Outline for an Operational-Semantics Definition of PROMELA

V. Natarajan *
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
nvaidhy@eos.ncsu.edu

Gerard J. Holzmann
Bell Laboratories
Murray Hill, NJ, USA
gerard@research.bell-labs.com

July 7, 1996

Abstract

PROMELA is a high-level specification language for modeling interactions in distributed systems, and for expressing logical correctness requirements about such interactions. The model checker SPIN accepts specifications written in this language, and it can produce automated proofs for each type of property. SPIN either proves that a property is valid in the given system, or it generates a counter-example that shows that it is not. This paper contains the outline for an operational-semantics definition of PROMELA.

1 Introduction

PROMELA is a language for reasoning about concurrent systems. The language has developed over the last fifteen years from a simple modeling language into a more complete specification language. The language, supported by the model checker SPIN, is used in a growing number of design projects, both large and small.

As new constructs are introduced into the language (e.g., [3]), their interplay with existing constructs can have subtle consequences. Some combinations of constructs can reasonably be interpreted in more then one way. So far, only the implementation of the model checker SPIN can provide definitive answers about semantics issues. A formal semantics definition can provide an implementation independent way to resolve possible conflicts of interpretation.

This paper gives an outline for a semantics basis of the language. To simplify the exposition, it imposes a number of restrictions. It will not discuss, for instance, *array variables* or *structures*, nor will it discuss *assert* statements, *never claims*, or correctness properties expressed by *progress* and *accept* labels. To simplify the definitions somewhat, we will also restrict *run statements* to one single parameter, and *channel declarations* to a single message field.

The remainder of this paper is organized as follows. Section 2 introduces notational conventions, and some semantic preliminaries. Section 3 formalizes the notion of *executability* of PROMELA statements. Section 4 presents how global states are transformed by the execution of statements. The notion of the execution of a program is captured in Section 5. Section 6 gives a small example of the conversion of a program into a labeled transition system. Section 7 summarizes the report.

2 Preliminaries

A PROMELA program consists of zero or more global variable declarations, and one or more process type declarations (*proctypes*). The verifier SPIN translates each process type declaration into a labeled transition system, in which the labels are primitive statements [2]. We define below

*Work done during the first author's stay at Bell Laboratories in the summer of 1995.

what a *primitive statement* is. For the time being it will suffice to note that a primitive statement is a state transformer: when it is *executable* (to be defined) it modifies the *current state* (to be defined) in a precise way (to be defined). Note that compound statements (such as *if .. fi*, *do .. od*, *unless* (introduced in [3]), and *goto* are not primitive statements, and neither are *else* statements or declarations.

Converting compound statements, or generally process type declarations, into labeled transition systems is a fairly straightforward procedure that is not detailed here. (It will be detailed in the full version.) Labeled transition systems are formally defined in Section 2.4. An example of the translation from a *proctype* declaration into a labeled transition system is given in Section 6. The conversion is performed by SPIN when it parses its input. The result can be seen after a verifier is generated and compiled, with: `pan -d`. (A graphical representation can also be seen with the *FSM View* option in XSPIN.)

2.1 Notational Conventions

If S is a nonempty set then S^i will denote the set of sequences of length i whose elements belong to S. If x is a sequence, then $x.i$ denotes the ith element of x. We will often assign symbolic names to specific positions in a sequence; if *sym* is the symbolic name for the ith position of the sequence x then $x.sym$ will denote $x.i$. If x is any mathematical structure then $x[y/sym]$ will stand for the same structure in which the substructure denoted by *sym* is replaced by y. If f is a function then $f[sym = y]$ will denote the function which takes the value y at *sym* and for all other arguments has the same value as f. If x is a sequence and y is some element then $x \circ y$ is the sequence obtained by appending y to the end of x. If x is a sequence then $cdr(x)$ is the sequence obtained from x by deleting its first element.

2.2 Syntactic Entities

We define the following syntactic entities: *Vars*, *Consts*, *Exprs*, *Asgns*, *SendExprs*, *RecExprs* and *RunExprs*.

Vars is the set of all variable names.

