and logical assertions. The requisites on these parameters allows for a wide range of process calculi, for instance, the terms can be the lambda calculus terms, and the conditions and the assertions can be formulae in some higher order logic. In the Psi-calculi framework many proposed pi-calculus extensions are expressible, so Psi-calculi is also an attempt to unify them (see [4] for a survey of pi-calculus extensions).

The theory of Psi-calculus is formulated in Nominal Logic [25]. Nominal Logic allows for a rigorous treatment of languages with binding constructs such as Psi-calculi and their parameters. Psi-calculi have been formalised [5] in the theorem prover assistant Nominal Isabelle [28].

The operational semantics of the Psi-calculi is not directly implementable in an automated tool because of possibly infinite branching of the concrete values received in an input action. By representing the branching of concrete values with a single name and by pushing the derivation decision procedure to a later stage, the symbolic Psi-calculi semantics [17] avoids infinite branching. The symbolic semantics and non-symbolic semantics of Psi-calculi agree on the derivation of agents. The symbolic Psi-calculi semantics is used in an algorithm [18] for computing weak and strong bisimulations. The symbolic Psi-calculi semantics work is largely based on the work of [14, 15, 6].

The Psi-calculus is intended for practical modeling of concurrent communicating systems. For it to be useful in practice, an automated tool is

needed, since real-world models tend to be complex, tedious to handle, and have many details which are prone to a human error. Traditionally, automated tools with simulation and bisimulation checking have been developed for all major process calculi: tools for the pi-calculus include [30, 8], and for the spi-calculus [7].

We developed a tool for Psi-calculi called Psi-calculi Workbench. Its main components are a symbolic simulator [17] and a weak symbolic bisimulation checker [18]. We also developed a framework for implementing Psi-calculi parameters and symbolic constraint solvers for the simulator and the weak bisimulation checker. The symbolic constraints are generated by the simulator and the weak bisimulation algorithm [18].

As part of this thesis, we defined terminating symbolic operational semantics for the Psi-calculi. We showed that the original symbolic semantics and the terminating symbolic semantics coincide up to the bisimulation. We implemented the terminating symbolic operational semantics in the tool.

We extend Psi-calculi with process constants (definition and environment for processes) for conveniently defining larger models. This meant adding another form of process to Psi-calculi, and adding an invocation rule to the terminating symbolic operational semantics.

Outline: this text is divided into two major parts. In the theory part (Section 2), we first establish the theory of Nominal Logic (Section 2.1), next, the theory of Psi-calculi (Section 2.2) and the symbolic operational semantics (Section 2.3), then we discuss the terminating symbolic operational semantics (Section 2.4) and process constants (Section 2.5) in greater detail giving theorems and proofs where appropriate. In the tool part (Section 3), we discuss the tool from two perspectives: from perspective of a user operating the tool on predefined Psi-calculus instances, and from perspective of a user implementing an instance in the tool framework. We also give examples of the implementation of three Psi-calculus instances (Section 3.2).

2 Theory

In this section, we introduce Psi-calculi [4] and its symbolic operational semantics [17, 18]. We also introduce Nominal Logic [25] which we use to define Psi-calculi. We define the terminating symbolic operational semantics and establish equivalence with the symbolic operational semantics. Lastly, we introduce an extension of Psi-calculi with process constants.

2.1 Nominal logic

When doing mathematics with pen and paper on formal languages involving variable-binding constructs, bindings seemingly do not pose any difficulties. One usually needs to be careful when introducing free variables to avoid unintentional capture by a binder. Words 'careful' and 'avoid' are well understood in the context of binders, and are established as a convention. This practice is not always sound, since the intention is to work with equivalence classes, but proofs are instead done on well chosen representatives and induction on representatives may be too weak, yet this is usually glossed over. Clearly, this style of proof is not suited for machine checking or an implementation.

Nominal Logic [25] is a solution that bridges the gap between the formal and informal practice. Its purpose is twofold: give a solid mathematical footing to pen-and-paper proofs [25], and allow machine checked proofs to be as close as possible to the pen-and-paper counterparts [28]. What is more, the underlying concepts of Nominal Logic are well suited [27] to be integrated into a programming language to ease the programming task of manipulating and constructing syntactical structures. Even when such an extended programming language is not available, the Nominal Logic theory has a functional 'feel' [25]; it allows for a straightforward implementation in a functional programming language, and removes the need for case-by-case solutions.

We first examine some examples of Nominal Logic application to the lambda calculus. In the first part of this section, we introduce concepts informally, while in the second part, we give formal definitions of Nominal Logic. This section is not intended to be a complete exposition of the theory, we only present key concepts see [25] for full account.

Let us recall the lambda calculus.

$$M, N ::= \lambda x.M \mid MN \mid y$$

where x and y are variables, and x binds into M.

The cornerstone of Nominal logic is the concept of atom (name) swapping. By swapping we mean a function $(x\ y) \cdot M$ which exchanges every occurrence of x with y in M and vice versa, no matter where they occur in

M. For instance:

$$(x\ y)\cdot(\lambda y.\lambda x.y) = \lambda x.\lambda y.x$$

Note that atom swapping preserves α-equivalence. Compare this with a renaming:

$$\{x/y\}(\lambda y.\lambda x.y) = \lambda x.\lambda x.x$$

Swapping, in a sense, is a more fundamental notion than renaming, since swapping does not need to know where the variable occurs and whether it is a binder or a free variable. Swapping preserves α-equivalence if an atom is swapped with an atom which is fresh for a term.

$$(x\ y)\cdot M =_\alpha M \text{ if } y\#M$$

where $y\#M$ denotes the atom y is fresh for the lambda term M, or in this case, that the name y is not in free names of the lambda term M.

Nominal logic moves α-equivalence into the logical framework itself. In Nominal logic the following is true:

$$\lambda x.x = \lambda y.y$$

In Nominal logic, the usual capture avoiding substitution is a total function:

$$\begin{aligned} x[x := L] &= L \\ (NM)[x := L] &= N[x := L]M[x := L] \\ (\lambda y.M)[x := L] &= \lambda y.M[x := L] \quad \text{if } x\#y \end{aligned}$$

Let us now see the formal definitions of Nominal Logic. All the definitions, except for the definition of support, are first-order derivable and in fact Nominal Logic provides axioms which entail them.

Definition 1. *(Atoms (or Names)) A countably infinite set $\mathcal{N}$. Ranged over by $a, b, \ldots$.*

Any infinite countable set is suitable as a atom set. Atoms are subjected to binding, atom swapping, etc.

Definition 2 (Atom swapping)**.** *The swapping function on atoms a, b, d is defined as follows:*

$$(a\ b)\cdot d \stackrel{\text{def}}{=} \begin{cases} a & \text{if } d = b \\ b & \text{if } d = a \\ d & \text{otherwise} \end{cases}$$

A nominal set is intended to interpret the syntax of a formal language. For instance, the first-order feature allows us to express lambda-calculus as a set: $\Lambda = \mathcal{N} \cup \{\lambda x.N : x \in \mathcal{N} \wedge N \in \Lambda\} \cup \{NM : N, M \in \Lambda\}$.

Definition 3 (Nominal set)**.** *A nominal set X is a set $|X|$, such that for every element $x \in |X|$ and every pair of atoms $a, b \in \mathcal{N}$ there is a swapping $(a\ b)\cdot x$ defined such that $(a\ b)\cdot x \in |X|$.*

A nominal set is required to have the following properties.

- *Properties of swapping. For all $a, b, c, d \in \mathcal{N}$ and all $x \in |X|$:*

$$\begin{aligned}(a\ a)\cdot x &= x\\ (a\ b)\cdot(a\ b)\cdot x &= x\\ (a\ b)\cdot(c\ d)\cdot x &= ((a\ b)\cdot c\ \ (a\ b)\cdot d)\cdot(a\ b)\cdot x\end{aligned}$$

- *Finite support property. Each member of a nominal set $x \in |X|$ involves a finite number of atoms: given x, there exists a finite subset $w \subseteq_{fin} \mathcal{N}$ such that for all $a, b \in \mathcal{N} \setminus w$ holds $(a\ b)\cdot x = x$.*

The notion of support is a direct consequence of the finite support property and the following definition is provable [25].

Definition 4 (Support)**.** *Let x be a member of some nominal set, then the support of x is*

$$n(x) \stackrel{\text{def}}{=} \{a \in \mathcal{N} : \{b \in \mathcal{N} : (a\ b)\cdot x \neq x\} \text{ is not finite}\}$$

The intuition of the support of a term is a set of atoms which modify the term when swapped.

Definition 5 (Fresh)**.** *Let a be an atom and X be a nominal set, then a is said to be fresh in X*

$$a\#X \stackrel{\text{def}}{=} a \notin n(X)$$

Morphisms (functions on a underlying nominal set) on nominal sets must be equivariant. Equivariance, intuitively, means that the equality is preserved by atom swapping.

Definition 6 (Equivariance)**.** *Let $f : X \to Y$ be a morphism of nominal sets X, Y. The morphism f is equivariant if*

$$f((a\ b)\cdot x) = (a\ b)\cdot f(x)$$

In particular, a capture avoiding substitution is such a morphism (function).

Definition 7 (Nominal datatype)**.** *A Nominal datatype* **T** *is a nominal set T with a set of equivariant functions defined on it.*

Lastly, we need to be able to tell which syntactic constructs are binders.

Definition 8 (Nominal logic syntax)**.** *Nominal logic syntax is that of first-order many-sorted logic with equality, plus the following:*

- *Sorts S are divided into two: sorts of atoms (A) and sorts of data (D). Sorts S can also be formed by atom abstraction $[A]S$.*

$$S ::= A \mid D \mid [A]S$$

- *The swap function symbol $(\bullet\ \bullet)\cdot\bullet$, with arity $A, A, S \to S$.*
- *The freshness relation symbol $\bullet\#\bullet$, with arity A, S.*

Nominal Logic is a first order logic with equality, coupled with the nominal logic syntax with the obvious interpretation of nominal symbols, with equivariant relations and functions, and with the set of first-order definable underlying nominal logic axioms (see [25]).

Definitions and reasoning in Psi-calculus theory is done in Nominal Logic formalism. By using the Nominal Isabelle [28] theorem prover which is based on Nominal Logic, much of Psi-calculi meta-theory has been formalised and machine checked [5].

2.2 Psi-calculus

In this section, we give basic definitions of the Psi-calculi framework [4] together with additional requirements [18] on them needed to define the symbolic operational semantics (Section 2.3). For full account of Psi-calculi and the symbolic operational semantics we refer to [4, 18].

Psi-calculi is a family of process calculi, a member of this family is called a Psi-calculus instance. A Psi-calculus process models a synchronous concurrent communicating (message passing) system. A process P interacts with other processes in an environment, or the processes which compose the process P interact internally, and P transitions into a new process P'. A transition is labelled with an action.

We first cover the possible forms of Psi-calculus agents. Let us fix a countably infinite set of *names* $\mathcal{N}$ (Definition 1) ranged over by $a, b, \ldots, z$ for Psi-calculi.

1. An empty process $\mathbf{0}$ which performs no actions.
2. An *output prefix* process $\overline{M}\,N\,.\,P$ performs the output action $\overline{M}\,(\nu\tilde{a})N$ and transitions into the process P. The intuition is that the process sends the data (the object) N through the channel (the subject) M. Moreover, the process may send a set of private names enclosed within the object N, which are denoted with $\widetilde{a}$ in the action. A process cannot disclose more private names than necessary for the transmission of the object N.

3. An *input prefix* process $\underline{M}(x).P$ performs the input action $\underline{M}(x)$ and transitions into the process P. Intuitively, the process receives an object N through the channel M, the name x refers to the object N, and x binds into P.

4. A *case* process $\mathbf{case}\ \varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$ may enact one of the subprocesses P_i whenever respective condition φ_i holds. If several conditions hold, a process is non-deterministically chosen. The process transitions into (and performs the action) that the chosen subprocess P_i transitions into. If none of the conditions hold, the process does not have any actions, like the $\mathbf{0}$ process.

5. A *restriction* process $(\nu a)P$ acts as P but with the name a made private to the process P. The name a is considered to be distinct from names in the process' environment but it can be transmitted to other processes.

6. A *parallel* process $P \mid Q$ models concurrent execution. The processes P and Q communicate and issue the silent action τ if one process does an input action and the other does an output action with channel equivalent subjects. The processes P and Q can also act independently.

7. A *replication* process $!P$ can be though as a process with infinitely many copies of P in parallel.

8. An *assertion* process $(\!|\Psi|\!)$ is an environment which can interact with agents in parallel by triggering conditions.

Now we turn to formal definitions. We first give the formal syntax of a Psi-calculus agents P, Q. We give the terms M, N, conditions φ_i, and assertions Ψ after.

Definition 9 (Psi-calculus agents)**.** *Given valid Psi-calculus parameters as in Definitions 13 and 15, the Psi-calculus* agents*, ranged over by $P, Q, \ldots$, are of the following forms.*

$\mathbf{0}$	*Nil*
$\overline{M}\,N\,.\,P$	*Output*
$\underline{M}(x).P$	*Input*
$\mathbf{case}\ \varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$	*Case*
$(\nu a)P$	*Restriction*
$P \mid Q$	*Parallel*
$!P$	*Replication*
$(\!\|\Psi\|\!)$	*Assertion*

In the Input $\underline{M}(x)\,.\,P$, x binds its occurrences in P. Restriction $(\nu a)P$ binds a in P.

Not all agents generated by the above rules are considered wellformed.

Definition 10 (Guarded assertion)**.** *An assertion is* guarded *if it is a subterm of an Input or Output.*

Definition 11 (Wellformed agent)**.** *An agent is* well formed *if in a replication* $!P$ *there are no unguarded assertions in* P*, and there are no unguarded assertion in any* P_i *in* **case** $\varphi_1 : P_1 \,[]\, \cdots \,[]\, \varphi_n : P_n$*.*

Definition 12 (Actions)**.** *The* actions *ranged over by* α, β *are of the following three kinds:*

$\overline{M}\,(\nu\tilde{a})N$	*Output*
$\underline{M}(x)$	*Input*
τ	*Silent*

where in Output $\widetilde{a}$ *binds into* N*, and all memebers of* $\widetilde{a}$ *must occur in* N*, i.e.* $\tilde{a} \subseteq \mathrm{n}(N)$*.*

As mentioned above, Psi-calculi is parameterised with three nominal datatypes and four equivariant operations. By providing these parameters and by satisfying the requisites, we obtain a concrete process calculus, a Psi-calculus instance. The parameters are for data (including communication channels), conditions, and logical assertions.

Definition 13 (Psi-calculus parameters)**.** *A Psi-calculus requires the three (not necessarily disjoint) nominal data types:*

$\mathbf{T}$	*the (data) terms, ranged over by* M, N
$\mathbf{C}$	*the conditions, ranged over by* φ
$\mathbf{A}$	*the assertions, ranged over by* Ψ

and the four equivariant operators:

$\leftrightarrows$	:	$\mathbf{T} \times \mathbf{T} \to \mathbf{C}$	*Channel Equivalence*
$\otimes$	:	$\mathbf{A} \times \mathbf{A} \to \mathbf{A}$	*Composition*
$\mathbf{1}$	:	$\mathbf{A}$	*Unit*
$\vdash$	$\subseteq$	$\mathbf{A} \times \mathbf{C}$	*Entailment*

and substitution functions $[\tilde{a} := \tilde{M}]$*, substituting terms for names, on all of* $\mathbf{T}$*,* $\mathbf{C}$*, and* $\mathbf{A}$*.*

The channel equivalence $\leftrightarrows$ operator is intended to tell if two terms represent the same communication channel. The composition $\otimes$ operator merges two assertions into one assertion. A unit assertion $\mathbf{1}$ which does not introduce any new information to compositions. The entailment relation $\vdash$ interprets a condition based on the information in an assertion. Note that channel equivalence produces a condition, so communication can be influenced by the environment (assertion).

Two assertions are equivalent if and only if they entail the same conditions. This relation is used in requisites on the parameters.

Definition 14 (Assertion equivalence)**.** $\Psi \simeq \Psi'$ *if* $\forall \varphi.\ \ \Psi \vdash \varphi \Leftrightarrow \Psi' \vdash \varphi$

Note that the following requirements are quite lax: the channel equivalence is not required to be reflexive, and assertion composition is not required to be idempotent.

Definition 15 (Requisites on valid Psi-calculus parameters)**.**

Channel symmetry:	$\Psi \vdash M \leftrightarrows N \implies \Psi \vdash N \leftrightarrows M$
Channel transitivity:	$\Psi \vdash M \leftrightarrows N \wedge \Psi \vdash N \leftrightarrows L \implies \Psi \vdash M \leftrightarrows L$
Composition:	$\Psi \simeq \Psi' \implies \Psi \otimes \Psi'' \simeq \Psi' \otimes \Psi''$
Identity:	$\Psi \otimes \mathbf{1} \simeq \Psi$
Associativity:	$(\Psi \otimes \Psi') \otimes \Psi'' \simeq \Psi \otimes (\Psi' \otimes \Psi'')$
Commutativity:	$\Psi \otimes \Psi' \simeq \Psi' \otimes \Psi$
Weakening:	$\Psi \vdash \varphi \implies \Psi \otimes \Psi' \vdash \varphi$
Names are terms:	$\mathcal{N} \subseteq \mathbf{T}$

The last two requisites *weakening* and *names are terms* are not among the original requisites [4]. They are part of the requisites of the symbolic operational semantics (Section 2.3). Although they are quite natural, in particular *names are terms* requisite is satisfied whenever $\mathbf{T}$ is the carrier set of some term algebra (see Appendix B).

We require a substitution function defined on all the nominal datatype parameters of the Psi-calculus instance. The requirements on it give the expected substitution function.

Definition 16 (Substitution function)**.** *The Psi-calculus requisites for a substitution function on nominal datatypes are:*

1. *If* $\widetilde{a} \subseteq n(X)$ *and* $b \in n(\widetilde{T})$ *then* $b \in n(X[\widetilde{a} := \widetilde{T}])$.
2. *If* $\widetilde{b} \# X$ *then* $X[\widetilde{a} := \widetilde{T}] = ((\widetilde{b}\,\widetilde{a}) \cdot X)[\widetilde{b} := \widetilde{T}]$.

The first requirement says that a substitution function does not lose any names when substituting. If we apply the substitution on X, we can still find all the names in X that are in all the supports of terms which are in the range of the substitution function. In other words, the substitution is a capture avoiding substitution.

The second requirement tells that a substitution function is not affected by the concrete name, so that we can do substitution on α-variants.

Additionally, we require the following properties on a substitution function for use with the symbolic operational semantics (see [18, Section 7]).

Definition 17 (Symbolic substitution function)**.** *We require the following of a substitution function on nominal datatype X.*

$$\begin{aligned} X[x := x] &= X \\ x[x := M] &= M \\ X[x := M] &= X \text{ if } x\#X \\ X[x := L][y := M] &= X[y := M][x := L] \text{ if } x\#y, M \text{ and } y\#L \end{aligned}$$

2.3 Symbolic operational semantics

The Psi-calculi operational semantics [4] is not suitable for a direct implementation due to a possibly infinite branching of concrete values in an input action. We instead use the Psi-calculi symbolic operational semantics [17, 18] which is finitely branching. The main idea behind it is to abstract the possible concrete values with a single name and to collect a set of conditions which the derivation of a transition needs to satisfy, called a constraint, in order to decide if the derivation is valid. The constraint solving is performed at a later stage. In contrast with the Psi-calculi operational semantics, the Psi-calculi symbolic operational semantics may produce impossible transition derivations, but a derived transition is valid only if we can find a solution to the simultaneously derived constraint. The constraint solver can be thought as an additional parameter to the Psi-calculus instance. The symbolic Psi-calculi operational semantics is fully abstract with regard to bisimulation congruence in Psi-calculi operational semantics, i.e. the semantics agree on the derived agents.

The definitions that follow are required for the definition of the symbolic operational semantics (Figure 1).

Definition 18 (Frame)**.** *A* frame *is of the form $(\nu\widetilde{b})\Psi$ where $\widetilde{b}$ is a sequence of names that bind into the assertion Ψ. We identify alpha variants of frames.*

Definition 19 (Frame of an agent)**.** *The* frame $\mathcal{F}(P)$ of an agent P *is defined inductively as follows:*

$$\begin{aligned} &\mathcal{F}(\mathbf{0}) = \mathcal{F}(\underline{M}(x).P) = \mathcal{F}(\overline{M}\,N.P) = \mathcal{F}(\mathbf{case}\ \widetilde{\varphi} : \widetilde{P}) = \mathcal{F}(!P) = \mathbf{1} \\ &\mathcal{F}((\!|\Psi|\!)) = (\nu\epsilon)\Psi \\ &\mathcal{F}(P \mid Q) = \mathcal{F}(P) \otimes \mathcal{F}(Q) \\ &\mathcal{F}((\nu b)P) = (\nu b)\mathcal{F}(P) \end{aligned}$$

Definition 20 (Equivalence of frames)**.** *We define $F \vdash \varphi$ to mean that there exists an alpha variant $(\nu\widetilde{b})\Psi$ of F such that $\widetilde{b}\#\varphi$ and $\Psi \vdash \varphi$. We also define $F \simeq G$ to mean that for all φ it holds that $F \vdash \varphi$ iff $G \vdash \varphi$.*

$$\textsc{In}\ \frac{}{\Psi \rhd \underline{M}(x).P \xrightarrow[\{\!|\Psi \vdash M \dot{\leftrightarrow} y|\!\}]{\underline{y}(x)} P}\ y\#\Psi, M, P, x$$

$$\textsc{Case}\ \frac{\Psi \rhd P_i \xrightarrow[C]{\alpha} P'}{\Psi \rhd \mathbf{case}\ \widetilde{\varphi} : \widetilde{P} \xrightarrow[C \wedge \{\!|\Psi \vdash \varphi_i|\!\}]{\alpha} P'}$$

$$\textsc{Out}\ \frac{}{\Psi \rhd \overline{M}\,N.P \xrightarrow[\{\!|\Psi \vdash M \dot{\leftrightarrow} y|\!\}]{\overline{y}\,N} P}\ y\#\Psi, M, N, P$$

$$\textsc{Com}\ \frac{\begin{array}{c}\Psi_Q \otimes \Psi \rhd P \xrightarrow{\overline{y}(\nu \widetilde{a})N} P' \\ (\nu \widetilde{b_P})\{\!|\Psi' \vdash M_P \dot{\leftrightarrow} y|\!\} \wedge C_P \\ \Psi_P \otimes \Psi \rhd Q \xrightarrow{\underline{z}(x)} Q' \\ (\nu \widetilde{b_Q})\{\!|\Psi' \vdash M_Q \dot{\leftrightarrow} z|\!\} \wedge C_Q\end{array}}{\Psi \rhd P \mid Q \xrightarrow[C_{com}]{\tau} (\nu \widetilde{a})(P' \mid Q'[x := N])}\ \begin{array}{c}\widetilde{a}\#Q, \\ y\#z\end{array}\ \Psi' = \Psi \otimes \Psi_P \otimes \Psi_Q$$

$$\textsc{Par}\ \frac{\Psi \otimes \Psi_Q \rhd P \xrightarrow[C]{\alpha} P'}{\Psi \rhd P \mid Q \xrightarrow[(\nu \widetilde{b_Q})C]{\alpha} P' \mid Q}\ \begin{array}{c}\mathrm{bn}(\alpha)\#Q \\ \alpha = \tau \vee \mathrm{subj}(\alpha)\#Q\end{array}$$

$$\textsc{Scope}\ \frac{\Psi \rhd P \xrightarrow[C]{\alpha} P'}{\Psi \rhd (\nu b)P \xrightarrow[(\nu b)C]{\alpha} (\nu b)P'}\ b\#\alpha, \Psi$$

$$\textsc{Open}\ \frac{\Psi \rhd P \xrightarrow[C]{\overline{y}(\nu \widetilde{a})N} P'}{\Psi \rhd (\nu b)P \xrightarrow[(\nu b)C]{\overline{y}(\nu \widetilde{a} \cup \{b\})N} P'}\ \begin{array}{c}b \in \mathrm{n}(N) \\ b\#\widetilde{a}, \Psi, y\end{array} \qquad \textsc{Rep}\ \frac{\Psi \rhd P \mid !P \xrightarrow[C]{\alpha} P'}{\Psi \rhd !P \xrightarrow[C]{\alpha} P'}$$

Figure 1: Transition rules for the symbolic semantics. Symmetric versions of Com and Par are elided. In the rule Com we assume that $\mathcal{F}(P) = (\nu \widetilde{b_P})\Psi_P$ and $\mathcal{F}(Q) = (\nu \widetilde{b_Q})\Psi_Q$ where $\widetilde{b_P}$ is fresh for all of $\Psi, \widetilde{b_Q}, Q$ and P, and that $\widetilde{b_Q}$ is correspondingly fresh. We also assume that $y, z\#\Psi, \widetilde{b_P}, P, \widetilde{b_Q}, Q, N, \widetilde{a}$. In Com, $C_{com} = (\nu \widetilde{b_P}, \widetilde{b_Q})\{\!|\Psi' \vdash M_P \leftrightarrow M_Q|\!\} \wedge (\nu \widetilde{b_Q})C_P \wedge (\nu \widetilde{b_P})C_Q$. In the rule Par we assume that $\mathcal{F}(Q) = (\nu \widetilde{b_Q})\Psi_Q$ where $\widetilde{b_Q}$ is fresh for Ψ, P and α. In Open the expression $\nu\tilde{a} \cup \{b\}$ means the sequence $\tilde{a}$ with b inserted anywhere.

