A Parametric Tool for Applied Process Calculi

Johannes Borgström   Ramūnas Gutkovas   Ioana Rodhe   Björn Victor
Department of Information Technology
Uppsala University
Email: {johannes.borgstrom, ramunas.gutkovas, ioana.rodhe, bjorn.victor}@it.uu.se

***Abstract*—**
High-level formalisms for concurrency are often defined as extensions of the the pi-calculus; a growing number is geared towards particular applications or computational paradigms. Psi-calculi is a parametric framework that can accommodate a wide spectrum of such calculi. It allows the definition of process calculi that extend the pi-calculus with arbitrary data, logic and logical assertions. All such calculi inherit machine-checked proofs of the meta-theory such as compositionality and bisimulation congruence.

We present a generic tool for implementing instances of psi-calculi, and for analysing psi-calculus processes using symbolic execution and bisimulation techniques for both unicast and wireless broadcast communication. We illustrate the tool by examples from pi-calculus and the area of wireless sensor networks.

***Keywords*-Process Calculi; Symbolic semantics; Wireless Sensor Networks**

I. Introduction

The development of concurrent systems is greatly helped by the use of precise and formal models of the system. There are many different formalisms for concurrent systems, often in specialized versions for particular application areas. Tool support for the formalism is necessary for constructing and reasoning about models of non-trivial systems.

Psi-calculi [2], based on the pi-calculus [16], is a parametric semantic framework, where the data language and logic can be tailored for each application. The framework provides a variety of features, such as lexically scoped local names for resources, both unicast and broadcast communication [5], first-order and higher-order communication [17] etc. Psi-calculi can capture the same phenomena as other proposed extensions of the pi-calculus such as the applied pi-calculus, the spi-calculus, the fusion calculus, the concurrent constraint pi-calculus, and calculi with polyadic communication channels or pattern matching. A major advantage is that all meta-theoretical results, including algebraic laws and congruence properties of bisimilarity, apply to any valid instantiation of the framework. Additionally, most of these results have been proved with certainty, using the Nominal Isabelle theorem prover. These features of psi-calculi save a lot of effort for anyone using it — psi-calculi is a reusable framework.

This paper describes the Psi-Calculi Workbench (Pwb) which is a generic tool for implementing psi-calculus instances, and for analysing processes in the resulting instances. Like psi-calculi, it is parametric: it is based on core functionality for bisimulation equivalence checking and symbolic simulation (or execution) of processes, but modules implementing the parameters for the particular instance of psi-calculi are supplied by the user. The implementation is facilitated by functionality in Pwb. Pwb thus has two types of users: the user analysing systems in an existing instance, and the instance implementor.

We illustrate the tool firstly by implementing the pi-calculus and showing bisimulation equivalence checking, and secondly in the more complex example of a protocol from the area of wireless sensor networks, incorporating specialized data structures, logics, and both unicast and broadcast communication.

II. Psi-calculi

This section is a brief recapitulation of psi-calculi, that mainly serves to introduce a few notions used in the examples below. For a more extensive treatment including motivations and examples see [2], [5], [10], [11].

We assume a countably infinite set of atomic *names* $\mathcal{N}$ ranged over by $a, b, \ldots, z$. A *nominal set* [6], [18] is a set equipped with a formal notion of what it means for a name a to occur (free) in an element X of the set, written $a \in \mathrm{n}(X)$. We write $\tilde{X}$ for a sequence $X_1, \ldots, X_n$ for some n. A psi calculus is parametric in data M and a logic with an entailment relation $\Psi \vdash \varphi$.

