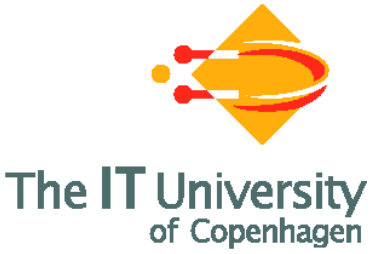



# The BPL Tool

## A Tool for Experimenting with Bigraphical Reactive Systems

**Espen Højsgaard**
**Arne John Glenstrup**

**IT University Technical Report Series** **TR-2011-145**

**ISSN 1600–6100** **October 2011**

**ISSN 1600–6100**

**ISBN 978-87-7949-244-8**

**Copies may be obtained by contacting:**

**IT University of Copenhagen**
**Rued Langgaards Vej 7**
**DK-2300 Copenhagen S**
**Denmark**

**Telephone: +45 72 18 50 00**
**Telefax: +45 72 18 50 01**
**Web** `www.itu.dk`

# The BPL Tool
# A Tool for Experimenting with Bigraphical Reactive Systems

Espen Højsgaard Arne John Glenstrup*

November 2, 2011

We present the BPL Tool, a first implementation of bigraphical reactive systems with binding. The BPL Tool provides manipulation, simulation and visualisation of bigraphs and bigraphical reactive systems, and can be used either through the included web and command line user interfaces or as a programming library.

## Contents

**1 Introduction** **2**
1.1 Related work . . . 2
1.2 Outline . . . 3

**2 Installation** **4**
2.1 User installation . . . 4
2.2 Developer installation . . . 4

**3 Example: Polyadic $\pi$ and Mobile Phones** **5**
3.1 A mobile phone system . . . 5
3.2 Polyadic $\pi$ . . . 5

**4 Reference** **9**
4.1 Preliminaries . . . 9
4.2 Signatures . . . 11
4.3 Types for bigraph terms . . . 12
4.4 Bigraphs . . . 12
4.4.1 Syntactic Sugar . . . 13
4.5 Bigraph Operations . . . 14
4.6 Matching . . . 15
4.7 Lazy lists . . . 15
4.8 Reaction rules . . . 16
4.9 Simulation . . . 17
4.10 Pretty printing . . . 17
4.11 Visualization . . . 18

*{espen,panic}@itu.dk. IT University of Copenhagen, Denmark

4.12 Controlling tool behaviour . . . . . 19
4.13 Exceptions . . . . . 20

**5 Conclusions and Future Work** **21**

# 1 Introduction

The theory of bigraphical reactive systems [19] provides a general meta-model for describing and analyzing mobile and distributed ubiquituous systems. Bigraphical reactive systems form a graphical model of computation in which graphs that embody both locality and connectivity can be reconfigured using reaction rules. So far it has been shown how to use the theory to recover behavioural theories for various process calculi [16, 15, 17] and how to use the theory to model context-aware systems [6].

In this report, we describe the BPL Tool, a first prototype implementation of bigraphical reactive systems, which can be used for experimenting with bigraphical models with binding. The theoretical foundations for the implementation have been developed in detail in [13], but in summary the BPL Tool is based on Damgaard et al.'s axiomatization of binding bigraphs [9] (i.e. it is term based) and the inductive characterization of matching [5] by the same authors. In [13] we have extended the inductive characterization from *graphs* to a *term* representation of bigraphs and have given an algorithmic interpretation of this characterization of matching. This required the development of some additional algorithms for bigraph terms: normalisation, renaming, and regularisation.

The BPL Tool is written in SML, consists of parser, normalisation and matching kernel, and includes web and command line user interfaces. To ensure correctness, we have implemented normalisation, renaming, regularisation and matching faithfully by implementing one SML function for every inference rule – in the case of matching, two: one for applications above and one for below the SWX rule.

The BPL Tool has been used to model the following:

| | |
|---|---|
| **the ARAN protocol** | (Bentzen, [2]) |
| **the GeoCast protocol** | (Niss, unpublished) |
| **IEEE 802.11 MAC 4-way handshake** | (Bentzen, [2]) |
| **the Insider Problem** | (Bentzen, [2]) |
| **a mobile phone system** | (Glenstrup, included in this report) |
| **platographical models** | (Elsborg, [12]) |
| **WS-BPEL and HomeBPEL** | (Bundgaard et al., [8]) |

The BPL Tool is available at `http://www.itu.dk/research/pls/wiki/index.php/BPL_Tool` where also some additional material can be found, such as API documentation and slides from presentations.

## 1.1 Related work

A number of implementations of bigraphs are being developed at various institutions. Unfortunately, it is hard to find the implementations themselves or papers describing them – until now,

this has also been the case for the BPL Tool – but here is a complete list of the implementations of which we are aware:

**BigMC:** A model checker for bigraphs which includes a command line interface and visualisation [4].

**bigraphspace:** A Java library which provides a tuple-space-like API based on bigraphs [14].

**Big Red:** A graphical editor for bigraphs with easily extensible support for various file formats [3].

**BigWB:** A graphical workbench for bigraphs, aiming at providing a unifying GUI for the various bigraph tools (no website or papers at the time of writing).

**DBtk:** A tool for directed bigraphs, which provides calculation of IPOs, matching, and visualisation [1].

**SAT based algorithm:** Sevegnani et al. has presented a SAT based algorithm for matching in place graphs with sharing [21] and an implementation is in progress based on MiniSAT [11].

**SBAM:** A stochastic simulator for bigraphs, aimed at simulation of biological models [20].

## 1.2 Outline

In the remainder of this report, we assume a basic knowledge of bigraphs; we refer the uninitiated reader to Milner's book [19]. We shall use the bigraph notation from [13].

This report encourages a hands-on approach, and our focus is therefore on getting the tool installed and trying an example. The tool has built-in documentation, which we include (and slightly expand) for easy reference.

**Section 2, Installation**

Instructions for how to obtain, install and run the BPL Tool.

**Section 3, Example: Polyadic $\pi$ and Mobile Phones**

We demonstrate the features of the BPL Tool using Milner's polyadic $\pi$ calculus model of mobile phones [18].

**Section 4, Reference**

We present the BPL language (BPLL) for bigraphical reactive systems and the various functions that the BPL Tool provides.

**Section 5, Conclusions and Future Work**

We present our experiences with the BPL Tool and conclude on its strengths and weaknesses, and present our plans for future improvements.

# 2 Installation

The BPL Tool is distributed as source code as it relies on an SML compiler with an interactive mode to provide a command line interface. The source code can be obtained from the BPL Tool website [7].

We shall here distinguish between two types of installation: user and developer installations.