Definition 21 (Transition constraint)**.** *A* solution *is a pair* (σ, Ψ) *where* σ *is a substitution sequence of terms for names, and* Ψ *is an assertion. The* transition constraints*, ranged over by* C, C_t *and corresponding solutions,* $\mathrm{sol}(C)$ *are defined by:*

	Constraint	*Solutions*
$C, C' ::=$	$\mathbf{true}$	$\{(\sigma, \Psi) \ : \ \sigma$ *is a subst. sequence* $\wedge \ \Psi \in \mathbf{A}\}$
	$\mathbf{false}$	$\emptyset$
	$(\nu \widetilde{a})\{\!\{\Psi \vdash \varphi\}\!\}$	$\{(\sigma, \Psi') \ : \ \exists \widetilde{b}.\widetilde{b}\#\sigma, \Psi' \ \wedge \ ((\widetilde{a}\ \widetilde{b}) \cdot \Psi)\sigma \otimes \Psi' \vdash ((\widetilde{a}\ \widetilde{b}) \cdot \varphi)\sigma\}$
	$C \wedge C'$	$\mathrm{sol}(C) \cap \mathrm{sol}(C')$

In $(\nu \widetilde{a})\{\!\{\Psi \vdash \varphi\}\!\}\ \widetilde{a}$ *are binding occurrences into* Ψ *and* φ*. We let* $(\nu \widetilde{a})(C \wedge C')$ *mean* $(\nu \widetilde{a})C \wedge (\nu \widetilde{a})C'$*, and we let* $(\nu \widetilde{a})\mathbf{true}$ *mean* $\mathbf{true}$*, and similarly for* $\mathbf{false}$*. We adopt the notation* $(\sigma, \Psi) \models C$ *to say that* $(\sigma, \Psi) \in \mathrm{sol}(C)$*.*

In the symbolic operational semantics, the transition relation has the following form.

$$P \ \xrightarrow[C]{\alpha} \ P'$$

The weak symbolic transitions abstract from the internal interactions of processes.

Definition 22 (Weak symbolic transitions)**.**

$$\Psi \rhd P \ \Longrightarrow_{\mathbf{true}} \ P$$

$$\textit{if } \Psi \rhd P \ \xrightarrow[C]{\tau} \ P'' \ \wedge \ \Psi \rhd P'' \ \Longrightarrow_{C'} \ P' \textit{ then } \Psi \rhd P \ \Longrightarrow_{C \wedge C'} \ P'$$

$$\textit{if } \Psi \rhd P \ \Longrightarrow_{C} \ P'' \wedge \Psi \rhd P'' \ \xrightarrow[C']{\alpha} \ P''' \wedge \Psi \rhd P''' \ \Longrightarrow_{C''} \ P'$$
$$\textit{then } \Psi \rhd P \ \Longrightarrow_{C \wedge C' \wedge C''} \ P'$$

2.4 Terminating symbolic operational semantics

The derivation of transitions using the symbolic operational semantics discussed in the previous Section 2.3 is non-terminating. The reason for non-termination is the replication rule Rep (Figure 1); the Rep rule generates infinite derivation trees (see [9] for treatment of termination, as the one presented here, and other forms of replication in process calculi).

Consider the following agent.

$$!\underline{a}\,(x).P$$

The following is a possible derivation tree of a transition for this agent.

$$\textsc{Rep}\ \dfrac{\textsc{Par-R}\ \dfrac{\textsc{Rep}\ \dfrac{\textsc{Par-R}\ \dfrac{\vdots}{\underline{a}\,(x).P \mid !\underline{a}\,(x).P \xrightarrow[\cdots]{\cdots} \cdots}}{!\underline{a}\,(x).P \xrightarrow[\cdots]{\cdots} \cdots}}{\underline{a}\,(x).P \mid !\underline{a}\,(x).P \xrightarrow[\cdots]{\cdots} \cdots}}{!\underline{a}\,(x).P \xrightarrow[\cdots]{\cdots} \cdots}$$

so if we kept applying the Par rule to the right agent of the parallel while applying the Rep rule we would be repeating this procedure ad infinitum. Hence the symbolic operational semantics is non-terminating.

But how to know when to stop? Intuition tells us that we do not need to generate an infinite number of same agents to derive a transition, and it is surely not the intention of the Psi-calculus designers. But how many do we need? Since Psi-calculi features a point-to-point communication between agents and we can express non-deterministic choice with a case form, a replication process may produce agents which communicate among themselves, for instance the following agent

$$!(\textbf{case}\ a = a : \underline{b}\,(x).P\ [\!]\ a = a : \overline{b}\,y.Q)$$

silently transitions into

$$P[x := y]\ \mid\ Q\ \mid\ !(\textbf{case}\ a = a : \underline{b}\,(x).P\ [\!]\ a = a : \overline{b}\,y.Q)$$

Another possibility is, as in the first example, one replicated copy does an input or an output action. But instead of counting the uses of Rep while computing derivation trees, and as a result complicating the derivation logic, we replace the Rep rule with two new rules which make the symbolic operational semantics terminating.

In figure 2, we present the Psi-calculi terminating symbolic operational semantics. The rule Rep-Com derives a silent transition if the replication produces self communicating agents. Note new rules have structurally smaller agents in their premises.

The idea and structure of Rep-Com is simple, and yet it may be lost in the verbosity arising from the need of stating constraints explicitly; for clarity we give the rule stripped from the symbolic constraints:

$$\frac{P \xrightarrow{\overline{M}\,N} P' \qquad P \xrightarrow{\underline{M}\,(x)} P''}{!P \xrightarrow{\tau} P \mid P''[x := N] \mid !P}$$

Up until now we only discussed the feasibility of the terminating symbolic operational semantics and argued for it informally. Here we establish

$$\textsc{Rep-Com}\ \frac{\Psi \rhd P \xmapsto{(\nu\widetilde{b})\{\!\{\Psi \vdash M \dot{\leftrightarrow} y\}\!\} \wedge C_{P'}} P' \quad guarded(P) \quad \widetilde{b},\widetilde{b'}\#\Psi,!P \quad \widetilde{b'}\#\widetilde{b} \quad \Psi \rhd P \xmapsto{(\nu\widetilde{b'})\{\!\{\Psi \vdash M' \dot{\leftrightarrow} z\}\!\} \wedge C_{P''}} P'' \quad y\#z \quad \widetilde{a}\#P}{\Psi \rhd !P \xmapsto{(\nu\widetilde{b},\widetilde{b'})\{\!\{\Psi \vdash M \dot{\leftrightarrow} M'\}\!\} \wedge (\nu\widetilde{b})C_{P'} \wedge (\nu\widetilde{b'})C_{P''}} (\nu\widetilde{a})(P' \mid P''[x := N]) \mid !P}$$

$$\textsc{Rep-I}\ \frac{\Psi \rhd P \xmapsto[C]{\alpha} P' \quad guarded(P)}{\Psi \rhd !P \xmapsto[C]{\alpha} P' \mid !P}\ bn(\alpha)\#!P$$

Figure 2: The Psi-calculic terminating symbolic operational semantics. The semantics are obtained by replacing the Rep rule in the Psi-calculi symbolic operational semantics (Figure 1) with rules Rep-Com and Rep-I.

that the terminating symbolic operational semantics derive the same transitions as the symbolic operational semantics and vice versa (Lemma 24 and Lemma 23).

Using the terminating symbolic operational semantics it is possible to contract $P \mid !P$ to $!P$ modulo bisimilarity.

Lemma 23 (Replication contraction)**.** *If* $\Psi \rhd P \mid !P \xmapsto[C]{\alpha} P'$ *then there is* P'' *such that* $\Psi \rhd !P \xmapsto[C']{\alpha} P''$ *and* $P' \sim P''$ *and* $sol(C) = sol(C')$.

Proof. The proof goes by investigation of the possible derivations of

$$\Psi \rhd P \mid !P \xmapsto[C]{\alpha} P'$$

First we determine all possible transition derivation cases of $P \mid !P$. By doing this, we get assumptions which are the top-most premises and from these assumptions and $!P$ we derive a bisimilar agent.

- Suppose $\Psi \rhd P \mid !P \xmapsto[C]{\alpha} P'$ was derived as follows.

$$\textsc{Par-L}\ \frac{\Psi \otimes \mathbf{1} \rhd P \xmapsto[C]{\alpha} P''}{\Psi \rhd P \mid !P \xmapsto[C]{\alpha} P'' \mid !P}$$

This can be simulated by the following and by using $\Psi \otimes \mathbf{1} \simeq \Psi$:

$$\textsc{Rep-I}\ \frac{\Psi \rhd P \xmapsto[C]{\alpha} P''}{\Psi \rhd !P \xmapsto[C]{\alpha} P'' \mid !P}$$

- The next derivation to consider:

$$\textsc{Par-R}\ \frac{\textsc{Rep-I}\ \dfrac{\Psi\otimes\mathbf{1}\ \rhd\ P\ \overset{\alpha}{\underset{C}{\longmapsto}}\ P'}{\Psi\otimes\mathbf{1}\rhd !P\ \overset{\alpha}{\underset{C}{\longmapsto}}\ P'\mid !P}}{\Psi\ \rhd\ P\mid !P\ \underset{(\nu\widetilde{b}_P)C}{\overset{\alpha}{\longmapsto}}\ P\mid(P'\mid !P)}$$

  This can be simulated by:

$$\textsc{Rep-I}\ \frac{\Psi\ \rhd\ P\ \overset{\alpha}{\underset{C}{\longmapsto}}\ P'}{\Psi\rhd !P\ \overset{\alpha}{\underset{C}{\longmapsto}}\ P'\mid !P}$$

  It is easy to see that both derivatives are structurally equivalent: $P|(P'|!P)\ \dot{\sim}_\Psi\ (P'|!P)|P\ \dot{\sim}_\Psi\ P'|(!P|P)\ \dot{\sim}_\Psi\dot{\sim}_\Psi\ P'|(P|!P)\ \dot{\sim}_\Psi\ P'|!P$.

  We know $\mathcal{F}(P)=(\nu\widetilde{b}_P)\mathbf{1}$ and $\widetilde{b}_P\#\Psi,!P,\alpha$ in the Par-R rule above, hence $\widetilde{b}_P\#C$. And from Definition 21 of solutions we know that $sol(C)=sol((\nu\widetilde{b}_P)C)$.

- The final derivation to consider is when $!P$ is "self communicating":

$$\textsc{Com}\ \frac{\textsc{Rep-I}\ \dfrac{\begin{array}{c}\Psi\otimes\mathbf{1}\ \rhd\ P\ \underset{(\nu\widetilde{b_P})\{\!\{\Psi'\vdash M\dot{\leftrightarrow}y\}\!\}\wedge C}{\overset{\overline{y}(\nu\widetilde{a})N}{\longmapsto}}\ P'\\ \Psi\otimes\mathbf{1}\ \rhd\ P\ \underset{(\nu\widetilde{b'_P})\{\!\{\Psi'\vdash M'\dot{\leftrightarrow}z\}\!\}\wedge C'}{\overset{\underline{z}(x)}{\longmapsto}}\ P''\end{array}}{\Psi\otimes\mathbf{1}\rhd !P\ \underset{(\nu\widetilde{b'_P})\{\!\{\Psi'\vdash M'\dot{\leftrightarrow}z\}\!\}\wedge C'}{\overset{\underline{z}(x)}{\longmapsto}}\ P''\mid !P}}{\Psi\ \rhd\ P\mid !P\ \underset{(\nu\widetilde{b_P},\widetilde{b'_P})\{\!\{\Psi'\vdash M\dot{\leftrightarrow}M'\}\!\}\wedge(\widetilde{b'_P})C\wedge(\widetilde{b_P})C'}{\overset{\tau}{\longmapsto}}\ (\nu\widetilde{a})(P'\mid(P''\mid !P)[x:=N])}$$

  where $\Psi'=\Psi\otimes\mathbf{1}\otimes\mathbf{1}\simeq\Psi$.

  This is simulated by the Rep-Com rule given above. Now to get the required derivative:

$$\begin{array}{l}(\nu\widetilde{a})(P'\mid(P''\mid !P)[x:=N])\\ \quad=(\nu\widetilde{a})(P'\mid(P''[x:=N]\mid !P))\\ \qquad x\#!P \text{ side condition of Rep-I}\\ \quad\dot{\sim}_\Psi\ (\nu\widetilde{a})((P'\mid P''[x:=N])\mid !P)\\ \qquad\text{associativity}\\ \quad\dot{\sim}_\Psi\ (\nu\widetilde{a})(P'\mid P''[x:=N])\mid !P\\ \qquad\text{due to } \widetilde{a}\#!P \text{ side condition of Com rule}\end{array}$$

No further possible derivations; proof is complete.

□

Lemma 24 ($\rightarrow$ simulated by $\mapsto$)**.**
Whenever $\Psi \rhd P \xrightarrow[C]{\alpha} P'$ *then there is* P'' *such that* $\Psi \rhd P \overset{\alpha}{\underset{C'}{\longmapsto}} P''$ *and* $P' \sim P''$ *and* $sol(C) = sol(C')$.

Proof. Proof goes on the length of derivation of $\xrightarrow[C]{\alpha}$.

We only consider Rep rule case, as other rules can be trivially simulated. By induction hypothesis we have $\Psi \rhd P \mid !P \overset{\alpha}{\underset{C}{\longmapsto}} P''$ and $P' \sim P''$; we need to prove (simulate) $\Psi \rhd !P \overset{\alpha}{\underset{C}{\longmapsto}} P'''$ and $P' \sim P'''$. By applying Lemma 23 to the induction hypothesis, we get the right conclusion and that $P'' \sim P'''$, and finally by transitivity $P' \sim P'''$.

□

Lemma 25 ($\mapsto$ simulated by $\rightarrow$)**.**
Whenever $\Psi \rhd P \overset{\alpha}{\underset{C}{\longmapsto}} P'$ *then there is* P'' *such that* $\Psi \rhd P \xrightarrow[C]{\alpha} P''$ *and* $P' \sim P''$.

Proof. By the length of the derivation of $\mapsto$. We will consider the two cases Rep-Com, and Rep-I as others are trivial.

Rep-Com Suppose we used the Rep-Com rule for derivation, then we simulate that as in figure 3. The premises come from the induction hypothesis and to get the desired derivative we follow the same reasoning as in the last case of the proof of Lemma 23.

Rep-I This case is simulated by the following.

$$\textsc{Rep}\ \dfrac{\textsc{Par-L}\ \dfrac{\Psi \otimes \mathbf{1} \rhd P \overset{\alpha}{\underset{C}{\longmapsto}} P'}{\Psi \rhd P \mid !P \overset{\alpha}{\underset{C}{\longmapsto}} P' \mid !P}}{\Psi \rhd !P \overset{\alpha}{\underset{C}{\longmapsto}} P' \mid !P}$$

□

We defined terminating symbolic operational semantics and established that it gives equivalent transitions with regard to the symbolic operational semantics up to bisimulation.

$$\textsc{Com}\ \dfrac{\mathbf{1}\otimes\Psi\ \rhd\ P\ \overset{\overline{y}(\nu\widetilde{a})N}{\underset{(\nu\widetilde{b})\{\!|\Psi\vdash M\dot{\leftrightarrow}y|\!\}\wedge C_{P'}}{\longmapsto}}\ P' \qquad \textsc{Rep}\ \dfrac{\textsc{Par-L}\ \dfrac{\mathbf{1}\otimes\Psi\otimes\mathbf{1}\ \rhd\ P\ \overset{\underline{z}(x)}{\underset{(\nu\widetilde{b'})\{\!|\Psi\vdash M'\dot{\leftrightarrow}z|\!\}\wedge C_{P''}}{\longmapsto}}\ P''}{\mathbf{1}\otimes\Psi\ \rhd\ P\mid !P\ \overset{\underline{z}(x)}{\underset{(\nu\widetilde{b'})\{\!|\Psi\vdash M'\dot{\leftrightarrow}z|\!\}\wedge C_{P''}}{\longmapsto}}\ P''\mid !P}}{\mathbf{1}\otimes\Psi\rhd !P\ \overset{\underline{z}(x)}{\underset{(\nu\widetilde{b'})\{\!|\Psi\vdash M'\dot{\leftrightarrow}z|\!\}\wedge C_{P''}}{\longmapsto}}\ P''\mid !P}}{\Psi\ \rhd\ P\mid !P\ \overset{\tau}{\underset{(\nu\widetilde{b},\widetilde{b'})\{\!|\Psi\vdash M\dot{\leftrightarrow}M'|\!\}\wedge(\nu\widetilde{b'})C_{P'}\wedge(\nu\widetilde{b})C_{P''}}{\longmapsto}}\ (\nu\widetilde{a})(P'\mid(P''\mid !P)[x:=N])}$$

$$\textsc{Rep}\ \dfrac{\Psi\rhd !P\ \overset{\tau}{\underset{(\nu\widetilde{b},\widetilde{b'})\{\!|\Psi\vdash M\dot{\leftrightarrow}M'|\!\}\wedge(\nu\widetilde{b'})C_{P'}\wedge(\nu\widetilde{b})C_{P''}}{\longmapsto}}\ (\nu\widetilde{a})(P'\mid(P''\mid !P)[x:=N])}{}$$

Figure 3: Derivation tree, a part of the proof of lemma 25.

2.5 Process constants: abstraction and parameters

The Psi-calculi does not provide a direct way of defining and invoking process constants, since it does not have the usual environment for definitions, and a construct for invoking those definitions in processes. Let us illustrate what we mean, the example below defines two process constants SEND and RECV, where SEND takes two arguments as terms and RECV takes one argument as a term.

$$\begin{array}{rcl} \textsc{Send}(ch, obj) & \Leftarrow & \overline{ch}\, obj \\ \textsc{Recv}(sock) & \Leftarrow & \underline{sock}(x).\overline{sock}\, x \end{array}$$

The following is a valid agent with two process constant invocations, where x, y are names (and terms).

$$\begin{array}{rcll} \textsc{Send}\langle y, x\rangle & | & \textsc{Recv}\langle y\rangle & = \\ \overline{ch}\, obj[ch := y, obj := x] & | & \underline{sock}(x).\overline{sock}\, x[sock := y] & = \\ \overline{y}\, x & | & \underline{y}(x).\overline{y}\, x & \xrightarrow{\tau} \overline{y}\, x \end{array}$$

The intention of the process constants is to provide a convenient way of defining aliases for more complicated agents when composing larger models.

Instead of using terms in the *invocation* form, we chose to restrict it to constants in order to be able to do static analysis, for instance, to inform the user of exceptional cases: whenever the environment contains several definitions with the same constant, or an invocation is used of a clause which has a non-empty support.

Formally, we introduce the following new agent form called *invocation*.

$$A\langle \widetilde{M}\rangle$$

where A is a process constant (ranged over by $A, B, \ldots$) with empty support, which identifies an agent (possibly, multiple agents) in an environment consisting of clauses:

$$A(\widetilde{x}) \Leftarrow P$$

where $n(P) \subseteq \widetilde{x}$. The set of clauses is required to be a nominal datatype, here Ct is a fixed countably infinite set of constants.

$$\mathbf{Cl}_{Ct} = \{A(\widetilde{x}) \Leftarrow P \;:\; A \in \mathrm{C} \wedge \widetilde{x} \in \mathcal{N}^{|\widetilde{x}|} \wedge P \in \mathbf{P} \wedge \text{guarded } P\}$$

Additionally, we also allow a non-deterministic behaviour, such as

$$\begin{array}{rcl} \textsc{SendRecv}(sock) & \Leftarrow & \overline{sock}\, sock \\ \textsc{SendRecv}(sock) & \Leftarrow & \underline{sock}(x) \end{array}$$

allowing the invocation of $\textsc{SendRecv}\langle y\rangle$ to either send or receive.

We say that a function $e : Ct \to \mathcal{P}_{fin}(\mathbf{Cl}_{Ct})$ is an environment. We parameterise the symbolic operational semantics with an environment, and we get the following transition system:

$$\Psi, e \rhd P \xrightarrow[C]{\alpha} P'$$

Furthermore, we add the following rule to the symbolic operational semantics (Figure 1) to obtain Psi-calculi with process constants.

$$\textsc{Invocation}\ \frac{n(P) \subseteq \widetilde{x} \qquad |\widetilde{x}| = |\widetilde{M}| \qquad guarded_e(P) \qquad A(\widetilde{x}) \Leftarrow P \in e \qquad \Psi, e \rhd P[\widetilde{x} := \widetilde{M}] \xrightarrow[C]{\alpha} P'}{\Psi, e \rhd A\langle\widetilde{M}\rangle \xrightarrow[C]{\alpha} P'}$$

Note that we fix the environment when deriving, therefore we do not need to merge environments in Par and Com (Figure 1) rules. We only need to propagate the fixed one.

What is more, the A in $A\langle\widetilde{M}\rangle$ is a constant, consequentially, it is immune to substitution $A\langle\widetilde{M}\rangle[\widetilde{x} := \widetilde{L}] = A\langle\widetilde{M}[\widetilde{x} := \widetilde{L}]\rangle$.