Consts is the set of all constants.

The term *var* refers to an element of *Vars*, *const* refers an element of *Consts* and *convar* refers to an element of either *Vars* or *Consts*.

Exprs is the set of expressions.

The term *expr* refers to an element of *Exprs*. The elements of *Exprs* are defined by the context-free grammar:

$bop ::= + \mid - \mid * \mid > \mid < \mid EQ \mid \mathrm{NE} \mid \mathrm{AND} \mid \mathrm{OR}$

$uop ::= - \mid !$

$expr ::= convar \mid (expr) \mid expr\ bop\ expr \mid uop\ expr \mid \mathrm{LEN}\ var \mid timeout \mid var?[convar]$

The next four sets define different types of statements, or *state transformers*.

Asgns is the set of assignment statements, with elements of the form '*var* = *expr*' .

SendExprs is the set of send statements, with elements of the form: '*var*!*expr*' .

RecExprs is the set of receive statements, with elements of the form '*var*?*convar*' .

RunExprs is the set of run statements, with elements of the form '*run Name expr*' .

Definition 2.1 *A* **primitive statement** *is an element of one of the sets: Exprs, Asgns, SendExprs, RecExprs, or RunExprs.*

Note that we have restricted *run* statements to a single parameter (in addition to the *proctype* name) and similarly we have restricted send and receive operations (and thereby channel declarations) to a single message field. These restrictions are meant to improve the readability of the definitions that follow.

2.3 Data Types and Value Domains

BTypes is the set of *basic* types, which we restrict here to $\{bit, byte, short, int\}$. *Types* is the set of types given by $BTypes \cup \{chan\}$.

To ensure that only finite state programs are defined, PROMELA sets finite upper bounds on certain parameters. We introduce symbolic names for these upper bounds. *MaxChanCap* is the maximum number of messages a channel can hold (all channels have finite capacity). *MaxNumProc* is maximum number of processes that can be active at a time, and *MaxNumChan* is the maximum number of channels that can be accessible at a time. For notational convience, we also define the following sets.

$MaxChanCapSet = \{\ i \mid 0 \leq i < MaxChanCap \wedge i \in Naturals\}$
$MaxNumChanSet = \{\ i \mid 0 \leq i \leq MaxNumChan \wedge i \in Naturals\}$
$MaxNumProcSet = \{\ i \mid 0 \leq i < MaxNumProc \wedge i \in Naturals\}$

The function *ValDomMap* specifies the values a variable of a given type can take during the execution of a program (Page 93 of [1]).

ValDomMap is a map from *BTypes* to $2^{Integers}$ defined as follows:
$ValDomMap(bit) = \{\ i \mid 0 \leq i \leq 1 \wedge i \in Naturals\}$
$ValDomMap(byte) = \{\ i \mid 0 \leq i \leq 255 \wedge i \in Naturals\}$
$ValDomMap(short) = \{\ i \mid -2^{15} \leq i \leq 2^{15} - 1 \wedge i \in Integers\}$
$ValDomMap(int) = \{\ i \mid -2^{31} \leq i \leq 2^{31} - 1 \wedge i \in Integers\}$
$ValDomMap(chan) = MaxNumChanSet$

The above definition states, for example, that a variable of type *bit*, can take only one of the two values 0 or 1.

2.4 Labeled Transition Systems

A *symbolic labeled transition system* is the mathematical representation of the information contained in the body of a process type declaration (i.e., a PROMELA *proctype*).

Definition 2.2 *A* **symbolic labeled transition system** *is a tuple of the form*

$$\langle Name, Structure, Start, locals, ChansOwned, Active, Param\rangle$$

. *where*

- *Name is an element of Vars* (the name of the proctype).
- *Structure is* $\langle lstates, Act, \longmapsto\rangle$ *where lstates is a finite set of local states and transition relation* $\longmapsto \subseteq lstates \times Act \times lstates$. *Act is* $PLabels \times EModes \times Priorities$ *and its elements will be ranged over by* $\alpha, \beta, \cdots$. *PLabels is the set of primitive statements, i.e., the union of Exprs, Asgns, SendExprs, RecExprs and RunExprs. EModes is the set of execution modes* $\{normal, atomic, deterministic\}$. *Priorities is the set of natural numbers.*

