A Parametric Tool for Applied Process Calculi: Psi-Calculi Workbench

Johannes Borgström    Ramūnas Gutkovas
Iona Rodhe    Björn Victor

Uppsala University, Sweden

Supported by ProFuN and UPMARC
Motivation

- Application Specific Reasoning
- Parametric
- Dynamic Connectivity
- Mobility
- Psi-Calculi Workbench
Psi-Calculi Workbench (Pwb) Features

Communication Primitives

- Unicast
- Wireless Broadcast

Parametric

On

Data Structures
- e.g., Names, Bits, Vectors, ADTs, Trees, ...

Logics
- e.g., EUF, FOL, Equational Theory, ...

Logical Assertions
- e.g., Knows a secret, Connectivity, Constraints...
Pwb Functionality

Symbolic Execution

\[ \Psi > P \xrightarrow{\alpha} P' \]

Constraints

Symbolic Behavioral Equivalence Checking

\[ P \sim Q \]
Psi-Calculi is a **parametric** process calculi framework

- Designed for **applications** (WSNs, Cache Coherence, Security Protocols, etc.)
- The **soundness** of the **meta-theory** has been **machine-checked** with **Isabelle** (algebraic laws, bisimulation theory, compositional semantics, etc.) **inherited** by all calculi [to appear in JAR]
- **Symbolic Semantics** and **Bisimulation** Checking Algorithm [JLAP’12]
Psi-Calculi Expressiveness

- **Captures:**
  - **Cryptography** as in the applied pi-calculus by (Abadi, Fournet 2001)
  - **Fusion** as in the explicit fusion calculus by (Wischik, Gardner 2005)
  - **Concurrent Constraints** as in the concurrent constraint calculus by (Buscemi, Montanari 2007)
  - **Polyadic Synchronization** (Carbone, Maffeis 2002)

- **Extensions:**
  - **Higher Order Processes** [to appear in MSCS]
  - Wireless **Broadcast** Communication [SEFM’11]
  - Generalized **pattern matching** and **sorts** [to appear in TGC’13]
  - Parametrized **type system** for Psi-calculi by (Hüttel 2011)
Psi-Calculi Workbench via an Example

Data Collection in Wireless Sensor Network
Purpose of the Example

- Demonstrate the *features* of Pwb
- Demonstrate the *functionality* of Pwb
- Applicability of Pwb to WSNs
Data Collection in Wireless Sensor Networks

- Network consists of a set of nodes and one distinguished node sink.
- Protocol has two phases:
  1. Establishment of a routing tree (rooted at sink): nodes wirelessly broadcast a special initialization message.
  2. Data collection: nodes send and forward data via established route using (reliable) unicast messages.
Specification in Pwb

Node Behavior

Sink(nodeId, sinkChan) <=
  "init(nodeId)! <sinkChan> .
  ! "data(sinkChan)"(x). ProcData<x> ;

Node(nodeId, nodeChan, datum) <=
  "init(nodeId)? (chan) .
  "init(nodeId)! <nodeChan> .
  "data(chan)"<datum> .
  ! "data(nodeChan)"(x).
  "data(chan)"<x> ;
Specification in Pwb

Node Behavior

Sink(nodeId, sinkChan) <=
  "init(nodeId)"! <sinkChan> .
  ! "data(sinkChan)"(x). ProcData<x> ;

Node(nodeId, nodeChan, datum) <=
  "init(nodeId)"? (chan) .
  "init(nodeId)"! <nodeChan> .
  "data(chan)"<datum> .
  ! "data(nodeChan)"(x).
  "data(chan)"<x> ;

1. Route Tree Establishment
Specification in Pwb

Node Behavior

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

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

1. Route Tree Establishment

2. Data Collection
Specification in Pwb

Node Behavior

Sink(nodeId, sinkChan) <=
  "init(nodeId)"! <sinkChan> .
  ! "data(sinkChan)"(x). ProcData<x> ;

Node(nodeId, nodeChan, datum) <=
  "init(nodeId)"? (chan) .
  "init(nodeId)"! <nodeChan> .
  "data(chan)"<datum> .
  ! "data(nodeChan)"(x).
  "data(chan)"<x> ;

System

(new sinkChan)  Sink<0, sinkChan>
(new chan1)     Node<1, chan1, datum1>
(new chan2)     Node<2, chan2, datum2>

