# Contents

1 Introduction 5

2 Processes, Channels and Communication: The II Calculus in SpiFcp Syntax 6
   2.1 Simple Processes 6
   2.2 Channels and communication 7
      2.2.1 Public Channels 7
      2.2.2 Choice 8
      2.2.3 Scopes and Private Channels 8
      2.2.4 Comparison 10
   2.3 Parametric and Recursive Processes 10

3 Getting Started with SpiFcp 11
   3.1 Hello 11
   3.2 Which 12
   3.3 Boolean And 13
   3.4 Modular Boolean And 14

4 Stochastic Programs 16
   4.1 Stochastic II Calculus 16
   4.2 Simple stochastic programs 17

5 Testing, Tracing and Debugging 18
   5.1 Interruption and Inspection 19
   5.2 Debugging 19
      5.2.1 Trace a Computation 19
      5.2.2 Interrupt and Inspect a Computation 21
   5.3 Tree trace 22

6 Ambient Stochastic II Calculus Programs 24
   6.1 Channels 24
   6.2 Ambient Declaration 24
   6.3 Ambient Stochastic II Calculus Processes 25
      6.3.1 Inter-Ambient Communication 25
6.3.2 Capability Communication ........................................ 25
6.4 Testing Ambient Programs ............................................. 26
6.5 Interruption and Inspection ............................................. 26

7 Using FCP in BioSpi modules .............................................. 27
  7.1 LOGIX Variables as Counters and Limits ............................. 27
  7.2 Semi-LOGIX Processes ................................................ 28

A Syntax and Semantics ..................................................... 29
  A.1 BNF for Stochastic Π Calculus ..................................... 29
  A.2 Alternate BNF for Ambient Stochastic Π Calculus ............... 35
  A.3 Primitives ............................................................ 36
  A.4 Semantics ............................................................. 36
  A.5 Logix Terms .......................................................... 37
  A.6 Notes ................................................................. 38

B BioSpi commands .......................................................... 41
  B.1 Channel Management and Message Transmission Macros ....... 41
    B.1.1 Create Channel - pc ........................................... 41
    B.1.2 Send Message - ps ............................................. 42
    B.1.3 Receive a Message - pr ..................................... 42
    B.1.4 Set Default Weighter - weighter ........................... 42
  B.2 Program Execution Macros ......................................... 42
    B.2.1 Execute Goals - run .......................................... 43
    B.2.2 Execute Goals - run .......................................... 43
  B.3 Execution Control ................................................... 43
    B.3.1 Suspend Execution - suspend ................................. 43
    B.3.2 Resume Execution - resume .................................. 43
    B.3.3 Abort Execution - abort ..................................... 44
  B.4 Debugging SpiFcp Processes ....................................... 44
    B.4.1 Set Display Options - options ................................ 44
    B.4.2 Show a Channel - spc ........................................ 44
    B.4.3 Show Goal - spg .............................................. 45
  B.5 Debugging SpiFcp ..................................................... 45
    B.5.1 Show Resolvent - spr ........................................ 45
    B.5.2 Display Communicating Channels - cta ...................... 45
    B.5.3 Debug a SpiFcp Goal - pdb .................................. 45
    B.5.4 Create an Execution Tree - vtree ........................... 45
  B.6 Debugging ............................................................. 46
    B.6.1 Show Ambient Tree - atr,ctr ................................ 46
    B.6.2 Show Resolvent - rtr ........................................ 47
  B.7 Miscellaneous Macros ................................................ 47
    B.7.1 Reset the System - reset ..................................... 47
    B.7.2 Input Commands - input ...................................... 47
    B.7.3 Call a UNIX Command { ...} .................................. 48
    B.7.4 Display Named Variables - .................................. 48

3
B.7.5 Change Current Computation .......................... 48

C Auxilliary LOGIX Procedures 49
C.1 record .................................................. 49
C.2 repeat .................................................... 51
C.3 Weight Computation for a Channel ......................... 52
Chapter 1

Introduction

Π Calculus is an abstract model of concurrent communication developed in the late 1980s to express the behavior of mobile systems [2]. This document describes an implementation of Π Calculus and its stochastic extension in Flat Concurrent Prolog (FCP) [6]. The implementation is embedded in The Logix System (LOGIX) [8, 7]. Unlike most previous implementations (e.g. Pict [3]), we implement the fully synchronous calculus, where a send and its corresponding receive are completed simultaneously. We further extend it to the stochastic variant ([4],[5]), where communication actions are assigned rates, and communication events are selected on a probabilistic basis, rather than a non-deterministic one. The SpiFcp system is developed as part of the BioSpi project, whose goal is to study biomolecular systems as concurrent processes. Therefore, we adhere closely to the concise Stochastic Π Calculus, and only rarely add functionalities beyond its original core.

The tutorial is constructed as follows: In Chapter 2, we informally introduce the basic entities and operators of the Π Calculus, and their appropriate representation in SpiFcp syntax. The full correspondence between the SpiFcp syntax and Π Calculus is given in Appendix A.1. We follow, in Chapter 3, with several simple programs, their compilation and execution. Chapter 4 extends these example to the stochastic variant. We then describe, in Chapter 5, the available testing, tracing and debugging tools. Chapter 6 describes the extension to Ambient Stochastic Π Calculus. Chapter 7 explains how LOGIX procedures, guard and goal predicates may be used to perform arithmetic computations and to produce other useful side-effects.

The full set of commands for both systems, is in Appendix B.
Chapter 2

Processes, Channels and Communication: The $\Pi$ Calculus in SpiFcp Syntax

$\Pi$ Calculus describes systems of agents which exist in parallel. These agents communicate with each other by sending and receiving messages on channels identified by names\(^1\). A message may be a simple signal or a tuple of $\langle$channel$\rangle$ names\(^2\), which can be used by the receiving process for further communication. This behavior, which allows the topology of communication networks to change over time, is called mobility.

2.1 Simple Processes

The basic unit of computation is the Process. The simplest process, 0, has no observable behavior. To make this into a SpiFcp program, we need to declare the process, by assigning it a Capitalized name. The LOGIX language attribute is also required as the first line of a module, containing a SpiFcp program. We will omit it in the remaining examples in this chapter.

- language(spiFcp).
  Try ::= 0 .

Systems are composed of multiple concurrent processes, which are composed in parallel. The $PAR$ operator ( $|$ ) is used for such composition. Each of the sub-processes must be declared separately.

\(^1\)A channel name is a string of letters, numbers and underscores, beginning with a lower-case letter.

\(^2\)In Chapter 7 we extend the definition of a message to include LOGIX variables.
Try ::= Try_again | Try_another .
Try_again ::= x! , 0 .
Try_another ::= ?x , 0 .

2.2 Channels and communication

2.2.1 Public Channels

What processes mostly do is communicate with each other. This is done on
channels, on which processes may send messages to each other. A Channel is
denoted by a name, starting with a lower-case letter. There are several kinds of
channels. We start with public channels; a public channel is common to all the
processes in the system for which it is declared. Public channels are declared
for the processes of a program (file) by a public attribute at the beginning of
the program.

Processes communicate on channels by sending and receiving messages. The
send action \( x ! [\ ] \)\(^3\) denotes a process that sends a signal, \( [\ ] \) on the channel
\( x \). This transmission can be completed when another concurrent process is
ready to receive a signal on the same channel, as for example by the receive
action \( x ? [\ ] \)\(^4\).

When a transmission is completed, the constituent actions of the sending and
receiving processes are no longer available. The sending and receiving processes
are released together - communication is synchronized. The following action in
each process may now begin. A sequence of actions is separated by the infix
comma operator ( , ). The last action in a sequence is delimited from the
continuation of the process by a comma. The 0 process terminates with no
further action.

\[
\text{public}(x).
\]

Try ::= Try_again | Try_another .
Try_again ::= x! , 0 .
Try_another ::= ?x , 0 .

A signal, or nil message, is the simplest message; it can be used for synchro-
nization. Messages with content can be sent as well. This content is a tuple of
one or more channels, which the receiver can use for further communications.
For example, in the following programs, the Try_again process sends the
channel \( z \) (by the action \( x ! \{ z \} \)) to Try_another , which uses channel
\( z \) in its communication with Another_try . Note, that channel \( w \) is not
declared; it is dynamically bound by the receive action.