  Informally, *Structure* encodes the control-flow of the sequential program named *Name* as a labeled directed graph. The nodes of this graph are the elements of set *lstates*. Each node represents a distinct local state of *Name*. Relation $\longmapsto$ represents the labeled edges between the nodes. If $\langle n_1, \alpha, n_2\rangle \in \longmapsto$ then it is possible to move to state n_2 from state n_1 by executing the primitive statement $\alpha.1$. The label of the edge also contains information about whether the statement $\alpha.1$ is to executed *normally*, *atomically* or *deterministically*. We call this the execution mode, and encode it as $\alpha.2$. Finally, $\alpha.3$ represents the *priority* of the transition. Priorities will be used to properly encode *unless* statements. A transition with a lower priority will not be selected for execution if one with strictly higher priority is also executable.

- *Start is an element of lstates.*

  It represents the initial state of the proctype.

- *locals is a* partial *function from the Vars to* $\cup\{$ *ValDomMap*$(\tau) \mid \tau \in$ *BTypes*$\}$.

  The partial function *locals* identifies the local variables of *Name*. Note that the function *locals* is partial; given a variable name *var*, *locals*(*var*) is undefined if *var* is not a local variable of *Name*. If *var* is indeed a local variable, then *locals*(*var*) is 0 (the default value, it it is uninitialized) or the value to which it is initialized. *locals* does not contain information about channel variables.

- *ChansOwned is a finite* sequence *of elements from* (*Vars* $\times$ (*MaxChanCapSet* $\cup \{\Omega\}) \times$ (*Types* $\cup \{\Omega\}$).

  The entity *ChansOwned* records the information about the channel variables accessed by this sequential program; specifically, it records the capacity of each channel (the maximum number of messages the individual channel can hold) and the data type of its message field. For an uninitialized variable of type *chan* these values above are set to Ω.

- *Active is an integer which is either 0 or 1.*

- *Param is a member of Vars.*

  This is a single formal parameter that is bound to a value when the process is instantiated.

(End of definition 2.2.)

2.5 Local and Global States

Lstates is the set of *local states* (i.e., process states). *Gstates* is the set of *global states* (i.e., system states). Formally, a *global state* Σ is a pair of the form $\langle X, g\rangle$. X is a finite set of elements of the form $\langle$*Name*, *pid*, *pc*, *locals*, *lChansOwned*$\rangle$. Each process active in the state Σ is represented by some member of X exactly once.

1. *Name* is an element of *Vars*.

   It is the name of the proctype corresponding to this process. Note that there could be more than one active process corresponding to the same proctype.

2. *pid* belongs to $MaxNumProcSet$.

   It is the unique identifier corresponding to an active process. In other words, if x and x' are two *distinct* members of X then $x.2 \neq x'.2$. Thus any member of X can be uniquely identified by *pid*. Thus we denote $X : i$ to denote that particular element of x of X such that $x.pid = i$.

3. *pc* belongs to *Lstates*

   It is the program counter of the active process; it points to some local state.

4. *locals* is a partial function from *Vars* to $\cup\{$ *ValDomMap*$(\tau) \mid \tau \in$ *BTypes*$\}$.

   It is the set of local variables of the active process.

5. *lChansOwned* is a finite sequence of elements from (*Vars* $\times$ (*MaxChanCapSet* $\cup \{\Omega\}) \times$ (*Types* $\cup \{\Omega\}$).

   It is the list of channels owned by this active process.

The second of component g of the global state Σ is a tuple of the kind

$\langle$*globals*, *gChansOwned*, *NrProcs*, *Handshake*, *Exclusive*, *ChIdMap*, *ChContMap*, *timeout*, *dstep*$\rangle$

where

1. *globals* is a partial function from *Vars* to $\cup\{$ *ValDomMap*$(\tau) \mid \tau \in$ *BTypes*$\}$

   It is the set of global variables.