Node Connectivity for Broadcasting

Sink

0

1

2

graph represented as edge list

(0,1), (0,2), (1,2)
**Pwb Features**

**Node Behavior**

Sink(nodeId, sinkChan) <=
"init(nodeId)"! <sinkChan> .
! "data(sinkChan)"(x). ProcData<x> ;

Node(nodeId, nodeChan, datum) <=
"init(nodeId)"? <chan> .
"init(nodeId)"! <nodeChan> .
"data(chan)"<datum> .
! "data(nodeChan)"(x).
"data(chan)"<x> ;

**System**

(new sinkChan) Sink<0, sinkChan>
(new chan1) Node<1, chan1, datum1>
(new chan2) Node<2, chan2, datum2>

**Node Connectivity for Broadcasting**

Sink

0

1

2

(graph represented as edge list)

(0,1), (0,2), (1,2)
Pwb Features

Node Connectivity for Broadcasting

Sink

Node<0, sinkChan>

Node<1, chan1, datum1>

Node<2, chan2, datum2>

(input channel) =>

(input channel)

(broadcast output channel)

(broadcast input channel)

System

all channels are structured

input channel

output channel

(graph represented as edge list)

(0,1), (0,2), (1,2)
The document contains a diagram and some text related to Pwb Features. The diagram includes nodes and edges, with process definitions and connectivity for broadcasting. The text describes the behavior of sink and node processes and the structure of channels.

Key points:
- All channels are structured.
- There is a broadcast output channel and a broadcast input channel.
- Node behavior is defined with process definitions.
- Process invocations are shown.
- Env. assertions and system broadcast output channel are also mentioned.

The diagram shows nodes with channels and data, connected with arrows indicating the flow of data and processes.
Establishment of a Routing Tree (1)

Connectivity as current assertion:

(new sinkChan) Sink<0, sinkChan>
(new chan1) Node<1, chan1, datum1>
(new chan2) Node<2, chan2, datum2>

"init(0)"!(new sinkChan)sinkChan

true

(!"data(sinkChan)"(gnb). ProcData<gnb>))
(((new chan1)(
  "init(1)"!<chan1>. 
  "data(sinkChan)"<datum1>. 
  !("data(chan1)"(gnb). 
  "data(sinkChan)"<gnb>)))
(((new chan2)(
  "init(2)"!<chan2>. 
  "data(sinkChan)"<datum2>. 
  !("data(chan2)"(gnb). 
  "data(sinkChan)"<gnb>))))

broadcasts

can unicast
Establishment of a Routing Tree (2)

(!"data(sinkChan)"(gnb). ProcData<gnb>)) |
(((new chan1)(
  "init(1)!<chan1>.
  "data(sinkChan)"<datum1>.
  !("data(chan1)"(gnb).
   "data(sinkChan)"<gnb>))})) |
(((new chan2)(
  "init(2)!<chan2>.
  "data(sinkChan)"<datum2>.
  !("data(chan2)"(gnb).
   "data(sinkChan)"<gnb>))))

"init(1)!<new chan1>chan1

(!"data(sinkChan)"(gnc). ProcData<gnc>)) |
("data(sinkChan)"<datum1>.
 !("data(chan1)"(gnc). "data(sinkChan)"<gnc>)) |
(((new chan2)(
  "init(2)!<chan2>.
  "data(sinkChan)"<datum2>.
  !("data(chan2)"(gnc).
   "data(sinkChan)"<gnc>))))

broadcasts can unicast
Data Collection

(!"data(sinkChan)"(gnc). ProcData<gnc>)) |
(!"data(sinkChan)"<datum1>. "data(sinkChan)<gnc>") |
((new chan2)(
  "init(2)!<chan2>.
  "data(sinkChan)<datum2>. "data(sinkChan)<gnc>
))

(!"data(sinkChan)"(gnb). ProcData<gnb>)) |
(!"data(chan1)"(gnc). "data(sinkChan)<gnc>") |
((new chan2)(
  "init(2)!<chan2>.
  "data(sinkChan)<datum2>. "data(sinkChan)<gnc>
))

(!"data(chan1)"(gna). "data(sinkChan)<gna>") |
((new chan2)(
  "init(2)!<chan2>.
  "data(sinkChan)<datum2>. "data(sinkChan)<gna>
))
Example Summary

- Structured channels
- **Broadcast** and **Unicast** Communication
- Broadcast **Connectivity** as an **assertion**
- Parameters of Pwb for WSN
How to Implement an Instance
Parametric Pwb Architecture

Supporting library of Solvers, Nominal, Parser, Printer, etc.