## 2.1 User installation

The BPL Tool requires the following software to be installed on your system:

**SML compiler** preferably SML of New Jersey but Moscow ML and MLton should work as well.

**GNU make**

**GNU sed**

The BPL Tool has been known to run on the following platforms: Linux (Ubuntu), OS X (ver. 10.4-10.6), and Windows XP (using Cygwin).

When you have installed the above and obtained a copy of the BPL Tool sources, you need to configure the tool to your setup. This is done by executing the following command in the $BPL/src directory:

```
./configure
```

To use the BPL Tool CLI, you need to use an SML compiler with an interactive mode (i.e. not MLton) – it works particularly well with SML of New Jersey, since SML/NJ allows the use of custom pretty-printers for values.

To start the CLI, execute the following command in the $BPL/src directory:

```
./bpltoolcli.sh
```

Once the BPL Tool has loaded you should be met with the prompt:

```
BPL (revision 3294) interactive prompt.  Type 'help[];' for help.
-
```

## 2.2 Developer installation

BPL Tool developers will – in addition to the basic installation – need the following software:

**GNU autoconf:** needed if you change configure.in.

**SML#:** BPL Tool uses some of the tools, SMLUnit and SMLDoc, from the SML# distribution, but not SML# itself.

To run the unit tests, execute the following command in the $BPL/src directory (or one of its sub-directories):

```
make test
```

If you have multiple SML compilers installed, you can switch between them by running the configure script with the MLC option set to one of the values `mlton`, `mosml`, or `smlnj`. E.g. to use the Moscow ML compiler, run

```
./configure MLC=mosml
```

# 3 Example: Polyadic $\pi$ and Mobile Phones

We model the polyadic $\pi$ calculus, running the mobile phone system introduced in Milner's $\pi$ book [18].

## 3.1 A mobile phone system

The mobile phone system we shall model is the following: there is a static network of transmitters which are all connected to a central control. Each mobile phone is located in a car and is connected to a single transmitter using a unique frequency. On some events, e.g. signal fading, the mobile phone may switch to another transmitter.

A simple example of such a system, call it $System_1$, is shown in Figure 1, where we also show how to define the system in the BPL Tool: the system consists of a car, one active and one idle transmitter, and a control centre. Note that the controls are atomic, since nodes with these controls

```
val (switch1,  talk1,  lose1,  gain1) =
  ( "switch1","talk1","lose1","gain1")
val (switch2,  talk2,  lose2,  gain2) =
  ( "switch2","talk2","lose2","gain2")

val Car      = atomic ("Car"     -: 2)
val Trans    = atomic ("Trans"   -: 4)
val Idtrans  = atomic ("Idtrans" -: 2)
val Control  = atomic ("Control" -: 8)

val System1 =
      Car[talk1,switch1]
  `|` Trans[talk1,switch1,gain1,lose1]
  `|` Idtrans[gain2,lose2]
  `|` Control[lose1,talk2,switch2,gain2,
              lose2,talk1,switch1,gain1]
```

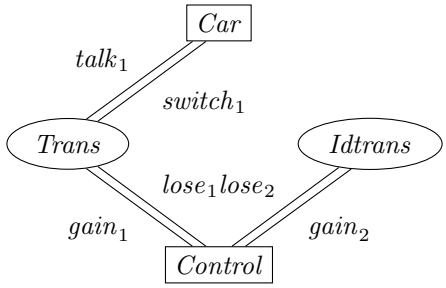



Figure 1: Definition of the mobile phone system, $System_1$

should not contain other nodes.

The illustration of $System_1$ in Figure 1 is "hand-drawn" in TikZ, but we can also use the BPL Tool to render the system using either SVG or TikZ. For example, Figure 2 shows how to generate TikZ for $System_1$ and the resulting diagram. This uses the default configuration, but it is possible to obtain more fine grained control over the appearance of roots, nodes and sites. As is evident from the diagram in this example, it cannot replace hand-drawn figures. Nevertheless, it is our experience that such automatic visualisation is very helpful when working with bigraphs in the BPL Tool.

## 3.2 Polyadic $\pi$

Let us now examine how we can model a dynamic aspect of the mobile phone system, namely the hand-over protocol for when a mobile phone switches from one transmitter to another. We shall model this in the polyadic $\pi$ calculus which again can be modelled directly in the BPL Tool.

```
- print (tikz System1);
\tikzstyle nametext=[font=\footnotesize\itshape,inner sep=0pt]%
\tikzstyle root=[dashed,rounded corners]%
\tikzstyle binder=[draw,fill=white]%
\tikzstyle node=[draw]%
\tikzstyle nodetext=[font=\sffamily\bfseries,text=blue,inner sep=0pt]%
\tikzstyle site=[fill=gray!25,rounded corners]%
\tikzstyle sitetext=[font=\sffamily,inner sep=0pt]%
\tikzstyle link=[draw]%
\begin{tikzpicture}[x={(0.02cm,0cm)},y=-0.02cm,baseline=-1cm]
  \draw[style=root] (0,16) rectangle +(260,88);
  \draw[style=node] (29,64) ellipse (0.5cm and 0.4cm);
  \draw (4,100) node [style=nodetext,anchor=south west] {Car};
  \draw[style=node] (83,64) ellipse (0.5cm and 0.4cm);
  \draw (58,100) node [style=nodetext,anchor=south west] {Trans};
  \draw[style=node] (147,64) ellipse (0.7cm and 0.4cm);
  \draw (112,100) node [style=nodetext,anchor=south west] {Idtrans};
  \draw[style=node] (221,64) ellipse (0.7cm and 0.4cm);
  \draw (186,100) node [style=nodetext,anchor=south west] {Control};
  \draw (231,45) .. controls +(0,-24) and +(0,20) .. (109,13);
  \draw (72,46) .. controls +(0,-24) and +(0,20) .. (109,13);
  \draw (25,44) .. controls +(0,-24) and +(0,20) .. (109,13);
  \draw (109,10) node [style=nametext,anchor=south] {talk1};
  \draw (238,47) .. controls +(0,-24) and +(0,20) .. (158,13);
  \draw (79,44) .. controls +(0,-24) and +(0,20) .. (158,13);
  \draw (32,44) .. controls +(0,-24) and +(0,20) .. (158,13);
  \draw (158,10) node [style=nametext,anchor=south] {switch1};
  \draw (196,50) .. controls +(0,-24) and +(0,20) .. (207,13);
  \draw (93,46) .. controls +(0,-24) and +(0,20) .. (207,13);
  \draw (207,10) node [style=nametext,anchor=south] {lose1};
  \draw (245,49) .. controls +(0,-24) and +(0,20) .. (249,13);
  \draw (86,44) .. controls +(0,-24) and +(0,20) .. (249,13);
  \draw (249,10) node [style=nametext,anchor=south] {gain1};
  \draw (217,44) .. controls +(0,-24) and +(0,20) .. (291,13);
  \draw (143,44) .. controls +(0,-24) and +(0,20) .. (291,13);
  \draw (291,10) node [style=nametext,anchor=south] {gain2};
  \draw (224,44) .. controls +(0,-24) and +(0,20) .. (333,13);
  \draw (150,44) .. controls +(0,-24) and +(0,20) .. (333,13);
  \draw (333,10) node [style=nametext,anchor=south] {lose2};
  \draw (203,47) .. controls +(0,-24) and +(0,20) .. (375,13);
  \draw (375,10) node [style=nametext,anchor=south] {talk2};
  \draw (210,45) .. controls +(0,-24) and +(0,20) .. (424,13);
  \draw (424,10) node [style=nametext,anchor=south] {switch2};
\end{tikzpicture}
```