2. *gChansOwned* is a finite sequence of elements from $(\mathit{Vars} \times (\mathit{MaxChanCapSet} \cup \{\Omega\}) \times (\mathit{Types} \cup \{\Omega\})$.

   It is the list of globally declared channels.

3. *NrProcs* is an element of *MaxNumProcSet*.

   $\mathit{NrProcs} + 1$ is the number of processes currently active at this global state. It cannot exceed *MaxNumProc*.

4. *Handshake* is either an empty set or an element of $\mathit{MaxNumProcSet} \times \mathit{MaxNumChanSet} \times \mathit{Consts}$.

   If *Handshake* is not empty it means that there is a synchronous communication (a rendezvous handshake) in progress; the active process with process-id *Handshake*.1 is sending the value *Handshake*.3 over the channel *Handshake*.2.

5. *Exclusive* is either empty or an element of *MaxNumProcSet*.

   If *Exclusive* is non-empty the process with process-id *Exclusive* is executing atomically or deterministically, and cannot be interrupted by other process executions.

6. *ChIdMap* is a partial function from $(\mathit{MaxNumProcSet} \cup \{-1\}) \times \mathit{Vars}$ to *MaxNumChanSet*.

   This partial function assign an unique channel-id to every every active channel; given a process-id and a variable name as arguments this function provides the channel-id. Global channels are assigned process-id -1. If the variable name is not a channel name for that process-id then the function is undefined on these arguments.

7. *ChContMap* is a partial function from *MaxNumChanSet* to

$$\bigcup\{\ [\cup\{\ \mathit{ValDomMap}(\tau) \mid \tau \in \mathit{Types}\}]^i \mid 0 \leq i \leq \mathit{MaxChanCap}\}$$

   Given a channel-id, this maps provides the current contents of that channel.

8. *timeout* and *dstep* are integers which take value either 0 or 1.

   If all processes in the system are blocked (i.e., there is no executable statement) when the value of *timeout* equals 0, then the value of changes to 1 for one execution step, and returns to its default value 0. If *dstep* equals 1, some process is executing deterministically.

2.6 Auxiliary Functions

This subsection is devoted a list of auxiliary functions that are used for querying and updating global states.

2.6.1 Characteristic Functions

When the same identifier is used to define both a global variable and a local variable, the usual scoping rules determine which variable is intended. The following functions can be used to resolve these issues.

Islchan is a function defined from $\mathit{Vars} \times \mathit{MaxNumProcSet} \times \mathit{Gstates}$ to the set of truth values $\{\ \mathit{tt}, \mathit{ff}\ \}$ as follows. $\mathit{Islchan}(\mathit{var}, \mathit{pid}, \langle X, g\rangle)$ is *tt* if there exists an x in $(X : \mathit{pid}).\mathit{lChansOwned}$ such that $x.1$ equals *var* and *ff* otherwise. Intuitively, $\mathit{Islchan}(\mathit{var}, \mathit{pid}, \langle X, g\rangle)$ is true iff in the global state $\langle X, g\rangle$, *var* is a local channel owned by a process whose process-id is *pid*.

Islnumvar is a function defined from $\mathit{Vars} \times \mathit{MaxNumProcSet} \times \mathit{Gstates}$ to the set of truth values $\{\ \mathit{tt}, \mathit{ff}\ \}$ as follows. $\mathit{Islnumvar}(\mathit{var}, \mathit{pid}, \langle X, g\rangle)$ is *tt* if $(X : \mathit{pid}).\mathit{locals}(\mathit{var})$ is well defined and *ff* otherwise.

Isgchan is a partial function defined from $\mathit{Vars} \times \mathit{MaxNumProcSet} \times \mathit{Gstates}$ to the set of truth values $\{\ \mathit{tt}, \mathit{ff}\ \}$ as follows. $\mathit{Isgchan}(\mathit{var}, \mathit{pid}, \langle X, g\rangle)$ is *tt* if $\mathit{Islnumvar}(\mathit{var}, \mathit{pid}, (X : \mathit{pid}).)$ as well

Islchan($var, pid, \langle X, g\rangle$) are *ff* and there exists an x in $g.gChansOwned$ such that $x.1$ equals var and *ff* otherwise. Intuitively, a variable is a global channel variable only if there is no local numeric or local channel variable of the same name.