As the last step we need to extend the *guarded* definition for taking into account the passed environment. Let *guarded* be defined as in [3, Section 24.5] and extended with a second parameter which is passed unchanged, and equipped with an additional clause, so the full definition:

$$\begin{aligned}
&guarded_e((\!|\Psi|\!), V) = false\\
&guarded_e(\mathbf{0}, V) = guarded_e(\underline{M}\,(x).P, V) = guarded_e(\overline{M}\,N.Q, V) = true\\
&guarded_e(P \mid Q, V) = guarded_e(P, V) \wedge guarded_e(Q, V)\\
&guarded_e((\nu x)P, V) = guarded_e(!P, V) = guarded_e(P, V)\\
&guarded_e(\mathbf{case}\ \varphi_i : P_i, V) = (\forall i) guarded_e(P_i, V)\\
&guarded_e(A\langle\widetilde{M}\rangle, V) =\\
&\qquad (\forall (A(\widetilde{x}) \Leftarrow P) \in e(A))\ \ \langle A, |\widetilde{M}|\rangle \notin V \ \wedge\ guarded_e(P, V \cup \{\langle A, |\widetilde{M}|\rangle\})
\end{aligned}$$

In this section we defined a new form of process, invocation, and extended the symbolic operational semantics with the Invocation rule. Although invocations need to be guarded, process constants are useful as a convenience for developing larger Psi-calculi models.

3 Workbench

In this section we present the tool. In the first part, we present the user's guide of the tool. In the second part, we present sample implementations for extending the tool with Psi-calculus instances.

3.1 Tool

3.1.1 Syntax

We have three levels of syntactical categories (inspired by the Isabelle syntax [23]):

- *Command interpreter.* Handled by the command interpreter parser.
- *Psi-calculus agents.* Handled by the Psi-calculus agent parser, a part of command interpreter parser. Some of the parsing is delegated to the user provided instance implementation.
- *Instance.* Handled by user implemented parsers.

The *Psi-calculus agents* syntax is summarised in the table 1. The agent parser accepts an input in `ASCII`.

Form	Notation	ASCII
Nil	$\mathbf{0}$	`0`
Output	$\overline{M}N.P$	`'M<N>.P`
AbbrInput	$\underline{M}(x).P$	`M(x).P`
Input	$\underline{M}(\lambda\tilde{x})N.P$	`M(\x1, ···, xn)N.P`
Restriction	$(\nu a)P$	`(new a)P`
Replication	$!P$	`!P`
Assertion	$(\!\vert\Psi\vert\!)$	`(| Psi |)`
Invocation	$A\langle\tilde{M}\rangle$	`A<M1, ···, Mn>`
Parallel	$P \mid Q$	`P | Q`
Case	$\mathbf{case}\ \phi_1 : P_1 \ [] \ \cdots \ [] \ \phi_n : P_n$	`case phi1 : P1 [] ··· [] phin : Pn`

Table 1: Forms handled by the parser. Note: the abbreviated input form **AbbrInput** is defined as $\underline{M}(x).P =_{\text{def}} \underline{M}(\lambda x)x.P$.

In Table 1 **T** are ranged over by `M`,`N`, **C** are ranged over by `phi`, **A** ranged by `Psi`, and names $\mathcal{N}$ are ranged over by `a`, `x`. These parameters belong to the *instance* syntactical category. The parser either accepts alpha-numeric strings or quoted strings. Since the parameters may have an arbitrary syntax, in most occurrences they need to be quoted. The following is a list of quotations.

- {* ··· *} is a string, where ··· is any string of characters not containing '*}', and '*}' cannot be escaped.
- " ··· " is a string of characters, where ··· does not contain '"', but it can be escaped as '\"'.
- ' ··· ', similarly, as above, just with ''' as quotation character.

The parser treats the following lexical objects as whitespace:

- In-line comments are started with '--' and they span until the end of the line.
- Multi-line comments are (* ··· *). Can be nested.
- The usual whitespace characters.

The forms in Table 1 is listed in a precedence order from highest to the lowest. The **Parallel** is right associative. For instance, the following agent

$$\textbf{case}\ \varphi : (\nu a)P \mid P' \,[]\, \varphi :!Q \mid Q' \mid Q''$$

would be parsed as

$$(\textbf{case}\ \varphi : (((\nu a)P) \mid P') \,[]\, \varphi : (!Q)) \mid (Q' \mid Q'')$$

The following is an example Pi-calculus instance script accepted by the parser:

```
(*
    Multi-line comment
*)
case "a = b" : (new a)'b<a>
  [] "a = b" : b(x)
   | -- in parallel with the above 'case' (in-line comment)
     ! 'b<a>.0 ;
```

and the interpreter outputs the accepted agent:

```
case "a = b" : (new a)( 'b<a> ) [] "a = b" : (b(x)) | (!('b<a>))
```

Notice that term equality in above example needs to be quoted, as it is not alpha-numeric.

The formal grammar is given in Appendix D.

3.1.2 Command interpreter

The command interpreter manages the process clause environment and provides various commands for manipulating and analysing agents. In the command interpreter every command must be separated by a semicolon ';'. If the command interpreter does not recognise a command then it treats the input as an agent and passes that agent to the `agent` command. The following is a list of commands supported by the command interpreter.

- `agent` P accepts an agent P and pretty prints it. This command also supports the following forms:
  - `agent n(`P`)` computes the support of the agent P and prints the set of names of that support.
  - `agent` P`[x:=M,y:=N,...]` applies the provided substitution function to the agent P and pretty prints the result.
  - `agent guarded(`P`)` checks if the assertions in the agent P are guarded, prints `true` or `false`.
  - `agent` P `=` Q checks if P and Q are α-equivalent, and print `true` or `false`.

  The keyword `agent` may be ommited.
- `sstep` P enters the *strong* symbolic execution simulator for the agent P. Simulator supports the following commands, which must be separated by a *newline*.
  - `N` where `N` is the number of a transition derived by the simulator. Upon entering this number the simulator chooses the derivative with this number and computes new transitions from that derivative.
  - `b` backtracks to the previous derivative.
  - `q` quits the simulator.
- `wsstep` P enters the *weak* symbolic execution simulator for the agent P using the. The command language is the same as for `sstep`.
- `env` prints the current process clause environment.
- `drop` A removes all process clauses for the constant A from the environment.
- `input "`$file$`"` reads commands from the file $file$ (quotes are not part of the file name, also any type of quotation can be used in place of quotes).
- `exit` exits the interpreter.
- `def { A(x,y,...) <= P ; B(x,y,...) <= Q ; ... }`.

  Inserts process clauses into the environment. Process clauses are separated by a semicolon ';'. Any number of clauses can be given. Newlines and other whitespace are insignificant. Clauses can be given the same name, introducing non-determinism, and the clauses may be mutually recursive.

- `A(x,y,...)<= P` is an abbreviation for `def {A(x,y,...)<= P;}`.

  Multiple clauses with the same name can only be introduced inside the `def` form, otherwise old clauses are replaced and the command interpreter issues a warning.

- P `~` Q runs the bisimulation algorithm (Appendix C.1) on P and Q, and prints the simplified constraint and the solution to that constraint if there is one.

Any of the P, Q above are of the syntax given in Section 3.1.1.

3.1.3 Loading the command interpreter

The command interpreter and Psi-calculus instance are loaded from a running SML interpreter. For instance the Pi-calculus instance can be loaded by the following:

```
use "workbench.ML";
use "pi.ML";
Pi.start();
```

where workbench.ML is the main Workbench SML file containing the required definitions for implementing a Psi-calculus instance for Workbench. The file pi.ML has the definition of the Pi-calculus instance, see the example in Section 3.2.1. The function Pi.start enters the Workbench command interpreter.

3.1.4 Sample session

In this section, we demonstrate a sample session of interaction with the command interpreter using the Pi-calculus instance. We load the command interpreter as described above in Section 3.1.3.

We typeset the command interpreter's prompt as

psi>

the input text as

```
command_name argument1 argument2 ... ;
```

and the output of the command interpreter as

Command output

Let us consider the following agent as an running example.

$$\underline{a}\,(x).\overline{hello}\,x \;\mid\; \overline{a}\,world$$

We expect the above agent to τ transition into

$$\overline{hello}\,world \mid \mathbf{0}$$

Let us try this in the loaded command interpreter with a command `sstep`, which expects an agent as argument.

```
psi> sstep a(x).'hello<x> | 'a<world> ;
Type <num> for selecting derivative, b - for backtracking, q - quit
3 possible derivative(s)
1 ---
  1 |>
    --|tau|-->

      Constraint:
        {| "a = a" |}
      Solution:
        ([], 1)
      Derivative:
        ('hello<world>) | (0)

2 ---
  1 |>
    --|ga(x)|-->

      Constraint:
        {| "a = ga" |}
      Solution:
        ([ga := a], 1)
      Derivative:
        ('hello<x>) | ('a<world>)

3 ---
  1 |>
    --|ga world|-->

      Constraint:
        {| "a = ga" |}
      Solution:
        ([ga := a], 1)
      Derivative:
        (a(x). 'hello<x>) | (0)

sstep>
```

We are presented with three possible derivatives of the agent, and we are also presented with a different prompt which signifies the command interpreter of `sstep` simulator. A brief description of the commands of `sstep` are given at the top of the output above. The one we are most

interested in is the first, the τ transition. The interpretation of the above output marked as `1 --` is under the constraint {`|"a = a"|`} and a solution `([],1)` found for the constraint, where `[]` is the identity substitution and `1` is the unit assertion, we can derive the agent `hello<world> | 0`.

The other possible derivatives are the agents transitioning separately. The second transition is an input whenever the name `a` is channel equivalent to a freely chosen name `ga`. The third transition is an output.

So, let us follow the first transition

```
sstep> 1

1 possible derivative(s)
1 ---
  1 |>
    --|ga world|-->

      Constraint:
        | "hello = ga" |
      Solution:
        ([ga := hello], 1)
      Derivative:
        (0) | (0)
```

Now the only possible transition for the agent is an output on the label `hello`. By choosing that derivative the interpreter outputs

```
sstep> 1
0 possible derivative(s)
```

At any point, we may quit the `sstep` command interpreter with the command `q` or return to the previous transitions (arbitrary times) by using the `b` command.

```
sstep> q
psi>
```

In the above interaction, we entered the process directly in the `sstep` command. Although this was convenient, for bigger agents it may become cumbersome. We may provide aliases for agents, for the left hand agent

```
psi> A(ch,hello) <= ch(x). 'hello<x>;

  A(ch,hello) <= ch(x). 'hello<x>
;
```

and the right hand agent

```
psi> B(ch, world) <= 'ch<world>;

  B(ch, world) <= 'ch<world>
;
```

We may inspect the current environment after the definitions

```
psi> env;

  B(ch, world) <= 'ch<world>

  A(ch, hello) <=  ch(x). 'hello<x>
```

By using the `sstep` command we get the same transitions as follows

```
psi> sstep A<a,hello> | B<a, world>;
Type <num> for selecting derivative, b - for backtracking, q - quit
3 possible derivative(s)
1 ---
  1 |>
    --|tau|-->

      Constraint:
        {| "a = a" |}
      Solution:
        ([], 1)
      Derivative:
        ('hello<world>) | (0)
⋮
```

Other derivatives are omitted.

We provide an alternative way of defining process constants, the `def` block. In the `def` block agents may reference other agents mutually recursively.

```
psi>
    def {
        C(a,hello,world) <= B<a,hello> | A<a,world> ;
        A(a,world) <= 'a<world>;
        B(a,hello) <= a(x).'hello<x>;
    };


def {
  C(a, hello, world) <= (B<a, hello>) | (A<a, world>);
  A(a, world) <= 'a<world>;
  B(a, hello) <= a(x). 'hello<x>;
};

  -- Warning: redefined clause A
  -- Warning: redefined clause B
```

The command interpreter issues two warnings. Both warnings are to notify us that we replaced previous definitions.

If we execute the following command, we get the expected transitions.

```
psi> sstep C<a,hello,world>;
```

Lastly, an important note is that data types with non alpha-numeric syntax need to be quoted, for instance, the condition $a \leftrightarrow b$ of the agent

$$\textbf{case } a \leftrightarrow b : \mathbf{0}$$

needs to be quoted in command interpreter as

```
psi> agent case "a = b" : 0;
 case "a = b" : 0
```

For a complete list of commands supported by the command interpreter refer to section 3.1.2.

3.2 Implementation of Psi-calculus instance

In this section we give examples on how to go from a mathematical description of a Psi-calculus instance to implementation in the tool. We consider three Psi-calculus instances: the first is the Pi-calculus instance where terms are names; the second is the Frame Hopping Spread Spectrum which has structured terms; and the third is Common Ether which has non-trivial assertions. These instances all appeared in [4].

The first example, the Pi-calculus instance, is a gentle introduction to an implementation of an instance. We describe this example in greater detail compared to the other examples we give. In this example only, we implement a bisimulation constraint solver, which can be found in Appendix A.

In all three examples, we mainly focus on developing and implementing a constraint solver for the transition constraints (see Section 2.3), since this is required by a working simulator.

We also give explanations of the underlying theory and present the definitions of the instances.

3.2.1 Pi-calculus instance

As our first example we will implement a Pi-calculus symbolic instance based on [4, Section 2.4] with some minor divergence to the Psi-calculus nominal datatypes. The Pi-calculus instance does *not* feature the more advanced capabilities of Psi-calculus, e.g. non-trivial assertions, and terms with binders, therefore it is a perfect means to show the mechanics of implementing an instance without concerning ourselves with more intricate details.

This section is a complete step-by-step presentation of the SML implementation of the Pi-calculus instance in literate programming style. SML

lines are numbered to make a clear distinction between the running text and the SML code.

The instance is defined in three steps:

- First we define nominal datatypes, equivariant operators, substitution functions and functions deciding alpha equality. These requirements are listed in the PSI_INSTANCE_NOM signature.
- Next we define the requirements needed to construct a symbolic Psi-calculus simulator on the instance. The requirements are a function which maps names to terms, a constraint solver for transition constraints, and a constraint solver for constraints produced by the bisimulation algorithm. The SYMBOLIC_PSI_FLAT signature satisfies the requirements for this step.
- The final step is to define functions for pretty printing and parsing the nominal datatypes. The signature for this is C_PSI.

The previously mentioned signatures in fact are part of the C_PSI interface and PSI_INSTANCE_NOM $\subset$ SYMBOLIC_PSI_FLAT $\subset$ C_PSI. Indeed, the file looks like the following:

```
structure PiInstanceNom       = struct
  ...
end;
structure PiSymbolicInstance = struct
  open PiInstanceNom
  ...
end;
structure PiCalculus : C_PSI = struct
  open PiSymbolicInstance
  ...
end;
Pi = Command(PiCalculus);
```

This splitting of structures for implementing one signature is because of SML lack of mutually recursive definitions for structures, as we need to refer to functions and datatypes when implementing the constraint solvers, printers and parsers. The first two structures are not restricted to the signature, otherwise it would close the datatype constructors.

The following text will be divided in three sections accordingly.

Instance definition

First we will define SML datatypes to represent nominal datatypes, then the SML functions implementing equivariant operators, next the nominal equivariant functions on SML datatypes, making those datatypes full fledged nominal datatypes, and substitution functions.

```
structure PiInstanceNom (* : PSI_INSTANCE_NOM *) =
struct
```

Before we begin writing down datatypes, let us look at the definition of nominal datatypes for Pi-calculus instance as defined in [4]. We diverge here from [4] by extending $\mathbf{C}$ with condition $\top$ to represent an always entailed condition, such that we can encode $P + Q$ as **case** $\top : P \,[]\, \top : Q$, rather than (νa)**case** $a = a : P \,[]\, a = a : Q$ where we would need to introduce a new name.

$$\begin{array}{lcl} \mathbf{T} & \stackrel{\text{def}}{=} & \mathcal{N} \\ \mathbf{C} & \stackrel{\text{def}}{=} & \{a = b : a, b \in \mathbf{T}\} \cup \{\top\} \quad \text{where } \top \notin \mathbf{T} \\ \mathbf{A} & \stackrel{\text{def}}{=} & \{1\} \end{array}$$

where $\mathcal{N}$ is countably infinite set of atomic *names* as usual.

A good candidate to represent names is the **string** type, since the framework provides the structure StringName with default functionality, and term is defined as name. $\mathbf{C}$ forms a carrier for a condition term algebra, hence it is only natural to represent it with algebraic datatypes[1] of SML, in condition below. Eq (a,b) corresponds to $a = b$ and T to $\top$, and assertion is an empty data constructor Unit.

```
  type name           = string
  type term           = name
  datatype condition = Eq of term * term | T
  datatype assertion = Unit
```

Next we turn to model the operators of Psi-calculus. Let us recall the definition of the operators. The difference with [4] is that we always entail $\top$.

$$\begin{array}{lcl} \dot{\leftrightarrow} & \stackrel{\text{def}}{=} & = \\ \otimes & \stackrel{\text{def}}{=} & \lambda\langle \Psi_1, \Psi_2 \rangle . 1 \\ \mathbf{1} & \stackrel{\text{def}}{=} & 1 \\ \vdash & \stackrel{\text{def}}{=} & \{\langle 1, a = a \rangle : a \in \mathcal{N}\} \cup \{\langle 1, \top \rangle\} \end{array}$$

As we can see, channel equivalence is defined as an equality condition. This is trivial in our SML representation, chaneq below. The composition and the unit of assertions are straightforward and the SML code is close to the above definition. The entailment, $\vdash$, relation is defined as a boolean function on the assertion and condition datatypes. The assertion datatype has only one constructor, therefore the function entails has two cases: Eq(m,n) computes a **string** equality on m and n, and T always returns *true*.

[1] It is worth noting that an instance implementor is free to choose any SML datatypes for implementing nominal datatypes as long it is possible to provide functions required by the NOMINAL signature for the corresponding types.

```
7   fun chaneq  (a,b)              = Eq (a,b)
8   fun compose (psi1,psi2)        = Unit
9   val unit                       = Unit
10  fun entails (Unit,Eq (m, n)) = (m = n)
11    | entails (Unit,T)           = true
```

The function swap_name is not necessary for the implementation of the instance and it is derived by the Command functor, but we need it for the definitions of the swapping function, and the substitution function. The swap_name function implements the usual swap function on names $(a\ b) \cdot n$. The StringName structure provides a default implementation for generating new names and swapping names for the type **string** which we use here.

```
12  fun swap_name (a,b) n = StringName.swap_name (a,b) n
```

The new function is used for generating fresh names, the requirement is that new provides a name which is not in the xvec list, i.e. $(\forall n \in \widetilde{x})n \neq new(\widetilde{x})$. Again StringName provides the default implementation.

```
13  fun new xvec = StringName.generateDistinct xvec
```

The Pi-calculus terms, conditions and assertions do not have binders, thus the implementation of the *support* is routine. The support is represented as a list of names. The function supportT computes the term support, and as we have names as our terms we only return the given name as a singleton list. The function supportC result can contain two names or none, and supportA returns an empty list.

```
14  fun supportT n             = [n]
15  fun supportC (Eq (a, b)) = [a, b]
16    | supportC  T            = []
17  fun supportA _             = []
```

As terms are just names, swapping a name in a term is the same as applying the swap_name function. Similarly for conditions while assertions are not affected by swapping.

```
18  fun swapT pi n = swap_name pi n
19  fun swapC _  T = T
20    | swapC pi (Eq (t1, t2)) =
21        Eq (swap_name pi t1, swap_name pi t2)
22  fun swapA _ _  = Unit
```

We also require an instance to provide an α-equivalence for the nominal datatypes. All the equality functions take as a first argument a function to be called for deciding α-equality for a datatype with binders. None of the nominal datatypes in Pi-calculus feature binders, and so α-equality is just syntactic equality, as can be seen below expressed with built-in SML equality and the first argument ignored. However, it is worth mentioning how to use these functions to implement an α-equality for nominal datatypes with binders which bind into the whole instance of a datatype.

Consider the lambda calculus language

$$M, N ::= \lambda x.M \mid MN \mid x$$

we define the nominal set of the terms of the lambda calculus as

$$\mathbf{T} = \{\lambda x.N : x \in \mathcal{N} \wedge N \in \mathbf{T}\} \cup \{MN : M \in \mathbf{T} \wedge N \in \mathbf{T}\} \cup \mathcal{N}$$

and the SML datatype for terms is

datatype lam = Lam **of** name * lam | App **of** lam * lam | Var **of** name

with nominal operations defined on it as expected. Thus a possible implementation of α-equality over lam terms may be:

```
fun eqT aEq (Lam (x,m), Lam (y,n)) =
        aEq ((x,m), (y,n))
  | eqT aEq (App (n,m), App (n',m')) =
        eqT aEq (n,n') andalso eqT aEq (m,m')
  | eqT aEq (Var x, Var y) =
        x = y
  | eqT _ _ = false
```

In the first case, two lambda abstractions are equated. We make use of the provided aEq function, which takes two tuples, where each tuple has as a first member a binder and as a second a term in which the binder binds into. The function aEq finds a fresh name for both terms and swaps both bound names with that name in those terms, and then calls back into eqT with the swapped name terms. This kind of mutual recursive behaviour of functions aEq and eqT relates to the Nominal Logic axiom **A1** from [25]

$$a.x = a'.x' \iff (a = a' \wedge x = x') \vee (a'\#x \wedge x' = (a\,a')x)$$

where $a.x$ is an abstraction with a being a binder binding into x: in our setting, the function eEq does the right hand side judgment, and the equating of terms x and x' are done by the function eqT.

The α-equality of application case is just a matter of recursively calling eqT with structurally smaller terms and by passing along the same aEq. Equating variables is just checking their name equality.

Since, as mentioned, Pi-calculus terms do not feature terms with binders, therefore we just ignore the provided function and use the builtin SML equality.

```
23 fun eqT _ (a,b) = a = b
24 fun eqC _ (a,b) = a = b
25 fun eqA _ (a,b) = a = b
```

Next we define a substitution function on the nominal datatypes. Each substitution function takes a substitution sequence sigma (a list of name and term pairs) and an instance of a nominal datatype.

The function substT finds a matching name in sigma, if found it returns the pairing term (name), otherwise it returns the name unchanged. substC distributes as substT, and assertions are not affected by the substitution.

```
26  fun substT sigma n =
27    case LIST.find (fn (x,_) => x = n) sigma of
28           NONE       => n
29         | SOME (_,t) => t
30  fun substC sigma T             = T
31    | substC sigma (Eq (t1, t2)) =
32        Eq (substT sigma t1, substT sigma t2)
33  fun substA sigma Unit          = Unit
```

This ends the first part of definitions.

```
34 end;
```

We need to close this structure as this structure's definitions will be used by functors in the structure PiSymbolicInstance.

Symbolic instance definition

We continue by defining extra requirements for the symbolic Psi-calculus and by implementing a constraint solver for a symbolic simulator and symbolic bisimulation checker.

```
35 structure PiSymbolicInstance (* : SYMBOLIC_PSI_FLAT *) =
36 struct
```

First we include the previously defined structure.

```
37   open PiInstanceNom
```

Additionally the symbolic Psi-calculus requires that names should be among the terms (Definition 15), i.e. $\mathcal{N} \subseteq \mathbf{T}$. This is modeled in the tool's framework by the function var : name →term; in our case it is the identity function.

```
38   fun var x = x
```

Now we turn our attention to solving the transition constraints. Before we can do anything useful we need to define the Constraint sub-structure which implements constraint datatypes and some useful functions, like substitution and nominal functions.