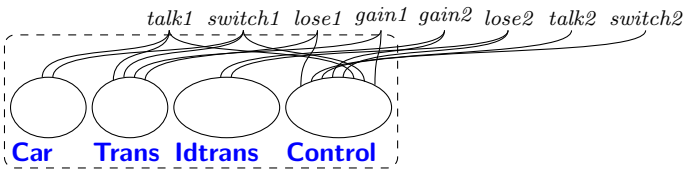
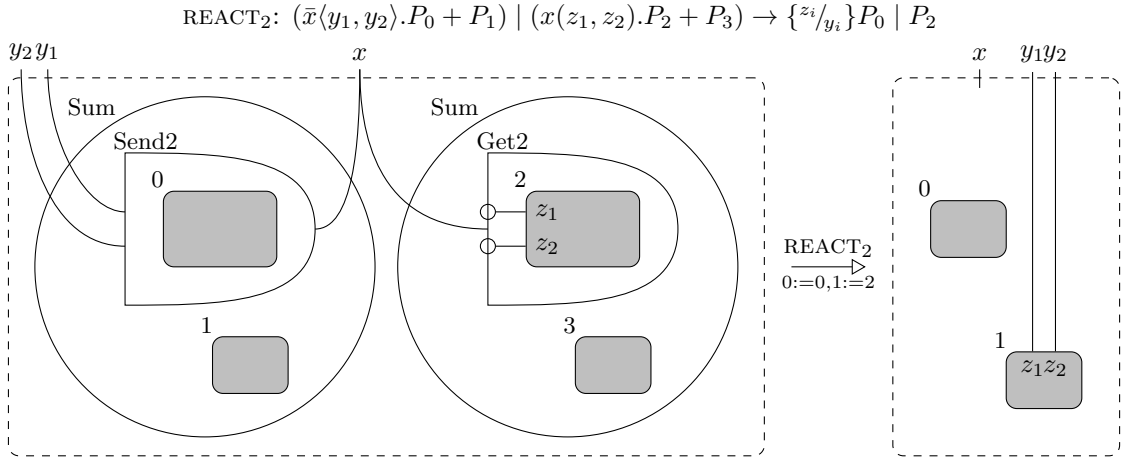



Figure 2: Generating TikZ

The polyadic $\pi$ calculus can be modeled by a family of reaction rules $\{\text{REACT}_i \mid i = 0, 1, \ldots\}$, one for each number of names that are to be communicated in a reaction [16]; $\text{REACT}_2$ is shown in Figure 3. The signature for the nodes modelling the polyadic $\pi$ calculus is constructed using




```
val REACT2 = "REACT2" :::

     Sum o (Send2[x,y1,y2] `|` `[]`)
  `|` Sum o (Get2[x][[z1],[z2]] `|` `[]`)

  --[0 |-> 0, 1 |-> 2]--|>

  (y1/z1 * y2/z2 * x//[]) o (`[]` `|` `[z1, z2]`)
```

Figure 3: $\pi$ calculus reaction rule shown as bigraphs and BPL values.

`passive` controls as shown in Figure 4. For this system, we only need `Send` and `Get` nodes for

```
val Sum   = passive0 ("Sum")
val Send0 = passive  ("Send0" -: 0  +  1)
val Get0  = passive  ("Get0"  -: 0  +  1)
val Send2 = passive  ("Send2" -: 2  +  1)
val Get2  = passive  ("Get2"  =: 2 --> 1)
```

Figure 4: Signature for polyadic $\pi$ calculus.

$\text{REACT}_0$ and $\text{REACT}_2$. Note that all reaction rule nodes are passive, preventing reaction within a guarded expression.

In the $\pi$ calculus the nodes in $System_1$ are defined as recursive equations. In the BPL tool, they are defined by a rule that unfolds an atomic node into a bigraph corresponding to the defining $\pi$ calculus expression. The definitional equations and BPL definitions are shown in Figure 5.

The definitions allows the control centre to switch *Car* communication between the two transmitters (supposedly when the car gets closer to the idle than the active transmitter), and allows the car to communicate with the active transmitter.

| *Defining equation* | *BPL definition* |
|---|---|
| $Car(talk, switch) \stackrel{\text{def}}{=}$<br>$\overline{talk}.\, Car\langle talk, switch\rangle$<br>$\quad + switch(t, s).\, Car\langle t, s\rangle$ | `val DEF_Car = "DEF_Car" :::`<br>`  Car[talk,switch]`<br>`  ----|>`<br>`  Sum o (    Send0[talk] o Car[talk,switch]`<br>`         '|' Get2[switch][[t],[s]] o Car[t,s])` |
| $Trans(talk, switch, gain, lose) \stackrel{\text{def}}{=}$<br>$talk\,.\, Trans\langle talk, \underline{switch}, gain, lose\rangle$<br>$\quad + lose(t, s).\overline{switch}\langle t, s\rangle$<br>$\quad\quad .\, Idtrans\langle gain, lose\rangle$ | `val DEF_Trans = "DEF_Trans" :::`<br>`  Trans[talk,switch,gain,lose]`<br>`  ----|>`<br>`  Sum o (    Get0[talk][]`<br>`              o Trans[talk,switch,gain,lose]`<br>`         '|' Get2[lose][[t],[s]]`<br>`              o Sum o Send2[switch,t,s]`<br>`                 o Idtrans[gain,lose])` |
| $Idtrans(gain, lose) \stackrel{\text{def}}{=}$<br>$gain(t, s).\, Trans\langle t, s, gain, lose\rangle$ | `val DEF_Idtrans = "DEF_Idtrans" :::`<br>`  Idtrans[gain, lose]`<br>`  ----|>`<br>`  Sum o Get2[gain][[t],[s]] o Trans[t,s,gain,lose]` |
| $Control(lose_1, talk_2, switch_2, gain_2,$<br>$\quad lose_2, talk_1, \underline{switch_1}, gain_1) \stackrel{\text{def}}{=}$<br>$\overline{lose_1}\langle talk_2, switch_2\rangle.\overline{gain_2}\langle talk_2, switch_2\rangle$<br>$\quad .\, Control\langle lose_2, talk_1, switch_1, gain_1,$<br>$\quad\quad lose_1, talk_2, switch_2, gain_2\rangle$ | `val DEF_Control = "DEF_Control" :::`<br>`  Control[lose1,talk2,switch2,gain2,`<br>`          lose2,talk1,switch1,gain1]`<br>`  ----|>`<br>`  Sum o Send2[lose1,talk2,switch2]`<br>`   o Sum o Send2[gain2,talk2,switch2]`<br>`    o Control[lose2,talk1,switch1,gain1,`<br>`              lose1,talk2,switch2,gain2]` |