\(^3\)Sending a signal \((x ! [\ ])\) may be abbreviated \( x! \).
\(^4\)Receiving a signal \((x ? [\ ])\) may be abbreviated \( ?x \).
public(x, z).

Try := Try_again | Try_another | Another_try .
Try_again ::= x ! {z} , 0 .
Try_another ::= x ? {w} , w!, 0 .
Another_try ::= ?z , 0 .

Tuples of more than one channel can be sent and received as messages - e.g. x ! {z, y, w} , x ? {a, b, c} .

2.2.2 Choice

A process may offer more than one communication. We have already seen that several sub-processes, each with a different communication may be spawned concurrently, using the infix PAR operator. A normal process may offer several mutually exclusive communications, by using the infix choice operator ( ; ). When one communication is completed, all other offers of that process are discarded. A normal process may be constructed either by the choice operator between sequences of communication actions, or by the sum operator (+) from simple normal processes. Having the syntactic ability to sum either actions or processes can often simplify programs considerably, as we shall see in the discussion of recursive processes. In the following example Try_again is declared using the choice operator, while Try_another is declared with the sum operator.

public(w, x, y, z).

Try ::= Try_again | Try_another | Another_try .
Try_again ::= x ! {z} , 0 ;
y ! {w} , 0 .
Try_another ::= First_try + Second.Try .
First.try ::= x ? {a} , a! , 0 .
Second.try ::= y ? {a} , a! , 0 .
Another_try ::= ?z , 0 .

Note, that Try_again and Try_another can interact either on channel x or on channel y, with different outcomes. In the original Π Calculus this choice is resolved in a non-deterministic way. Later we will see how it can be resolved in a probabilistic way in the stochastic variant.

2.2.3 Scopes and Private Channels

The scope of communication may be restricted to a particular process, by declaring a new private channel in a process; the private channel is known only within the declaring process and its declared sub-processes, and is distinguished from any other channel (public or private) with the same name. Importantly, the
scope of a private channel may be expanded by sending it to outside processes, an event called scope extrusion. Private channels can be declared during process declaration, using the + operator. For example, the private channel $x$ is declared in Try_again below. It is distinct from and cannot communicate with the public channel $x$ in Another_try.

public($w, x, y, z$).

Try ::= Try_again | Try_some_more | Another_try.

Try_again + $x ::= y !\{x\}, (x !\{w\}, 0 ; y !\{z\}, 0 ).$

Another_try ::= ?$x , 0 .$

Process scopes may be restricted as well. In this case, a local process can only be called within the scope in which it was declared. This process scoping, which is not part of the original Π calculus, was added in order to simplify the writing of complex programs. A process scope is delimited by the paired brackets << and >>. For example, First_try and Second_try are scoped within Try_another in the following program; they cannot be referenced in the outer scope.

public($x, y, z$).

Try ::= Try_another | Another_try.

Try_another ::= <<
First_try + Second Try.
First_try ::= x ?\{a\}, a!, 0 .
Second_try ::= y ?\{a\}, a!, 0
>> .

Another_try ::= x !\{z\}, ?z , 0 .$

Note that the last process within the scope is delimited by >>, followed by a full stop ( . ) to end the declaration. See Appendix A.1 for further details.

A delimited process scope can also be used for the declaration of new channels, without the need of an explicit process declaration. This possibility is often very convenient. Here, the new predicate is subsumed into a prefix <parameters>, declaring the private channels $x, y$.

Try ::= <<
$x, y. First_try + Second Try |$
Another_try(x) | Another_try(y) .
First_try ::= x ?\{a\}, a!, 0 .
Second_try ::= y ?\{a\}, a!, 0
>> .

Another_try(x) + $b ::= x !\{b\}, ?b , 0.$
2.2.4 Comparison

Sequences of actions may also include comparison actions in addition to communication actions. This provides an if-then-else construct based on the comparison of two channels. Similar to communication actions, comparison actions can be disjunctively chosen, to yield a case-like construct. Note, that a full if-then-else structure is required, so an otherwise action must be included. More than one pair of channels may be compared in a single comparison guard. In the following program, Choose receives a 2-tuple of channels ( \{x_1, x_2\} ) on channel w , and selects the continuation by matching them to a pair of channels from x, y, z .

\[
\text{public}(w, x, y, z).
\]

\[
\text{Try ::= Choose | First_choice | Second_choice | Third_choice .}
\]

Choose ::= <<
\[
\text{w ? } \{x_1, x_2\}, <<
\]
\[
\text{x_1 != x & x_2 != y , Choice1 ;}
\]
\[
\text{x_1 != x & x_2 != z , Choice2 ;}
\]
\[
\text{otherwise , Choice3}
\]
\[
\text{>> .}
\]
\[
\text{Choice1 ::= z!, 0 .}
\]
\[
\text{Choice2 ::= y!, 0 .}
\]
\[
\text{Choice3 ::= x!, 0 .}
\]
\[
\text{First_choice ::= w! \{x, y\}, 0 .}
\]
\[
\text{Second_choice ::= w! \{x, z\}, 0 .}
\]
\[
\text{Third_choice ::= w! \{y, z\}, 0 .}
\]

2.3 Parametric and Recursive Processes

In the original Π Calculus, parametric and recursive processes are derived forms, based on a replication operator (Bang). This operator provides an unlimited number of concurrent processes, and is therefore inappropriate for realistic implementations. On the other hand, parametric and recursive process definitions are extremely useful, and are therefore primitive in the SpiFcp syntax, in a straightforward way, as seen in the following example:

\[
\text{public(x).}
\]

\[
\text{Try ::= A(x) .}
\]
\[
\text{A(a) ::= a ? \{b, c\} , B(b, c) .}
\]
\[
\text{B(e, f) ::= e! \{f\} , A(f)}
\]

10
Chapter 3

Getting Started with SpiFcp

In this chapter we present several programs written in SpiFcp.

3.1 Hello

One thing that a process can do is to cause an observable event in the outside world. In our case, this world is LOGIX. For example, the process `Main`, in the program below will cause LOGIX to display the string `Hello World` on the screen.

```
Main ::= screen#display("Hello World").
```

The program declares the process `Main`. In order for it to execute, it must be run. To run this program, put it in a file, e.g. `hello.cp`, preceded by the LOGIX `language` attribute:

```
-language(spifcp).
```

Now enter the UNIX command `spifcp`. When the LOGIX prompt, “@” appears, enter the LOGIX command `hello#"Main"`. The screen will look something like this:

```
% spifcp
Emulated Flat Concurrent Prolog 21/01/12 - 18:58:18
Open Source spifcp 2.4 Enquire about "license", "warranty" !
26/03/12 - 16:01:36

@hello#"Main"
<1> started
```

11
3.2 Which

A slightly more complex program is which. In this program we declare four processes (True, False, Send, and Answer). The process True, for example, calls two sub-processes, Send and Answer, composed in parallel with the PAR operator. Both Send and Answer are parametric processes. Answer, for instance, is called with the channels true and false. These channels are private channels, declared in True by the new (+) operator. Thus, channel true is shared only by the Send and Answer sub-processes spawned by True.

True + (true, false) ::=  
  Send(true) | Answer(true, false) .

False + (true, false) ::=  
  Send(false) | Answer(true, false) .

Send(it) ::=  
  it! , 0 .

Answer(yes, no) ::=  
  ?yes , screen#display("Too true!");
  ?no , screen#display("Too bad!") .

The calls to True and False behave differently. The process Answer receives a nil message ([ ]) on one of its two channels and displays a corresponding result.

@which#"True"
<1> started
  Too true!
<1> terminated
@which#"False"
<2> started
  Too bad!
<2> terminated

1Later when the values are non-zero, their utility will be apparent.
An alternative form of \texttt{which}, using \texttt{public} channels instead of \texttt{new} channels is \texttt{public\_which}. In this case the \texttt{yes} and \texttt{no} channels are declared by the public command, and are common to all the instances of \texttt{Send} and \texttt{Answer}.

\begin{verbatim}
public(yes, no).

True ::= Send(yes) | Answer.
False ::= Send(no) | Answer.
Send(it) ::= it! , 0 .
Answer ::= ?yes , screen#display("Too true!") ;
            ?no , screen#display("Too bad!") .
\end{verbatim}

It has the same behavior as \texttt{which}.

