Appendix A

BNF

BNF for PSI

```
<program>            ::= <psi_attributes> . <process_definitions> .
                         <process_definitions> .

<psi_attributes>     ::= <psi_attribute>
                         <psi_attribute> . <psi_attributes>

<psi_attribute>      ::= <export_declaration>
                         <global_declaration>
                         <default_base_rate_declaration>
                         <default_weighter_declaration>

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

<global_declaration> ::= global(<global_declaration_list>)

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

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

<process_name_list>  ::= <process_name>
                         <process_name> , <process_name_list>
```

```
<weighter_declaration>
                    ::= <weighter>
                        <weighter>(<weighter_parameter_list>)

<weighter_parameter_list>
                    ::= <real number>
                        <real number> , <weighter_parameter_list>

<global_declaration_list>
                    ::= <channel_declaration>
                        <channel_declaration> , <global_declaration_list>

<channel_declaration>
                    ::= <channel>
                        <channel>(<base_rate>)
                        <channel>(<base_rate> , <weighter_declaration>)

<base_rate>         ::= <number>
                        infinite

<process_definitions>
                    ::= <process>
                        <process> . <process_definitions>

<process>           ::= <left_hand_side> ::= <right_hand_side>

<left_hand_side>    ::= <atom>
                        <atom> + <new_declaration>
                        <atom> + (<new_declaration_list>)

<atom>              ::= <process_name>
                        <process_name>(<argument_list>)

<new_declaration_list>
                    ::= <new_declaration>
                        <new_declaration> , <new_declaration_list>

<new_declaration>   ::= <channel_declaration>
                        <logix_variable>
```

```
<argument_list>      ::= <argument>
                         <argument> , <argument_list>

<argument>           ::= <channel>
                         <logix_variable>

<right_hand_side>    ::= <communication_clauses>
                         <comparison_clauses>
                         <logix_guard_clauses>
                         <body>

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

<communication_clause>
                     ::= <communication> , <right_hand_side>

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

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

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

<transmission>       ::= <message>
                         <multiplier> * <message>
                         <message> * <multiplier>

<message>            ::= []
                         {<argument_list>}

<multiplier>         ::= <positive_integer>

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

<comparisons>        ::= <comparison_clause>
```

```
                         <comparison_clause> ; <comparisons>

<comparison_clause>  ::= <comparison> , <right_hand_side>

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

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

<logix_guard_clauses>::= <logix_guard_clause>
                         <logix_guard_clause> ; <logix_guard_clauses>

<logix_guard_clause> ::= {<logix_ask_guard>} , <right_hand_side>
                         {(<logix_guard>)} , <right_hand_side>

<logix_guard>        ::= <logix_ask_guard_list>
                         <logix_ask_guard_list> : <logix_tell_guard_list>

<logix_ask_list>     ::= <logix_ask_guard>
                         <logix_ask_guard> , <logix_ask_list>

<logix_tell_list>    ::= <logix_tell_guard>
                         <logix_tell_guard> , <logix_tell_list>

<body>               ::= <call>
                         <call> | <body>

<call>               ::= <local_call>
                         <local_sum>
                         << <new_scope> >>
                         <external_call>
                         {<logix_goal>}
                         {(<logix_goal_list>)}
                         self
                         self( <substitution_list> )
                         true
                         0
```

```
<local_sum>          ::= <local_call>
                         <local_call> + <local_sum>

<local_call>         ::= <atom>
                         <process_name>( <substitution_list> )

<substitution_list>  ::= <substitution>
                         <substitution> , <substitution_list>

<substitution>       ::= <argument> = <argument>

<new_scope>          ::= <new_declaration_list> . <scope_content>
                         <scope_content>

<scope_content>      ::= <right_hand_side>
                         <right_hand_side> . <process_definitions>

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

<logix_goal_list>    ::= <logix_goal>
                         <logix_goal> , <logix_goal_list>
```

Primitives:

- `<channel>` is an alpha-numeric string, beginning with a lower-case alphabetic character; the string may contain underscore (_) characters. It represents a π-calculus channel.