Figure 5: Definitions of Car, Trans, Idtrans and Control nodes.

Our BPL definition of the initial system in Figure 1, $System_1$, is the folded version; querying the tool reveals the four possible unfolding matches, illustrated in Figure 6. Here `mkrules` constructs

```
- val rules = mkrules [REACT0, REACT2, DEF_Car, DEF_Trans, DEF_Idtrans, DEF_Control];
[...]
- print_mv (matches rules System1);
[{rule = "DEF_Car",
 context
  = (talk1/talk * switch1/switch) ||
    `[]` `|` Trans[talk1, switch1, gain1, lose1] `|`
    Idtrans[gain2, lose2] `|`
    Control[lose1, talk2, switch2, gain2, lose2, talk1, switch1, gain1],
 parameter = idx0},
{rule = "DEF_Control", [...] },
{rule = "DEF_Idtrans", [...] },
{rule = "DEF_Trans", [...] }]
```

Figure 6: Determining which rules match $System_1$.

the internal representation of a rule set, and `print_mv` prettyprints a lazy list of matches, produced by the `matches` function, cf. Sections 4.6 and 4.9.

We can unfold the four nodes into their defining $\pi$ calculus expressions by using the reaction tactic `TAC_unfold`, shown in Figure 7. The tactic is constructed using the `react_rule` tactic which simply applies a named reaction rule and the `++` tactic which runs its arguments sequentially, cf. Section 4.9. Applying this tactic using the function `run`, we get an unfolded version of the system.

Querying the BPL Tool for matches in the unfolded system reveals exactly the switch and talk actions, initiated by $\text{REACT}_2$ and $\text{REACT}_0$ rules, respectively, cf. Figure 8. Applying the $\pi$ calculus reaction rules for switching, using the `TAC_switch` tactic, we arrive at $System_2$, where *Car* communication has been switched to the other transmitter, as witnessed by the outer names to which *Car* ports link, as well as the order of names to which *Control* ports link.

# 4 Reference

The language used in the BPL Tool is called BPLL, and it consists of a number of SML constructs which allows you to write BPLL directly in SML programs. This also means that your favorite interactive SML environment doubles as BPLL environment.

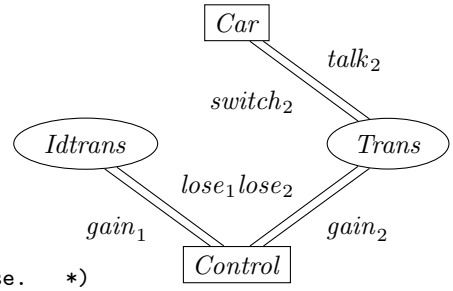
In this section we present the BPLL syntax for bigraphs and bigraphical reactive systems. Much of this information is also accessible through the help function in the BPL Tool CLI.

## 4.1 Preliminaries

BPLL is a DSL embedded in Standard ML. This has the benefit of allowing easy extensions to the language and easy integration into SML programs. But it also imposes some restrictions on the syntax; for example, we cannot use `|` to denote the prime parallel product operator, so instead we use `` `|` ``, i.e. we wrap the operator in backquotes. In general, we have attempted to choose a syntax which visually closely resembles Milner's bigraph notation. While we believe we have been

```
- val TAC_unfold =
  react_rule "DEF_Car"     ++ react_rule "DEF_Trans"   ++
  react_rule "DEF_Idtrans" ++ react_rule "DEF_Control";
[...]
- val System1_unfolded = run rules TAC_unfold System1;
val System1_unfolded =
  Sum o
  (Send0[talk1] o Car[talk1, switch1] `|` Get2[switch1][[t], [s]] o Car[t, s]) `|`
  Sum o
  (Get0[talk1] o Trans[talk1, switch1, gain1, lose1] `|`
   Get2[lose1][[t], [s]] o Sum o Send2[switch1, t, s] o Idtrans[gain1, lose1]) `|`
  Sum o Get2[gain2][[t], [s]] o Trans[t, s, gain2, lose2] `|`
  Sum o
  Send2[lose1, talk2, switch2] o
  Sum o
  Send2[gain2, talk2, switch2] o
  Control[lose2, talk1, switch1, gain1, lose1, talk2, switch2, gain2]
  : 0 -> <{talk1, switch1, gain1, lose1, gain2, lose2, talk2, switch2}>
  : agent
```

Figure 7: Unfolding $System_1$, using the TAC_unfold tactic.




```
- print_mv (matches rules System1_unfolded);
[{rule = "REACT0", [...] }, {rule = "REACT2", [...] }]
[...]
- val TAC_switch =
  react_rule "REACT2" ++ (* Control tells Trans to lose.   *)
  react_rule "REACT2" ++ (* Control tells Idtrans to gain. *)
  react_rule "REACT2";   (* Trans tells Car to switch.     *)
[...]
- val System2 = run rules TAC_switch System1_unfolded;
val System2 =
  Idtrans[gain1, lose1] `|` Car[talk2, switch2] `|`
  Control[lose2, talk1, switch1, gain1, lose1, talk2, switch2, gain2] `|`
  Trans[talk2, switch2, gain2, lose2]
  : 0 -> <{lose1, talk2, switch2, gain2, lose2, talk1, switch1, gain1}>
  : agent
-
```

Figure 8: Checking possible matches, then switching to $System_2$, using the TAC_switch tactic.

reasonably successful at this, we also appreciate that the notation is a bit heavy and we welcome any suggestions for improvements.