### 3.3 Boolean And

A more complex program, \texttt{booland}, performs a boolean \texttt{AND} computation. In this program we make use of the mobility of the calculus, receiving channels and using them in further communications. For example, The process \texttt{TT} receives two channels on channel \texttt{b}, and transmits a signal on the first of them. Three public channels, \texttt{b1, b2, c}, ensure initial communication between the constituent sub-processes of each \texttt{Run}. Then, the newly declared channels, \texttt{t, f, x}, are transmitted and allow additional interaction, e.g. between \texttt{TT} and \texttt{Test}. The \texttt{export} attribute specifies which processes may be called externally (If it is omitted, all outer scope processes may be called.)

\begin{verbatim}
export(RunTT, RunTF, RunFT, RunFF).
public(b1, b2, c).

TT(b) ::= b ? {t,f} , t! , 0 .
FF(b) ::= b ? {t,f} , f! , 0 .
Test(b) ::= << t, f . b ! {t,f} ,
            << ?f , screen#display("It’s false");
            ?t , screen#display("It’s true") >> >> .
AndB ::= c ? {t,f} , << x . b1 ! {x,f} , ?x , b2 ! {t,f} , 0 >> .
\end{verbatim}

RunTT ::= Test(c) | AndB | TT(b1) | TT(b2) .
RunFT ::= Test(c) | AndB | FF(b1) | TT(b2) .
RunTF ::= Test(c) | AndB | TT(b1) | FF(b2) .
RunFF ::= Test(c) | AndB | FF(b1) | FF(b2) .

The paired brackets \texttt{<<} and \texttt{>>} declare a nested scope in which channels may be declared, and an un-named process is defined and executed.

Try compiling \texttt{booland} and calling the exported processes. Note that \texttt{RunFT} and \texttt{RunFF} do not terminate.

13
You can inspect the state of a computation by entering the SpiFcp macro command `spr` following the LOGIX prompt.

```plaintext
@booland"RunFF"
<1> started
It's false
@spr
<1> suspended
booland # AndB.1.1.comm(Test.1.t, Test.1.f, b2!, AndB.1.x!)
booland # FF.comm(b2!)
```

The two line resolvent of the computation says that a nested sub-process of `AndB` (using channels `t, f, b2, x`) and the process `FF` (using channel `b2`) are both waiting to communicate. By inspection of the program, we see that the action `?x` of `AndB` is waiting to receive a signal, and the action `b ? {t,f}` of `FF` is waiting to receive a 2-tuple of channels. The enhanced channel names indicate the clauses which created them. The suffix exclamation points (!), indicate which channels are active (offering a `<send>` or `<receive>`).

### 3.4 Modular Boolean And

Another way to write the program `booland` is to divide it into separate modules. If a sub-process from a different module is needed by a process, we use the module name in the command, e.g. `boolean#TT(b1)` in the `RunFT` process.

```plaintext
tand.cp
-language(spifcp).

public(b1,b2).

RunTT ::= boolean#TT(b1) | boolean#TT(b2) | btest#Test | band#AndB .
RunFT ::= boolean#TT(b1) | boolean#FF(b2) | btest#Test | band#AndB .
RunTF ::= boolean#FF(b1) | boolean#TT(b2) | btest#Test | band#AndB .
RunFF ::= boolean#FF(b1) | boolean#FF(b2) | btest#Test | band#AndB .

-tnot.cp
-language(spifcp).

public(b).

RunT ::= btest#Test | boolean#TT(b) | bnot#NotB .
RunF ::= btest#Test | boolean#FF(b) | bnot#NotB .
```
boolean.cp
-language(spifcp).

TT(b) ::= b ? \{t,f\}, t!, 0.
FP(b) ::= b ? \{t,f\}, f!, 0.

btest.cp
-language(spifcp).

public(c).

Test+(t,f) ::= c ! \{t,f\},
<<
   ?f , screen#display("It's false") ;
   ?t , screen#display("It's true")
>>.

band.cp
-language(spifcp).

public(b1,b2,c).

AndB ::= c ? \{t,f\} , << x . b1 ! \{x,f\} , ?x , b2 ! \{t,f\} , 0 >> .

bnot.cp
-language(spifcp).

public(b,c).

NotB ::= c ? \{t,f\} , b ! \{f,t\} , 0 .

This makes it easy to add boolean not ( bnot.cp ) and other boolean functions, modularly. The modules may be compiled separately (e.g. by LOGIX command compile(band) ) or by calling an initial process, causing compilation of all modules recursively required by the computation (e.g. LOGIX command tand#RunTF ).
Chapter 4

Stochastic Programs

4.1 Stochastic Π Calculus

In all of the programs in the preceding chapters communication on channels was completed as soon as possible. Furthermore, the choice of the next communication event to occur was non-deterministic. In this chapter we present the stochastic variant, in which communications are assigned different rates. Based on these rates, weighted, random delays are calculated, and used to select a complementary pair of communications to complete and the amount to advance an internal clock. The stochastic version [4], has been modified and implemented by us specifically for biochemical reactions [5].

We distinguish several channel types. Each type corresponds to a different kind of communication or reaction. A bimolecular channel represents a chemical reaction involving two different molecules. Upon declaration (as a public or private channel) it is assigned a non-negative base-rate, which represent the base rate of the reaction. A homodimerized channel represents a chemical reaction involving two molecules of the same kind. It is also assigned a non-negative base rate. It is distinguished however by its use as both an input channel and an output channel in the same choice construct, and cannot be used in any other kind of construct. Finally instantaneous channels do not represent actual reactions, but are rather used for encoding purposes. They are declared with the rate \textit{infinite}.

When a stochastic program is run, an actual rate is calculated for each stochastic channel (reaction) based on its base rate and the number of processes offering to transmit on the channel. A communication offer may be modified optionally by an integer multiplier, adjusting its weight in the calculation; the default multiplier is 1.

For a \textit{bimolecular} channel the default actual rate is:

\[ \text{baserate} \times \sum (\text{receivemultipliers}) \times \sum (\text{sendmultipliers}) \]

For a \textit{homodimerized} channel the default actual rate is:
\[
\frac{\text{baserate} \times \sum \text{multipliers} \times (\sum \text{multipliers} - 1)}{2}
\]

Using these actual rates, the next time step and the next communication are selected, by a standard algorithm [1].

The default actual rate may be computed by a custom computation, by specifying an explicit computation function (\texttt{<weighter>}).

### 4.2 Simple stochastic programs

The programs described in this tutorial and interesting programs submitted by users are in sub-directory \texttt{Examples}. Heavy annotation is encouraged to allow the programs to serve pedagogical purposes.
LOGIX supports computations. A computation is initiated by a Remote Procedure Call (RPC). We saw some examples of this in previous sections. In general, an RPC has the form:

LogixPath#LogixGoal

LogixPath may be the name of a module or a transformation of a UNIX path:

\[
dir_1/dir_2/\cdots/dir_n/module.cp \Rightarrow dir_1#dir_2#\cdots#dir_n#module
\]

where \(dir_1\) is a directory which is an immediate sub-directory of the current directory, the current directory itself, or a directory which contains the current directory. Since LOGIX treats an alphanumeric name which begins with a lower-case letter as a string, such directory and module names need not be quoted; SpiFcP and BioSpi treat alphanumeric names within a module which begin with an uppercase letter and appear in the place of a functor as strings - all other names, which are not intended as \texttt{<logix_variables>} (alphanumeric, beginning with an upper-case letter or underscore), should be quoted - e.g. a pathologically complicated example:

\[
tests#Cases"#13"#sub_cases#Module#"TestIt"(3)
\]

The terms tests, #13 and sub_cases are the names of unix directories along the path to the process TestIt; Cases and Module are \texttt{<logix_variables>} which should be instantiated to directory and module names, respectively. Because the two sub-systems are imbedded within LOGIX, TestIt should be quoted in command lines entered via the keyboard\(^1\).

LogixGoal may be any atom, referring to a process which is exported by the target module.