The functor Constraint defines the constraints needed for the simulator (symbolic operational semantics), of the form:

$$\begin{array}{rcl} C, C' & ::= & (\nu\widetilde{a})\{\!|\Psi \vdash \varphi|\!\} \\ & | & C \wedge C' \end{array}$$

The C constrain Constraint.constraint is a synonym for Constraint.atomic **list** and $(\nu\widetilde{a})\{\!|\Psi \vdash \varphi|\!\}$ is Constraint.atomic (avec, psi, phi). These datatypes are nominal datatypes, meaning they have swap, new, etc. defined on them in

the Constraint structure. The Constraint functor also provides a default implementation of capture avoiding substitution. As we can see, the functor uses the definitions from PiInstanceNom, which is one of the reasons we need to split the structure.

```
39 | structure Constraint = Constraint(PiInstanceNom)
```

The Pi-calculus instance conditions are syntactic and their interpretation is syntactical equality on names defined by the entailment relation. Furthermore we can view, the transition constraint as a syntactic equation system. In order to solve this system, we implement a syntactic unification algorithm known as Martelli-Montanari algorithm. We use the form of algorithm given in [19], which treats equations as a sequence of goals rather than a term rewriting system since it is easier to implement in our framework.

The outcome of a successful unification is a substitution sequence. This sequence is computed by composing smaller sequences at an appropriate algorithm step, as we will see later. We only need a special case where substitution sequence is extended with a new substitution clause.

$$[\widetilde{x} := \widetilde{L}][x' := L'] = \begin{cases} [\widetilde{x} := \widetilde{L}[x' := L']] & \text{if } x' \in \widetilde{x} \\ [\widetilde{x} := \widetilde{L}[x' := L']] \cup [x' := L'] & \text{if } x' \notin \widetilde{x} \end{cases}$$

In SML a substitution sequence is represented as a list of pairs of name and term. Hence getting the domain $\widetilde{x}$ of $[\widetilde{x} := \widetilde{L}]$ is just a matter of taking the first member of each pair.

```
40 | fun dom sigma = map (fn (n,t) => n) sigma
```

Below is the implementation of the substitution sequence composition given above, where sigma is $[\widetilde{x} := \widetilde{L}]$ and s is $[x' := L']$. So firstly it applies the substitution s to the range of sigma and then adds s to the sequence if x' is not in the domain of sigma.

```
41 | fun composeSubst sigma (s as (x',l')) =
42 | let
43 |   val app = map (fn (n,t) => (n, substT [s] t)) sigma
44 | in
45 |   if Lst.member x' (dom sigma)
46 |     then app
47 |     else s :: app
48 | end
```

Next we present the unification rules tailored to the Pi-calculus case. We omit the assertion from the constraint, since it is always unit, we write $(\nu\widetilde{a})\{\!|\varphi|\!\}$ for $(\nu\widetilde{a})\{\!|\mathbf{1} \vdash \varphi|\!\}$, and we write $\{\!|\varphi|\!\}$ for $(\nu\varepsilon)\{\!|\varphi|\!\}$.

These rules solve the unification problem for the Pi-calculus instance.

$$
\begin{array}{llll}
(\nu\widetilde{a})\{\!|a \leftrightarrows b|\!\} \wedge C & \overset{[b:=a]}{\rightarrowtail} & C[b := a] \text{ if } a, b\#\widetilde{a} \text{ and } a\#b & (\textsc{Elim}) \\
(\nu\widetilde{a})\{\!|a \leftrightarrows a|\!\} \wedge C & \rightarrowtail & C & (\textsc{Tr1}) \\
(\nu\widetilde{a})\{\!|\top|\!\} \wedge C & \rightarrowtail & C & (\textsc{Tr2}) \\
(\nu\widetilde{a})\{\!|a \leftrightarrows b|\!\} \wedge C & \rightarrowtail & \blacksquare \text{ if } a\#b \text{ and } a \in \widetilde{a} \vee b \in \widetilde{a} & (\textsc{Fail})
\end{array}
$$

The first rule, (Elim), produces a singleton substitution sequence and applies that to the residual whenever both names are free and are not the same. Notice that in the substitution the order of the names are swapped. This is done because when an agent does an input or an output the fresh name generated for the subject name is placed on the right hand side of the equality. In this manner we will always be substituting away the generated name, in order to give a more "informative" substitution sequence. (Fail) rule says that there is no name which can be equated to a restricted name, and stops with a failure. (Tr1) and (Tr2) handle trivial cases. The algorithm succeeds with $\square$, when there are no more constraints and the labels of the rewrite steps gives a sequence of substitution sequences.

As an example of the application of the above rules, let us consider this example:

$$\textbf{case } a \leftrightarrows c : \textbf{case } c \leftrightarrows b : \textbf{case } a \leftrightarrows b : d(x)$$

This agent would give rise to the following constraint.

$$\{\!|d \leftrightarrow f|\!\} \wedge \{\!|a \leftrightarrow b|\!\} \wedge \{\!|c \leftrightarrow b|\!\} \wedge \{\!|a \leftrightarrow c|\!\}$$

where f is a freshly chosen name.

One possible sequence of unification rule application is the following

$$
\begin{array}{rl}
\{\!|d \leftrightarrow f|\!\} \wedge \{\!|a \leftrightarrow b|\!\} \wedge \{\!|c \leftrightarrow b|\!\} \wedge \{\!|a \leftrightarrow c|\!\} & \overset{[f:=d]}{\rightarrowtail} \\
\{\!|a \leftrightarrow b|\!\} \wedge \{\!|c \leftrightarrow b|\!\} \wedge \{\!|a \leftrightarrows c|\!\} & \overset{[b:=a]}{\rightarrowtail} \\
\{\!|c \leftrightarrows a|\!\} \wedge \{\!|a \leftrightarrows c|\!\} & \overset{[a:=c]}{\rightarrowtail} \{\!|c \leftrightarrows c|\!\} \rightarrowtail \square
\end{array}
$$

We just showed that the channel equivalence, $=$, is transitive. By composing all the substitutions on the arrows in sequence we get a solution substitution $[f := d][b := a][a := c] = [f := d, b := c, a := c]$. It is clear that this solution is not unique, for instance the substitution $[f := d, b := a, c := a]$ also satisfies the above constraint. In fact, the unification algorithm gives a substitution unique up to a permutation of names, i.e. a bijective renaming (cf. [19]).

In contrast to the above example, below is an agent which gives rise to a transition constraint with no solutions.

$$(\nu a)a(x)$$

The constraint to be solved is $(\nu a)\{\!|a \dot{\leftrightarrow} b|\!\}$ (where b is chosen fresh $b\#a$); this satisfies the (FAIL) rule.

Before we continue implementing the unification, we define a convenience sub-structure which provides nominal functions for a list of names, such that we can compute $a\#\widetilde{x}$ with L.fresh a xvec.

```
49  structure L = NominalNameList(
50      struct type atom = name val new = new end)
```

The function mgu is an implementation of the above unification rules. We represent success, $\square$, by returning Either.RIGHT sigma with the substitution sequence, and failure, $\blacksquare$, by returning Either.LEFT phi with the offending condition. The function mgu takes a constraint as the first argument, and the accumulated substitution sequence as the second. The function mgu computes a new substitution sequence when the (ELIM) rule is applicable.

The lines that correspond to each of the above unification rules are annotated to the right.

```
51  fun mgu [] sigma = Either.RIGHT sigma                          □
52    | mgu ((avec, Unit, T) :: cs) sigma =                        (TR2)
53      mgu cs sigma
54    | mgu ((avec, Unit, (Eq (a,b))) :: cs) sigma =
55      if a = b then mgu cs sigma                                 (TR1)
56      else
57        if L.fresh a avec andalso L.fresh b avec                 (ELIM)
58          then mgu (Constraint.subst cs [(b,a)])
59                   (composeSubst sigma (b,a))
60          else Either.LEFT [Eq (a,b)]                            ■
```

The last thing we need to provide to complete the implementation of a constraint solver for transition constraints is the solve function. The solve function accepts a constraint and produces either a list of solutions or a list of failed conditions. In our case, the function solve is just a wrapper for the mgu function. A solution to the constraint is only the substitution sequence, as Pi-calculus has trivial assertions. And the mgu function gives a Most General Unifier (up to a name permutation, or bijective renaming substitution [19]). That is, it is sufficient to return only the substitution sequence produced by mgu as all other solutions are entailed by the mgu (again, up to name permutation).

```
61  fun solve cs =
62    case mgu cs [] of
63         Either.RIGHT sigma ⇒ Either.RIGHT [(sigma, Unit)]
64       | Either.LEFT phi    ⇒ Either.LEFT  [phi]
```

⋮

This Pi-calculus instance implementation also hosts a constraint solver for the constraints generated by the bisimulation algorithm (Appendix A). The

constraints are more complicated to solve[2] and require a more complicated algorithm for finding solutions. However, the approach is in the same vein as presented here: a bisimulation constraint solver is a function that takes a constraint and produces either a solution or a counter example. Such a constraint solver is not required for the simulation of agents, therefore we omit it here and direct an interested reader to Appendix A for a full implementation.

⋮

```
65 end;
```

Implementation of printers and parsers

The last piece in the construction of a working simulator with command interpreter is ability to pretty print and parse nominal datatypes.

At this stage we can restrict the PiCalculus structure to the C_PSI signature.

```
66 structure PiCalculus : C_PSI =
67 struct
```

As before we input the previously defined structure.

```
68   open PiSymbolicInstance
```

First we need to settle on the `ASCII` syntax of the members of the nominal datatypes. For names we choose an alphanumeric representation, conditions for equality will be formed with `=`, the top value is written `T` and the unit assertion is `1`.

In summary

Notation	`ASCII`
a	`a`
$a \leftrightarrows b$	`a = b`
$\top$	`T`
$\mathbf{1}$	`1`

First we turn to printing. Every nominal datatype and names have corresponding print functions which take a member from nominal datatype and return a string representation of it.

Names are represented by strings so printing functions are almost trivial, as the printC inserts a `=` between the channel equality names.

[2] The bisimulation constraints expresses at least a first order logic with equality and freshness constraints and without quantifiers.

```
69  fun printN a = a
70  fun printT a = a
71  fun printC (Eq (a, b)) = a ^ " = " ^ b
72    | printC  T = "T"
73  fun printA psi = "1"
```

Parsing is more intricate. We parse nominal datatypes with the combinator library provided with the tool, although it would not be very difficult to write a custom parser for the terms of the Pi-calculus instance.

First we need to construct a parser structure which provides the combinators on string streams.

```
74  structure Parser = Parser(StringStream)
```

The functor PsiParserBase provides the Psi-calculus language lexical parser combinators: the notion of whitespace, identifier, etc.

```
75  structure Lex = PsiParserBase(Parser)
```

None of the structures defined above are opened here in order to make the origin of functions clear.

Parsing combinators deserve some treatment here. A parsing combinator is a value or a function with a result of the type 'a parser. The type 'a denotes a parse result. The basic combinators are provided by the Parser structure, we use the return combinator, which does not consume any input and always succeeds by returning the value passed to it as a parse result. A parser combinator may fail resulting in a parse error and/or backtracking if it is a part of another parser combinator. This means that parser combinators implement top down recursive descent parser[3]. In contrast, the zero value is a combinator which does not consume any input and always fails.

We also use lexical parser combinators provided by the structure Lex. Lexical combinators are in an accord with the lexical rules of Psi-calculi tool (Section 3.1.1). The combinator identifier parses a series of alphanumeric characters but removes any whitespace (including comments), and it first returns the concatenated characters as a string. The other combinator of interest is the function stok (mnemonic for 'string token') which also first removes any whitespace and then matches the remaining input with its string argument; if there is no match it fails.

More complex parsers are built from basic combinators by using sequencing functions p >>= q with type 'a parser * ('a → 'b parser) → 'b parser, >> with type 'a parser * 'b parser → 'b parser. p >>= q sequences two parsers, by first applying the parser p, and then giving parse result of p to the function q which returns a new parser, which in turn can be used for further sequencing or application. p >> q is the same as >>=, but ignores the result of p, and applies q directly. Another crucial building block of parsers is a

[3]The problem of not handling left recursive grammars is inherited.

choice[4] between combinators, which comes in two flavours: deterministic and non-deterministic. The one we use is the deterministic choice combinator. The choice parser combinator composes two parser combinators by trying to apply the first parser combinator to a stream; if the combinator succeeds, choice returns that parser's result and otherwise it tries the second parser combinator.

These parser combinators are modelled after [16], which contains a reference and examples.

Let us first define some aliases.

```
76 | fun p >>= q = Parser.>>= (p,q)
77 | fun p >>  q = Parser.>>  (p,q)
```

We are now at the point where we can define the syntax of the datatypes. We will first define parsers for the names, conditions and assertion. And afterwards we provide the requisites for the signature. As mentioned before, the syntax of a name will coincide with the identifier syntax of Psi Workbench. The value Lex. identifier is a parser combinator which define Psi Workbench's identifier syntax. The value Lex. identifier is of the type **string** parser.

```
78 | val name = Lex.identifier
```

Now we define the syntax for conditions, which can be expressed in the following grammar production rules.

⟨*cond-eq*⟩ ::= ⟨*name*⟩ '=' ⟨*name*⟩

⟨*cond-t*⟩ ::= 'T'

⟨*cond*⟩ ::= ⟨*cond-eq*⟩ | ⟨*cond-t*⟩

The condEq parser combinator defines the syntax of the ⟨*cond-eq*⟩. The parser combinator condEq parses a name binds it to a, then expects some whitespace and matches "=", and then again a name by binding it to b and if none of the parsers failed builds a channel equality datatype. Hence, the combinator's condEq type is condition parser.

```
79 | val condEq =  name >>=
80 |   (fn a =>     Lex.stok "=" >>
81 |               name >>=
82 |   (fn b =>     Parser.return (Eq (a,b))
83 |   ))
```

The sequencing functions form a production rule, and >>= provides a way to refer to the results (in similar fashion as in attribute grammars). The

[4] It is worh noting that >>= is associative, the combinator zero is a left and right unit for choice combinator. In fact, the type 'a parser together with the discussed functions is a monad, cf. [16].

Parser.return function lifts an ordinary SML value into a parser combinator; this kind of parser combinator does not consume any input and always succeeds by returning the value provided. The combinator Lex.stok takes a string as an argument and treats it as a token by matching it to the input before discarding any whitespace including comments; if the input matches then it succeeds, otherwise it fails.

The parser to for the ⟨*cond-t*⟩ is now straightforward.

```
84 | val condT = Lex.stok "T" ≫ Parser.return T
```

By using deterministic choice to combine the condT and condEq we complete the ⟨*cond*⟩ implementation.

```
85 | val cond = Parser.choice (condT, condEq)
```

Assertion is similar to the condT.

```
86 | val assr = Lex.stok "1" ≫ Parser.return Unit
```

Parsing functions of nominal datatypes are required to take a string and return either a corresponding nominal datatype or an error string. But Parser.parse takes a parser combinator and a stream, so to bridge this we define an auxiliary function. If a parser succeeds it returns a list of results, and since we only used deterministic choice to construct parsers there will be one result returned. The second case clause is just for inhibiting a compiler warning of non exhaustive pattern matching.

```
87 | fun parseResult p s =
88 |   case Parser.parse p (StringStream.make s) of
89 |        Either.RIGHT [(r,s)] ⇒ Either.RIGHT r
90 |      | Either.RIGHT _ ⇒ Err.undefined ()
91 |      | Either.LEFT  _ ⇒ Either.LEFT "Error␣parsing"
```

We can now fill in all the required parsing functions.

```
92 | fun parseN s = parseResult name s
93 | fun parseT s = parseN s
94 | fun parseC s = parseResult cond s
95 | fun parseA s = parseResult assr s
```

An important design decision when implementing printers and parsers is to make them circular, i.e., parse a = parse (print (parse a)) and print a = print (parse (print a)). This is not strictly enforced, but it is a good convention, as the output of the command interpreter then can be reused as an input. Let's take conditions as an example, say we have a string "a=b" which would be parsed into Eq("a","b") by parseC. The function printC would produce a string "a␣=␣b" (note the spaces). The parser parseC would again, given the string "a␣=␣b", return Eq("a","b"), which is the same as previously. The parser function parseC is implemented in terms of stok and identifier which contract any whitespace and only parses the 'essence'.

```
96 | end;
```

The final matter is to construct a structure with the Command functor. The structure provides a start : unit → unit function to start the command interpreter.

```
97 structure Pi = Command(PiCalculus);
```

In order to load this file, the file `workbench.ML` must be loaded first which provides all the definitions required above (see Section 3.1.3).

This ends the Pi-calculus instance implementation using Psi-calculus workbench framework. Most of the mathematical machinery involved in defining a Psi-calculus instance is quite straightforward to carry over to SML code. This is not very surprising as we use term algebras, and SML's algebraic datatypes are designed to easily represent term algebras. The same is true for most of the associated operations. The most involved part is the design and the implementation of a constraint solver for both transition and bisimulation constraints.

3.2.2 Frequency hopping spread spectrum

In this section, we go through an implementation of a Psi-calculus instance modelling Frequency hopping spread spectrum (FHSS). This instance showcases terms with more complex structure than a name, although they are still syntactic. Additionally, this instance features additional rules to the transition constraint solver of the Pi-calculus instance to handle these terms.

This time we diverge from following the literate-programming style as closely as in the Pi-calculus instance example (Section 3.2.1) by omitting non essential code but preserving the order of code as it would appear in a file, thus we abstain from numbering code lines. We concentrate on the differences to the Pi-calculus instance arising from the introduction of non-trivial terms; we leave out straightforward implementation details and functionality which is transferable without much effort from the Pi-calculus instance; we also omit the SML structure declarations as it should be clear now of which structures various SML datatypes and functions are part of.

Here we reproduce from [4, section 4.1.1] a short description of what FHSS is and what FHSS instance is modelling.

> Wireless communication over a constant radio frequency has a number of drawbacks. In a hostile environment a radio can be tuned in to the correct frequency and monitor the communication which is also vulnerable to jamming. A solution to these problems is to jump quickly between different frequencies in a scheme called frequency hopping spread spectrum (FHSS), first patented in 1942. To eavesdrop it would then be necessary to

match both the order of the frequencies and the pace of switching. Jamming is also made more difficult since the available power would have to be distributed over many frequencies.

Let us start with a simple example, the following is a valid agent of the FHSS instance:

$$\overline{\mathsf{nextFreq}(b)}\,\langle \mathsf{nextFreq}(b)\rangle.P \;\mid\; \underline{\mathsf{nextFreq}(\mathsf{nextFreq}(a))}\,(x).Q$$

which tries to communicate through channels $\mathsf{nextFreq}(\mathsf{nextFreq}(a))$ and $\mathsf{nextFreq}(b)$, so in order for the communication to succeed b must be equal to $\mathsf{nextFreq}(a)$. After a communication the agent continues as

$$P \;\mid\; Q[x := \mathsf{nextFreq}(b)]$$

or with a solution applied,

$$(P \;\mid\; Q[x := \mathsf{nextFreq}(b)])[b := \mathsf{nextFreq}(a)]$$

Let us define the FHSS instance formally (this definition is based on [4, Section 4.1.1]), we additionally introduce a $\top$ condition as in the Pi-calculus instance.

$$\begin{array}{rcl}
\mathbf{T} & \stackrel{\text{def}}{=} & \mathcal{N} \cup \{\mathsf{nextFreq}(M) : M \in \mathbf{T}\} \\
\mathbf{C} & \stackrel{\text{def}}{=} & \{M = N : M, N \in \mathbf{T}\} \cup \{\top\} \\
\mathbf{A} & \stackrel{\text{def}}{=} & \{1\} \\
\mathbf{1} & \stackrel{\text{def}}{=} & 1 \\
\dot{\leftrightarrow} & \stackrel{\text{def}}{=} & = \\
\otimes & \stackrel{\text{def}}{=} & \lambda\langle \Psi_1, \Psi_2\rangle.1 \\
\vdash & \stackrel{\text{def}}{=} & \{\langle 1, M = M\rangle : M \in \mathbf{T}\} \cup \{\langle 1, \top\rangle\}
\end{array}$$

where $\mathcal{N}$ is countably infinite set of atomic names as usual and $\top \notin \mathbf{T}$.

The difference with Pi-calculus is the $\mathbf{T}$ nominal datatype. Other nominal datatypes and operations are unchanged.

Unsurprisingly, the definition of the terms nominal datatype $\mathbf{T}$ forms a carrier set for a term algebra. In a term algebra variables are part of the carrier set (cf. Appendix B.3).

The signature is

$$\Sigma = \{\mathsf{nextFreq}\}$$

where the function symbol $\mathsf{nextFreq}$ arity is 1. Since we can use names as variables, we invoke the T constructor (Definition 32 in Appendix B) on the above signature and names to get a set. Given that $\mathbf{T}$ is a least fixed-point of the above definition, and $T(\Sigma, \mathcal{N})$ constructs a least set, inductively, they coincide

$$T(\Sigma, \mathcal{N}) = \mathbf{T}$$

and thus **T** is a carrier set of the Σ-term algebra $\mathcal{T}(\Sigma, \mathcal{N})$.

A natural choice to implement this datatype is SML's algebraic datatypes. We have two cases: one for names, and one the function symbol.

```
datatype term = Name     of name
              | NextFreq of term
```

Because no binders are present in **T**, the computation of the equality on terms in the entailment is trivial, and we use the built-in SML equality to compute the syntactic equality.

```
fun entails (Unit,Eq (m, n)) = (m = n)
  | entails (Unit,True)      = true
```

Obviously, any term of **T** always contains exactly one name. A channel equivalence condition contains exactly two names, so we just do a structural recursion to extract those names.

```
fun supportT (Name n)      = [n]
  | supportT (NextFreq m) = supportT m
fun supportC (Eq (m, n))  = supportT m @ supportT n
  | supportC True          = []
```

The swapping of names in terms are easily implemented with the function swap_name. We only need to propagate the swap_name function down to the contained name.

```
fun swapT pi (Name n)       = Name (swap_name pi n)
  | swapT pi (NextFreq t)  = NextFreq (swapT pi t)
fun swapC _  True           = True
  | swapC pi (Eq (t1, t2)) = Eq (swapT pi t1, swapT pi t2)
```

Again, as we do not have binders in terms we do not need to concern our selves with a possibility of name capture whenever we do a substitution. The function substT implements a substitution function for terms. Given a substitution sequence sigma it first recursively gets to the name of a term, and then tries to find a substitution mapping with that name. If found returns the term from the mapping, otherwise it returns the name unchanged.

```
fun substT sigma (Name a)      =
      (case List.find (fn (b,_) => a = b) sigma of
            NONE       => Name a
          | SOME (_,t) => t)
  | substT sigma (NextFreq n) = NextFreq (substT sigma n)
```

As mentioned previously, alpha equivalence is the same as syntactic equivalence for terms.

```
fun eqT _ (a,b) = a = b
```

For finding a substitution function for a syntactic equation system (this is what a transition constraint for FHSS is) we again employ the Martelli-Montanari (MM) unification algorithm. This time, as we are dealing with more complex terms, we reintroduce and adapt the rules used to solve the

Pi-calculus instance transition constraints (see section 3.2.1) to the FHSS instance case and add two new rules[5] for dealing with non-trivial terms. We refer to [19] for more details on MM, and to [2] for the alternative presentation of the algorithm.