- `<base_rate>` is a real number, which is a factor in the stochastic weighting of a `<channel>` . When the `<base_rate>` of a `<channel>` is 0, the channel acts as a sink - all messages on that channel are discarded. A `<base_rate>` may be specified by a LOGIX variable for a new channel, but not for the default nor for a global channel. The LOGIX variable should be instantiated to a number.

- `<process_name>` is an alpha-numeric string, beginning with an upper case letter, which may contain underscore (_) characters.

- `<weighter>` is an alpha-numeric string, beginning with a lower-case alphabetic character; the string may contain underscore (_) characters.

  A `<weighter>` may be specified by a LOGIX variable for a new channel, but not for the default nor for a global channel. The LOGIX variable should be instantiated to an alpha-numeric string as above.

  Any `<weighter_parameter_list>` element may be specified by a LOGIX variable for a new channel, but not for a global channel. The LOGIX variable should be instantiated to an alpha-numeric string as above (for the `<weighter>` name) or to a number (for any other parameter).

- `<logix_variable>` is an alpha-numeric string, beginning with an upper-case alphabetic character; the string may contain underscore (_) characters, or it may be a single underscore character. By convention the single underscore character is an anonymous `<logix_variable>` ; it may only appear in a `<message>` or within a `<logix_guard>` , a `<logix_ask_guard>` or a `<logix_goal>` [1] .

- `<logix_ask_guard>` is any predicate permitted in the ask of a guard in LOGIX language(compound).

- `<logix_tell_guard>` 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).

[1] WARNING: when the single underscore character LOGIX variable appears in a `<message>` , it should not appear in place of a `<channel>` .

Notes:

- The "new" predicate has been subsumed into a prefix `<new_declaration_list>` of a `<new_scope>` and the added `<new_declaration_list>` of a `<process>` .
- The `<argument>s` of a `<process>` are provided by a caller. An `<argument>` may be "extruded" from a `<process>` by a `<send>` .
- A `<process>` which is exported, may be called by an external process.
- Nested "new" processes are scoped with double angle brackets (see definition of `<new_scope>`).
- The 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>` .
- A new `<argument>` which is a `<logix_variable>` 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 values may be tested by a `<logix_ask_guard>` or manipulated by a `<logix_goal>` or by a `<logix_tell_guard>` . An arbitrary `<logix_term>` may be sent in a `<message>` by the `<logix_goal>` `psi_send/2` , or received by the `<logix_goal>` , `psi_receive/2` :

```
psi_send(<message_content>, <channel>)
psi_receive(<channel>, <message_content>)
```

e.g.

```
psi_send({"String", {tuple}, [list | Tail]},
         "_var"(channel))

psi_receive("_var"(channel), {String, Tuple, List})
```

- The library `<logix_goal>s` `psi_send/3` , `psi_receive/3` , `psi_send/4` , `psi_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.*

```
P + (<new_declaration_list>) ::= <right_hand_side>.
```

  is equivalent to:

```
P ::= << <new_declaration_list> .
         P1 .
         P1 ::= <right_hand_side> >> .
```

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

```
P ::= <communication1> ,
          {<logix_ask_guard>} , Q;
      <communication2> ,
          (<comparison> , R;
           otherwise , S).
```

  is equivalent to:

```
P ::= <communication1> ,
           << P1 .
              P1 ::= {<logix_ask_guard>} , Q >>;
      <communication2> ,
           << P2 .
              P2 ::= <comparison> , R ;
                     otherwise , S >>.
```

and

```
P ::= <communication> , ( {<logix_ask_guard1>} , Q ;
                          {<logix_ask_guard2>} , R ) .
```

is equivalent to:

```
P ::= <communication> ,
           << P1 .
              P1 ::= {<logix_ask_guard1>} , Q ;
                     {<logix_ask_guard2>} , R >> .
```

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

Appendix B

PSI commands

The macro commands supplied for PSI 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

B.1.1 Create Channel - pc

pc(Channel)
pc(Channel,Creator,BaseRate)
pc(Channel,Creator,BaseRate,ComputeWeight)

are used to create channels. The first macro creates an "instantaneous" (infinite rate) Channel. The second 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 <weighter_declaration> for the new channel.
Channel may be a string, in which case a 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.

```
@pc(a)
@ps([],a)
@a^
a = (psi.a = [send])
```

B.1.2 Send Message - ps

ps(Message,Channel)
ps(Message,Channel,Multiplier)

offer to send Message on Channel, where Multiplier is a positive integer; the likelihood that Channel will be selected for transmission increases with Multiplier.