\(^1\) However, see B.2.1
5.1 Interruption and Inspection

An SpiFcp (or BioSpi) program may reach an impasse (e.g. boolean#RunFF above), reach a limit (see B.2.1 and B.2.2), be interrupted manually (see B.3.1) or terminate normally. At an impasse, limit or interruption, the current state of the computation can be inspected by the command “spr” (see B.5.1), and the communicating channel set can be inspected by the command “cta” (see B.5.2).

5.2 Debugging

A computation may be run under control of a debugger.

pdb(RPC)

The computation may be traced, interrupted at break points and inspected. Here are some simple examples using the module boolean - see Section 3.3.

5.2.1 Trace a Computation

Following a query -> prompt, enter the command trace.

```plaintext
@pdb(boolean#RunTF)
<1> started
boolean Debug Reduction Started
boolean RunTF ::= ?
query -> trace
boolean RunTF ::= RunTF.0.
boolean RunTF.0(b1, b2, c) ::= Test(c) | AndB(b1, b2, c) | TT(b1) | FF(b2).
boolean Test(?c) ::= Test.1(?c).
boolean AndB(?b1, ?b2, ?c) ::= AndB_(?b1, ?b2, ?c).
boolean TT(?b1) ::= TT_(?b1).
boolean FF(?b2) ::= FF_(?b2).
boolean Test.1(?c) ::= Test.1.(?c, Test.1.f, Test.1.t).
boolean Test.1.(c, Test.1.f, Test.1.t) ::= Test.1._(c, Test.1.f, Test.1.t).
boolean Test.1._(c, Test.1.f, Test.1.t) ::= Test.1.1(Test.1.f, Test.1.t).
boolean AndB_(?b1, ?b2, c) ::= AndB.1(Test.1.t, Test.1.f, ?b1, ?b2).
```
Each traced reduction above has the form:

```
booland <process_name>(<arguments>) ::= <spifc_body>
```

Channels which are annotated with a prefix question mark (?) or a suffix exclamation mark (!) are active at the time at which the reduction is displayed. Channels which are displayed within square brackets have been closed, and are no longer available for communication.

The suffixes following many process names, indicate a derived sub-process, which may result from prefix guards or from new scope declarations. Where the suffix ends in “.”, the sub-process is one which completes a communication.

---

2The debugger displays the reductions after they have occurred, but not synchronized with the reduction itself.

3However, the channel may have been open at the time that the process was reduced.
Where a process is reduced to multiple parallel processes, they are displayed separated by commas, instead of PARs.

5.2.2 Interrupt and Inspect a Computation

Following a query -> prompt, enter a break command. Following the next query -> prompt, press <enter>.

@pdb(booland#RunFT)
<1> started
booland Debug Reduction Started
booland RunFT :- ?
query -> break("AndB")
booland RunFT :- ?
query ->
booland AndB(?b1, ?b2, c) ::= 
       AndB_(?b1, ?b2, c).
query ->

When the computation reaches the specified process, the debugger prompts "query -> "; entering debug, brings the debugger to inspection mode. Following the debug? -> prompt, entering resolvent, produces a list of goals which have not yet been reduced. Entering resume turns off suspension; entering query, returns to execution mode, where pressing <enter> continues the computation.

@debug? -> resolvent
booland Debug Reduction Suspended
booland goal - 1 Test.1.1_(?Test.1.f, ?Test.1.t)
booland goal - 2 AndB(?b1, ?b2, c)
booland goal - 3 FF_(?b1)
booland goal - 4 TT_(?b2)
@debug? -> resume
booland Debug Reduction Resumed
@debug? -> query
booland AndB(?b1, ?b2, c) ::= 
       AndB_(?b1, ?b2, c).
query ->
It's false
@debug ->

If the computation spontaneously enters inspection mode, no active processes remain; the residual processes may be inspected.

@debug -> resolvent
booland Debug Reduction Suspended
booland goal - 1 AndB.1.1_(Test.1.t, Test.1.f, ?b2, ?AndB.1.x)
booland goal - 2 TT_(?b2)
@debug ->

To terminate the computation and the debugging session, enter abort.

@debug -> abort
booland Debug Reduction Aborted
<1> terminated
@

5.3 Tree trace

A computation tree may be produced, using the vtree command - see Section 3.3.

@vtree(booland,"RunFT",Tree)
<1> started
It's false

When the LOGIX prompt appears, the ctree command closes the tree, terminating all further construction.
The command ptree(Tree) prints the tree in prefix order.

@ctree(Tree)
@ptree(Tree)
begin : booland # RunFT
 | RunFT
  | .RunFT(b1, b2, c)
  | Test(c)
  | AndB(b1, b2, c)
  | FF(b1)
  | TT(b2)
   ? TT.comm(b2)
   | FF.comm(b1)
   | FF.1(Test.1.f)
   | FF.1.comm(Test.1.f)
   | AndB.comm(b1, b2, c)
   | AndB.1(Test.1.t, Test.1.f, b1, b2, AndB.1.x)
   | AndB.1.comm(Test.1.t, Test.1.f, b1, b2, AndB.1.x)
   | AndB.1.1(Test.1.t, Test.1.f, b2, AndB.1.x)
   ? AndB.1.1.comm(Test.1.t, Test.1.f, b2, AndB.1.x)
  | Test.1(c)
  | Test.1.(c, Test.1.f, Test.1.t)
  | Test.1.comm(c, Test.1.f, Test.1.t)
  | Test.1.1(Test.1.f, Test.1.t)

22
The meaning of the suffixes and annotation is the same as for debugging in Section 5.2.1. Indentation illustrates the depth of call within a module. The prefix “|” indicates a reduced goal; the prefix “#” indicates a remote process call; the prefix “?” indicates a goal which has not been reduced. The prefixes “begin :” and “end :” delimit a remote process call.

To trace tand (see Section 3.4) call, e.g.

@vtree(boolean#tand,"RunFT",Tree, 2)
Chapter 6

Ambient Stochastic Π Calculus Programs

Stochastic Π Calculus supports a flat process space. Ambient Stochastic Π Calculus supports nested process spaces, organized in a tree, and communication between nearby nodes (called ambients) of the tree.

6.1 Channels

In addition to a type, a channel in Ambient Stochastic Π Calculus has a locus as well. The locus determines whether communication is local (within the ambient) or non-local (between nearby ambients). Possible non-local loci of communication are: Parent to Child, Child to Parent, Sibling to Sibling.

6.2 Ambient Declaration

Ambients are declared dynamically. An ambient is declared as a named <new_scope>, e.g.

ACell ::= cell(<< x, y. Cytoplasm(x,y) | Membrane(x,y) >>)

The new ambient is created as a child of the ambient which declares it. It inherits copies of all of the local channels known to its parent. When it is created the ambient is assigned a unique positive integer as part of its identifier (e.g. cell(17)), which may be used to distinguish it from other ambients with the same name.

Ambients are mobile. An ambient may exit its parent, enter a sibling (become its child) or merge with a sibling.
6.3 Ambient Stochastic $\Pi$ Calculus Processes

The first line of an Ambient Stochastic $\Pi$ Calculus module must be:

- language(biospi).

A Stochastic $\Pi$ Calculus module may be changed to an Ambient Stochastic $\Pi$ Calculus module by replacing the language name in the first line. The semantics of the program are unchanged.

6.3.1 Inter-Ambient Communication

A $\langle$send$\rangle$ or $\langle$receive$\rangle$ may be prefixed by a $\langle$direction$\rangle$, specifying the locus of communication. The communication is between ambients. The two communicating processes do so on their shared channel.

<table>
<thead>
<tr>
<th>direction</th>
<th>kind of communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>intra-{$\bf ambient$} (may be omitted)</td>
</tr>
<tr>
<td>p2c</td>
<td>parent to child</td>
</tr>
<tr>
<td>c2p</td>
<td>child to parent</td>
</tr>
<tr>
<td>s2s</td>
<td>sibling to sibling</td>
</tr>
</tbody>
</table>

Examples:

p2c a ! 3*[]
s2s b ? {x,y}

6.3.2 Capability Communication

A communication may be an assertion of the form $\langle$capability$\rangle$ $\langle$channel$\rangle$. The communication is between ambients. The two processes which communicate do so on the shared channel.
capability  action

enter    enter a sibling which asserts
          accept <channel>
accept   accept as a child a sibling which asserts
          enter <channel>
exit     become a sibling of its parent which asserts
          expel <channel>
expel    expel a child which asserts
          exit <channel>
merge -  merge with a sibling which asserts
          merge + <channel>'
merge +  merge with a sibling which asserts
          merge - <channel>

Examples:

enter a
expel b
merge - c

6.4 Testing Ambient Programs

Unlike SpiFcp programs, BioSpi programs cannot be executed directly under LOGIX. However, the commands run (see B.2.1) and record (see B.2.2) provide means to execute biospi programs.