Definition 1 (Psi-calculus parameters)**.** *A psi-calculus is defined by three data types: the (data) terms* $\mathbf{T}$*, ranged over by* M, N*, the conditions* $\mathbf{C}$*, ranged over by* φ*, the assertions* $\mathbf{A}$*, ranged over by* Ψ*, and six equivariant operators:*

$\dot{\leftrightarrow}$	:	$\mathbf{T} \times \mathbf{T} \to \mathbf{C}$	Unicast Channel Equivalence
$\dot{\prec}$	:	$\mathbf{T} \times \mathbf{T} \to \mathbf{C}$	Broadcast Output Connectivity
$\dot{\succ}$	:	$\mathbf{T} \times \mathbf{T} \to \mathbf{C}$	Broadcast Input Connectivity
$\otimes$	:	$\mathbf{A} \times \mathbf{A} \to \mathbf{A}$	Assertion Composition
$\mathbf{1}$	:	$\mathbf{A}$	Unit Assertion
$\vdash$	$\subseteq$	$\mathbf{A} \times \mathbf{C}$	Logical Entailment

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

We impose certain requisites on these operators. In brief, channel equivalence must be symmetric and transitive, $\otimes$

must be compositional, and the assertions with $(\otimes, \mathbf{1})$ form an abelian monoid.

Definition 2 (Psi-calculus agents)**.** *Given psi-calculus parameters as in Definition 1, the psi-calculus* agents*, ranged over by $P, Q, \ldots$, are of the following forms.*

$\pi \,.\, P$	Prefix
case $\varphi_1 : P_1 \,[\!]\, \cdots \,[\!]\, \varphi_n : P_n$	Case
$(\nu a)P$	Restriction
$P \mid Q$	Parallel
$!P$	Replication
$(\!\mid \Psi \mid\!)$	Assertion
$\mathtt{A}\langle \tilde{M} \rangle$	Invocation

where the prefixes π are given by

$$\pi ::= \overline{M}\, \tilde{N} \mid \underline{M}(\tilde{x}) \mid \overline{M}!\tilde{N} \mid \underline{M}?(\tilde{x})$$

denoting (polyadic) unicast output and input, and broadcast output and input, respectively. Restriction binds a in P and input prefixes bind $\tilde{x}$ in the suffix. We identify alpha-equivalent agents.

Case agents are sometimes abbreviated as **if** φ **then** P when $n = 1$, or as **0** or nothing when $n = 0$. To simplify the implementation of PWB and improve usability, we use polyadic rather than monadic communication (in contrast to [11]) and distinguish between unicast and broadcast prefixes (in contrast to [5]).

The *actions* ranged over by α are of the following kinds: Input $\underline{M}\, \tilde{N}$ denotes the reception of data $\tilde{N}$ on channel M. Output $\overline{M}\, (\nu\tilde{a})\tilde{N}$ represents an action sending $\tilde{N}$ along M and opening the scopes of the names $\tilde{a}$ (similarly to the polyadic pi-calculus). Silent actions τ result from internal communication. We also treat synchronous non-blocking lossy (wireless) broadcast communication, with corresponding input ($\underline{M}?\, \tilde{N}$) and output ($\overline{M}!\, (\nu\tilde{a})\tilde{N}$) actions.

A *concrete transition* is written $\Psi \,\triangleright\, P \xrightarrow{\alpha} P'$, meaning that in environment Ψ agent P can do an α to become P'.

III. Symbolic Semantics

When a process performs an input action $\underline{M}\, \tilde{N}$, there are related transitions for all other possible received values $\tilde{N}$. To avoid this infinite branching, we use a symbolic semantics [11] extended with rules for broadcast communication. *Symbolic transitions* are of the form $P \xrightarrow[C]{\alpha} P'$ where C is a constraint (similar to those for the pi-calculus [4], [12]). In psi-calculi the context can enable a transition if it contains an assertion, such as $(\!\mid x = 3 \mid\!)$. Therefore a solution of a constraint contains both a substitution (representing earlier inputs) and an assertion (representing the parallel context).

Definition 3. *A* solution *is a pair (σ, Ψ) where σ is a substitution of terms for names, and Ψ is an assertion. The* transition constraints*, ranged over by C and their solutions, $\mathrm{sol}(C)$ are defined by:*

	Constraint	*Solutions*
$C ::=$	$\mathbf{true}$	$\{(\sigma, \Psi)\}$
	$\mathbf{false}$	$\emptyset$
	$(\nu\tilde{a})\{\!\mid \Psi \vdash \varphi \mid\!\}$	$\{(\sigma, \Psi') \;:\; \exists\tilde{b}.\tilde{b}\#\sigma, \Psi', \Psi, \varphi \wedge ((\tilde{a}\;\tilde{b}) \cdot \Psi)\sigma \otimes \Psi' \vdash ((\tilde{a}\;\tilde{b}) \cdot \varphi)\sigma\}$
	$C \wedge C'$	$\mathrm{sol}(C) \cap \mathrm{sol}(C')$

The symbolic semantics are fully abstract w.r.t. the concrete semantics: $P \xrightarrow[C]{\alpha} P'$ iff $\Psi \,\triangleright\, P\sigma \xrightarrow{\alpha\sigma} P'\sigma$ where $(\Psi, \sigma) \in \mathrm{sol}(C)$.