Here we present the rules for solving transition constraint for FHSS, a brief description follows afterwards.

$$(\nu\widetilde{a})\{\!|\mathrm{nextFreq}(N) \leftrightarrows \mathrm{nextFreq}(M)|\!\} \wedge C \rightarrowtail (\nu\widetilde{a})\{\!|N \leftrightarrows M|\!\} \wedge C \qquad \text{(DECOM)}$$

$$(\nu\widetilde{a})\{\!|\mathrm{nextFreq}(N) \leftrightarrows a|\!\} \wedge C \rightarrowtail (\nu\widetilde{a})\{\!|a \leftrightarrows \mathrm{nextFreq}(N)|\!\} \wedge C \qquad \text{(SWAP)}$$

$$(\nu\widetilde{a})\{\!|\top|\!\} \wedge C \rightarrowtail C \qquad \text{(TRT)}$$

$$(\nu\widetilde{a})\{\!|a \leftrightarrows a|\!\} \wedge C \rightarrowtail C \qquad \text{(TREQ)}$$

$$(\nu\widetilde{a})\{\!|a \leftrightarrows N|\!\} \wedge C \overset{[a:=N]}{\rightarrowtail} C[a := N] \quad \text{if } a, N\#\widetilde{a} \ \wedge \ a\#N \qquad \text{(ELIM)}$$

$$(\nu\widetilde{a})\{\!|a \leftrightarrows N|\!\} \wedge C \rightarrowtail \blacksquare \quad \text{if } a \neq N \ \wedge \ (a \in n(N) \vee a \in \widetilde{a} \vee n(N) \subseteq \widetilde{a}) \qquad \text{(FAIL)}$$

The (DECOM) rule, as the name suggests, decomposes terms into structurally smaller terms and returns the decomposition for further unification, since only the difference between terms is at play, e.g. the constraint $\mathsf{nextFreq}(\mathsf{nextFreq}(a)) \leftrightarrows \mathsf{nextFreq}(b)$ entails the solutions $[b := \mathsf{nextFreq}(a)]$. The (SWAP) rule exchanges the positions of terms in channel equivalence if the right hand side term is a name. With this rules we do not need to introduce symmetric versions of other rules. The (ELIM) rule produces a substitution mapping (a partial solution). Such a substitution mapping is valid only when neither side of the equation has restricted names, and the name on the left side is not in the term of the right hand side (as this would give rise to an infinite term), e.g. constraints $(\nu\widetilde{a})\{\!|a \leftrightarrows \mathsf{nextFreq}(a)|\!\}$ are disallowed. The second check is also known by the name "occurs check" [19]; more will be said about it later. The second check has a second purpose, it also says that this is not a trivial case (handled by (TREQ)). And the (FAIL) rule is triggered if it is not a trivial case and not a (ELIM) case. As before a solution – a substitution sequence – is produced when there are no more rules to apply. See 3.2.1 for a simpler case.

The following example should make it clearer how the above rules interact. Let us return to the agent we gave at the beginning of this section, but

[5] In fact, these two new rules are required by the MM algorithm, but are not needed for Pi-calculus instance case.

let us swap the constituent agents over the parallel.

$$\underline{\mathsf{nextFreq}(\mathsf{nextFreq}(a))}\,(x).Q \;\mid\; \overline{\mathsf{nextFreq}(b)}\,\langle \mathsf{nextFreq}(b)\rangle.P$$

This agent gives rise to the transition constraint:

$$(\nu\varepsilon)\{\!|\mathsf{nextFreq}(\mathsf{nextFreq}(a)) \leftrightarrow \mathsf{nextFreq}(b)|\!\}$$

The course of unification rule application might look like:

$$\begin{aligned}
(\nu\varepsilon)\{\!|\mathsf{nextFreq}(\mathsf{nextFreq}(a)) \leftrightarrow \mathsf{nextFreq}(b)|\!\} &\;\underset{(\textsc{Decom})}{\rightarrowtail}\\
(\nu\varepsilon)\{\!|\mathsf{nextFreq}(a) \leftrightarrow b|\!\} &\;\underset{(\textsc{Swap})}{\rightarrowtail}\\
(\nu\varepsilon)\{\!|b \leftrightarrow \mathsf{nextFreq}(a)|\!\} &\;\underset{[b:=\mathsf{nextFreq}(a)]}{\underset{(\textsc{Elim})}{\rightarrowtail}}\\
\mathbf{true}[b := \mathsf{nextFreq}(a)] &= \mathbf{true}
\end{aligned}$$

Since there is only one (Elim) rule above, the computed substitution sequence is $[b := \mathsf{nextFreq}(a)]$, which is identical to the substitution sequence we deduced at the beginning of the section.

In contrast, suppose we have a constraint:

$$(\nu\varepsilon)\{\!|a \leftrightarrow \mathsf{nextFreq}(a)|\!\}$$

The (Fail) rule would be triggered as a is in the support of $\mathsf{nextFreq}(a)$, i.e. $a \in \{a\}$. The motivation of this rule is that this kind of constraint gives rise to the infinite term:

$$\mathsf{nextFreq}(\mathsf{nextFreq}(\cdots(\mathsf{nextFreq}(\cdots))\cdots))$$

Obviously, this kind of term cannot be computed and it is not part of the least fixed-point definition of $\mathbf{T}$.

The rules above produces a substitution sequences which composed gives a substitution which is a Most General Unifier (cf. Section 3.2.1 and [19]). The following function implements the above rules. It takes a transition constraint and an accumulated substitution sequence and returns either a complete substitution sequence for the constraint or a condition which triggered a failure rule. We also assume that the function composeSubst composing substitution sequences is defined (Section 3.2.1), and that the function freshL, computing $\widetilde{a}\#\widetilde{b}$, is also defined.

```
fun mgu [] sigma = Either.RIGHT sigma
    (* (TrT) case *)
  | mgu ((avec, Unit, True)::cs) sigma =
    mgu cs sigma
    (* (Decom) case *)
  | mgu ((avec, Unit, (Eq (NextFreq a, NextFreq b)))::cs)
        sigma =
```

```
      mgu ((avec, Unit, (Eq (a,b))) :: cs) sigma
    (* (SWAP) case *)
  | mgu ((avec, Unit, (Eq (NextFreq a, Name b))) :: cs)
        sigma =
      mgu ((avec, Unit, (Eq (Name b, NextFreq a))) :: cs)
          sigma
    (* If the name is on the left hand side of the channel equivalence *)
  | mgu ((avec, Unit, (Eq (Name a, n))) :: cs) sigma =
    if Name a = n then mgu cs sigma (* (TRN) case *)
      else
        (* if a#ã ∧ ã#N ∧ a#N *)
        if L.fresh a avec andalso
           freshL avec (supportT n) andalso
           L.fresh a (supportT n)
         then
            (* compose the produced substitution
             * sequence with the accumulated and
             * apply it to the residual *)
           mgu (Constraint.subst cs [(a, n)])
               (composeSubst sigma (a, n))
         else (* otherwise (FAIL) *)
           Either.LEFT [(Eq (Name a, n))]
```

Just like in the Pi-calculus example we implement the function solve by using the function mgu.

```
fun solve cs =
  case mgu cs [] of
       Either.RIGHT sigma ⇒ Either.RIGHT [(sigma, Unit)]
     | Either.LEFT phi    ⇒ Either.LEFT  [phi]
```

At this point, we are only left with implementing the printing and parsing functions, and most of them are directly transferable from Pi-calculus instance example. We only describe the printing and parsing of terms; other details are exactly as in the Pi-calculus instance.

Recall that we are using strings to represent names. The matter of printing a term then becomes only to prepend the required amount of "nextFreq" strings.

```
fun printT (Name n) = n
  | printT (NextFreq t) = "nextFreq(" ^ printT t ^ ")"
```

We use the same structures for parsing as before in the Pi-calculus instance (Section 3.2.1) but this time we make their definitions visible in the current namespace. In addition, we open the Missing structure which provides some auxiliary functions. In particular, we use the construct </.../>, which makes any function, accepting a tuple as an argument, into a left associative infix operator, e.g. the function **fun** add (x,y) = x + y could be used infix in 2 </add/> 5 to add two numbers and produce the result 5.

```
structure Parser = Parser(StringStream)
structure Lex    = PsiParserBase(Parser)
```

The grammar production rule for the terms are bit more interesting this time, since it is recursive.

⟨*term*⟩ ::= '`nextFreq`' '`(`' ⟨*term*⟩ '`)`' | ⟨*identifier*⟩

This rule is straightforward to express with an parser combinators. But before we do that, we need to address a technicality arising from the usage of recursive parser combinators. SML is a functional programming language with eager evaluation strategy, i.e. function arguments are evaluated before they are applied to a function. When using recursion in parser combinators we may get an undesired effect that the parser combinator is applied too "early" i.e. too eagerly. In most cases we want the parser combinator to be applied when the previous combinator in the sequence is completed, and not when it becomes an argument to a sequencing operator. For tackling this we use the delayed function which takes a parser combinator function with type unit → 'a parser and calls it only when the previous parser combinator completes. An alternative is to use the sequencing operator ≫= even when the result of a previous combinator is not needed and there is a need for a recursive application.

Let us return to the function implementing the above grammar production rule. The two alternatives of a grammar production are modelled with the combinator choice and using the previously described construct to make it into an infix operator. The first alternative is to match the string "nextFreq" and the opening parenthesis as tokens, then try to parse an inner term and finally to match the matching closing parenthesis and returning the resulting term. The second alternative is to parse an identifier and return it as a name term. The function is quite similar to the production rule, alas a bit more verbose since it is intertwined with actions.

```
fun term () =
      (stok "nextFreq" ≫ stok "(" ≫
      (delayed term) ≫=
      (fn t ⇒ stok ")" ≫ return (NextFreq t)))
  </choice/>
      (Lex.identifier ≫= return o Name)
```

This ends the FHSS instance example. Adding structured terms does not complicate the instance significantly since the same known techniques apply, e.g. the standard mgu computation.

3.2.3 Common ether

In this section, we explore non-trivial assertions. The Psi-calculus instance that we implement is called common ether [4, Section 2.6]. Although the assertions that we define are quite simple they induce a surprisingly big state

space for even the transition constraint solving. We implement a constraint solver for transition constraints and describe the details as in other examples.

We take the same approach when presenting as in the FHSS instance example (Section 3.2.2). We describe only the differences with the Pi-calculus instance example (Section 3.2.1). Not the whole code is listed (although it is listed in the same order as it appears in a file), and we skip the SML structure definitions.

As usual, we begin with an example of an agent of this instance. In the following example, we introduce step-by-step the features of this instance. Let us take the following agent and let us assume $a\#b$:

$$\underline{a}\,(x).P \;\mid\; \overline{b}\,y.Q$$

because of the above assumption, communication would not be possible in the Pi-calculus instance. But using assertions, we can enable communication. We model a single global communication channel, without requiring a global knowledge of available receiving or transmitting channel names. In the environment we record the available names for accessing the global communication channel.

$$(\!|\{a\}|\!) \mid \underline{a}\,(x).P) \quad\mid\quad (\overline{b}\,y.Q \mid (\!|\{b\}|\!))$$

may silently transition into

$$(\!|\{a\}|\!) \mid P)[x := y] \quad\mid\quad (Q \mid (\!|\{b\}|\!))$$

How can we make these seemingly independent environments coalesce? This is a feature of the operational semantics: recall that the rules COM and PAR compose juxtaposed assertions of the frame of an agent with the current assertion (Figure 1). By taking assertion composition as set union we derive the following

$$\textsc{Com}\;\frac{\textsc{Par}\;\dfrac{\{a,b\} \rhd \underline{a}\,(x).P \xrightarrow[\cdots]{\cdots} \ldots}{\{b\} \rhd (\!|\{a\}|\!) \mid \underline{a}\,(x).P \xrightarrow[\cdots]{\cdots} \ldots} \qquad \textsc{Par}\;\dfrac{\{a,b\} \rhd \overline{b}\,y.Q \xrightarrow[\cdots]{\cdots} \ldots}{\{a\} \rhd \overline{b}\,y.Q \mid (\!|\{b\}|\!) \xrightarrow[\cdots]{\cdots} \ldots}}{\mathbf{1} \rhd (\!|\{a\}|\!) \mid \underline{a}\,(x).P) \quad\mid\quad (\overline{b}\,y.Q \mid (\!|\{b\}|\!)) \xrightarrow[C]{\tau} \ldots}$$

Note that at the top of the derivation tree the whole environment is available and there it is trivial to make a decision.

One peculiarity arises, that channel equivalence is not reflexive. But this is allowed by Psi-calculus since channel equivalence must be transitive and symmetric but not necessarily reflexive (Definition 15 in Section 2.2). Consider the following agent with no environment.

$$\underline{a}\,(x) \mid \overline{a}\,y$$

the above agent does not have a τ transition, while the following has

$$\underline{a}\,(x) \mid \overline{a}\,y \mid (\!|\{a\}|\!)$$

so by adding the environment $(\!|\{a\}|\!)$ the above agent τ transitions into

$$\mathbf{0} \mid \mathbf{0} \mid (\!|\{a\}|\!)$$

A more interesting example agent

$$((\nu a)(\!|\{a\}|\!) \mid \underline{a}\,(x).P) \quad | \quad ((\nu b)(\!|\{b\}|\!) \mid \overline{b}\,y.Q)$$

silently transitions into

$$((\nu a)(\!|\{a\}|\!) \mid P)[x := y] \quad | \quad ((\nu b)(\!|\{b\}|\!) \mid Q)$$

Let us formalise the above discussion into the Psi-calculus instance definition.

$$\begin{array}{rcl}
\mathbf{T} & \stackrel{\text{def}}{=} & \mathcal{N} \\
\mathbf{C} & \stackrel{\text{def}}{=} & \{a = b : a, b \in \mathbf{T}\} \\
\mathbf{A} & \stackrel{\text{def}}{=} & \mathcal{P}_{\text{fin}}(\mathcal{N}) \\
\leftrightarrow & \stackrel{\text{def}}{=} & = \\
\otimes & \stackrel{\text{def}}{=} & \cup \\
\mathbf{1} & \stackrel{\text{def}}{=} & \emptyset \\
\vdash & \stackrel{\text{def}}{=} & \{\langle \Psi, a \leftrightarrow b\rangle : a, b \in \Psi \ \wedge \ \Psi \in \mathbf{A}\}
\end{array}$$

While the agent's observed behaviour is quite complex, the instance definition is compact and succinct. To represent the communication channel, a set of names is sufficient. That is the nominal datatype for assertions is all possible subsets of names. Composition of environments is set union.

Probably the simplest way to represent a set in SML is with the list data structure. The assertion is a set of names so in SML we choose to represent it as a name list. We allow duplicates in a list, this does not introduce problems but we need to be careful and have this in mind.

```
type assertion = name list
```

Implementing set operations is straightforward, and the unit constant and entail operations are almost a direct translation of the above definition. We use the auxiliary function Lst.member to test membership in a list.

```
val unit = []
fun entails (psi,Eq (a, b)) =
  Lst.member a psi andalso Lst.member b psi
```

The support of an assertion is exactly the assertion as it is only a list of names.

```
fun supportA psi = psi
```

The functor NominalNameList derives a nominal datatype structure of an atom list. An atom list is the data of the nominal datatype and the structure L contains all the nominal operations defined on it. An assertion is such a nominal datatype.

```
structure L = NominalNameList(struct
  type atom = name
  val  new = StringName.generateDistinct end)
```

The above structure L provides the needed implementation of the swapping function for an assertion.

```
fun swapA pi psi = L.swap pi psi
```

The function substA implements a substitution function by distributing a function over a list which for every element tries to find a substitution mapping and replaces the name if found and leaves it unchanged if not. Compare this with the function substT in the Pi-calculus instance example: the anonymous function is the same as the function substT. In fact, the function substA is a unique extension of substT to an endomorphism (see [2] or Appendix B.3).

```
fun substA sigma psi =
  map (fn n =>
      case List.find (fn (a,b) => a = n) sigma of
           SOME (_,x) => x
         | NONE => n) psi
```

Because we use lists to represent sets and we allow duplicates, we cannot use built-in SML equality to decide assertion equivalence. The order and the frequency of elements in a list has no significance. We check if every element in an assertion appears at least once in the other assertion, and vice versa. If it is the case then both assertions are deemed to be equivalent.

```
fun eqA _ (psi,psi') =
  Lst.all (fn a => Lst.member a psi') psi andalso
  Lst.all (fn a => Lst.member a psi ) psi'
```

Now we turn to the description and implementation of a transition constraint solver. We will express the constraint solver as a transition system. This transition systems is non-deterministic and forms a tree of solutions. The function implementing the transition system follows the branches of a solution tree and returns the leafs as a solution to the constraint.

In Figure 4 the transition system $\rightarrowtail$ for solving a transition constraint is given. The state (configuration) of the system is of the form

$$\langle(\sigma, \Psi'), C\rangle$$

where the tuple (σ, Ψ') represents a partial solution to some constraint and the constraint C is a constraint which still needs to be satisfied. A transition

$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma[a':=b'],\Psi'),C\rangle$$
$$\text{if } a\#\widetilde{a}\wedge a'\#\Psi''\wedge b\#\widetilde{a}\wedge b'\in\Psi'' \qquad (\textsc{Ce}1)$$
$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma,\Psi'\otimes\{a'\}),C\rangle$$
$$\text{if } a\#\widetilde{a}\wedge a'\#\Psi''\wedge b'\in\Psi'' \qquad (\textsc{Ce}2)$$

$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma[b':=a'],\Psi'),C\rangle$$
$$\text{if } a\#\widetilde{a}\wedge a'\in\Psi''\wedge b\#\widetilde{a}\wedge b'\#\Psi'' \qquad (\textsc{Ce}3)$$
$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma,\Psi'\otimes\{b'\}),C\rangle$$
$$\text{if } a'\in\Psi''\wedge b\#\widetilde{a}\wedge b'\#\Psi'' \qquad (\textsc{Ce}4)$$

$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma,\Psi'\otimes\{a',b'\}),C\rangle$$
$$\text{if } a\#\widetilde{a}\wedge a'\#\Psi''\wedge b\#\widetilde{a}\wedge b'\#\Psi'' \qquad (\textsc{Ce}5)$$
$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma[a':=b'],\Psi'\otimes\{b'\}),C\rangle$$
$$\text{if } a\#\widetilde{a}\wedge a'\#\Psi''\wedge b\#\widetilde{a}\wedge b'\#\Psi'' \qquad (\textsc{Ce}6)$$
$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma[b':=a'],\Psi'\otimes\{a'\}),C\rangle$$
$$\text{if } a\#\widetilde{a}\wedge a'\#\Psi''\wedge b\#\widetilde{a}\wedge b'\#\Psi'' \qquad (\textsc{Ce}7)$$

$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \langle(\sigma,\Psi'),C\rangle$$
$$\text{if } \Psi''\vdash a'=b' \qquad (\textsc{Ce}8)$$

$$\langle(\sigma,\Psi'),(\nu\widetilde{a})\{\!|\Psi\vdash a\doteq b|\!\}\wedge C\rangle \rightarrowtail \blacksquare$$
$$\text{if } \Psi''\nvdash a'=b'\wedge(a\in\widetilde{a}\wedge b\in\widetilde{b}) \qquad (\textsc{Ce}9)$$

Figure 4: Common ether constraint refinement transition rules. The last rule is not strictly necessary, but it is included for completeness. We also define for each rule $\Psi''=\Psi\sigma\otimes\Psi'$ and $a'=a\sigma$ and $b'=b\sigma$. Note a',b',Ψ'' may be variants of an alpha conversion in order to respect $\widetilde{a}\#\sigma,\Psi'$.

of $\rightarrowtail$ is a refinement of a partial solution to extend the partial solution to include the next conjunct in the configuration

$$\langle(\sigma,\Psi'),C\wedge C'\rangle \rightarrowtail \langle(\sigma',\Psi''),C'\rangle$$

that is the tuple $(\sigma',\Psi'')\in sol(C)$. Recall that we treat transition constraints as lists

$$(\nu\widetilde{a_1})\{\!|\Psi_1\vdash\varphi_1|\!\}\wedge((\nu\widetilde{a_2})\{\!|\Psi_2\vdash\varphi_2|\!\}\wedge\cdots(\cdots\wedge\mathbf{true})\cdots)$$

and there is no generality lost since $\wedge$ is associative.

In order to find a solution (σ,Ψ') to a constraint C we invoke

$$\langle(\mathrm{Id},\emptyset),C\rangle \rightarrowtail^* \langle(\sigma,\Psi'),\mathbf{true}\rangle$$

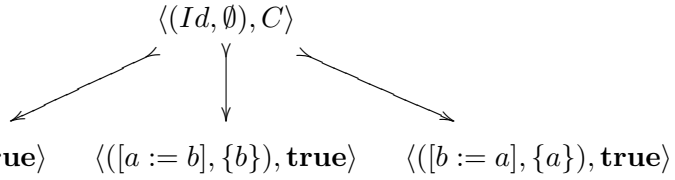
where $\rightarrowtail^*$ is a transitive reflexive closure of $\rightarrowtail$, and Id is the identity (empty) substitution. We may also extract all the solutions generated by the rules in figure 4 by taking all possible paths.

Now that the mechanics of the transition system is established, we turn to the motivation of the rules. We start with (CE5) - (CE7). Suppose we have the constraint

$$C = (\nu\varepsilon)\{\!|\emptyset \vdash a \dot{\leftrightarrow} b|\!\}$$

where neither a, nor b are bound and $a\#b$. Note that $sol(C) = sol(C \wedge \mathbf{true})$. What solutions to C we can deduce? We can always make names channel equivalent by just putting them in the environment, thus one possible solution is $(Id, \{a, b\})$ (rule (CE5)), i.e., $\{a, b\} \vdash a \dot{\leftrightarrow} b$ holds by definition. Another possibility is putting only one name into an environment and replacing the other with that name, this gives us this solution $([a := b], \{b\})$, i.e., $\{b\} \vdash b \dot{\leftrightarrow} b$ (rule (CE6)). And the last possibility is a symmetric version of the previous solution (rule (CE7)). There are more solutions to the C constraint, but we do not want to include "junk".

This generalises to partial solutions. For the above example the tree of solutions look like



Now suppose we have the constraint

$$(\nu\varepsilon)\{\!|\{b\} \vdash a \dot{\leftrightarrow} b|\!\}$$

there are two possible solutions either we substitute a with b (rule (CE1)) or add the name a to the environment (rule (CE2)). If we were to restrict the name b as

$$(\nu b)\{\!|\{b\} \vdash a \dot{\leftrightarrow} b|\!\}$$

then we would be facing one possibility – adding a to the environment (rule (CE2)). The rules (CE2) and (CE4) are symmetrical versions of the rules (CE1) and (CE3), respectively.