In order to keep bigraph terms as readable as possible, the BPL Tool assumes that bigraph names are bound to string variables of the same name. For example, the identity wiring $\mathsf{id}_{\{x\}}$ will be printed as `idw[x]`, thus assuming the preceding declaration of `x`: `val x = "x"`. The same goes for named ports (see below). This saves a lot of quotes when the BPL Tool prints bigraph terms.

## 4.2 Signatures

The definition of signatures in the BPL Tool is centered around controls: to define a control called $K$ which has *status* $s \in \{$`active`,`passive`,`atomic`$\}$, *global arity* $m$ and *local arity* $n$ one writes:

`val` $K$ `=` $s$ `("`$K$`" =:` $m$ `-->` $n$`)`

For instance, in the example in Section 3 we needed a passive control with global arity 1 and local arity 2 called `Get2`, which was defined as follows:

```
val Get2 = passive ("Get2" =:  2 --> 1)
```

There is syntactic sugar for the common cases where the local arity is zero or both arities are zero, and for named ports:

**general case:**

`val` $K$ `=` $s$ `("`$K$`" =:` $m$ `-->` $n$`)`

**local arity = 0:**

`val` $K$ `=` $s$ `("`$K$`" -:` $n$`)`

**global and local arity = 0:**

`val` $K$ `=` $s$`0 ("`$K$`")`

**named ports, general case:**

`val` $K$ `=` $s$ `("`$K$`" ==:` `[`$p'_1$`,...,`$p'_m$`] --->` `[`$p_1$`,...,`$p_n$`])`

**named ports, local arity = 0:**

`val` $K$ `=` $s$ `("`$K$`" --:` `[`$p_1$`,...,`$p_n$`])`

where $K$ is the control name, $s \in \{$`active`,`passive`,`atomic`$\}$ is the status, $m$ is the global arity, $n$ is the local arity, and the $p_i$ and $p'_i$ are the names of global and local ports respectively.

It is not quite precise to say that the latter four cases are just syntactic sugar, as the way the control is used depends on how it was declared (i.e. their SML types are different) (cf. Ions below).

## 4.3 Types for bigraph terms

The BPL Tool uses the following types to distinguish bigraph terms on certain forms:

`bgterm:` a bigraph term that might not be well-formed, i.e. interfaces in compositions might not match;

`bgval:` a well-formed bigraph term with interfaces;

`'a bgbdnf:` a bigraph term which is on a binding discrete normal form indicated by the phantom type used for `'a`:

- `M:` molecule
- `S:` singular top-level node
- `G:` global discrete prime
- `N:` name-discrete prime
- `P:` discrete prime
- `D:` discrete bigraph
- `B:` bigraph
- `DR:` discrete, regular bigraph
- `BR:` regular bigraph

The normal forms are shown in Figure 9.

$$
\begin{array}{rll}
M & ::= (\mathsf{id}_Z \otimes K_{\vec{y}(\vec{X})})N & \textit{molecule} \\
S & ::= \ulcorner\alpha\urcorner \quad | \quad M & \textit{singular top-level node} \\
G & ::= (\mathsf{id}_Y \otimes \mathit{merge}_n)(\bigotimes_i^n S_i)\pi & \textit{global discrete prime} \\
N & ::= (X)G & \textit{name-discrete prime} \\
P,Q & ::= (\mathsf{id}_Z \otimes \widehat{\sigma})N & \textit{discrete prime} \\
D & ::= \alpha \otimes (\bigotimes_i^n P_i)\pi & \textit{discrete bigraph} \\
B & ::= (\omega \otimes \mathsf{id}_{(\vec{X})})D & \textit{bigraph} \\
DR & ::= \alpha \otimes (\bigotimes_i^n P_i) & \textit{discrete, regular bigraph} \\
BR & ::= (\omega \otimes \mathsf{id}_{(\vec{X})})DR & \textit{regular bigraph}
\end{array}
$$

Figure 9: Normal forms for binding bigraphs.

## 4.4 Bigraphs

Bigraphs are built from elementary bigraphs and operators:

| Ions: | (`K,L,M` : `control` ; `x,y,p,p'` : `string`) |
|---|---|
| `K[y,...]` | Ion with control of global arity |
| `L[y,...][[x,...],...]` | Ion with control of global/local arity |
| `K[p==y,...]` | Ion with control of global arity with named ports |

| | |
|---|---|
| `L[p==y,...][p'==x,...]` | Ion with control of global/local arity with named ports |

Wirings: (`x,y : string`)

| | |
|---|---|
| `y/x` | Renaming link |
| `y//[x,...]` | Substitution link |
| `y//[]` | Name introduction |
| `-/x` | Closure edge |
| `-//[x,...]` | Multiple closure edges |
| `idw[x,...]` | Identity wiring |

Concretions: (`n >= 0` ; `x : string`)

| | |
|---|---|
| `` `[x,...]` `` | Concretion of names `x`,... |

Merges: (`n >= 0`)

| | |
|---|---|
| `merge(n)` | Merge of inner width n |
| `<->` | Barren root (= `merge 0`) |

Permutations: (`0 <= i`$_k$ `< m` ; `x : string`)

| | |
|---|---|
| `@[..., i`$_k$`, ...]` | Permutation mapping site $k$ to root `i`$_k$ |
| `@@[..., i`$_k$`&[x,...], ...]` | Permutation with local names |
| `idp(m)` | Identity permutation of width m |

Operators: (`x : string` ; `A,B : bgval` ; `P : prime bgval`)

| | |
|---|---|
| `<[x,...]> P` | Abstract names x,... of a prime P |
| `A * B` | Tensor product |
| `A \|\| B` | Parallel product |
| `A` `` `\|` `` `B` | Prime product |
| `**[A,...]` | Tensor procuct of n factors |
| `\|\|\|[A,...]` | Parallel product of n factors |
| `` `\|`[A,...] `` | Prime product of n factors |
| `A o B` | Composition |

Precedence: (`x : string` ; `P : prime bgval`)

| | |
|---|---|
| `o` | Composition (strongest) |
| `*, \|\|,` `` `\|` `` | Product, left associative |
| `<[x,...]> P` | Abstraction (weakest) |

Bigraphs built using these combinators have the SML type `bgval`. Note that since the BPL Tool binds bigraph composition to '`o`', we rebind function composition to '`oo`'.