B.1.3 Receive Message - pr

pr(Channel,Message)
pr(Channel,Message,Multiplier)

offer to receive Message on Channel, where Multiplier is a positive integer, as above.

B.1.4 Set Default Weighter - weighter

weighter(Weighter)

sets the default weight computation. Weighter is an atom: a computation name (string) or a tuple Name(P1, $\cdots$, Pn), where P1, $\cdots$, 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:

Path#Goal

The system creates a computation, uniquely identified by a positive integer throughout the session.
To execute a PSI program, *e.g.* RunTT in module bland in directory boolean, call:

boolean#bland#"RunTT"

The call above is an example of a Remote Process Call (RPC).
The quotation marks around RunTT are necessary, since LOGIX treats names beginning with a capital letter or an underscore as logic variables.

B.2.1 Execute Goals - run

run(Goals)
run(Goals,Limit)

execute all of the Goals as a single computation: 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 - record

record(Goals,File,Limit) record(Goals,File,Limit,Scale)

also executes all of the goals until Limit, and 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.
A computation may be suspended , resumed or aborted. You may also inspect its resolvent - the set of unterminated processes - see Section **??** for examples.

B.2.3 Suspend Execution - suspend

s
suspend
s(all)
suspend(all)
s(N)
suspend(N)

suspend the current or the specified computation(s) (N or all). The last two forms also reset the current computation number.

B.2.4 Resume Execution - resume

re
resume
re(all)
resume(all)
re(N)
resume(N)

resume the current or the specified computation(s), as above.

B.2.5 Abort Execution - abort

a
abort
a(all)
abort(all)
a(N)
abort(N)

abort the current or the specified computation(s), as above.

B.3 Debugging

Debugging aids consist of inspection and execution control macros.

B.3.1 Set Display Options - options

options(New)
options(New,Old)

set new display control options and (optionally) return old ones. New may be a single option or a LOGIX list of options.

- none: Don't display any messages.
- active: Display all active message actions (send, receive, dimer); this is the usual default.

- 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.3.2 Show Goal - spg

spg
spg(N)
spg(N,Options)

display the goal of the current or of the specified computation. The last two forms also reset the current computation number.

B.3.3 Show Resolvent - spr

spr
spr(N)
spr(N,Options)

display the resolvent of the current or of the specified computation, as above.

B.3.4 Show a Channel - spc

spc(Channel)
spc(Channel,Options)

display Channel.

B.3.5 Show Global Channels - spgcs

spgcs
spgcs(Options)

display the global channels.

B.3.6 Debug a Goal - pdb

pdb(RPC)
pdb(RPC,Options)

start debugging of the Remote Process Call (e.g. hysteresis#"MODULE"). The debugger provides help in reponse to the command "help".

B.3.7 Create an Execution Tree - vtree

vtree(Path,Goal,Tree) vtree(Path,Goal,Depth)

executes Goal, with respect to Path and prepares Tree, which may be displayed using macros below; Depth is the depth of remote process call to be included in Tree. For example, if Path is boolean#bland and goal is "RunTT", Tree represents the execution of the RPC:

boolean#bland#"RunTT"

To include all of the processes executed in a call to tand in Section **??**. 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:

ptree(Tree)
ptree(Tree,Options)

The default option for messages to be displayed is "none" instead of "active". 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:

ctree(Tree)

This terminates the system's participation in the execution of the computation.

B.4 Miscellaneous Macros

B.4.1 Reset the System - reset

This command closes all PSI activities, effectively returning the system to its initial state, except for the random seed and the current options.