The rule (CE8) is a trivial case when the constraint is entailed without need for further refinement. The rule (CE9) signifies failure where there are no solutions for a constraint. This is the case when no other rule applies.

The function tr implements the rules given in figure 4. The function is presented with a partial solution and a constraint, and tries to determine which of the rules are applicable by testing side conditions of rules (the first element of a tuple of the list rules). Then the list of applicable rules is constructed by collecting only the functions with a true side condition and those functions are evaluated, i.e. a partial solution is computed. If there are no such functions tr signals a failure with the function fail. If there

are such functions then a node is constructed with the function node, and all the branches are recursively computed. The functions node and fail are discussed later.

```
fun tr ((sigma, psi'), []) = node ((sigma,psi'),[])
  | tr ((sigma, psi'),((c as (avec, psi, (Eq (a,b)))) :: cs))
  = Constraint.subst [c] sigma |>
  (fn [(_, psi'', (Eq (a',b')))] =>
      let
        val psi'' = compose (psi'', psi')
        (* menmonic af = a is fresh in avec *)
        val af    = L.fresh a avec
        val bf    = L.fresh b avec
        val af'   = L.fresh a' psi''
        val bf'   = L.fresh b' psi''
        (* mnemonic am = a is member of avec *)
        val am    = not af
        val bm    = not bf
        val am'   = not af'
        val bm'   = not bf'
        val rules = [
        (* CE1 *)
        ((af andalso af' andalso bf andalso bm'),
          (fn () => (composeSubst sigma (a',b'), psi'))),

        (* CE2 *)
        ((af andalso af' andalso bm'),
          (fn () => (sigma, compose(psi', [a'])))),

        (* CE3 *)
        ((af andalso am' andalso bf andalso bf'),
          (fn () => (composeSubst sigma (b',a'), psi'))),

        (* CE4 *)
        ((am' andalso bf andalso bf'),
          (fn () => (sigma, compose(psi', [b'])))),

        (* CE5 *)
        ((af andalso af' andalso bf andalso bf'),
          (fn () => (sigma, compose(psi', [a',b'])))),

        (* CE6 *)
        ((af andalso af' andalso bf andalso bf'),
          (fn () => (composeSubst sigma (a',b'),
                     compose(psi', [b'])))),

        (* CE7 *)
        ((af andalso af' andalso bf andalso bf'),
          (fn () => (composeSubst sigma (b',a'),
                     compose(psi', [a'])))),

        (* CE8 *)
        ((entails (psi'', Eq (a',b'))),
          (fn () => (sigma, psi')))
```

```
          ]

          val valid = map (fn (_,c) ⇒ c ())
            (List.filter (fn (cond, sol) ⇒ cond) rules)
        in
          case valid of
               [] ⇒ fail ()
             | _  ⇒ node ((sigma,psi'),
                          map (fn sol ⇒ tr (sol, cs)) valid)
        end
      | _ ⇒ Err.undefined ())
```

These two functions imitate the construction of the solution tree and can be thought of as typical SML data constructors. But we do not need the internal nodes of the tree, hence these functions discard the intermediate results. The function fail just returns an empty list. The function node takes two arguments (as a tuple) where the first is the node – a partial solution (σ, Ψ) – and the second is a list of nodes branches. Since we do not keep the intermediary structure, when a we encounter a leaf node (a node without branches) we return a singleton list with that node, otherwise we drop the node and return the union of branches.

```
and fail () = []
and node (n,[]) = [n]
  | node (n,l)  = List.concat l
```

The only thing is left to do is to plug the function tr into the function solve.

```
fun solve cs =
let
  val sols = tr (([], []), cs)
in
  case sols of
       [] ⇒ Either.LEFT []
     | _  ⇒ Either.RIGHT sols
end
```

This ends the common ether example. The common ether Psi-calculus instance introduces the non-trivial assertions, due to this extra effort is required for designing a transition constraint solver. It is a mix of finding a mgu and careful case analysis. This demonstrates the Psi-calculus versatility, showing that we can implement very interesting logics completely outside of the Psi-calculi meta-theory.

4 Conclusion

In this thesis, we presented an automated tool for the Psi-calculi framework. We defined a terminating operational semantics which we implemented in the tool and proved equivalent to the original symbolic operational semantics up to bisimulation. We also extended the Psi-calculi framework with process constants.

This tool is the second automated tool developed for the Psi-calculi framework, the first one [31] is for verifying Psi-calculus instances in a proof assistant.

4.1 Future work

We chose simplicity and clarity over performance. In many instances, the performance could be improved by simply using better optimised data structures. The bisimulation checking algorithm suffers from a diamond graph problem that is the algorithm may revisit the same agent exponentially many times, we could solve this with some form of caching of agents.

Currently, in order to define a new instance or to modify an instance we need to implement it in SML together with constraint solvers, we intend to provide a generic framework for known instance classes, for example Psi-calculi parameters that form free algebras.

We intend to integrate the tool with the Nominal Isabelle [28] theorem prover assistant and the tool [31]. We intend to use the bisimulation checker for generating proofs for the assistant.

The current implementation of the bisimulation algorithm has some drawbacks. The constraints generated by the algorithm include freshness constraint which complicates the constraint solver. We would like to investigate an algorithm which does not require freshness constraints. Another drawback is that the constraints generated by the algorithm entail solutions which contain too little useful information about the reason two agents are bisimilar. We also would like to add an implementation of the strong bisimulation algorithm to the tool.

4.2 Related work

There have been a number of various automated tools developed for process calculi. The most relevant to our work are Mobility Workbench (MWB) [29, 30], SBC [7], and Another Bisimulation Checker (ABC) [8]. MWB is a tool for the polyadic pi-calculus [21]. Its bisimulation checker is based on a different approach than symbolic semantics called open bisimulation. ABC is also a tool for checking open bisimulation of the pi-calculus. SBC is bisimilarity checker for the spi-calculus, its bisimilarity checker is of interest as it is based on the symbolic spi-calculus semantics.

References

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:36–47, 1999.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, 1998.

[3] Jesper Bengtson. *Formalising process calculi*. PhD thesis, Uppsala UniversityUppsala University, Division of Computer Systems, Computer Systems, 2010.

[4] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *CoRR*, abs/1101.3262, 2011.

[5] Jesper Bengtson and Joachim Parrow. Psi-calculi in Isabelle. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 99–114, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] Michele Boreale and Rocco De Nicola. A Symbolic Semantics for the pi-calculus. *Information and Computation*, 126(1):34 – 52, 1996.

[7] Johannes Borgström and Sébastien Briais. SBC. `http://lamp.epfl.ch/~jobo/sbc/`.

[8] Sébastien Briais. ABC. `http://sbriais.free.fr/tools/abc/`.

[9] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *Mathematical Structures in Computer Science*, 19:1191–1222.

[10] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[11] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[12] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7:301–303, May 1964.

[13] Igor Gammer and Eyal Amir. Solving satisfiability in ground logic with equality by efficient conversion to propositional logic. In *In Proceedings of the 7th Symposium on Abstraction, Reformulation, and Approximation*, 2007.

[14] M. Hennessy and H. Lin. Symbolic bisimulations. *Theor. Comput. Sci.*, 138:353–389, February 1995.

[15] Matthew Hennessy and H. Lin. Proof systems for message-passing process algebras. *Formal Asp. Comput.*, 8(4):379–407, 1996.

[16] Graham Hutton. Monadic parsing in Haskell. *Journal of functional programming*, 1998.

[17] Magnus Johansson, Björn Victor, and Joachim Parrow. A fully abstract symbolic semantics for psi-calculi. *CoRR*, abs/1002.2867, 2010.

[18] Magnus Johansson, Björn Victor, and Joachim Parrow. Computing strong and weak bisimulations for psi-calculi. Submitted, 2011.

[19] J. W. Klop. Term rewriting systems. In S. Abramsky et al., editor, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[20] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1st edition, June 1999.

[21] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. 100:1–77, September 1992.

[22] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53:2006, 2006.

[23] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[24] Joachim Parrow. An introduction to the pi-calculus. In Jan Bergstra, Alban Ponse, and Scott Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.

[25] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:2003, 2002.

[26] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[27] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: programming with binders made simple. In *International Conference on Functional Programming*, volume 38, pages 263–274.

[28] Christian Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40:327–356, May 2008.

[29] Björn Victor. *A Verification Tool for the Polyadic π-calculus*. Licentiate thesis, Department of Computer Systems, Uppsala University, Sweden, May 1994. Available as report DoCS 94/50.

[30] Björn Victor, Faron Moller, Mads Dam, and Lars-Henrik Eriksson. MWB. `http://www.it.uu.se/research/group/mobility/mwb`.

[31] Johannes Åman Pohjola. Verifying psi-calculi. Master's thesis, Uppsala University, Department of Information Technology, 2010.

A Pi-calculus bisimulation constraint solver

In this section, we continue the Pi-calculus instance example implementation (Section 3.2.1). We implement a bisimulation constraint solver (Appendix C.1) for the instance. As in other examples (Section 3.2), we use a literal programming style for describing the constraint solver and we omit the line numbering.

The constraints used by the symbolic bisimulation algorithm is more complex than that of symbolic transition constraints. Even when a Psi-calculus instance does not provide any additional logic, bisimulation constraints entail a first-order logic without quantifiers and with equality. Knowing this, to find a solution to a bisimulation constraint is the same problem as to find a model satisfying a preposition logic formula (SAT). We use one of the classical algorithms for SAT in propositional logic, DPLL [11, 10]. The DPLL algorithm is extendible with additional theories [22]. The theories enable the algorithm to be used to solve formulae of superset of propositional logic. Our implementation of the algorithm is not intended to be efficient, the main purpose of it is to provide a sample implementation of a bisimulation constraint solver. Furthermore, the freshness constraint (Definition 38 in Appendix C) solution depends on a complete model of the constraint which it is part of. We introduce an additional step to handle freshness constraints. This step backtracks if it fails. This kind of additional steps might be complicated to use in an off-the-shelf SAT solver.

While it is feasible to translate a first-order logic without quantifiers and with an equality (ground logic) into a propositional logic [13] and thus solve it with a SAT algorithm, it is not clear that such translation exists for freshness constraints.

A constraint that is produced by the bisimulation algorithm contains trivially solvable sub-constraints and a constraint accepted by the DPLL algorithm must be in CNF form. Trivial constraints can be solved at a separate stage with a simpler constraint solver. We use a term rewriter [2] to convert constraints into CNF constraints and to solve trivial constraints. A term rewriter is provided by the tool's framework, we demonstrate how to implement term rewriting rules.

Recall that the structure PiSymbolicInstance (Section 3.2.1) contains the SML code presented here.

A.1 Constraints

The table below lists the bisimulation algorithm constraint forms (see Appendix C.1 for the bisimulation algorithm) and the SML representation.

Notation	SML code
true	True
false	False
$(\nu\widetilde{a})\{\!\|\mathbf{1} \vdash a \dot{\leftrightarrow} b\|\!\}$	Atomic (avec, unit, Eq(a,b))
$\{\!\|a = b\|\!\}$	TermEq (a,b)
$\{\!\|a\#P\|\!\}$	FreshP (a,p)
$C \wedge C'$	Conj (c,c')
$C \vee C'$	Disj (c,c')
$C \Rightarrow C'$	Imp (c,c')

First we define negation since we need it translation into CNF.

$$\neg C \stackrel{\text{def}}{=} C \Rightarrow \mathbf{false}$$

We write $(\nu\widetilde{a})\{\!|a \dot{\leftrightarrow} b|\!\}$ for $(\nu\widetilde{a})\{\!|\mathbf{1} \vdash a \dot{\leftrightarrow} b|\!\}$, and $\{\!|a = b|\!\}$ denotes a term equality constraint.

Similarly to the transition constraints, the bisimulation constraint is a nominal datatype and has a substitution function defined on it, we get the default implementation of bisimulation constraint by applying the functor BisimConstraint.

```
structure BisimConstraint = BisimConstraint(PiInstanceNom)
```

The functor BisimConstraint defines the constraints required by the bisimulation constraint generation algorithm.

A.2 Term rewriting and all that

A term rewriter is a syntactic transformation function which maps terms to terms by applying a set of term rewriting (transformation) rules, usually a set of identities (one direction) [2]. In our case, the terms are constraints. A term rewriter is provided by the BisimConstraint structure. For a full account of term rewriting systems see [19, 2].

As mentioned in the introduction, we use a term rewriter for simplifying constraints (solving trivial sub-constraints) produced by the bisimulation algorithm, we also use a term rewriter for translating a constraint to a Conjunctive Normal Form (CNF).

The function rewrite (from the structure BisimConstraint) takes a list of term rewriting rules and a constraint. A term rewriting rule is a function which given a constraint and returns either a constraint if the rewrite rules is applicable or, otherwise, a failure. For instance a term rewriting rule

$$C \wedge \mathbf{true} \rightarrow C$$

can be implemented as a SML function

```
val conjRightUnit = fn (Conj (c,True)) => SOME c
                     | _               => NONE
```

For instance, in order to remove all the right units in a constraint c with the above term rewriting rule, we do rewrite [conjRightUnit] c.

Now we are ready to express the term rewriting rules for solving the trivial constraints, they are as follows.

$$\{\!|\mathbf{1} \leq \mathbf{1}|\!\} \to \mathbf{true} \qquad (\nu\widetilde{a})\{\!|\top|\!\} \to \mathbf{true} \qquad (\nu\widetilde{a})\{\!|a \dot{\leftrightarrow} a|\!\} \to \mathbf{true}$$

$$\{\!|a = a|\!\} \to \mathbf{true} \qquad (\nu\widetilde{a})\{\!|a \dot{\leftrightarrow} b|\!\} \to \mathbf{false} \text{ if } a\#b \wedge (a \in \widetilde{a} \vee b \in \widetilde{b})$$

Compared with rules for solving trivial transition constraints in Section 3.2.1, we add two more rules to solve additional cases. The static implication case is always **true** as the Pi-calculus instance does not feature assertions; term equality is the same as channel equivalence without binders.

Additionally, we can trivially solve freshness constraints if there are no sub-constraints in a constraint C with the channel equivalence (or the term equivalence) which contain a name used in a freshness constraint. By parameterising the term rewriter on a constraint C we get

$$\{\!|a\#P|\!\} \to_C \mathbf{true} \quad \text{if } a\#C$$

Since in Pi-calculus instance we can regard channel equivalence and term equality as the same constraint, we convert channel equivalence to term equality and we name this transformation a normalisation.

$$(\nu\widetilde{a})\{\!|a \dot{\leftrightarrow} b|\!\} \to \{\!|a = b|\!\} \quad \text{if } a, b\#\widetilde{a}$$

After solving the trivial constraints, we get a number of redundant constraints which we can prune by simplification. For example, the simplification rules simplificationRules provided by the functor BisimConstraint are the following (the symmetric versions are elided).

$$\mathbf{false} \wedge C \to \mathbf{false} \qquad \mathbf{true} \wedge C \to C \qquad C \wedge C \to C \qquad C \wedge (C \vee C') \to C$$

$$\mathbf{false} \vee C \to C \qquad \mathbf{true} \vee C \to \mathbf{true} \qquad C \vee C \to C \qquad \mathbf{false} \Rightarrow C \to \mathbf{true}$$

$$C \Rightarrow \mathbf{true} \to \mathbf{true} \qquad \cdots \wedge C \wedge \cdots \wedge \neg C \wedge \cdots \to \mathbf{false}$$

$$\cdots \vee C \vee \cdots \vee \neg C \vee \cdots \to \mathbf{true} \qquad \neg\neg C \to C$$

Note the above rewriting rules are identities in first-order logic. It is not hard to show that these hold for the bisimulation constraints.

First, we make all the definition locally visible.

```
local open BisimConstraint in
```

The following code implements the trivial constraint solving and simplification rules.

```
val trivialConditionRules =
  (fn (StImp (psi,psi')) ⇒ SOME True
    | _ ⇒ NONE) ::

  (fn (Atomic (avec, unit, T)) ⇒ SOME True
    | _ ⇒ NONE) ::

  (fn (Atomic (avec, unit, Eq (a, b))) ⇒
      if a = b then SOME True else NONE
    | _ ⇒ NONE) ::

  (fn (TermEq (a, b)) ⇒
      if a = b then SOME True else NONE
    | _ ⇒ NONE) ::

  (fn (Atomic (avec, unit, Eq (a, b))) ⇒
      if a ≠ b andalso
        (Lst.member a avec orelse Lst.member b avec)
          then SOME False else NONE
    | _ ⇒ NONE) ::

  (fn (Atomic ([], unit, Eq(a,b))) ⇒ SOME (TermEq (a,b))
    | _ ⇒ NONE) ::

  (fn (Atomic ((avec as (n::ns)), unit, Eq (a,b))) ⇒
      if L.fresh a avec andalso L.fresh b avec
          then SOME (Atomic ([], unit, Eq(a,b)))
          else NONE
    | _ ⇒ NONE) ::

  []
```

The trivial freshness constraint rule.

```
fun freshnessRule c =
  (fn (FreshP (a,p)) ⇒
      if BisimConstraint.fresh a c then SOME True
      else NONE
   | _ ⇒ NONE) :: []
```

As we already mentioned, we employ the term rewriter for transforming a constraint into a CNF form. Let us first recall that CNF is a formula of the form

$$(\ell_{1,1} \vee \cdots \vee \ell_{1,m_1}) \wedge \cdots \wedge (\ell_{n,1} \vee \cdots \vee \ell_{n,m_n})$$

i.e. a conjunction of disjunctions of literals. A literal can be of two kinds a positive and a negative of some atomic formula A:

$$\ell ::= A \mid \neg A$$

In the Pi-calculus instance case, an atomic formula is one of:

$$A ::= \{\!|a = b|\!\} \mid \{\!|a\#P|\!\}$$

In SML code.

```
fun isLiteral (FreshP _)                = true
  | isLiteral (Imp (FreshP _, False)) = true
  | isLiteral (TermEq _)                = true
  | isLiteral (Imp (TermEq _, False)) = true
  | isLiteral _                         = false
```

For obtaining a CNF formula from an arbitrary constraint, the following procedure is used:

- Remove implications.
- Move all the negations inwards.
- Distribute disjunctions over conjunctions.

The first step is this rewrite rule

$$C \Rightarrow C' \rightarrow \neg C \vee C'$$

The second step is de Morgan's laws

$$\neg(C \wedge C') \rightarrow \neg C \vee \neg C' \qquad \neg(C \vee C') \rightarrow \neg C \wedge \neg C'$$

And the last step

$$C \vee (C' \wedge C'') \rightarrow (C \vee C') \wedge (C \vee C'')$$

The above procedure is implemented by using these term rewriting rules in the function cnf in the BisimConstraint structure.

A.3 DPLL algorithm

After the simplification, normalisation, and transformation to a CNF form we use the Davis-Putnam-Logemann-Loveland[11, 10] algorithm to find a model, i.e. a solution to the constraint. The simplest form of DPLL algorithm is used to find a model (a valuation to variables) to a propositional logic formula but the bisimulation constraint give more expressiveness than propositional logic. The DPLL algorithm can be extended to supersets of propositional logic by treating the atomic formulas of background theory T as propositional variables, and by defining custom decision procedures for the theory T in the model. This method generally known as SAT Modulo Theory (SMT), see [22] for a description and survey of SMT approaches. We also use [22] as a guideline and we present here their formulation of the algorithm as a transition system.

The DPLL algorithm accepts a formula in CNF form. When describing the algorithm we use the following notation for CNF formula.

$$\ell_{1,1} \vee \cdots \vee \ell_{1,m_1}, \ldots, \ell_{n,1} \vee \cdots \vee \ell_{n,m_n}$$

i.e., we separate the conjuncts with commas.

A central concept in DPLL is a (partial) model. A model is a consistent set of literals $\ell_1, \ldots, \ell_n$. The set can be thought as a conjunction so consistency means an absence of a contradiction. The decision procedure of the theory T is parameterised with a relation $\mathcal{M} \models C$. This relation, intuitively, means if the constraint C can be solved trivially by considering literals in the model $\mathcal{M}$ as true. In our presentation, by replacing literals of a model $\mathcal{M}$ in the constraint C with constraints **true** and by expecting the outcome of the simplification and the trivial constraint solving to reduce the constraint C to the constraint **true**. If a model is inconsistent, we say it is in a fail state $\bot$.

The algorithm is formulated with a transition systems where transitions take the form:

$$\mathcal{M} \Vdash C \hookrightarrow \mathcal{M}' \Vdash C$$

the model $\mathcal{M}$ is a partial model, and the transition goes into a partial model $\mathcal{M}'$ with more literals accounted for. Note the constraint is not changed although in the implementation we simplify the constraint according the partial model.

We implement the simplest form of DPLL algorithm, given by the state transition rules below, taken from [22] with modifications discussed later.

$$\mathcal{M} \Vdash C, \mathrm{c} \vee \ell \;\; \hookrightarrow \;\; \mathcal{M}\ell \Vdash C, \mathrm{c} \vee \ell$$
$$\text{if } \mathcal{M} \models \neg \mathrm{c} \;\wedge\; l \notin \mathcal{M} \qquad (\textsc{UnitProp})$$
$$\mathcal{M} \Vdash C \;\; \hookrightarrow \;\; \mathcal{M}\ell \Vdash C$$
$$\text{if } (\exists \mathrm{c} \in C)\ell \in \mathrm{c} \;\wedge\; (\forall \mathrm{c} \in C)\neg\ell \notin \mathrm{c} \;\wedge\; \ell \notin \mathcal{M} \qquad (\textsc{PureLit})$$
$$\mathcal{M} \Vdash C \;\; \hookrightarrow \;\; \mathcal{M}\ell^d \Vdash C$$
$$\text{if } \ell \notin \mathcal{M} \wedge (\exists \mathrm{c} \in C)\ell \in \mathrm{c} \vee \neg\ell \in \mathrm{c} \qquad (\textsc{Decide})$$
$$\mathcal{M} \Vdash C, \mathrm{c} \;\; \hookrightarrow \;\; \blacksquare$$
$$\text{if } \mathcal{M} = \bot \vee (\mathcal{M} \models \neg \mathrm{c} \wedge \nexists \ell^d \in \mathcal{M}) \qquad (\textsc{FailC})$$
$$\mathcal{M}\ell^d\mathcal{N} \Vdash C, \mathrm{c} \;\; \hookrightarrow \;\; \mathcal{M}\neg\ell \Vdash C, \mathrm{c}$$
$$\text{if } (\mathcal{M}\ell^d\mathcal{N} = \bot \vee \mathcal{M}\ell^d\mathcal{N} \models \neg \mathrm{c}) \;\wedge\; \nexists \ell'^d \in \mathcal{N} \qquad (\textsc{Backtrack})$$

Recall the constraint C is in a CNF form. We mark a literal ℓ as ℓ^d to denote a decision point where we might backtrack if a consistent model is not found. Note the literal is positive. If there is no more rules to apply then $\mathcal{M}$ is a complete model.