### 4.4.1 Syntactic Sugar

The BPL Tool allows some of the syntactic shorthands used in the bigraph literature – the shorthands for each relevant combinator are as follows:

**abstraction:** In an abstraction $(X)P$, one may abstract names that are not in the outer face of $P$ and we allow abstractions on name introductions. For example, `<[x]> y//[]` is allowed. The desugared form of such an abstraction is

$$(X)P \stackrel{\text{def}}{=} (X)(P \otimes Y \otimes \mathsf{id})$$

where $Y$ are the names of $X$ which are not in the outer face of $P$ and $\mathsf{id}$ is either $\mathsf{id}_1$ if $\mathsf{width}(P) = 0$ and $\mathsf{id}_0$ otherwise. Thus, `<[x]> y//[]` $\stackrel{\text{def}}{=}$ `<[x]>(y//[] * x//[] * idp(1))`.

**composition:** The BPLL composition operator is a generalization of Milner's nesting operator: in a composition $A \circ B$ the outer names of $A$ and $B$ may be shared. For example, `K[x] o K[x]` is allowed. The desugared form of such a composition is

$$A \circ B \stackrel{\text{def}}{=} (A \parallel \mathsf{id}_X) \circ B$$

where $X$ are the outer names of $B$. Thus, `K[x] o K[x]` $\stackrel{\text{def}}{=}$ `(K[x] || idw[x]) o K[x]`.

Also, it allows implicit abstraction of names in primes: in a composition $A \circ P$ where $P$ is prime, the local inner names of $A$ that are not local in the outer face of $P$ will be abstracted. The desugared form of such a composition is

$$A \circ P \stackrel{\text{def}}{=} A \circ ((X)P)$$

**ion:** The global ports of an ion $K_{\vec{y}}$ are allowed to use the same name, i.e. the names of $\vec{y}$ need not be distinct. For example, `K[x,x]` is allowed. The desugared form of such an ion is

$$K_{\vec{y}(\vec{X})} \stackrel{\text{def}}{=} (\omega \otimes \mathsf{id}_1) \circ K_{\vec{z}(\vec{X})}$$

where $\vec{z}$ is a vector of $n$ distinct names, $\mathsf{ar}(K) = m \to n$, and $\omega : \{\vec{z}\} \to \{\vec{y}\}$ is a substitution satisfying $\vec{y}_i = \omega(\vec{z}_i)$ $(i \in n)$. Thus, `K[x,x]` $\stackrel{\text{def}}{=}$ `(x//[x,y] * idp(1)) o K[x,y]`.

Note that the BPL Tool will do its best (subject to the configuration options discussed in Section 4.12) to use the syntactically sugared forms whenever possible.

Also, the BPL Tool internally works on bigraph terms in the normal forms shown in Figure 9. Such terms are not easily readable, in particular because `` `|` `` and `||` are treated as derived operators. The BPL Tool will try (again subject to configuration options) to simplify the terms and use `` `|` `` and `||` whenever possible.

## 4.5 Bigraph Operations

| | | | |
|---|---|---|---|
| `===` | : | `bgval * bgval -> bool` | Equality (infix) |
| `====` | : | `'a bgbdnf * 'a bgbdnf -> bool` | Equality (infix) |
| `norm_v` | : | `bgval -> B bgbdnf` | Normalise |
| `denorm_b` | : | `'a bgbdnf -> bgval` | Denormalise |
| `regl_v` | : | `bgval -> BR bgbdnf` | Regularise |
| `regl_b` | : | `B bgbdnf -> BR bgbdnf` | Regularise |
| `simpl_v` | : | `bgval -> bgval` | Attempt to simplify |
| `simpl_b` | : | `'a bgbdnf -> bgval` | Attempt to simplify |

## 4.6 Matching

Matching is computationally intensive, so the BPL Tool uses lazy lists to represent sets of matches.

`match_v` : `{agent:bgval, redex:bgval} -> match lazylist`
Match redex in agent, returning a lazy list of matches.

`match_b` : `{agent:B bgbdnf, redex:B bgbdnf} -> match lazylist`
Match redex in agent, returning lazy list of matches.

`print_mv` : `match lazylist -> unit`
Print lazy list of matches.

`print_mb` : `match lazylist -> unit`
Print lazy list of matches.

`print_mtv` : `match lazylist -> unit`
Print lazy list of matches with trees.

`print_mtb` : `match lazylist -> unit`
Print lazy list of matches with trees.

## 4.7 Lazy lists

The main functions for working with lazy lists are the following; see the online API for a complete list [7].

`lznull` : `'a lazylist -> bool`
Test whether the lazy list is empty.

`lzhd` : `'a lazylist -> 'a`
Return the first element of a lazy list.

`lztl` : `'a lazylist -> 'a lazylist`
Return the tail of a lazy list.

`lzunmk` : `'a lazylist -> 'a lazycell`
Return the head and tail of a lazy list, or Nil if it is empty.

`lzmap` : `('a -> 'b) -> 'a lazylist -> 'b lazylist`
Map a function on all elements of a lazy list.

## 4.8 Reaction rules

Reaction rules are constructed using the following combinators:

Instantiations: ($i_k$,$j_k$ : `int` ; $x_k$,$y_k$ : `string`)

`[..., `$i_k$ `|-> `$j_k$`, ...]`
Instantiation mapping reactum site $i_k$ to redex site $j_k$

`[..., `$i_k$`&[`$x_0$`, ..., `$x_{m-1}$`] |--> `$j_k$`&[`$y_0$`, ..., `$y_{m-1}$`], ...]`
Instantiation mapping local reactum name $x_k$ to redex name $y_k$

Rules: (`R`,`R'` : `rule` ; `rho` : `instantiation` ; `N` : `string`)

`R ----|> R'` Rule with redex R, reactum R' and default instantiation
`R --rho--|> R'` Rule with redex R, reactum R' and instantiation rho
`N ::: R ----|> R'` Named rule

Operators on rules: (`R` : `rule`)

`redex R` Extract the redex of a rule
`reactum R` Extract the reactum of a rule
`inst R` Extract the instantiation of a rule

For convenience, instantiations in rules need not be fully specified; if an instantiation $\rho : J \to I$, where $J$ and $I$ are the reactum and redex innerfaces respectively, is partially specified, the BPL Tool will automatically add missing mappings as follows:

1. if a site of $J$ is not mentioned, it is assumed to map to the same site at $I$, inferring the name map as in (2);
2. if the name lists of a map are empty, the local renaming will be inferred as follows:

(a) if the relevant sites of $I$ and $J$ have the same local names, an identity renaming will be used;

(b) otherwise, if there is only one local name at both sites, say $x$ at $I_i$ and $y$ at $J_j$, the local renaming $(y)/(x)$ will be used.

An exception will be raised if this procedure not yield an instantiation.