IV. Implementation

The Psi-Calculi Workbench (PWB) is implemented in the Standard ML programming language and compiles under the Poly/ML compiler [19] version 5.4 and later. PWB is open source and freely available from [8].

PWB is a modular implementation of psi-calculi, and can be viewed both as a modelling tool and as a library for building tools for particular instances of psi-calculi. Used as a modelling tool, the user interacts with a command interpreter that provides commands for process definitions (manually or from files), manipulation of the process environment, stepping through symbolic (strong and weak) transitions of a process, and symbolic bisimilarity checking (strong and weak).

A. Psi-Calculus Instantiation

Section PWB implements a number of helper libraries for the instance implementor. We show the architecture of PWB in Figure 1. In this figure, the dependency relation goes from right to left: each component may depend only on components that are above it or to its left. All components build on the supporting library that provides the basic data structures and core algorithms for psi-calculi. The instance implementor provides definitions for the parameters of an instance, constraint solvers, and parsing and pretty-printing code. These user-implemented components are then called by the different algorithms implemented by the tool and by the command interpreter. Not all components are required to be implemented: for instance, the bisimulation constraint solver is only needed for bisimularity checking.

The parameters of an instance (Definition 1) are implemented as types name, term, condition and assertion corresponding to $\mathcal{N}$, $\mathbf{T}$, $\mathbf{C}$ and $\mathbf{A}$ respectively, and user-defined functions of the following types for the operators.

```
val chaneq    : term * term -> condition
val brReceive  : term * term -> condition
val brTransmit : term * term -> condition
val compose : assertion * assertion -> assertion
val unit     : assertion
val entails : assertion * condition -> bool
```

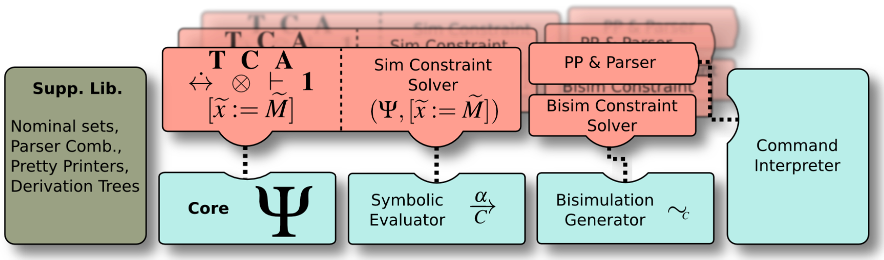
Supp. Lib.
Nominal sets, Parser Comb., Pretty Printers, Derivation Trees
T C A
$\dot{\leftrightarrow}$ $\otimes$ $\vdash$ $\mathbf{1}$
$[\widetilde{x} := \widetilde{M}]$
Sim Constraint Solver
$(\Psi, [\widetilde{x} := \widetilde{M}])$
PP & Parser
Bisim Constraint Solver
Command Interpreter
Core Ψ
Symbolic Evaluator $\xrightarrow[C]{\alpha}$
Bisimulation Generator $\sim_c$

Figure 1. Psi-Calculi Workbench Architecture

The parser for process terms makes calls to the user-specified parser for each data type; terms, assertions and conditions that are not representable as alphanumeric strings need to be quoted, as in "data(n)"(x).0. The agent syntax is the same as given in Definition 2, except for these changes: (νx) is spelled as (new x), the output prefix $\overline{M}\,\tilde{N}$ is written as 'M<N1,...,Nk>, and the input prefix $\underline{M}(\tilde{x})$ as M(x1 ,..., xk).