6.5 Interruption and Inspection

An Ambient Stochastic Π Calculus program may reach an impasse (e.g. booland#RunFF above), reach a limit (see B.2.1 and B.2.2), be interrupted manually (see B.3.1) or terminate normally. At an impasse, limit or interruption, the current state of the computation can be inspected by the command rtr (see B.6.2). At an impasse, limit or interruption, the tree of ambients can be inspected by the command atr; the tree including the communicating channel set of each ambient can be inspected by the command ctr (see B.6.1).
Chapter 7

Using FCP in BioSpi modules

This chapter describes the use of FCP guards, goals and variables inside BioSpi processes.

7.1 LOGIX Variables as Counters and Limits

The following process sends \( N \) nil messages to channel \( c \).

\[
\text{Send}(N, c) + I ::= \{I = 0\} | \text{SendAndCount}.
\]

\[
\text{SendAndCount}(I, N, c) ::= \\
\quad \{I++ < N\}, \text{SendNil} | \text{self}; \\
\quad \{I >= N\}, \text{screen#display(sent - I*[])}. \\
\text{SendNil}(c) ::= c!, 0.
\]

A process argument may be a \(<\text{logix}\_\text{variable}>\) - in this case, \( N \). Such a variable may be passed to another process, or it may be tested or operated upon in a LOGIX guard or a LOGIX goal.

Similarly, a \(<\text{logix}\_\text{variable}>\) may be declared in an added argument list or in a declaration list, or it may be sent or received in a message - in this example, \( I \) is declared as an added argument.

The LOGIX goal \( \{I = 0\} \) presets \( I \) to 0. The LOGIX guards test \( I \), comparing it to \( N \). The LOGIX guard \( \{I++ < N\} \) also increments \( I \), for the recursive call, \text{self}, to the process \text{SendAndCount}.

The first process may be rewritten:

\[
\text{Send}(N, c) + I ::= (\text{number}(N) : I = 0), \text{SendAndCount}.
\]
to wait until $N$ is known to be a number, before initializing $I$ and then continuing with **SendAndCount**. This might be useful if the reactions enabled by **SendAndCount** should not occur until $N$ has been initialized, for instance if **SendAndCount** were to enable the reactions, and subsequently count them.

Many other LOGIX guards and goals may be employed in **BioSpi** processes - see [8], Appendix 1.

### 7.2 Semi-LOGIX Processes

A process which has the form of a LOGIX process, is written:

$$\text{<left_hand_side>} :- \text{<logix_clauses>}.$$  

where **<left_hand_side>** has the same syntax and semantics as defined above, except that **<process_name>** begins with a lower-case letter.

A **<logix_clause>** may begin with an optional **<logix_guard>** followed by |, and ends with a **<logix_body>**.

**<logix_clause>s** are separated by choice operators ( ; )

A **<logix_guard>** consists of **<logix_asks>** optionally followed by : and **<logix_tells>**.

Terms of **<logix_asks>**, **<logix_tells>** and **<logix_body>** are separated by commas.

The process which appears in the first subsection of this chapter, can be re-written:

```
send(N, c) :- I := 0} \mid \text{sendAndCount}.
```

```
sendAndCount(I, N, c) :-
   I++ < N \mid \text{SendNil }, self;
   I >= N \mid \text{screen#display(sent - I*[])}. 
```

```
SendNil(c) ::= c! , 0.
```

See also, fib.cp in sub-directory Examples.
Appendix A

Syntax and Semantics

A.1 BNF for Stochastic Π Calculus

A Stochastic Π Calculus module begins with the line:

-language(spifcp).

That line is followed by one <program>.

<program> ::= <spi_attributes> . <process_definitions> . <process_definitions> .

<spi_attributes> ::= <spi_attribute> <spi_attribute> . <spi_attributes>

<spi_attribute> ::= <export_declaration> <public_declaration> <default_baserate_declaration> <default_weighter_declaration>

<export_declaration> ::= export(<process_names>)

<public_declaration> ::= public(<parameters>)

<default_baserate_declaration> ::= baserate(<base_rate>)

<default_weighter_declaration> ::= weighter(<weighter_declaration>)

<process_names> ::= <process_name> <process_name> . <process_names>
<spifcp_body>

<communication_clauses> ::= 
  <communication_clause> 
  <communication_clause> ; <communication_clauses>

<communication_clause> ::= <communication> , <spifcp_clauses>

<communication> ::= <receive> 
  <send> 
  <delay>

<receive> ::= <channel> ? <transmission> 
  ? <channel>

<send> ::= <channel> ! <transmission> 
  <channel> !

<delay> ::= delay(<base_rate>) 
  delay(<base_rate>, <weighter_declaration>)

<translation> ::= <message> 
  <multiplier> * <message> 
  <message> * <multiplier>

<message> ::= [] 
  {
    <arguments>}</n

<multiplier> ::= <positive_integer>

<comparison_clauses> ::= <comparisons> 
  ; otherwise , <spifcp_clauses>

<comparisons> ::= <comparison_clause> 
  ; <comparisons>

<comparison_clause> ::= <comparison> , <spifcp_clauses>

<comparison> ::= <compare> 
  & <comparison>

<compare> ::= <channel> =? <channel> 
  <channel> =\ <channel>

<spifcp_guard_clauses> ::= 
  <spifcp_guard_clause>
<spifcp_guard_clause> ::=<spifcp_logix_guard>, <spifcp_clauses>

<spifcp_logix_guard> ::= {{logix_ask}}
(logix_ask), logix_asks
(logix_asks : logix_tells)

<logix_clauses> ::= logix_clause
(logix_clause ; logix_clauses)

<logix_clause> ::= logix_guard | logix_body

<logix_guard> ::= logix_asks_tells
(logix_asks)

<logix_asks_tells> ::= logix_asks : logix_tells

<logix_asks> ::= logix_ask
(logix_ask , logix_asks)

<logix_tells> ::= logix_tell
(logix_tell , logix_tells)

<logix_body> ::= call
(call , logix_body)

<spifcp_body> ::= call
(call | spifcp_body)

<call> ::= local_call

<local_call_sum> ::= nested_scope

<external_call> ::= macro_call

<object_call> ::= object_call

<logix_goal> ::= true
0

<local_call_sum> ::= local_call + local_sum

<local_sum> ::= local_call
(local_call + local_sum)
<local_call> ::= <local_process_name> <local_process_name>({call_arguments})

<local_process_name> ::= <process_name> self

<call_arguments> ::= <arguments> <substitutions>

<substitutions> ::= <substitution> <substitution>, <substitutions>

<substitution> ::= <channel> = <channel> ::= <logix_variable> = <logix_variable>

<nested_scope> ::= << <new_scope> >>

<new_scope> ::= <parameters> . <scope_content> <scope_content>

<scope_content> ::= <spifcp_clauses> <spifcp_clauses>. <process_definitions>

<external_call> ::= <logix_path_term>#<atom> <logix_path_term>#<logix_goal> <logix_path_term>#<external_call>

<macro_call> ::= set_base_rate(base_rate, channels_and_reply) randomize_messages(channels_and_reply) serialize_messages(channels_and_reply) get_channel_status(channel, channel_attributes, logix_variable) object(logix_variable) object(logix_term, logix_variable)

<object_call> ::= <logix_variable>! <object_request>

<object_request> ::= close close(reply) name(logix_variable) name(logix_variable, reply) read(logix_variable) read(logix_variable, reply) store(logix_term) store(logix_term, reply)
values(<logix_variable>)
values(<logix_variable>, <reply>)

<channels_and_reply> ::= <reply>
  <channel> , <channels_and_reply>

<channel_attributes> ::= <reply>
  <channel_attribute> , <channel_attributes>

<reply> ::= <logix_variable>
A.2 Alternate BNF for Ambient Stochastic π Calculus

An Ambient Stochastic π Calculus module begins with the line:

\[-\text{language(biospi)}.\]