## 4.9 Simulation

Tactics: (`i` : `int` ; `N` : `string` ; $\texttt{t}_i$ : `tactic`)

| | |
|---|---|
| `react_rule N` | Apply rule `N` |
| `react_rule_any` | Apply any rule |
| `roundrobin` | Apply rules roundrobin until none match |
| $\texttt{t}_1$ `++` $\texttt{t}_2$ | Use $\texttt{t}_1$, then $\texttt{t}_2$ |
| `TRY` $\texttt{t}_1$ `ORTHEN` $\texttt{t}_2$ | If $\texttt{t}_1$ fails, use $\texttt{t}_2$ on its result |
| `IF` $\texttt{t}_1$ `THEN` $\texttt{t}_2$ `ELSE` $\texttt{t}_3$ | If $\texttt{t}_1$ finishes, use $\texttt{t}_2$, else $\texttt{t}_3$ on its result |
| `REPEAT t` | Repeat `t` until it fails |
| `i TIMES_DO t` | Use `t` `i` times |
| `finish` | Finish tactic |
| `fail` | Fail tactic |

Reaction operations: (`v` : `bgval`; `m` : `match`; $\texttt{r}_i$ : `rule` ; $\texttt{N}_i$ : `string` ; `rs` : `rules`)

| | |
|---|---|
| `react m` | Perform a single reaction step |
| `mkrules` [$\texttt{r}_0$, ..., $\texttt{r}_n$] | Construct a rule map |
| `mknamedrules [...,` ($\texttt{N}_i$, $\texttt{r}_i$)`, ...]` | Construct a rule map with explicit names |
| `matches rs v` | Return lazy list of all matches of all rules |
| `run rs t v` | Perform agent reactions using a tactic |
| `steps rs t v` | Return agent for each step using a tactic |
| `stepz rs t v` | Return lazily agent for each step using a tactic |

## 4.10 Pretty printing

Bigraphs:

| | | | |
|---|---|---|---|
| `str_v` | : | `bgval -> string` | Return as a string |
| `str_b` | : | `'a bgbdnf -> string` | Return as a string |
| `print_v` | : | `bgval -> unit` | Print to stdOut |
| `print_b` | : | `'a bgbdnf -> unit` | Print to stdOut |

Matches:

| | | | |
|---|---|---|---|
| `print_mv` | : | `match lazylist -> unit` | Print lazy list of matches |
| `print_mb` | : | `match lazylist -> unit` | Print lazy list of matches |
| `print_mtv` | : | `match lazylist -> unit` | Print lazy list of matches with trees |
| `print_mtb` | : | `match lazylist -> unit` | Print lazy list of matches with trees |

Rules:

| | | | |
|---|---|---|---|
| `str_r` | : | `rule -> string` | Return rule as a string |
| `print_r` | : | `rule -> unit` | Print rule |

## 4.11 Visualization

Configuration:

`makecfg` `(string * BG.PPSVG.path -> configinfo) -> config`
Construct a config

`unmkcfg` `config -> string * BG.PPSVG.path -> configinfo`
Deconstruct a config

`defaultcfg` `config`
Default config

Scalable Vector Graphics (SVG):

`svg_v` `config option -> bgval -> string`
Return as SVG fragment string

`svg_b` `config option -> B bgbdnf -> string`
Return as SVG fragment string

`svg` `bgval -> string`
Return as SVG fragment string

`svgdoc_v` `config option -> bgval -> string`
Return as SVG document string

`svgdoc_b` `config option -> B bgbdnf -> string`
Return as SVG document string

`svgdoc` `bgval -> string`
Return as SVG document string

`outputsvgdoc_v` `string -> config option -> bgval -> unit`
Output as SVG document to file

`outputsvgdoc_b` `string -> config option -> B bgbdnf -> unit`
Output as SVG document to file

`outputsvgdoc` `string -> bgval -> unit`
Output as SVG document to file

TikZ:

`tikz_v` `real option -> config option-> bgval -> string`
Return as TikZ string

`tikz_b` `real option -> config option -> B bgbdnf -> string`

Return as TikZ string

`tikz` `bgval -> string`
Return as TikZ string

`outputtikz_v` `string -> real option-> config option -> bgval -> unit`
Output as TikZ to file

`outputtikz_b` `string -> real option-> config option -> B bgbdnf -> unit`
Output as TikZ to file

`outputtikz` `string -> bgval -> unit`
Output as TikZ to file

## 4.12 Controlling tool behaviour

The behaviour of the BPL Tool can be modified by changing a number of configuration *flags*. Flags are accessed by two families of functions:

| | |
|---|---|
| `Flags.get`*Type*`Flag "`*name*`"` | Get the value of the named flag of the given type |
| `Flags.set`*Type*`Flag "`*name*`"` *value* | Set the value of the named flag of the given type |

The help function for flags displays and explains all the available flags as well as their current and default values:

```
help["flags"];
```

**Matching:**

| **name** | **type** | **description** |
|---|---|---|
| `/kernel/match/match/nodups` | `bool` | Remove duplicate matches. |

**Miscellaneous:**

| **name** | **type** | **description** |
|---|---|---|
| `/debug/level` | `int` | Level of debugging information (0 = no info, >0 = info). |
| `/dump/prefix` | `string` | Filename prefix for pretty print dumps to a file. |
| `/misc/timings` | `bool` | Enable timings. |

**Pretty printing:**

| name | type | description |
|---|---|---|
| /misc/indent | int | Set extra indentation at each level when prettyprinting to N. |
| /misc/linewidth | int | Set line width to W characters. |
| /kernel/ast/bgterm/pp0abs | bool | Explicitly display empty-set abstractions (ignored if ppabs is false). |
| /kernel/ast/bgterm/ppabs | bool | Explicitly display abstractions (abstractions on roots are always displayed). |
| /kernel/ast/bgterm/ppids | bool | Explicitly display identities in tensor and parallel products. |
| /kernel/ast/bgterm/ppmeraspri | bool | Replace merge with prime product (best effort). |
| /kernel/ast/bgterm/pptenaspar | bool | Replace tensor product with parallel product. |
| /kernel/ast/bgval/pp-merge2prime | bool | Substitute \| for ‖ by removal of merges before prettyprinting. |
| /kernel/ast/bgval/pp-simplify | bool | Simplify BgVal terms before prettyprinting. |
| /kernel/ast/bgval/pp-tensor2parallel | bool | Substitute ‖ for * by removal of y//X's before prettyprinting. |
| /kernel/bg/name/strip | bool | Strip trailing _xx off input names (xx are hex digits). |
| /kernel/match/rule/ppsimplereactum | bool | Simplify reactum when displaying rules. |
| /kernel/match/rule/ppsimpleredex | bool | Simplify redex when displaying rules. |

For convenience, one can switch the use of syntactic shorthands on and off with a single command:

use_shorthands on/off

This will modify the following flags appropriately:

```
/kernel/ast/bgterm/ppids
/kernel/ast/bgterm/ppabs
/kernel/ast/bgterm/pp0abs
/kernel/ast/bgterm/pptenaspar
/kernel/ast/bgterm/ppmeraspri
```

## 4.13 Exceptions

Exceptions can be explained by the BPL Tool using the following command:

`explain exn -> 'a` Explain exception in detail and raise it again

If the debug level is greater than 0, and the SML interpreter supports it, the exception history will also be printed.