B. Symbolic Execution

Symbolic execution of processes is a useful tool to explore the properties of a process, or indeed the model itself. Here values input by the process are represented by variables, and constraints are collected along the derivation of a transition. In this way, it is easy to see under which conditions transitions are possible, deferring instantiation of variables as long as possible. PWB provides symbolic execution using the symbolic semantics described in Section III. Both strong and weak (ignoring τ-transitions) semantics are available.

In order to concretize the conditions under which a transition may take place, the instance implementor may provide a constraint solver for the transition constraints. Since the connectives of Definition 3 are all positive, a simple unification-based solver often suffices. The solver should return either a list of combinations of unsatisfiable constraints, or a list of solutions each consisting of a substitution and an assertion.

```
val solve : constraint ->
  (condition list list ,
   ((name*term) list * assertion) list) either
```

C. Symbolic Bisimulation

PWB can also be used to check bisimulation equivalence of processes. To this end, we implement the bisimulation algorithm of [11] (with some minor corrections and optimizations). This algorithm takes two processes and yields a constraint in an extended constraint language; the two processes are bisimilar under all solutions to the constraint. The extended language for constraints additionally includes conjunction, disjunction and implication as well as constraints for term equality $\{\!|M = N|\!\}$, freshness $\{\!|a\#X|\!\}$, and static implication between frames $\{\!|(\nu\tilde{a})\Psi_1 \leq (\nu\tilde{c})\Psi_2|\!\}$. In order to simplify the development of a constraint solver for this richer language, PWB contains an SMT solver library with suitable helper functions. Unless the assertion language is trivial ($\mathbf{A} = \{\mathbf{1}\}$), most of the additional effort in extending a solver for transition constraints to one for bisimulation constraints lies in properly treating static implication constraints.

V. Examples

We demonstrate the use of PWB on two examples. First, we show how to encode the pi-calculus as a psi-calculus instance, and how to implement this instance in PWB. Second, we define a calculus for studying data collection in wireless sensor networks, and define and study a simple algorithm. This example highlights the flexibility of the parameterization of PWB.

A. The pi-calculus

As an example of a simple psi-calculus instance, we present our implementation of the pi-calculus in PWB. In the Pi instance the terms are names; conditions are equalities between names and the 'true' condition $\top$. We only have the unit assertion $*$.

$$\begin{array}{rcl} \mathbf{T} & \triangleq & \mathcal{N} \\ \mathbf{C} & \triangleq & \{a = b : a, b \in \mathcal{N}\} \cup \{\top\} \\ \mathbf{A} & \triangleq & \{*\} \end{array}$$

Two names are channel equivalent iff they are equal; the 'true' condition always holds.

$$\begin{array}{rcl} a \dot{\leftrightarrow} b & \triangleq & a = b \\ \mathbf{1} & \triangleq & * \\ \mathbf{1} \otimes \mathbf{1} & \triangleq & \mathbf{1} \end{array} \qquad \begin{array}{rcl} \mathbf{1} & \vdash & \top \\ \mathbf{1} & \vdash & a = a \end{array}$$

1) Implementation: The implementation of psi-calculus parameters is straightforward. We instantiate the types of the signature given in Section IV-A with

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

and implement the operators

```
fun chaneq   (a,b)                 = Eq (a,b)
val unit                           = Unit
fun compose (_,_)                  = Unit
fun entails (Unit,Eq (a, b)) = (a = b)
  | entails (Unit,T)               = true
```

The broadcast parameters br... simply throw an exception, to avoid using broadcast prefixes in agents by mistake.

2) Sample Transition: For example, we check whether the agent P(a,b) = a(x) | 'b<b> has a τ-transition. Upon running sstep P(a,b), the symbolic simulator of PWB yields three transitions: one output, one input, and one τ-transition (shown below).

```
--|tau|-->
   Source:        (a(x)) | ('b<b>)
   Constraint: {| "b = a" |}
   Solution:      ([a := b], 1)
   Derivative: (0) | (0)
```

The transition constraint is that names a and b are equal; one solution of the constraint is the substitution of b for a.