That line is followed by one \(<\text{program}>\). The program has alternative definitions as follows.

\[
\begin{align*}
<\text{communication}> & ::= <\text{receive}> \\
& \quad <\text{send}> \\
& \quad <\text{capability}> \\
<\text{receive}> & ::= <\text{channel}> ? <\text{transmission}> \\
& \quad <\text{direction}><\text{channel}> ? <\text{transmission}> \\
<\text{send}> & ::= <\text{channel}> ! <\text{transmission}> \\
& \quad <\text{direction}><\text{channel}> ! <\text{transmission}> \\
<\text{direction}> & ::= \text{local} \\
& \quad \text{p2c} \\
& \quad \text{c2p} \\
& \quad \text{s2s} \\
<\text{capability}> & ::= \text{enter} <\text{channel}> \\
& \quad \text{accept} <\text{channel}> \\
& \quad \text{exit} <\text{channel}> \\
& \quad \text{expel} <\text{channel}> \\
& \quad \text{merge} - <\text{channel}> \\
& \quad \text{merge} + <\text{channel}> \\
<\text{call}> & ::= <\text{local\_call}> \\
& \quad <\text{local\_call\_sum}> \\
& \quad <\text{external\_call}> \\
& \quad <\text{ambient}> \\
& \quad <\text{macro\_call}> \\
& \quad <\text{object\_call}> \\
& \quad <\text{logix\_goal}> \\
& \quad \text{true} \\
& \quad 0 \\
<\text{ambient}> & ::= <\text{ambient\_name}>(<\text{nested\_scope}>)
\end{align*}
\]
A.3 Primitives

- A `<process_name>` is an alpha-numeric string, which may contain underscore (_ _) characters, beginning with an upper-case letter for an `<spifcp_process>` or with a lower-case letter for a `<logix_process>`.

- A `<channel>` is an alpha-numeric string, which may contain underscore (_ _) characters, beginning with a lower-case letter. It represents a Π Calculus channel.

- A `<weighter>` is an alpha-numeric string, which may contain underscore (_ _) characters, beginning with a lower-case letter. The currently acceptable values of `<weighter>` are `default` and `michaelis`; additional values may be defined - see `weighter.txt`.

  An `<ambient_name>` is an alpha-numeric string, which may contain underscore (_ _) characters, beginning with a lower-case letter.

- A `<logix_variable>` is an alpha-numeric string, which may contain underscore characters, beginning with an upper-case letter, or it may be a single underscore character. The string may be followed by optional single-quote characters. By convention the single underscore character is an anonymous `<logix_variable>`; it may not appear in place of an `<argument>` in a `<left_hand_side>`.

A.4 Semantics

A Stochastic Π Calculus `<program>` is completely equivalent to an Ambient Stochastic Π Calculus `<program>` with the same syntactic content.

- A `<parameter>` in a `<public_declaration>` is an implicit argument of every process in the program.

- Any argument of a `<process>` may be provided by a caller in a `<call_arguments>`, replacing any implicit argument of the same name. An argument may be `extruded` from a `<process>` by a `<send>`. An argument may also be replaced in a `<parameter>` which is added to the `<atom>` in the `<left_hand_side>` of a `<process_declaration>`.

- A `<base_rate>` may be specified by a `<logix_variable>` in the `<default_baserate_declaration>` or in a `<channel_declaration>`. In the former case, or when the `<channel_declaration>` is within the `<public_declaration>`, the `<logix_variable>` itself must be declared in the `<public_declaration>`; in the latter case, when the channel is in the `<parameter>` which is added in the `<process_declaration>`, the `<logix_variable>` should be an argument of the process. In any case, the `<logix_variable>` should be instantiated to a non-negative number.
A `<weighter>` may be specified by a `<logix_variable>` in the `<default_weighter_declaration>` or in a `<channel_declaration>`. In the former case, or when the `<channel_declaration>` is within the `<public_declaration>`, the `<logix_variable>` itself must be declared in the `<public_declaration>`; in the latter case, when the channel is in a `<parameter>` which is added in the `<process_declaration>`, the `<logix_variable>` should be an argument of the process. In any case the `<logix_variable>` should be instantiated to a string (see above).

A `<weighter_parameter>` may be specified by a `<logix_variable>`, as for a `<base_rate>` (see above).

An argument in a `<macro_call>` which precedes the `<channels_and_reply>` or which is a `<channel_attribute>` may be a read-only-variable (`<logix_variable>`?). The `<logix_variable>` must be instantiated before the `<macro_call>` can be completed.

When a `<macro_call>` is completed, the trailing `<logix_variable>` is instantiated - its value is usually the string `true`, but it may vary in some cases.

The program macro `get_channel_status` instantiates the trailing `<logix_variable>` to the value(s) of the named attribute(s). See `program_macros.txt` for details regarding `<channel_attribute>` values.

WARNING: When the single underscore character `<logix_variable>` (`_`) appears in a `<receive> <transmission>` in place of a `<channel>`, the corresponding `<send> <channel>` should be declared in a `<public_declaration>`.

### A.5 Logix Terms

See `supplement.mss` for details of the LOGIX language.

- `<logix_ask>` is any predicate permitted in the ask of a guard in LOGIX language(compound).
- `<logix_tell>` is any predicate permitted in the tell of a guard in LOGIX language(compound).
- `<logix_goal>` is any predicate permitted in the right-hand-side of a clause in LOGIX language(compound).
- `<logix_path_term>` is any term permitted in the path specification of a remote procedure call in LOGIX language(compound).
- `<logix_term>` is any term permitted in a logix program.
A.6 Notes

- The new predicate of Π Calculus has been subsumed into a prefix <parameters> of a <new_scope> and the added <parameters> of a <process>.

- A <process> which is declared at level one of a <program> definition, may be called by an external process, if it is exported explicitly in an <export_declaration>, or if there is no <export_declaration> in the <program>, in which case all level one processes are exported.

- Nested new processes are scoped with double angle brackets (see definition of <nested_scope>).

- Each <local_call> in a <local_call_sum> must be to a <process> whose <right_hand_side> consists of <communication_clauses>.

- The basic reserved words are self, true and otherwise. They are reserved in context, and may be used as channel names.
  - self may be used to iterate any process, including anonymous processes.
  - true is an alternative name for process 0.
  - otherwise appears as the guard of the last clause of <comparisons>.

Additional reserved words, used in Ambient Stochastic Π Calculus are enter, accept, exit, expel, merge, p2c, c2p, s2s. These words are all prefix operators, except for merge, which is only reserved in context.

- An argument which is declared as a <logix_variable> in the added <parameters> of a <left_hand_side> is initially uninstantiated.

- Within a <logix_term>, normal LOGIX recognition of variables applies - i.e. variable names all begin with a capital letter or underscore; to reference a <channel> whose name begins with a lower case letter, within a <logix_term>, refer to "_var"(<channel>)

- A <logix_variable> may have a value which is an arbitrary LOGIX term. Such a value may be tested by a <logix_ask>, instantiated or used as an argument in a <logix_goal> or by a <logix_tell_guard>.

- A <logix_variable> may be instantiated by an assignment in a <call>:

  {<logix_variable> = <logix_term>}

38
• An arbitrary \texttt{logix\_term} may be sent in a \texttt{message} by the library \texttt{spi\_send/2}, or received by the library \texttt{spi\_receive/2}:

\begin{verbatim}
spi_send(<message\_content>, <channel>)
spi_receive(<channel>, <message\_content>)
\end{verbatim}
e.g.

spi_send({"A string", a(Tuple), [A, list | Tail]},  
            "_var"(channel))

spi_receive("_var"(channel), {String, Tuple, List})

• The library `<logix_goals>`:

  spi_send/3, spi_receive/3, spi_send/4, spi_receive/4

may be used as well, where the third argument is a multiplier (default 1), and the fourth argument is an identifier (default sender or receiver).

• The additive definition of `<left_hand_side>` is syntactic sugar - e.g.