- Service of existing channels is terminated;
- The list of existing global channels is discarded;
- The current internal clock is reset to 0
- The current time limit is reset to a very large number;
- The current options are conserved (See Appendix B.3.1).

The reset function is called automatically whenever the internal time Limit is exceeded (See Section B.2.1 and Section B.2.2).

B.4.2 Input Commands - input

```
i(Path)
input(Path)
```

Input the command file designated by Path. For example to execute the commands contained in the file test in sub-directory scripts:

```
input(scripts#test)
```

B.4.3 Call a UNIX Command - {· · ·}

```
{Command}
```

may be used to execute a UNIX command directly. Examples:

```
{ls}
{"cat notes"}
```

B.4.4 Display Named Variables ^

VariableName^

may be used to display a named variable.

^

may be used to display all named LOGIX variables.

Appendix C

Auxilliary LOGIX Procedures

C.1 record

The psi_record process can run a PSI process and record its behavior.

```
psi_record#run(<call program>, <time limit>)
psi_record#run(<call program>, <file_name>, <time limit>)
psi_record#run(<call program>, <file_name>, <time limit>, <scale>)
```

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

- `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 `<control>`C to kill LOGIX.

- `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 "time".
  - `+<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.

To analyze the file, producing a table suitable for plotting with Matlab, use the PERL program "psi2t" - *e.g.*

```
% psi2t fff
```

creates a table, where column 1 is time, and columns $2 \cdots n$ 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.

```
% psi2t fff 0.1
```

To combine columns sums add terms of the form:

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

```
% psi2t 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:

```
% psi2t 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, psixys, combines the functions with a call to gnuplot.

C.2 repeat

This LOGIX procedure is called by the `run` and `record` macros - see Appendix B.2.1 and Appendix B.2.2.
The repeat process can run a quantified set of PSI processes.

```
repeat#run(<quantified process set>)
```

where:

```
<quantified process set> ::=
    <external call>
    (<process set>)
    <repetition> * (<process set>)

<process set> ::=
    <quantified process set>
    <process set> , <quantified process set>

<repetition> ::= <integer>
```

A negative `<repetition>` is treated as zero. (See Appendix A for the definition of `<external_call>`.)
Examples:

- `repeat#run(64*(dimerization#"A_PROTEIN"))`
- ```
  repeat#run([6*(activator#"A_PROTEIN"),
              activator#"A_GENE",repressor#"R_GENE"])
  ```
- ```
  repeat#run([2*[activator#A_GENE,3*(repressor#R_GENE)],
              hysteresis#module])
  ```

Note that the parentheses are necessary in the case of:

```
<repetition>*(<external_call>)
```

Here is an example call to run

```
repeat#run([3*(bland#"RunTT"),4*(bland#"RunFT")])
<2> started
It's false
It's false
It's false
It's false
It's true
It's true
It's true
@spr
<2> suspended
bland # AndB.1.1.comm(Test.1.t, Test.1.f, global.b2!, AndB.1.x!)
bland # TT.comm(global.b2!)
bland # AndB.1.1.comm(Test.1.t, Test.1.f, global.b2!, AndB.1.x!)
bland # TT.comm(global.b2!)
bland # AndB.1.1.comm(Test.1.t, Test.1.f, global.b2!, AndB.1.x!)
bland # TT.comm(global.b2!)
bland # AndB.1.1.comm(Test.1.t, Test.1.f, global.b2!, AndB.1.x!)
bland # TT.comm(global.b2!)
@
```

C.3 Weight Computation for a Channel

Channels with finite rates are weighted for selection using the default computations.
The user may specify custom computations using the notation for `<weighter_declaration>` in Appendix A; the computation must be explicitly coded in the module `Logix/<emulator name>/psiweight.c` .
Module `<emulator>/psiweight.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 `psi_compute_bimolecular_weight` and/or in the function `psi_compute_homodimerized_weight` .

- Re-install LOGIX - `psiweight.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 "argn" 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.