3) Bisimulation Checking: To check if the agent P(c,b) = c(x) | b(x) is bisimilar to Q(a,b) = **case** T: a(x).b(x) [] T: b(x).a(x) (which represents a nondeterministic choice between a(x).b(x) and b(x).a(x)), we run symbolic bisimulation checker by giving the command P(c,b) ~ Q(a,b). The bisimulation algorithm computes a necessary and sufficient constraint for bisimilarity, which has a solution ([a := c], 1). Applying the substitution $[a := c]$ to P(c,b) and Q(a,b) yields bisimilar agents.

4) Constraint Solvers: The transition constraint solver attempts to find a satisfying substitution to a transition constraint. It performs unification, implemented as a transition system (given below). The nodes in the transition system are either a pair of a conjunction of atomic constraints (ranged over by C) and a substitution σ, or the failed state $\square$.

$$(\nu\tilde{a})\{\!|a = a|\!\} \wedge C, \sigma \to C, \sigma \qquad (\nu\tilde{a})\{\!|\top|\!\} \wedge C, \sigma \to C, \sigma$$
$$(\nu\tilde{a})\{\!|a = b|\!\} \wedge C, \sigma \to \square \text{ if } a \neq b \wedge (a \in \tilde{a} \vee b \in \tilde{a})$$
$$(\nu\tilde{a})\{\!|a = b|\!\} \wedge C, \sigma \to C[b := a], \sigma[b := a] \text{ otherwise}$$

The bisimulation constraint solver treats a richer constraint language, including disjunction and implication, and is therefore more complex. First, it simplifies and pre-solves the constraint by term rewriting on the constraint, supported by built-in helper functions. Second, PWB transforms the constraint into conjunctive normal form (CNF). Third, the CNF is passed to the built-in SMT solver of PWB, for which the instance implementor needs to provide a suitable theory.

B. Data collection in a Wireless Sensor Network

A wireless sensor network consists of numerous sensor nodes that sense environmental data. A special node, called the sink, is used to collect data from the network. Collection often uses multi-hop communication, building a routing tree rooted at the sink [15]. As wireless communication is unreliable, different trees may be built in each protocol run.

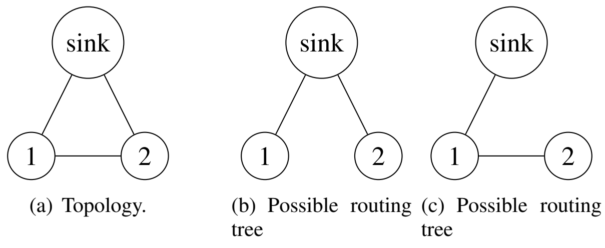

Figure 2. A simple topology with a sink and two sensor nodes where (a) shows the connectivity and (b)-(c) show some possible routing trees.

We present a simple algorithm to build a routing tree: the sink starts the tree building by broadcasting a special init message $\langle Sink, q\rangle$ containing its id $Sink$ and some query q. When a node n_i first receives an init message, it sets its parent $parent_{n_i}$ to the sender of the message, and broadcasts a new init message $\langle n_i, q\rangle$ containing its own id to continue building the next level of the tree. The response datum to the query is sent by each node to its parent. Moreover, each node forwards such response messages to its parent, ensuring that they eventually reach the sink.

1) Psi-calculus instance and protocol model: We present a model for the tree building and data collection protocol in a custom Psi-calculus instance. For simplicity, we abstract from the query. We assume a static topology $\mathsf{Top} \in \mathcal{P}(\mathbb{N} \times \mathbb{N})$; the topology in Figure 2(a) would be represented by $\mathsf{Top} = \{(0,1),(1,0),(0,2),(2,0),(1,2),(2,1)\}$ where the sink has id 0. We then define

$$\begin{array}{lll}
\mathbf{T} & \triangleq & \{\mathsf{init}(M), \mathsf{data}(M) : M \in \mathbf{T}\} \cup \mathbb{N} \cup \mathcal{N} \\
\mathbf{C} & \triangleq & \{K \dot{\succ} M, M \dot{\prec} K, M \dot{\leftrightarrow} N : M, N, K \in \mathbf{T}\} \\
\mathbf{A} & \triangleq & \{\mathsf{Top}\} \\
\mathbf{1} & \triangleq & \mathsf{Top}.
\end{array}$$