Enzyme + (rel_s, rel_p) ::= 
  bind_s ! {rel_s,rel_p} , EX(rel_s,rel_p) ;
  bind_p ! {rel_s,rel_p} , EX(rel_s,rel_p) .

is equivalent to:

Enzyme ::= 
  << rel_s, rel_p .
  bind_s ! {rel_s,rel_p} , EX(rel_s,rel_p) ;
  bind_p ! {rel_s,rel_p} , EX(rel_s,rel_p) 
  >> .

• The recursive definition of `<right_hand_side>` is syntactic sugar - e.g.

Fibonacci(ch, FN) ::= 
  ch ? {FA} , ((FN = {N, Fj, Fi}, N++, Fj' := Fi + Fj : 
  FA = FN, FN' = {N', Fj', Fj}) , Fibonacci ;
  (FA = [] : FN = _), 0) .

is equivalent to:

Fibonacci(ch, FN) ::= 
  ch ? {FA} ,
  << P1 .
  P1(FN) ::= 
  (FN = {N, Fj, Fi}, N++, Fj' := Fi + Fj :
  FA = FN, FN' = {N', Fj', Fj}) , Fibonacci ;
  (FA = [] : FN = _), 0)
  >> .

40
Appendix B

BioSpi commands

The macro commands supplied for BioSpi include and in some cases replace the user macros of LOGIX. They fall into three major categories:

- Channel management and message transmission
- Program execution
- Debugging.

B.1 Channel Management and Message Transmission Macros

These macros may be useful in auxiliary LOGIX programs. They may be used within an FCP goal; such use is not recommended.

B.1.1 Create Channel - pc

Create a new private channel.

\[
\text{pc(Channel)}
\]
\[
\text{pc(Channel,Creator,BaseRate)}
\]
\[
\text{pc(Channel,Creator,BaseRate,ComputeWeight)}
\]

The first macro creates an instantaneous (infinite rate) channel. The second macro creates a based Channel, whose name is derived from the string Creator, and whose base rate is specified by the non-negative number BaseRate (When BaseRate = 0, the created channel is a sink - i.e. all sends and receives on the channel are discarded, and no actual transmission occurs.) The third macro permits the user to specify a \(<\text{weighter_declaration}>\), Weighter, for the new channel.

Channel may be a string, in which case a \text{logix_variable}; named with that string is created. The name may be used to refer to the channel when using
the “ps” and “pr” commands to send and receive messages, or to inspect the channel - e.g. 

```c
@pc(a)
@ps([],a)
@options(full)
@a^ 
```

```
a = spi.a(1)!
```

B.1.2 Send Message - ps
Offer to send a message on a channel.

```c
ps(Message,Channel)
ps(Message,Channel,Multplier)
```

Multiplier is a positive integer; the likelihood that Channel will be selected for transmission increases with Multiplier.

B.1.3 Receive a Message - pr
Offer to receive a message on a channel.

```c
pr(Channel,Message)
pr(Channel,Message,Multplier)
```

Multiplier is a positive integer, as above.

B.1.4 Set Default Weighter - weighter
Set the default weight computation.

```c
weighter(Weighter)
```

Weighter is an atom: a computation name (string) or a tuple Name(P1, · · · , Pn), where P1, · · · , Pn are additional numeric parameters to the weight computation.

B.2 Program Execution Macros
The basic LOGIX command to execute a program has the form:

```c
Path#Goal
```

The system creates a computation, uniquely identified by a positive integer throughout the session.

To execute a spifcp program, e.g. RunTT in module booland in directory boolean , call:

```c
boolean#booland#"RunTT"
```

The call above is an example of a Remote Procedure Call (RPC).
B.2.1 Execute Goals - run
Reset the session (as in B.7.1) and execute all of the Goals as a single computation.

\[
\text{run} \text{(Goals)} \\
\text{run} \text{(Goals, Limit)}
\]

The first form continues indefinitely; the second continues until Limit units of internal time have elapsed. See Appendix C.2 for details on specification of multiple goals.

B.2.2 Execute Goals - run

\[
\text{run} \text{(Goals, File, Limit)} \\
\text{run} \text{(Goals, File, Limit, Scale)} \\
\text{run} \text{(Goals, File, Limit, Format)} \\
\text{run} \text{(Goals, File, Limit, Scale, Format)}
\]

Like run it resets the session and executes all of the goals until Limit; it also records their behavior on the named file, optionally scaling the output times by multiplying by Scale. See Appendix C.2 for details on specification of multiple goals. See Appendix C.1 for details about the file and formatting.

B.3 Execution Control

A computation may be suspended, resumed or aborted. You may also inspect its resolvent - the set of untermined processes - see Section 5.2 for examples.

B.3.1 Suspend Execution - suspend
Suspend the current or the specified computation(s).

\[
\text{s} \\
\text{s(all)} \\
\text{s(N)}
\]

The last form also resets the current computation number. While the program is suspended, it may be inspected (see B.5.1, B.5.2, B.6.2, B.6.1).

B.3.2 Resume Execution - resume
Resume the current or the specified computation(s), as above.

\[
\text{re} \\
\text{re(all)} \\
\text{re(N)}
\]
B.3.3 Abort Execution - abort
Abort the current or the specified computation(s), as above.

a
a(all)
a(N)

B.4 Debugging SpiFcp Processes
Debugging aids consist of inspection and execution control macros.

B.4.1 Set Display Options - options
Set new display control options and (optionally) return old ones.

options(New)
options(New,Old)

New may be a single option or a LOGIX list of options.

• none: Don’t display any messages; this is the usual default.
• active: Display all active message actions (send, receive, dimer).
• sender: Display each message’s sender in the form:
  Process(ChannelName, Multiplier, Action)
• no_sender: Only display a message’s action; this is the usual default.

An example list is:

[active,sender]

In the macros below, options may be specified explicitly in one variant of most groups. When the options are specified, they override the global options set by the options macro above.

B.4.2 Show a Channel - spc
spc(Channel)
spc(Channel,Options)

display Channel.
B.4.3 Show Goal - spg
Display the goal of the current or of the specified computation.

\[
\begin{align*}
\text{spg} \\
\text{spg}(N) \\
\text{spg}(N,\text{Options})
\end{align*}
\]

The last two forms also reset the current computation number.

B.5 Debugging SpiFcp

B.5.1 Show Resolvent - spr
Suspend the current or specified computation and display its resolvent as above.

\[
\begin{align*}
\text{spr} \\
\text{spr}(N) \\
\text{spr}(N,\text{Options})
\end{align*}
\]

To continue the computation, use the resume command (see B.3.2).

B.5.2 Display Communicating Channels - cta
Display communicating channels.

\[
\text{cta}
\]

B.5.3 Debug a SpiFcp Goal - pdb
Debug a single RPC.

\[
\begin{align*}
\text{pdb}(\text{RPC}) \\
\text{pdb}(\text{RPC},\text{Options})
\end{align*}
\]

The debugger provides help in response to the command “help”. See the document supplement.mss for details of the debugger commands.

B.5.4 Create an Execution Tree - vtree
Execute Goal, with respect to Path and prepare Tree.

\[
\begin{align*}
\text{vtree}(\text{Path},\text{Goal},\text{Tree}) \\
\text{vtree}(\text{Path},\text{Goal},\text{Tree},\text{Depth})
\end{align*}
\]

Tree may be displayed using macros “ctree” and “vtree” below. Depth is the depth of remote process call to be included in Tree - if omitted, all goals are included. For example, if Path is boolean#booland and goal is RunTT, Tree represents the execution of the RPC:
Ordinarily, you should wait until the system becomes idle, or the computation
has been suspended before attempting to view Tree. To view Tree, use either
of the macros:

\[
\begin{align*}
&\text{ptree(Tree)} \\
&\text{ptree(Tree,Options)}
\end{align*}
\]

See B.4.1 above for basic options. Additional options which may be specified
are:

- prefix: Display Tree in prefix order; this is the default.
- execute: Display Tree in execution order.

To close the execution tree:

\[
\text{ctree(Tree)}
\]

This terminates the system's participation in the execution of the computation.
See Section 5.3 for examples.