# 5 Conclusions and Future Work

We have introduced the BPL Tool, a first implementation of bigraphical reactive systems with binding, and have demonstrated its use by modeling a simple mobile phone system.

Our research group has used the BPL Tool to successfully model a number of sytems (cf. Section 1). Our experience is, that the BPL Tool is that it is quite useful for modeling as it validates well-formedness of terms and rules, and its visualization capabilities, in particular through the web interface, provides a good overview of reaction rules.

However, there is also room for improvement:

- The tool would benefit from a more complete graphical user interface than what the web interface provides. One approach would be to extend Big Red [3] as follows:
  - add support for binding,
  - add facilities for modeling reaction rules, and
  - add simulation facilities, by using the BPL Tool as a simulation backend.
- The BPLL syntax is a bit heavy due to the fact that it is embedded in Standard ML. By building a dedicated command line interface one would be free to choose a simpler syntax. The BPL Tool code base already contains a parser for an older version of BPLL, so the main task is to implement an interactive prompt. The compromise would of course be that end-users will have a harder time extending the tool.
- The implementation of matching is not very fast, due to the fact that it is derived directly from the inductive characterization of matching which is based on the binding discrete normal form. The main issues are the following:
  - Structural congruence is currently handled naïvely: when matching children of a node, one need to find partitions and permutations and the BPL Tool simply generates them all.
  - Matching currently follows the place graph structure, and the link graph is only matched at the root and leaves of the matching inference tree. By interleaving the matching of the two graphs in a more fine-grained manner, one could probably prune the search space significantly; this would perhaps be easier if one based matching on a *connected normal form* where edges are as close to their constituent points as possible instead of being at the outermost level.
  - Only one redex is matched at a time, as this is the algorithm that naturally falls out of the inductive characterization of matching. By matching all redexes simultaneusly, only one traversal of the agent term would be necessary.

  However, while we believe the suggested improvements are significant, we believe that more efficient matching will be achieved by using SAT-solvers, which is currently being investigated by Sevegnani et al. [21], or by the graph embedding based approach of Højsgaard et al. [20]. Note that matching is NP-complete [20] and thus no efficient algorithm exists unless P = NP.

- From a modeling perspective, is would be convenient if the BPL Tool was extended with support for sortings of some kind, such that modellers could specify the structure of well-formed bigraphs and then have the BPL Tool verify well-formedness of agents and rules and that the latter preserves well-formedness.
- Similarly, built-in support for datatypes and manipulation of data would make it easier to express models containing computations. We suggest that such that an extension should be founded on a solid formal foundation, such as the calculational bigraphical reactive systems of Debois [10].

# References


[1] Giorgio Bacci, Davide Grohmann, and Marino Miculan. DBtk: A toolkit for directed bigraphs. In *CALCO*, pages 413–422, 2009.

[2] Jørgen Eske Runge Bentzen. Master's thesis, IT University of Copenhagen, 2007.

[3] Big Red. `http://www.itu.dk/research/pls/wiki/index.php/Big_Red`, 2010.

[4] BigMC – Bigraphical Model Checker. `http://bigraph.org/bigmc/`.

[5] Lars Birkedal, Troels Christoffer Damgaard, Arne John Glenstrup, and Robin Milner. Matching of bigraphs. *Electronic Notes in Theoretical Computer Science*, 175(4):3–19, 2007.

[6] Lars Birkedal, Søren Debois, Ebbe Elsborg, Thomas Troels Hildebrandt, and Henning Niss. Bigraphical models of context-aware systems. In Luca Aceto and Anna Ingólfsdóttir, editors, *Proceedings of the 9th International Conference on Foundations of Software Science and Computation Structure*, volume 3921 of *LNCS*, pages 187–201. Springer-Verlag, March 2006.

[7] BPL Tool. `http://www.itu.dk/research/pls/wiki/index.php/BPL_Tool`.

[8] Mikkel Bundgaard, Arne John Glenstrup, Thomas Hildebrandt, Espen Højsgaard, and Henning Niss. Formalizing WS-BPEL and higher order mobile embedded business processes in the bigraphical programming languages (BPL) tool. Technical Report TR-2008-103, IT University of Copenhagen, 2008.

[9] Troels Christoffer Damgaard and Lars Birkedal. Axiomatizing binding bigraphs. *Nordic Journal of Computing*, 13(1–2):58–77, 2006.

[10] Søren Debois. Computation in the informatic jungle. Draft, 2011.

[11] Niklas Eén and Niklas Sörensson. MiniSAT. `http://minisat.se`.

[12] Ebbe Elsborg. *Bigraphs: Modelling, Simulation, and Type Systems*. PhD thesis, IT University of Copenhagen, 2009.

[13] Arne John Glenstrup, Troels Christoffer Damgaard, Lars Birkedal, and Espen Hjsgaard. An implementation of bigraph matching. Technical Report TR-2010-135, IT University of Copenhagen, December 2010.

[14] Chris Greenhalgh. bigraphspace. `http://bigraphspace.svn.sourceforge.net/`, 2009.

[15] Ole Høgh Jensen. Mobile processes in bigraphs. Available at `http://www.cl.cam.ac.uk/~rm135/Jensen-monograph.html`, 2006.

[16] Ole Høgh Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report UCAM-CL-TR-580, University of Cambridge, February 2004.

[17] James Judi Leifer and Robin Milner. Transition systems, link graphs and Petri nets. Technical Report UCAM-CL-TR-598, University of Cambridge, August 2004.

[18] Robin Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, 1999.

[19] Robin Milner. *The Space and Motion of Communicating Agents.* Cambridge University Press, 2009.

[20] Stochastic Bigraphical Abstract Machine (SBAM). `http://www.itu.dk/research/pls/wiki/index.php/Stochastic_Bigraphical_Abstract_Machine_%28SBAM%29`.

[21] M. Sevegnani, C. Unsworth, and M. Calder. A SAT based algorithm for the matching problem in bigraphs with sharing. Technical Report TR-2010-311, University of Glasgow, Department of Computing Science, 2010.