Channels are of two kinds: broadcast channels $\mathsf{init}(M)$ and unicast channels $\mathsf{data}(M)$. We define $\vdash$ as follows.

$$\begin{array}{lll}
\Psi & \vdash & \mathsf{data}(M) \dot{\leftrightarrow} \mathsf{data}(N) \text{ iff } M = N \in \mathcal{N} \\
\Psi & \vdash & \mathsf{init}(M) \dot{\prec} \mathsf{init}(N) \text{ iff } M = N \in \mathbb{N} \\
\Psi & \vdash & \mathsf{init}(M) \dot{\succ} \mathsf{init}(N) \text{ iff } (M, N) \in \Psi
\end{array}$$

Here broadcast channels contain the node id of the sender (second line above), and can be received by any node connected to it (third line). The behaviour of the sink and the other nodes is given by the processes below.

```
Sink(nodeId, bsChan) <=
      '"init(nodeId)"!<bsChan> .
      ! "data(bsChan)"(x) ;

Node(nodeId, nodeChan, datum) <=
      "init(nodeId)"?(pChan) .
      '"init(nodeId)"!<nodeChan> .
      '"data(pChan)"<datum> .
      NodeForwardData<nodeChan, pChan> ;

NodeForwardData(nodeChan, pChan) <=
      ! "data(nodeChan)"(x). '"data(pChan)"<x> ;
```

2) Sample Transition: To illustrate the use of the tool, we consider a small system with a sink and two sensor nodes. Each node has a unique channel that is used for response messages.

```
System3(d1,d2) <=
     (new chanS) Sink<0,chanS>        |
     (new chan1)  Node<1, chan1, d1>  |
     (new chan2)  Node<2, chan2, d2>
```

We will show a possible transition sequence in PWB, using the topology shown in Figure 2(a). Below, we regard the system as closed and disregard reception from the environment. We thus only consider transitions labelled with broadcast output and unicast communication actions.

The following initial transition is obtained by executing the symbolic simulator of PWB on System3<d1,d2> and corresponds to the routing tree shown in Figure 2(b). It is one of seven possible initial transitions produced by PWB, of which three represent broadcast reception from the environment, and the other three situations where not all nodes receive the broadcast message. The transition label gna!(\bsChan)bsChan, represents the channel with a fresh name gna. The generated constraint requires $\mathsf{init}(0) \mathrel{\dot{\prec}} gna$, $gna \mathrel{\dot{\succ}} \mathsf{init}(1)$ and $gna \mathrel{\dot{\succ}} \mathsf{init}(2)$, or "node 0 is output connected to some channel gna which is input connected to nodes 1 and 2". The constraint solver finds a solution to the constraint, which substitutes $\mathsf{init}(0)$ for gna.

```
--|gna!(\bsChan)bsChan|-->
Source:
  System3<d1, d2>
Constraint:
  (new chan1, chan2, chanS){| "init(0)<gna" |} /\
  (new chanS, chan2, chan1){| "gna>init(1)" |} /\
  (new chanS, chan1, chan2){| "gna>init(2)" |}
Solution:
  ([gna := "init(0)"], 1)
Derivative:
  (!("data(chanS)"(x))) |
    (((new chan1)(
      '"init(1)"!<chan1>.
        '"data(chanS)"<d1>.
          NodeForwardData<chan1, chanS>
    )) |
      ((new chan2)(
        '"init(2)"!<chan2>.
          '"data(chanS)"<d2>.
            NodeForwardData<chan2, chanS>
      )))
```

In the derivative the Sink successfully communicated its unicast channel chanS to both nodes.