We briefly summarise the DPLL rules: (UnitProp) applies whenever there is a clause in a constraint such that by taking out the literal ℓ model entails the rest of the clause as false, therefore it forces the literal ℓ to be added as a positive in a model as it makes the clause true; (PureLit) applies when there is a clause with the literal ℓ which no other clause in the constraint C has a negative of that literal so this forces literal into a model as a positive, otherwise the decision rule applies (Decide), which adds a literal as a positive and marks it as a decision, such that the (Backtrack) rule

can return to that point where the decision was made and reverse it (negate the literal and unmark it as a decision) if the decision was wrong. Lastly, fail (FailC) if a model entails negative clause and there were no decisions made. If none of the above rules apply then we are left with a complete model which satisfies the constraint. We extend the original DPLL algorithm with the disjuncts in rules (FailC) and (Backtrack) where it is checked if a model itself is in a consistent state (cf. [22]).

Next we describe the model and its construction from literals.

A.4 Model

Here we consider the T part in the SMT problem. From the previous section recall the operation in the transition system

$$\mathcal{M}\ell$$

we call it *concatenation*. So, the objective of this section is to define this operation for the Pi-calculus instance bisimulation constraints, and by doing so establish the background theory.

Let us reiterate the constraints that form the model.

$$A ::= \{\!|a = b|\!\} \;\mid\; \{\!|a\#P|\!\}$$

The equality is transitive coupled with a fact that literals may be negated, the naïve approach does not work as it produces an inconsistent model. For instance, if we have a model

$$\mathcal{M} = \{\!|a = b|\!\}, \{\!|b = c|\!\}$$

and we have a literal to concatanate with

$$\ell = \neg\{\!|a = c|\!\}$$

by just adding the literal ℓ to the model $\mathcal{M}$ gives the following model $\mathcal{M}, \ell$

$$\mathcal{M}, \ell = \{\!|a = b|\!\}, \{\!|b = c|\!\}, \neg\{\!|a = c|\!\}$$

but this model is inconsistent because by transitivity the constraint $\{\!|a = c|\!\}$ is in the model $\mathcal{M}$.

The solution we use is to compute an equivalence closure (a transitive, reflexive, symmetric closure) every time a literal is added to a model, we denote it by

$$\mathcal{M}^{=}$$

Continuing with the example the model is (we omit the reflexive and symmetric literal)

$$\mathcal{M}^{=} = \{\!|a = b|\!\}, \{\!|b = c|\!\}, \{\!|a = c|\!\}$$

now it is trivial to detect the inconsistency. In fact, in our implementation we do not even need to check for such inconsistency as the model would entail the literal $\neg\{\!|a = c|\!\}$ in a constraint considered as the constraint **false** and it would be reduced by the simplifier. As we later see, the computation of equivalence closure will benefit us when extracting a solution from a model.

Formally, an equivalence closure on a binary relation R can be given as the least relation $R^{=}$:

$$\begin{array}{rcl} R^{=} & = & \{(b,a) : (a,b) \in R^{=}\} \\ & \cup & \{(a,c) : \exists b.(a,b) \in R^{=} \wedge (b,c) \in R^{=}\} \\ & \cup & R \end{array}$$

The equivalence closure on a model $\mathcal{M}$ is

$$\mathcal{M}^{=} = \{\{\!|a = b|\!\} : \{\!|a = b|\!\} \in \mathcal{M}\}^{=} \cup \mathcal{M}$$

i.e. treating literals of the form $\{\!|a = b|\!\}$ in $\mathcal{M}$ as binary tuple and by extracting these tuples, then computing an equivalence closure on them, and finally adding them back to the model.

The implementation of computing equivalence closure on a model is also split into two operations.

The least fixed-point definition of equivalence closure gives straightforward computation algorithm. The function eqClosure' computes $R^{=}$ given above when it is provided with a list of tuples.

```
fun eqClosure' ids =
let
  (* a tuple membership testing in a list *)
  fun eqInClosure (a,b) ids =
    Lst.exists (fn id => id = (a,b)) ids

  (* symmetry *)
  val r = map (fn (a,b) => (b,a)) ids
  val r = List.filter (fn (a,b) =>
                         not (eqInClosure (a,b) ids)) r
  val r = ids @ r

  (* transitivity *)
  val t = map (fn (a,b) =>
              map (fn (_,c) => (a,c))
                  (List.filter (fn (b',c) => b = b') r)) r
  val t = List.concat t
  val t = List.filter
            (fn (a,b) => not (eqInClosure (a,b) r)) t
in
  if null t
      then r
      else eqClosure' (r @ t)
end
```

The fixed-point computation for an equivalence closure presented above is not an efficient way for computing an equivalence closure but it is a straightforward way and it is easy to implement. For a more efficient solution see Galler-Fischer [12] data structure for computing equivalence classes.

The function eqClosure is the final step for computing and equivalnce closure of the model $\mathcal{M}$.

```
fun eqClosure m =
let
  val (eq,m') = Either.partition
      (map (fn c =>
              case c of
                   TermEq _ => Either.LEFT c
                 | _ => Either.RIGHT c ) m)
  val eqT = map (fn (TermEq eq) => eq
                  | _ => Err.undefined () ) eq
  val eqCl = eqClosure' eqT
  val eq' = map (fn eq => TermEq eq) eqCl
in
  eq @ m'
end
```

A syntactic equality on literals is not sufficient as the term equality and negation of the term equality is commutative, what is more, a freshness constraint may contain alpha equivalent agents. This motivates us to be explicit in the following definition of constraint equality

$$\begin{aligned}
\{\!|a = b|\!\} = \{\!|a' = b'|\!\} &\Leftrightarrow a = a' \wedge b = b' \;\vee\; a = b' \wedge b = a' \\
\neg\{\!|a = b|\!\} = \neg\{\!|a' = b'|\!\} &\Leftrightarrow a = a' \wedge b = b' \;\vee\; a = b' \wedge b = a' \\
\{\!|a\#P|\!\} = \{\!|a'\#P'|\!\} &\Leftrightarrow a = a' \wedge P =_\alpha P' \\
\neg\{\!|a\#P|\!\} = \neg\{\!|a'\#P'|\!\} &\Leftrightarrow a = a' \wedge P =_\alpha P'
\end{aligned}$$

The function lEq implements above equivalence. It expects two literals and returns true if they are equivalent and false if they are not.

```
fun lEq l l' =
  case (l, l') of
       (TermEq (a,b), TermEq (a',b')) =>
          (a = a' andalso b = b') orelse
          (a = b' andalso b = a')
     | (Imp(TermEq (a ,b ), False),
        Imp(TermEq (a',b'), False)) =>
          (a = a' andalso b = b') orelse
          (a = b' andalso b = a')
     | (FreshP(a,p), FreshP(b,q)) => a = b andalso
                                     Psi.eqD(p,q)
     | (Imp(FreshP(a,p),False),
        Imp(FreshP(b,q),False)) => a = b andalso
                                   Psi.eqD(p,q)
     | _ => false
```

Computing of $\ell \in \mathcal{M}$, now becomes straightforward.

```
fun inM l m = Lst.exists (fn l' ⇒ lEq l l') m
```

We also require a function which would compute a negation of a literal, i.e. $\neg(\neg\ell) = \ell$ and $\neg(\ell) = \neg\ell$.

```
fun negL (Imp (c, False)) = c
  | negL c = (Imp (c, False))
```

Now we have everything needed to define the concatenation operation $\mathcal{M}\ell$ on the model $\mathcal{M}$ and the literal ℓ. A concatenation may result in a new model with the literal taken into the account of a model or the inconsistent state $\perp$. The concatenation operation is the theory part of SMT, however, there is a need to do theory judgment at the point when a full model is constructed by the DPLL algorithm, see validateAndReturn and dpllSAT functions.

We have four cases of literals to consider. In all cases, we do not need to check for an inconsistency arising from a model already containing a negated literal to that being added. As we mentioned before, we are guaranteed that such negated literal would not be considered for an addition to a model.

When adding a term equality literal to a model, we check if the model would be consistent after the addition. A freshness constraint disallows a name to be used in the domain of a substitution, and a model serves as a means to extract a substitution sequence, consequentially an inconsistency may arise if both of the equality sides are disallowed in a substitution sequence. Note this needs to be checked for the model with an added literal and an equivalence closure, for instance if we have $\{\!|a = b|\!\}, \{\!|a\#P|\!\}, \{\!|c\#P|\!\}$ model and a candidate literal is $\{\!|b = c|\!\}$ and if we would not require an equivalence closure then such candidate would be added to a model but there is no substitution sequence which would satisfy such model, we may compute these substitution sequences from above model $[a := c, b := c]$, $[b := a]$, $[b := c]$, $[b := a, c := b]$ but neither of them satisfy such model. The second clause requires that there must not be an induced equivalence and disequality. A resulting model is an equivalence closure.

$$\mathcal{M}\{\!|a = b|\!\} = \begin{cases} \perp & \text{if } \exists P, P', c, d.\{\!|c\#P|\!\}, \{\!|d\#P'|\!\}, \{\!|c = d|\!\} \in \mathcal{M}' \\ \perp & \text{if } \exists c, d.\neg\{\!|c = d|\!\}, \{\!|c = d|\!\} \in \mathcal{M}' \\ \mathcal{M}' & \text{otherwise} \end{cases}$$

where $\mathcal{M}' = (\mathcal{M}, \{\!|a = b|\!\})^{=}$.

The disequality case does not pose any complications as the negations are reduced before applying this operation (more on this later).

$$\mathcal{M}\neg\{\!|a = b|\!\} = \mathcal{M}, \neg\{\!|a = b|\!\}$$

In case when adding a freshness constraint, the only inconsistency that may arise is disallowing both of the sides of some equality in a model for a

substitution.

$$\mathcal{M}\{\!|a\#P|\!\} = \begin{cases} \bot & \text{if } \exists b, P'.\{\!|b\#P'|\!\}, \{\!|a = b|\!\} \in \mathcal{M} \\ \mathcal{M}, \{\!|a\#P|\!\} & \text{otherwise} \end{cases}$$

The last case is simply adding the literal into the model.

$$\mathcal{M}\neg\{\!|a\#P|\!\} = \mathcal{M}, \neg\{\!|a\#P|\!\}$$

Note the concatenation operation does not guaranty a model is free from inconsistencies. The clauses that check for inconsistencies arising from freshness constraints are there for an early detection, a recheck is needed for freshness constraints in a model as the last step since they *depend* on the substitution sequence constructed, i.e., a complete model.

The function extendM computes the concatenation operation given a model m and a literal l, returns a new model SOME on success, otherwise NONE if the model would be inconsistent after addition of the literal.

```
fun extendM m (l as TermEq (a,b)) =
let
  val meq = eqClosure (l::m)
  val fr  = List.concat
      (map (fn (FreshP (a,_)) => [a] | _ => []) meq)
in
  if Lst.exists
      (fn (TermEq (c,d)) => Lst.member c fr
                            andalso Lst.member d fr
        | _ => false) meq
   then NONE
   else
     if Lst.exists
          (fn (ell as TermEq (a,b)) => inM (negL ell) meq
            | _ => false) meq
       then NONE
       else SOME meq
end
  | extendM m (l as Imp(TermEq(a,b), False)) = SOME (l::m)
  | extendM m (l as FreshP (a,_)) =
      if Lst.exists (fn (FreshP (b,_)) =>
          inM (TermEq (a,b)) m  | _ => false) m
      then NONE
      else SOME (l::m)
  | extendM m (l as (Imp (FreshP _, False))) = SOME (l::m)
  | extendM _ _ = Err.error "An undefined case in extendM"
```

A.5 The implementation of DPLL

The implementation we give here is reminiscent of On-Line SAT solver (see [22, Section 3.2] for a survey of SMT techniques), i.e., a theory judgement is made every time a literal is added to a model, and instead of restarting

a SAT procedure whenever model is inconsistent, the algorithm backtracks to the last known good point where model was still consistent.

In our implementation of the DPLL algorithm, a state transition simplifies a CNF constraint

$$\mathcal{M}, C \Vdash \mathcal{M}', simp_{\mathcal{M}'}(C)$$

such that literals entailed by the model with an added literal are simplfied. This gives us a strict termination condition

$$\mathcal{M}, C \Vdash \mathcal{M}', \mathbf{true}$$

i.e., $\mathcal{M}'$ is a complete model to the initial constraint.

For implementing the $simp_{\mathcal{M}}$ function we yet again use the term rewriter. We introduce additional to the simplification term rewriting rules parameterised over the model $\mathcal{M}$.

$$\ell \rightarrow_{\mathcal{M}} \mathbf{true} \text{ if } \ell \in \mathcal{M} \qquad\qquad \ell \rightarrow_{\mathcal{M}} \mathbf{false} \text{ if } \neg\ell \in \mathcal{M}$$

The function modelRules implements the above term rewriting rules. The function given a model m produces a singleton list with term rewriting rule.

```
fun modelRules m =
  [fn l ⇒
      if isLiteral l then
        if inM l m then SOME True
        else if inM (negL l) m then SOME False
        else NONE
      else NONE]
```

The above entailment rewriting rules are applied together with the simplification rules. The function simpM runs a term rewriter on c constraint using the simplification rules an the rules we defined above.

```
fun simpM m c =
  rewrite (simplificationRules @ modelRules m) c
```

In order to simplify the implementation of the DPLL algorithm, we treat constraints in CNF form as a list of lists, that is, a list of list of disjuncts. The function disjToList accepts a constraint which is disjunction of atomic constraints and returns a list of atomic constraints.

```
fun disjToList (Disj (c,c')) = disjToList c @ disjToList c'
  | disjToList c = [c]
```

Similarly, the function cnfToList accepts a constraint in CNF form and produces a list of lists of atomic constraints.

```
fun cnfToList (Conj (c,c')) = cnfToList c @ cnfToList c'
  | cnfToList c = [disjToList c]
```

The function simpClause uses the above simplification on a clause c. The implementation is very simple: we reconstruct a disjunction, then we apply the term rewriter and finally we split the disjunction into a list. So this function implements the relation $\mathcal{M} \models c \vee c' \vee \ldots$ for a disjunction.

```
fun simpClause m c =
  disjToList (simpM m (disjunct c))
```

The function simpCNF is similar to the above simpClause but it accepts a CNF. This is the relation $\mathcal{M} \models C$.

```
fun simpCNF m c =
  cnfToList (simpM m (conjunct (map disjunct c)))
```

The problem that we are solving is finding a solution for a bisimulation constraint C, while the model we get from the DPLL algorithm is conjunction of constraints. We face with yet another layer of constraint solving but we already solved a similar problem for the transition constraints. The function modelSubst computes a *mgu* given a model m.

```
fun modelSubst m = modelSubst' m []
and modelSubst' [] sigma = sigma
  | modelSubst' (TermEq(a,b)::eqs) sigma =
      modelSubst'
          (map (fn e => BisimConstraint.subst e [(a,b)]) eqs)
          (composeSubst sigma (a,b))
  | modelSubst' (_::eqs) sigma = modelSubst' eqs sigma
```

We may not get a correct solution by extracting it from any model. We first need to prepare a model for extraction. This is due to the freshness constraints since it constrains the domain of the substitution function. We are helped with the fact that the model is an equivalence closure, so we only need to orient the equalities, more specifically delete the non conforming equalities. The function orientM given a model m returns a model consisting only with literals of kind of term equality.

```
fun orientM m =
let
  val fr =
    List.concat (map (fn (FreshP (a,_)) => [a] | _ => []) m)
  val eq =
    List.filter (fn (TermEq (a,_)) => not (Lst.member a fr)
                  | _ => false) m
in
  eq
end
```

By inspecting the DPLL rules, we notice that the model entailment relation comes only in the form $\mathcal{M} \models \neg$C, where C is a clause. Therefore we can compute such relation just by simplifying the clause w.r.t. a model $\mathcal{M}$ and expecting to find a constant **false**. This is exactly what the function modelsNegC does, it expects a model m and a clause c.

```
fun modelsNegC m c =
  case simpClause m c of
       [False] ⇒ true
     | _       ⇒ false
```

We are now at a position where we have definitions required to implement DPLL algorithm. We will implement the rules (UnitProp) and (PureLit) as separate functions. They are called by the main function dpllSAT. The failure is represented by `NONE`. The decision rule (Decide) is implemented as two recursive calls returning the first one that succeeded that we get a backtracking rule for free by using Sml backtracking mechanics.

The function pureLit implements the rule (PureLit). It takes a model m and a CNF as a list, it traverses the list a clause at a time by checking each literal in a clause for a negation of a literal in the rest of the CNF. There is no need to check the whole CNF for the negation as the previous queries would have identified a literal at the current position. The function pureLit returns `SOME` literal if it finds a literal which has no negation in any of the clauses, otherwise returns `NONE`.

```
fun pureLit m [] = NONE
  | pureLit m (c::cnf) =
  case
    List.find
      (fn l ⇒ not (inM l m) andalso
          Lst.all (fn c ⇒ not (inC (negL l) c)) cnf) c
    of NONE ⇒ pureLit m cnf
     | SOME l ⇒ SOME l
```

This is just an auxiliary function for the above which checks if a literal l is in a clause c.

```
and inC l c = Lst.exists (fn l' ⇒ lEq l l') c
```

The other rule (UnitProp) is implemented by the function unitProp. It takes the same arguments as the pureLit function. The function tries to find a clause in a CNF formula in which taking out a literal would simplify that clause to **false** by examining each clause in order. The function returns `SOME` literal if such clause exist and otherwise `NONE`.

```
fun unitProp m [] = NONE
  | unitProp m (cl::cnf) =
  case
    List.find
      (fn (l,c) ⇒ (not (inM l m)) andalso modelsNegC m c)
      (splitL cl)
    of NONE ⇒ unitProp m cnf
     | SOME (l,_) ⇒ SOME l
```

The function splitL is an auxiliary function for the above. This function given a list, splitL returns a tuple list where every tuple contains an element from the original list and the original list without that element.

```
and splitL' [] pls = []
  | splitL' (l::ls) pls =
      (l, pls @ ls) :: splitL' ls (pls @ [l])
and splitL ls = splitL' ls []
```

Now we look at a series of mutually recursive functions which complete the DPLL algorithm.

The model may become inconsistent and therefore we should be able to backtrack, i.e., the function extendAndDpll first tries to extend the model m with the literal l and if it fails returns a failure, otherwise calls the main function dpllSAT with the extended model.

```
fun extendAndDpll m l cnf =
  case extendM m l of
       NONE ⇒ NONE
     | SOME m ⇒ dpllSAT m cnf
```

The function splitDpll is an implementation of both (BACKTRACK) and (DECIDE) rules. So given a model m, a literal l, and a CNF cnf first try to solve the cnf with positive literal, if that fails it backtracks and then tries to solve the cnf by extending the model with a negative literal.

```
and splitDpll m l cnf =
  case extendAndDpll m l cnf of
       SOME m ⇒ SOME m
     | NONE   ⇒ extendAndDpll m (negL l) cnf
```

The main function of an implementation of the DPLL algorithm. It accepts a model m and a constraint cnf in a CNF form. The first thing it does at each recursive call is simplify the given CNF and checks if it completed constructing a model and then validates it, if not, then it chains the above rules, first by trying unit propagation, then by trying pure literal, and lastly by doing a decision. The decision takes the first literal of the first clause, other rules use the found literal.

```
and dpllSAT m [] = validateAndReturn m
  | dpllSAT m (cnf as (c::cs)) =
let
  val cnf' = simpCNF m cnf
in
  case cnf' of
       [[True]] ⇒ validateAndReturn m
     | _ ⇒ (
  case unitProp m cnf' of
       SOME l ⇒ extendAndDpll m l cnf'
     | NONE ⇒
       (case pureLit m cnf' of
             SOME l ⇒ extendAndDpll m l cnf'
           | NONE ⇒
             let
               val l = hd c
             in
               splitDpll m l cnf'
```

```
                end) )
end
```

A.6 Additional consistency check

Even though we are checking for inconsistencies when we are constructing a model, we may not find all of them until a model is fully constructed. So, as a last step, we check if freshness constraints and freshness constraint negations hold with a complete model. Recall, freshness constraint's $\{\!|a\#P|\!\}$ solution space is defined by $a\#P\sigma \wedge a\#dom(\sigma)$, and for the negation we have $a \in n(P\sigma) \vee a \in dom(\sigma)$. For the former we only need to check the first conjunct $a\#P\sigma$ as the second is already true by the way we construct a model and extract a substitution sequence, and for the latter we check both disjuncts.

The function validateAndReturn takes a model m and returns a substitution sequence, it does this by first constructing a substitution sequence from a model and checking if all the freshness constraints and their negation hold, and if that is the case it returns the substitution sequence, otherwise it fails with `NONE`.

```
and validateAndReturn m =
let
  val sigma = modelSubst (orientM m)
in
  if
    Lst.all (
        fn (FreshP (a,p)) ⇒ Psi.fresh a (Psi.subst p sigma)
         | (Imp (FreshP (a,p), False)) ⇒
             not (Psi.fresh a (Psi.subst p sigma)) orelse
             Lst.member a (dom sigma)
         | _ ⇒ true
    ) m
  then SOME sigma
  else NONE
end
```

The final step is to 'plug in' the DPLL algorithm into the framework.

A.7 Putting it together

The function solveBisim is an implementation of the bisimulation constraint solver. It is similar to the solve function, it takes a constraint c and produces a list of substitution and assertion or a counter example.

```
fun solveBisim' _ c =
let
  val c    = rewrite ( trivialConditionRules
                     @ simplificationRules) c
  val c    = rewrite (freshnessRule c) c
  val c    = cnf c
```

```
    val cnf  = cnfToList c
    val sol  = dpllSAT [] cnf
    val sols = case sol of SOME sigma ⇒ [sigma]
                         | NONE ⇒ []
    val sols = map (fn sigma ⇒ (sigma, unit)) sols
  in
    case sols of
          [] ⇒ Either.LEFT []
        | _  ⇒ Either.RIGHT sols
  end

  val solveBisim = SOME solveBisim'
```

This ends the **local open** BisimConstraint.

```
end
```

This ends the implementation of the constraint solver for the constraints generated by the bisimulation algorithm. The solution to such constraints require more advanced techniques but SMT techniques are well established field [22]. We did not show how to interface with an external SMT solver, since we intended to show the principles underlying such solutions. We think that by exposing an implementation we make a clearer connection with SMT theory and how one would go about designing models and interfacing with an external SMT solver. The real challenge is to define a model and the decision procedure for that model.

B Theory preliminaries

B.1 Fixed point equations on sets

In this thesis we used equations on sets of the following form

$$T = \mathrm{F}(T)$$

where F is some total function that given a set returns a set. We tended to use it in the manner shown below

$$T = A_1 \cup \cdots \cup A_2 \cup \{f(x_1, \ldots, x_n) : P(x_1, \ldots, x_n) \wedge x_1 \in T \wedge \cdots \wedge x_n \in T\}$$

the right hand side is the function F and A_i are some constant sets.

To be able to tell if they are solvable we need two notions.

Definition 26 (Monotonicity)**.** *If whenever we have $T \subseteq T'$ then $F(T) \subseteq F(T')$ holds.*

Definition 27 (Continuity)**.** *If $T_1 \subseteq \cdots \subseteq T_n \subseteq \cdots$ is an increasing sequence (a chain) and the if following holds*

$$\bigcup_i F(T_i) = F(\bigcup_i T_i)$$

then F is continuous. Monotonicity is implied by continuity.

Any fixed-point equation is always solvable if the function is continuous.

Theorem 28 (First recursion theorem)**.** *Each continuous function F has a least fixed-point.*

All equations used in this thesis do have least fixed-points. In fact, continuity in sets imply computability.

These results generalise to system of equations, see [26].