Pwb

Command Interpreter
- Symbolic Equivalence Checker
- Symbolic Execution
- Psi Calculi Core
Implementing the Example in Pwb

**Type of Data**

```plaintext
datatype term = Init of term |
| Data of term |
| Name of name |
| Int of int |
```

**Type of Logic**

```plaintext
datatype condition = OutputConn of term * term |
| InputConn of term * term |
| ChEq of term * term |
```

**Type of Assertions**

```plaintext
datatype assertion = Top |
```
Implementing the Example in Pwb

<table>
<thead>
<tr>
<th>Type of Data</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datatype term</strong></td>
<td><strong>chaneq</strong> : ( \text{term} \times \text{term} \rightarrow \text{condition} )</td>
</tr>
<tr>
<td>(= \text{Init of term} )</td>
<td><strong>brReceive</strong> : ( \text{term} \times \text{term} \rightarrow \text{condition} )</td>
</tr>
<tr>
<td>(</td>
<td>\text{Data of term} )</td>
</tr>
<tr>
<td>(</td>
<td>\text{Name of name} )</td>
</tr>
<tr>
<td>(</td>
<td>\text{Int of int} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of Logic</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datatype condition</strong></td>
<td><strong>subst(\mathcal{A})</strong> : ( (\text{name} \times \text{term}) \times \text{list} \rightarrow \mathcal{A} \rightarrow \mathcal{A} )</td>
</tr>
<tr>
<td>(= \text{OutputConn of term} \times \text{term} )</td>
<td>for (\mathcal{A}) in (\text{term}, \text{condition}, \text{assertion})</td>
</tr>
<tr>
<td>(</td>
<td>\text{InputConn of term} \times \text{term} )</td>
</tr>
<tr>
<td>(</td>
<td>\text{ChEq of term} \times \text{term} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type of Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>datatype assertion</strong></td>
</tr>
</tbody>
</table>
Implementing the Example in Pwb

Execution Constraint Solver

solve : condition list -> (substitution*assertion) option

(simplified signature)

In the example solves a unification problem for unicast channels

e.g. ChEq (Data “sinkChan”, Data “sinkChan”)

In the example solves a membership problem in the connectivity graph for broadcast channels
Easy to Obtain a Tool

```
datatype term

datatype condition

datatype assertion

chaneq : term*term -> condition
brReceive : term*term -> condition
brTransmit : term*term -> condition
compose : assertion*assertion -> assertion

subst\(A\) : (name*term) list -> A -> A
for \(A\) in term, condition, assertion

solve : condition list ->
(substitution*assertion) option
```
Currently implemented Calculi (1)

- **pi-calculus (260 + 333 LOC)**: data are names, logic of equality on names. Also includes a equivalence solver.

- **Frame Hopping Spread Spectrum (292 LOC)**: structured data as channels, logic of equality on the data. Solver is implemented as unification.

- **Common Ether (331 LOC)**: data are names, assertions are sets of names and logic of membership of a name in the current assertion. Solver generates assertions.
Currently implemented Calculi (2)

- **The example WSN (462 LOC):** as already discussed.
- **Aggregating WSN (569 LOC):** similar to the example. The instance extends it with structured data with the aggregation operator and positive integers with arithmetic operations.
- **Alternating Bit Protocol (574 LOC):** structured data with bit operations, substitution does term rewriting, logic is over equations of bit operations and bit predicates.
- **Value-passing CCS (538 LOC):** structured data over a signature (of arithmetic), logic of linear equations, solver use an external linear programming solver.
Instance Summary

- All of the instances are done in the same tool Pwb
- Don’t need separate tools
- The calculi inherit machined checked proofs
Future Work

**Functionality**
- Model Checking
- Behavioral Preorder Checker
- Behavioral Types
- Modular Construction of Instances

**Application**
- Bigger Real-World Instances
- Interfacing with Isabelle (generating proofs)

**Extensions**
- Higher Order Processes
- Reliable Broadcast
- Pattern Matching
Conclusions

- Pwb is a *parametric* tool on data and logics for concurrency
- Pwb is *one tool* for *many calculi* inheriting machine checked proofs
- Pwb provides primitives for both *unicast* and *broadcast* communication
- Pwb provides symbolic *execution* and *equivalence* checking
Where to Get Pwb

Pwb Home
http://goo.gl/ZJPu9

Pwb on Github
http://goo.gl/aU40h

Pwb is free software (GPL)
Runs on UNIX like systems (on Windows, use cygwin)