From this point the system can evolve in two symmetrical ways: either of the nodes broadcasts an init message, but since no node in the (closed) system is listening on a broadcast channel, the message is not received. The following transition is for node 1.

```
--|gna!(\chan1)chan1|-->
Source:
    The same as the above derivative
Constraint:
  (new chan2, chan1){| "init(1)<gna" |}
Solution:
  ([gna := "init(1)"], 1)
Derivative:
  (!("data(chanS)"(x))) |
    (('"data(chanS)"<d1>.
      NodeForwardData<chan1, chanS>) |
     ((new chan2)(
       '"init(2)"!<chan2>.
         '"data(chanS)"<d2>.
           NodeForwardData<chan2, chanS>
     )))
```

The system is now in the state where node 1 can send data to the sink. By following the analogous transition for node 2, we get the system where both nodes are ready to communicate the data.

```
--|gna!(\chan2)chan2|-->
Source:
    The same as the above derivative
Constraint:
  (new chan2){| "init(2)<gna" |}
Solution:
  ([gna := "init(2)"], 1)
Derivative:
  (!("data(chanS)"(x))) |
    (('"data(chanS)"<d1>.
      NodeForwardData<chan1, chanS>) |
     ('"data(chanS)"<d2>.
       NodeForwardData<chan2, chanS>))
```

3) Implementation: We define the terms, condition, assertions and unit in the tool as:

```
datatype term = Init of term | Data of term
              | Name of name | Int of int
datatype condition = OutputConn of term * term
              | InputConn of term * term
              | ChEq of term * term
datatype assertion = Top
val unit  = Top
```

And the logical entailment relation as:

```
fun entails (psi, OutputConn (m, n)) = m = n
  | entails (psi, InputConn (m, n)) =
      Lst.member (m,n) top orelse Lst.member (n,m) top
  | entails (_  , ChEq (Name a, Name b)) = a = b
  | entails (_  , ChEq (_,_)) = false
```

4) Constraint Solver for Symbolic Transitions: Transition constraints are conjunctions of conditions. The constraints are solved in two phases, corresponding to the unicast connectivity constraints and the broadcast connectivity constraints, respectively. To simplify the solver, we treat all free names in the processes as distinct (cf. distinctions [16]). For unicast constraints, we thus fail if the constraint is not satisfied.

$$
\begin{array}{l}
(\nu\tilde{a})\{\!|data(a) \mathrel{\dot{\leftrightarrow}} data(b)|\!\} \wedge C \ \rightarrowtail\ C \text{ if } a = b \\
(\nu\tilde{a})\{\!|data(a) \mathrel{\dot{\leftrightarrow}} data(b)|\!\} \wedge C \ \rightarrowtail\ \Box \text{ if } a \neq b
\end{array}
$$

During the second phase, the constraint solver checks for broadcast connectivity in the given topology. Let O be the output constraints $\{\!|init(n) \mathrel{\dot{\prec}} a|\!\}$ and I the input constraints $\{\!|a \mathrel{\dot{\succ}} init(n)|\!\}$. We distinguish four different cases:

1) if $I = \emptyset$ and $O = \{\{\!|init(n) \mathrel{\dot{\prec}} a|\!\}\}$, then the solution is $[a := init(n)]$.

2) if $I \neq \emptyset$ and $O = \{\{\!|init(n) \dot{\prec} a|\!\}\}$, and we have $(n, m) \in \mathsf{Top}$ for every constraint $\{\!|a \dot{\succ} init(m)|\!\}$ in I, then the solution is $[a := init(n)]$. Otherwise the constraint is unsatisfiable, i.e. $\Box$.
3) if $I \neq \emptyset$ and $O = \emptyset$, then the constraint solver finds n such that for every $\{\!|a \dot{\succ} init(m)|\!\} \in I$ we have $(n, m) \in \mathsf{Top}$. For each such n, $[a := init(n)]$ is a possible solution.
4) if $I = \emptyset$ and $O = \emptyset$, then the broadcast part of the constraint is trivially true.

VI. Related work

ProVerif [3] is a specialised tool for security protocol verification. It accepts protocol specifications in a version of the applied pi-calculus [1]. The tool is parametric in a term language equipped with equations and unidirectional rewrite rules, but works in a fixed logic (predicate logic with equality). ProVerif does not include a symbolic simulator or a bisimulation checker. mCRL2 for ACP [7] allows higher order sorted free algebras and equational logics, and PAT3 [14] includes a CSP♯ [20] module where actions built over types like booleans, integers are extended with C♯ like programs.