B.2 Transition systems

Throughout this thesis we used and developed a number of transition systems. Here we make the notion of a transition system precise. These definition can be found in [26].

Definition 29 (ts)**.** *A transition system (ts) is a structure $\langle \Gamma, \rightarrow \rangle$ where Γ is a set of elements, γ, called configurations, and $\rightarrow \subseteq \Gamma \times \Gamma$ is a binary relation called the transition relation.*

Notation $\gamma \rightarrow \gamma'$ means that there is a transition $\rightarrow$ from configuration γ to configuration γ'.

A more familiar transition system is that of automaton.

Definition 30 (lts)**.** *A labelled transition system (lts) is a structure* $\langle \Gamma, A, \rightarrow \rangle$ *where* Γ *is a set of configurations, and* A *is a set of labels (actions), and* $\rightarrow \subseteq \Gamma \times A \times \Gamma$ *is the transition relation.*

The symbolic operation semantics is described in terms of a labelled transition system, for instance $P \xrightarrow[C]{\alpha} P'$ is $\rightarrow \subseteq P \times (A \times C) \times P$.

B.3 Some notions from universal algebra

This section is intended only just as a refresher on universal algebra. This section only includes very few definitions used in universal algebra, and it only includes those definitions that we use throughout this thesis. For an account on universal algebra from computer science perspective please see [2].

Definition 31 (Signature)**.** *A signature* Σ *is a set of function symbols. Each function symbol* $f \in \Sigma$ *is associated with an integer – the arity of the function symbol. Function symbols of arity* 0 *are called constants.* $\Sigma^{(n)}$ *denotes a set of function symbols of arity* n *from the signature* Σ*.*

Definition 32 (Sigma-terms)**.** *Let* Σ *be a signature and* $\mathcal{N}$ *be a set of variables (names), such that* Σ *and* $\mathcal{N}$ *are disjoint* $\Sigma \cap \mathcal{N} = \emptyset$*. The* T *operator constructs a set* $T(\Sigma, \mathcal{N})$ *called* Σ*-terms over* $\mathcal{N}$*, and is defined inductively as follows*

- $\mathcal{N} \subseteq T(\Sigma, \mathcal{N})$ *(variables are among the terms),*
- *for all* $n \geq 0$ *and all* $f \in \Sigma^{(n)}$*, and all* $t_1, \ldots, t_n \in T(\Sigma, \mathcal{N})$ *we have* $f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{N})$

Definition 33 (Sigma-algebra)**.** *Let* Σ *be some signature. A* Σ*-algebra* $\mathcal{A}$ *consists of*

- *the underlying* carrier *set* A *(a non empty set);*
- *mappings associated with each of the function symbols* $f \in \Sigma^{(n)}$

$$f^{\mathcal{A}} : A^n \rightarrow A$$

Definition 34 (Homomorphism)**.** *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be* Σ*-algebras. A* Σ*-homomorphism* $\phi : \mathcal{A} \rightarrow \mathcal{B}$ *is a mapping between carrier sets* $\phi : A \rightarrow B$*, such that for all* $f \in \Sigma^{(n)}$ *and all* $a_i \in A$ *we have*

$$\phi(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(\phi(a_1), \ldots, \phi(a_n))$$

An endomorphism *is homomorphism* $\phi : \mathcal{A} \rightarrow \mathcal{A}$*.*

Definition 35 (Congruence)**.** *Let $\mathcal{A}$ be a Σ-algebra. An equivalence relation $\equiv$ on the carrier set A of $\mathcal{A}$ is called a congruence iff for all $f \in \Sigma^{(n)}$ and $a_1 \equiv b_1, \ldots, a_n \equiv b_n$ where $a_i, b_i \in A$ we have*

$$f^{\mathcal{A}}(a_1, \ldots, a_n) \equiv f^{\mathcal{A}}(b_1, \ldots, b_n)$$

Finally, a term algebra is Σ-algebra which is obtained in a kind of trivial way.

Definition 36 (Term-algebra)**.** *Let Σ be a signature and $\mathcal{N}$ set of names (variables). A Σ-algebra $\mathcal{T}(\Sigma, \mathcal{N})$ is an algebra with a carrier set $T(\Sigma, \mathcal{N})$ and with the following mappings for $f \in \Sigma^{(n)}$ and terms $a_i \in T(\Sigma, \mathcal{N})$.*

$$f^{\mathcal{T}(\Sigma, \mathcal{N})} : T(\Sigma, \mathcal{N})^n \to T(\Sigma, \mathcal{N}) = (a_1, \ldots, a_n) \mapsto f(a_1, \ldots, a_n)$$

That is every function symbol is interpreted as itself.

C Symbolic strong and weak bisimulations

These equivalence definitions for Psi-calculus are included here for completeness, see [18, 17].

Definition 37 (Symbolic static equivalence)**.** *Two processes P and Q are* statically equivalent *for C, written $P \simeq_C Q$, if for each $(\sigma, \Psi) \in \mathrm{sol}(C)$ we have that $\Psi \otimes \mathcal{F}(P)\sigma \simeq \Psi \otimes \mathcal{F}(Q)\sigma$.*

Definition 38 (Bisimulation constraints)**.** *The* constraints*, ranged over by C, are of the forms*

	Constraint	*The solutions are all pairs (σ, Ψ) such that*
$C, C' ::=$	C_t	$(\sigma, \Psi) \models C_t$
	$\{\!\{M = N\}\!\}$	$M\sigma = N\sigma$
	$\{\!\{a\#X\}\!\}$	$(a\#X)\sigma$ *and* $a\#dom(\sigma), \Psi$
	$C \wedge C'$	$(\sigma, \Psi) \models C$ *and* $(\sigma, \Psi) \models C'$
	$C \vee C'$	$(\sigma, \Psi) \models C$ *or* $(\sigma, \Psi) \models C'$
	$C \Rightarrow C'$	$(\sigma, \Psi) \models C$ *implies* $(\sigma, \Psi) \models C'$
	$F \leq G$	$(\sigma, \Psi) \models C$ *s.t.* $\forall \varphi.\ \Psi \otimes (F\sigma) \vdash \varphi$ *iff* $\Psi \otimes (G\sigma) \vdash \varphi$

where C_t are the transition constraints. In $\{\!\{a\#X\}\!\}$, X is any nominal data type. We call $\{\!\{a\#X\}\!\}$ freshness constraint, and $\{\!\{M = N\}\!\}$ term equality constraint.

Definition 39 ((Early) Symbolic bisimulation)**.** *A* symbolic bisimulation *$\mathcal{S}$ is a ternary relation between constraints and pairs of agents such that $\mathcal{S}(C, P, Q)$ implies all of*

1. *$P \simeq_C Q$, and*
2. *$\mathcal{S}(C, Q, P)$, and*
3. *If $P \xrightarrow[C_P]{\tau} P'$ then there exists $\widehat{C}$ such that $C \wedge C_P \Leftrightarrow \bigvee \widehat{C}$ and for all $C' \in \widehat{C}$ there exists Q' and C_Q such that*
   (a) *$Q \xrightarrow[C_Q]{\tau} Q'$, and*
   (b) *$C' \Rightarrow C_Q$, and*
   (c) *$\mathcal{S}(C', P', Q')$*
4. *If $P \xrightarrow[C_P]{\underline{y}(x)} P'$, $x\#(P, Q, C, C_P, y)$ and $y\#(P, Q, C)$ then there exists $\widehat{C}$ such that $C \wedge C_P \Leftrightarrow \bigvee \widehat{C}$ and for all $C' \in \widehat{C}$ there exists Q' and C_Q such that*
   (a) *$Q \xrightarrow[C_Q]{\underline{y}(x)} Q'$, and*

(b) $C' \Rightarrow C_Q$, *and*

(c) $\mathcal{S}(C', P', Q')$

5. *If* $P \xrightarrow[C_P]{\overline{y}\,(\nu\widetilde{a})N} P'$, $\widetilde{a}\#(P, Q, C, C_P, y)$ *and* $y\#(P, Q, C)$ *then there exists* $\widehat{C}$ *such that* $C \wedge C_P \wedge \{\!|\widetilde{a}\#P, Q|\!\} \Leftrightarrow \bigvee \widehat{C}$ *and for all* $C' \in \widehat{C}$ *there exists* Q' *and* C_Q *such that*

   (a) $Q \xrightarrow[C_Q]{\overline{y}\,(\nu\widetilde{a})N'} Q'$, *and*

   (b) $C' \Rightarrow C_Q \wedge \{\!|N = N'|\!\}$, *and*

   (c) $\mathcal{S}(C', P', Q')$

We write $P \sim_s Q$ *if* $(\mathbf{true}, P, Q) \in \mathcal{S}$ *for some symbolic bisimulation* $\mathcal{S}$*, and say that* P *is* symbolically bisimilar *to* Q*.*

Definition 40 (Symbolic static implication)**.** *A process* P statically implies *another process* Q symbolically *for* C*, written* $P \leq_C Q$*, if for each* $(\sigma, \Psi) \in \mathrm{sol}(C)$ *we have that* $P\sigma \leq_\Psi Q\sigma$*.*

Definition 41 (Weak Symbolic bisimulation)**.** *A* weak symbolic bisimulation $\mathcal{S}$ *is a ternary relation between constraints and pairs of agents such that* $\mathcal{S}(C, P, Q)$ *implies all of*

1. *there exists a set of constraints* $\widehat{C}$ *such that* $C \Leftrightarrow \bigvee \widehat{C}$ *and for all* $C' \in \widehat{C}$ *there exists* Q' *and* C_Q *such that*

   (a) $Q \underset{C_Q}{\Longrightarrow} Q'$,

   (b) $C' \Rightarrow C_Q$,

   (c) $P \leq_{C'} Q'$, *and*

   (d) $(C', P, Q') \in \mathcal{S}$

2. $\mathcal{S}(C, Q, P)$, *and*

3. *If* $P \xrightarrow[C_P]{\tau} P'$ *then there exists* $\widehat{C}$ *such that* $C \wedge C_P \Leftrightarrow \bigvee \widehat{C}$ *and for all* $C' \in \widehat{C}$ *there exists* Q' *and* C_Q *such that*

   (a) $Q \underset{C_Q}{\Longrightarrow} Q'$, *and*

   (b) $C' \Rightarrow C_Q$, *and*

   (c) $\mathcal{S}(C', P', Q')$

4. *If* $P \xrightarrow[C_P]{y(x)} P'$, $x\#(P, Q, C, C_P, y)$ *and* $y\#(P, Q, C)$ *then there exists* $\widehat{C}$ *such that* $C \wedge C_P \Leftrightarrow \bigvee \widehat{C}$ *and for all* $C' \in \widehat{C}$ *there exists* Q' *and* C_Q *such that*

(a) $Q \xRightarrow[C_Q]{y(x)} Q'$, *and*

(b) $C' \Rightarrow C_Q$, *and*

(c) $\mathcal{S}(C', P', Q')$

5. *If* $P \xrightarrow[C_P]{\overline{y}\,(\nu\widetilde{a})N} P'$, $\widetilde{a}\#(P, Q, C, C_P, y)$ *and* $y\#(P, Q, C)$ *then there exists* $\widehat{C}$ *such that* $C \wedge C_P \wedge \{\!|\widetilde{a}\#P, Q|\!\} \Leftrightarrow \bigvee \widehat{C}$ *and for all* $C' \in \widehat{C}$ *there exists* Q' *and* C_Q *such that*

   (a) $Q \xRightarrow[C_Q]{\overline{y}\,(\nu\widetilde{a})N'} Q'$, *and*

   (b) $C' \Rightarrow C_Q \wedge \{\!|N = N'|\!\}$, *and*

   (c) $\mathcal{S}(C', P', Q')$

We write $P \dot{\approx}^s_C Q$ *if* $(C, P, Q) \in \mathcal{S}$ *for some symbolic bisimulation* $\mathcal{S}$. *We write* $P \dot{\approx}^s Q$ *for* $P \dot{\approx}^s_{\mathbf{true}} Q$, *and say that* P *is* symbolically bisimilar *to* Q.

C.1 Algorithm for computing weak bisimulation

In this section we reproduce from [18] the algorithm for computing weak bisimulations C. The algorithm is presented in figures 5, 6, 7. This algorithm can be transformed into computing strong bisimulations [18].

The algorithm requires a more complex constraint language than transitional constraints definition 21 on page 18.

In appendix A we provide an example implemenation of a bisimulation constraint solver for Pi-calculus instance.

```
/* bisim(P,Q)
   P and Q are agents. Returns a pair (C,T) where C is a constraint such that
   P ≈̇_{C smp} Q and T is a table describing a witnessing bisimulation. */

bisim(P, Q) = close(P, Q, true, ∅)

/* close(P,Q,C,W)
   P and Q are agents, C are the constraints seen so far, and W is a set of pairs of
   agents that have already been visited by the algorithm. Returns a pair (C',T),
   where C' is a constraint necessary for P and Q to be bisimilar, and T is a table
   describing a partial witnessing bisimulation. */

close(P, Q, C, W)
   if (P, Q) ∈ W then
      (true, ∅)
   else let (C_stimp, T_stimp) = match-stimp(P, Q, C, W)
            (C'_stimp, T'_stimp) = match-stimp(Q, P, C, W)
            (C_τ, T_τ) = match-τ(P, Q, C, W)
            (C'_τ, T'_τ) = match-τ(Q, P, C, W)
            (C_out, T_out) = match-out(P, Q, C, W)
            (C'_out, T'_out) = match-out(Q, P, C, W)
            (C_in, T_in) = match-in(P, Q, C, W)
            (C'_in, T'_in) = match-in(Q, P, C, W)
        in (C_stimp ∧ C'_stimp ∧ C_τ ∧ C'_τ ∧ C_out ∧ C'_out ∧ C_in ∧ C'_in,
            T_stimp ⊔ T'_stimp ⊔ T_τ ⊔ T'_τ ⊔ T_out ⊔ T'_out ⊔ T_in ⊔ T'_in ⊔
            {(P, Q) ↦ {C ∧ C_stimp ∧ C_τ ∧ C_out ∧ C_in}} ⊔
            {(Q, P) ↦ {C ∧ C'_stimp ∧ C'_τ ∧ C'_out ∧ C'_in}})
```

Figure 5: `bisim` and `close` functions

```
/* match-stimp(P, Q, C, W)
   The parameters are as in close(P, Q, C, W). Returns a pair (C', T) where C'
   is a constraint that is necessary for P to statically imply Q, and T is a table
   describing a partial witnessing bisimulation. */

match-stimp(P, Q, C, W)
   let Qtr = {(C_Qi, Q_i) : Q ⟹_{C_Qi} Q_i}
       (C̃, T̃) = map λ(C_Qi, Q_i).
                    let (C_i, T_i) =
                       close(P, Q_i, C ∧ C_Qi, W ∪ {(P, Q)})
                    in (C_Qi ∧ C_i ∧ (C_i ∧ C_Qi ⇒ F(P) ≤ F(Q_i)), T_i)
                Qtr
   in (true ⇒ ⋁C̃, ⨆T̃)

/* match-τ(P, Q, C, W)
   The parameters are as in close(P, Q, C, W). Returns a pair (C', T) where C'
   is a constraint that is necessary for Q to simulate P for τ-actions, and T is a
   table describing a partial witnessing bisimulation. */

match-τ(P, Q, C, W)
    let Ptr = {(C_Pi, P_i) : P ⟶_{C_Pi} P_i}
        Qtr = {(C_Qj, Q_j) : Q ⟹_{C_Qj} Q_j}
        (C̃, T̃) = map (λ(C_Pi, P_i).
                    let (C̃_i, T̃_i) = map (λ(C_Qj, Q_j) .
                        let (C_ij, T_ij) =
                           close(P_i, Q_j, C ∧ C_Pi ∧ C_Qj, W ∪ {(P, Q)})
                        in (C_Qj ∧ C_ij, T_ij)) Qtr
                    in (C_Pi ⇒ ⋁C̃_i, ⨆T̃_i)) Ptr
    in (⋀C̃, ⨆T̃)
```

Figure 6: `match-stimp` and `match-`τ functions

/* match-out(P, Q, C, W)
The parameters are as in close(P, Q, C, W). Returns a pair (C', T) where C' is a constraint that is necessary for Q to simulate P for outputs, and T is a table describing a partial witnessing bisimulation. */

match-out$(P,\ Q,\ C,\ W)$
  let Ptr = $\{(\overline{y}\,(\nu\widetilde{a})N,\ C_{Pi},\ P_i)\ :\ P \xrightarrow[C_{Pi}]{\overline{y}\,(\nu\widetilde{a})N} P_i$
    $\wedge\ y =$ newName$(P, Q, C, \mathcal{X})\ \wedge\ \widetilde{a}\#P, Q, C, C_{Pi}, y\}$
    $(\widetilde{C},\ \widetilde{T})$ = map $\lambda(\overline{y}\,(\nu\widetilde{a})N,\ C_{Pi},\ P_i)$.
    let Qtr = $\{(\overline{z}\,(\nu\widetilde{c})N',\ C_{Qj},\ Q_j)\ :\ Q \xRightarrow[C_{Qj}]{\overline{z}\,(\nu\widetilde{c})N'} Q_j$
      $\wedge\ y = z\ \wedge\ \widetilde{a} = \widetilde{c}\}$
      $(\widetilde{C_i},\ \widetilde{T_i})$ = map $\lambda(\overline{z}\,(\nu\widetilde{c})N',\ C_{Qj},\ Q_j)$
      let $(C_{ij},\ T_{ij})$ =
        close$(P_i, Q_j, C \wedge C_{Pi} \wedge C_{Qj} \wedge \{\!|N = N'|\!\} \wedge \{\!|\widetilde{a}\#P, Q|\!\},\ W \cup \{(P, Q)\})$
      in $(C_{Qj} \wedge \{\!|N = N'|\!\} \wedge C_{ij},\ T_{ij})\ Qtr$
    in $(C_{Pi} \wedge \{\!|\widetilde{a}\#P, Q|\!\} \Rightarrow \bigvee \widetilde{C_i},\ \bigsqcup \widetilde{T_i})\ Ptr$
  in $(\bigwedge \widetilde{C},\ \bigsqcup \widetilde{T})$

/* match-in(P, Q, C, W)
The parameters are as in close(P, Q, C, W). Returns a pair (C', T) where C' is a constraint that is necessary for Q to simulate P for inputs, and T is a table describing a witnessing bisimulation. */

match-in$(P,\ Q,\ C,\ W)$
  let Ptr = $\{(\underline{y}(x),\ C_{Pi},\ P_i)\ :\ P \xrightarrow[C_{Pi}]{\underline{y}(x)} P_i$
    $\wedge\ y =$ newName$(P, Q, C, \mathcal{X})\ \wedge\ x\#P, Q, C, C_{Pi}, y\}$
    $(\widetilde{C},\ \widetilde{T})$ = map $\lambda(\underline{y}(x),\ C_{Pi},\ P_i)$.
      let Qtr = $\{(\underline{z}(x'),\ C_{Qj},\ Q_j)\ :\ Q \xRightarrow[C_{Qj}]{\underline{z}(x')} Q_j$
        $\wedge\ y = z\ \wedge\ x = x'\}$
        $(\widetilde{C_i},\ \widetilde{T_i})$ = map $\lambda(\underline{z}(x'),\ C_{Qj},\ Q_j)$.
          let (C_{ij}, T_{ij}) =
            close$(P_i, Q_j, C \wedge C_{Pi} \wedge C_{Qj}, W \cup \{(P, Q)\})$
          in $(C_{Qj} \wedge C_{ij},\ T_{ij})\ Qtr$
      in $(C_{Pi} \Rightarrow \bigvee \widetilde{C_i},\ \bigsqcup \widetilde{T_i})\ Ptr$
  in $(\bigwedge \widetilde{C},\ \bigsqcup \widetilde{T})$

Figure 7: match-out and match-in functions

D Grammar

D.1 Command grammar

⟨*script*⟩ ::= ⟨*clause*⟩*

⟨*clause*⟩ ::= ⟨*term*⟩ ⟨*clause-args*⟩ '`<=`' ⟨*agent*⟩ ⟨*tr*⟩

⟨*clause-args*⟩ ::= '`<`' ⟨*name-list*⟩ '`>`' | ε

⟨*tr*⟩ ::= '`;`' | ⟨*eof*⟩ | ε

⟨*name-list*⟩ ::= ε | ⟨*name*⟩ ('`,`' ⟨*name*⟩)*

D.2 Agent grammar

⟨*agent*⟩ ::= ⟨*parallel*⟩ | ⟨*restriction*⟩ | ⟨*replication*⟩ | ⟨*parens*⟩ | ⟨*prefix*⟩
| ⟨*case*⟩ | ⟨*assertion*⟩ | ⟨*nil*⟩ | ⟨*invocation*⟩

⟨*parallel*⟩ ::= ⟨*agent*⟩ '`|`' ⟨*agent*⟩

⟨*restriction*⟩ ::= '`(`' '`new`' ⟨*name*⟩ ('`,`' ⟨*name*⟩)* '`)`' ⟨*agent*⟩

⟨*replication*⟩ ::= '`!`' ⟨*agent*⟩

⟨*parens*⟩ ::= '`(`' ⟨*agent*⟩ '`)`'

⟨*prefix*⟩ ::= ⟨*prefix'*⟩ '`.`' ⟨*agent*⟩
| ⟨*prefix'*⟩

⟨*prefix'*⟩ ::= ⟨*input*⟩ | ⟨*output*⟩

⟨*input*⟩ ::= ⟨*term*⟩ '`(`' '`\`' ⟨*name*⟩ ('`,`' ⟨*name*⟩)* '`)`' ⟨*term*⟩
| ⟨*term*⟩ '`(`' ⟨*name*⟩ '`)`'

⟨*input"*⟩ ::= '`,`' ⟨*input*⟩

⟨*output*⟩ ::= '`'`' ⟨*term*⟩ '`<`' ⟨*term*⟩ '`>`'

⟨*case*⟩ ::= '`case`' ⟨*case-clause*⟩ ('`[]`' ⟨*case-clause*⟩)*

⟨*case-clause*⟩ ::= ⟨*cond*⟩ '`:`' ⟨*agent*⟩

⟨*assertion*⟩ ::= '`(|`' ⟨*assr*⟩ '`|)`'

⟨*nil*⟩ ::= '`0`'

⟨*invocation*⟩ ::= ⟨*identifier*⟩ '`<`' ⟨*term-sequence*⟩ '`>`'

⟨*term-sequence*⟩ ::= ε | ⟨*term*⟩ ('`,`' ⟨*term*⟩)*

⟨*name*⟩ ::= ⟨*literal*⟩

⟨*term*⟩ ::= ⟨*literal*⟩

⟨*cond*⟩ ::= ⟨*literal*⟩

$\langle assr \rangle$ ::= $\langle literal \rangle$

$\langle literal \rangle$ ::= ‘"’ . . . ‘"’
| ‘’’ . . . ‘’’
| ‘{*’ . . . ‘*}’
| $\langle identifier \rangle$

$\langle identifier \rangle$ ::= $\langle id \rangle^{+}$ ‘’’*

$\langle id \rangle$ ::= $\langle alpha\text{-}numeric \rangle$ | ‘_’

$\langle alpha\text{-}numeric \rangle$ ::= [a-zA-Z0-9]