**B.6 Debugging**

### B.6.1 Show Ambient Tree - atr,ctr

Display the ambient tree, a subtree, a node or a set of nodes.

\[
\begin{align*}
&\text{atr} \\
&\text{atr(AmbientSelector)} \\
&\text{ctr} \\
&\text{ctr(AmbientSelector)}
\end{align*}
\]

The form of the display is specified by the AmbientSelector (default is the entire
tree). An ambient is uniquely identified (within a run) by a positive integer.
Its full identifier is a 2-tuple, \(<\text{name}>(<\text{integer}>)\). The AmbientSelector may specify:

- if omitted or the empty string, the entire tree.
- if a positive integer, the entire subtree, starting with the designated ambient.
- if a name, all ambients with that name; The name “system” designates
  the root of the tree as well as any other ambient whose full identifier is
  system(<integer>)
- if a 2-tuple \(\text{(e.g. cell(6)})\) or a negative integer \(\text{(e.g. -6)}\), the absolute
  value of the integer is the unique identifier of the single node displayed; in
  the former case, the name of the ambient is ignored.
B.6.2 Show Resolvent - rtr
Suspend the computation and display the resolvent as a tree of ambients.

\[
\text{rtr} \\
\text{rtr}(\text{AmbientSelector})
\]

The active processes in each ambient are indented immediately below the node identifier. To continue the computation, use the resume command (see ref{resume}).

The AmbientSelector is treated as for atr/ctr above.

B.7 Miscellaneous Macros

B.7.1 Reset the System - reset
This command closes all spifcp activities, effectively returning the system to its initial state, except for the random seed, the ordinals assigned to private channels and the current options.

- Activity of existing channels is terminated;
- The list of existing public channels is discarded;
- The current internal clock is reset to 0 (See Chapter 4);
- The current time limit is reset to a very large number;
- No computation, ambient or process is terminated.

The reset function is called automatically at the beginning of any run or record command and whenever the internal time Limit is exceeded (See B.2.1 and B.2.2).

B.7.2 Input Commands - input
Input the command file designated by Path.

\[
\text{i(Path)} \\
\text{input(Path)}
\]

For example to execute the commands contained in the file test in sub-directory scripts:

\[
\text{input(scripts#test)}
\]
B.7.3 Call a UNIX Command \{...\}
Execute a UNIX command directly.

\{Command\}

Examples:

\{ls\}
\{"cat notes"\}

B.7.4 Display Named Variables - ^
Display a named variable.

VariableName^

Display all named \{logix variables\}.

^

B.7.5 Change Current Computation
Change the current computation according to the specified number.

state(Number)

The current computation number is set to Number and that computation’s goal and state are displayed,
Appendix C

Auxilliary LOGIX Procedures

C.1 record

The spi_record process can run a spifcp process and record its behavior.

\[
\text{spi_record#run(\langle call program\rangle, \langle time limit\rangle)}
\]

\[
\text{spi_record#run(\langle call program\rangle, \langle file_name\rangle, \langle time limit\rangle)}
\]

\[
\text{spi_record#run(\langle call program\rangle, \langle file_name\rangle, \langle time limit\rangle, \langle scale\rangle)}
\]

\[
\text{spi_record#run(\langle call program\rangle, \langle file_name\rangle, \langle time limit\rangle, \langle format\rangle)}
\]

\[
\text{spi_record#run(\langle call program\rangle, \langle file_name\rangle, \langle time limit\rangle, \langle scale\rangle, \langle format\rangle)}
\]

\[
\text{spi_record#run(\langle call program\rangle, \langle file_name\rangle, \langle time limit\rangle, \langle format\rangle, \langle scale\rangle)}
\]

It is normally called by the macros run in Section B.2.1 and record in Section B.2.2.

- \text{run(hysteresis#MODULE, 100)}

starts the process MODULE in program module hysteresis and terminates the run after 100 time units have elapsed. To terminate the run prematurely, suspend the LOGIX computation or enter \text{<control>} C to kill LOGIX.

- \text{record(hysteresis#MODULE, fff, 100)}

does the same thing, and records the events of the run on file fff. The elements of the file are lines which have one of three forms.

- A real-valued \text{time}.
- \text{+<procedure name>}, which records the start of a procedure.
- `<procedure name>`, which records the termination of a procedure.

- `record(hysteresis#MODULE, fff, 100, 10)`

  does the same thing, and multiplies each time recorded in the file by 10.

The `<format>` argument may be one of "none", "process", "creator", "full". The default is "none"; the other three annotate the records of communication with the name or identifier of the channel over which the communication occurred.

To analyze the file, producing a table suitable for plotting with Matlab, use the PERL program "spi2t" - e.g.

```
% spi2t fff
```

creates a table, where column 1 is time, and columns 2-⋯n are totals of active processes. A short file, with one long line, listing the column (process) names, and n-1 lines associating process names with array columns is also produced. For example:

```
fff.table and fff.names
```

Column one of the .table file is incremented approximately by 1 between rows (lines). To change the increment to another positive number, N, add the argument N to the call to tally - e.g.

```
% spi2t fff 0.1
```

To combine columns sums add terms of the form:

```
<summed_column_name>+=<absorbed_column_name>
```

e.g.

```
% spi2t fff GENE+=BASAL+PROMOTED GENE+=ACTIVATED_TRANSCRIPTION
```

To rescale the output times, specify a negative rescale value - e.g. to rescale output times by 1/10:

```
% spi2t fff -10
```

To split the table into multiple 2-column files, suitable for gnuplot, use the PERL program "t2xys" - e.g.

```
% t2xys fff
% gnuplot
  gnuplot> plot "fff.3" smooth unique
```

A short shell script, spixys, combines the functions with a call to gnuplot.
C.2 repeat

This LOGIX procedure is called by the run and record macros (see B.2.1 and B.2.2). The repeat process can run a quantified set of spifcp processes.

\[
\text{repeat\#run(}\langle\text{quantified process set}\rangle)\]

where:

\[
\langle\text{quantified process set}\rangle ::= \langle\text{external call}\rangle \\
\langle\text{process set}\rangle \\
\langle\text{repetition}\rangle * \langle\text{process set}\rangle
\]

\[
\langle\text{process set}\rangle ::= \langle\text{quantified process set}\rangle, \langle\text{quantified process set}\rangle
\]

\[
\langle\text{repetition}\rangle ::= \langle\text{integer}\rangle
\]

A negative \(\langle\text{repetition}\rangle\) is treated as zero. (See Appendix A.1 for the definition of \(\langle\text{external call}\rangle\).

Examples:

- \text{repeat\#run(}64*(\text{dimerization#A\_PROTEIN})\text{)}
- \text{repeat\#run([}6*(\text{activator#A\_PROTEIN}), \\
  \text{activator#A\_GENE, repressor#R\_GENE})\text{)}
- \text{repeat\#run([}2*[\text{activator#A\_GENE, 3*(repressor#R\_GENE)}], \\
  \text{hysteresis#module})\text{)}

Note that the parentheses are necessary in the case of:

\[
\langle\text{repetition}\rangle*(\langle\text{external call}\rangle)
\]

Here is an example call to run (see Section 3.3).

\[
\text{repeat\#run([}3*(\text{booland#RunTT}),4*(\text{booland#RunFT})\text{])}
\]

\[<2>\text{ started}
\]

It’s false
It’s false
It’s false
It’s false
It’s true
It’s true
It’s true
@spr

51
C.3 Weight Computation for a Channel

Channels with finite rates are weighted for selection using the default computations in Section 4.1. The user may specify custom computations using the notation for `<weighter_declaration>` in Appendix A.1; the computation must be explicitly coded in the module Logix/<emulator name>/spiweight.c. Module <emulator>/spiweight.c may be modified to specify a custom computation.

- Choose a name for the computation, which is not used for some other computation. The name should be alpha-numeric, and it should start with a lower-case letter (Embedded underscores are permitted.)
- Choose an integer to represent the computation, which is not used for some other computation.
- Add an entry to the “weighter” array in the specified form.
- Add a case for the C-code of the computation to the switch(es) in the function `spi_compute_bimolecular_weight` and/or in the function `spi_compute_homodimerized_weight`.
- Re-install LOGIX - spiweight.c and the appropriate emulator are automatically re-compiled.

The arguments include the parameters of the `<weighter_declaration>` in their order of declaration in the array “argv”; the argument “argv” is the size of the array.

The computed weight should be stored in “result”. Two examples of custom computation, named “square” and “poly” are included in the module.
Bibliography