Our symbolic semantics and bisimulation generation algorithm (slight variations of our previous work [11]) are to a large extent based on the pioneering work by Hennessy and Lin [9] for value-passing CCS, later specialised for the pi-calculus by Boreale and De Nicola [4] and independently by Lin [12], [13].

Acknowledgement: This work has been supported by the ProFun project.

References

[1] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *Proceedings of POPL '01*. ACM, Jan. 2001, pp. 104–115.

[2] J. Bengtson, M. Johansson, J. Parrow, and B. Victor, "Psi-calculi: A framework for mobile processes with nominal data and logic," *Logical Methods in Computer Science*, vol. 7, no. 1, 2011.

[3] B. Blanchet, "Using Horn clauses for analyzing security protocols," in *Formal Models and Techniques for Analyzing Security Protocols*, ser. Cryptology and Information Security Series, V. Cortier and S. Kremer, Eds. IOS Press, Mar. 2011, vol. 5, pp. 86–111.

[4] M. Boreale and R. De Nicola, "A symbolic semantics for the π-calculus," *Information and Computation*, vol. 126, no. 1, pp. 34–52, 1996.

[5] J. Borgström, S. Huang, M. Johansson, P. Raabjerg, B. Victor, J. Åman Pohjola, and J. Parrow, "Broadcast psi-calculi with an application to wireless protocols," in *Proc. SEFM 2011*, ser. LNCS, vol. 7041. Springer, 2011, pp. 74–89.

[6] M. J. Gabbay and A. M. Pitts, "A new approach to abstract syntax with variable binding," *Formal Aspects of Computing*, vol. 13, pp. 341–363, 2001.

[7] J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg, "The formal specification language mCRL2," in *Methods for Modelling Software Systems (MMOSS)*, ser. Dagstuhl Seminar Proceedings, no. 06351. Dagstuhl, Germany: IBFI, 2007.

[8] R. Gutkovas, "Psi Calculi Workbench," 2013, a tool for Psi-calculi, developed at Department of Information Technology, Uppsala University. [Online]. Available: http://www.it.uu.se/research/group/mobility/applied/psiworkbench

[9] M. Hennessy and H. Lin, "Symbolic bisimulations," *Theoretical Computer Science*, vol. 138, no. 2, pp. 353–389, 1995.

[10] M. Johansson, J. Bengtson, J. Parrow, and B. Victor, "Weak equivalences in psi-calculi," in *Proc. of LICS 2010*. IEEE, 2010, pp. 322–331.

[11] M. Johansson, B. Victor, and J. Parrow, "Computing strong and weak bisimulations for psi-calculi," *Journal of Logic and Algebraic Programming*, vol. 81, no. 3, pp. 162–180, 2012.

[12] H. Lin, "Symbolic transition graph with assignment," in *Proceedings of CONCUR '96*, ser. Lecture Notes in Computer Science, U. Montanari and V. Sassone, Eds., vol. 1119. Springer, 1996, pp. 50–65.

[13] ——, "Computing bisimulations for finite-control pi-calculus," *Journal of Computer Science and Technology*, vol. 15, no. 1, pp. 1–9, 2000.

[14] Y. Liu, J. Sun, and J. S. Dong, "PAT 3: An extensible architecture for building multi-domain model checkers," in *ISSRE '11*, T. Dohi and B. Cukic, Eds. IEEE, 2011, pp. 190–199.

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: a Tiny AGgregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, Dec. 2002.

[16] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part I/II," *Information and Computation*, vol. 100, pp. 1–77, Sep. 1992.

[17] J. Parrow, J. Borgström, P. Raabjerg, and J. Åman Pohjola, "Higher-order psi-calculi," 2012, to appear in MSCS. Available from http://www.it.uu.se/research/group/mobility.

[18] A. M. Pitts, "Nominal logic, a first order theory of names and binding," *Information and Computation*, vol. 186, pp. 165–193, 2003.

[19] "Poly/ML," 2013, a full implementation of Standard ML. [Online]. Available: http://www.polyml.org

[20] J. Sun, Y. Liu, J. S. Dong, and C. Chen, "Integrating specification and programs for system modeling and verification," in *Proc. TASE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 127–135.