Ischan is a function defined from $Vars \times MaxNumProcSet \times Gstates$ to the set of truth values { *tt*,*ff* } as follows. *Ischan*(var, pid, Σ) is *tt* if either *Isgchan*(var, pid, Σ) is *tt* or *Islchan*(var, pid, Σ) is *tt* and *ff* otherwise.

Isgnumvar is a function defined from $Vars\times MaxNumProcSet\times Gstates$ to the set of truth values { *tt*,*ff* } as follows. *Isgnumvar*($var, pid, \langle X, g\rangle$) is *tt* if $g.globals(var)$ is well defined and *Islnumvar*(var) is *ff* and *Islchan*(var) is *ff*. Otherwise it is *ff*.

Isnumvar is a partial function defined from $Vars \times MaxNumProcSet \times Gstates$ to the set of truth values { *tt*,*ff* } as follows. *Isnumvar*(var, pid, Σ) is *tt* if either *Isgnumvar*(var, pid, Σ) is *tt* or *Islnumvar*(var, pid, Σ) is *tt* and *ff* otherwise.

2.6.2 Other Auxiliary Functions

We define some more auxiliary functions that depend on the characteristic functions. The function *ChanId*, given a variable name and a process-id provides the unique channel-id associated with it if it is indeed a channel. Its definition makes use of the characteristic functions to ensure the proper scoping rule is enforced. Similary the function *RightVal*, given a variable and a process-id provides the right value associated with the variable; if it is a channel variable then its channel-id is returned.

ChanId is a partial function defined from $Vars \times MaxNumProcSet \times Gstates$ to the set $(MaxNumChanSet \cup \Omega)$ as follows. *ChanId*(var, pid, Σ) is Ω if *Ischan*(var, pid, Σ) is *ff*. If *Islchan*(var, pid, Σ) is *tt* then *ChanId*(var, pid, Σ) is *ChIdMap*($\langle pid, var\rangle$). If *Isgchan*(var, pid, Σ) is *tt* then *ChanId*(var, pid, Σ) is *ChIdMap*($\langle -1, var\rangle$).

RightVal is a partial function defined from $Vars \times MaxNumProcSet \times Gstates$ to the set *Integers* as follows. If *Islchan*(var, pid, Σ) is *tt* then *RightVal*(var, pid, Σ) is *ChanId*(var, pid, Σ). If *Islnumvar*(var, pid, Σ) is *tt* then *RightVal*(var, pid, Σ) is *locals*(var). If *Isgnumvar*(var, pid, Σ) is *tt* then *RightVal*(var, pid, Σ) is *globals*(var). If *Isgchan*(var, pid, Σ) is *tt* then *RightVal*(var, pid, Σ) is *ChanId*($var, -1, \Sigma$).

capacity is a partial function defined from $Vars \times MaxNumProcSet \times Gstates$ to the set *MaxChanCapSet* as follows. If *Islchan*(var, pid, Σ) is *tt* then *capacity*(var, pid, Σ) is $x.2$ where x is a element of the sequence *lChansOwned* such that $x.1$ is same as *var*. If *Isgchan*(var, pid, Σ) is *tt* then *capacity*(var, pid, Σ) is $x.2$ where x is a element of the sequence *gChansOwned* such that $x.1$ is same as *var*.

compatible($var_1, var_2, pid, \Sigma$) is *tt* if *RightVal*($var_1, pid, \Sigma$) = *RightVal*($var_2, pid, \Sigma$) and *ff* otherwise. *compatible*($const, var, pid, \Sigma$) is *tt* if *RightVal*(var_1, pid, Σ) = *const* and *ff* otherwise. *compatible*($const_1, const_2$) is *tt* if $const_1 = const_2$. Synchronous communication can happen only if there is some compatibility between the sender and the receiver; the above function formalizes this relationship.

2.6.3 Evaluation of Expressions

We define the function $[\![\ ]\!]_{pid,\Sigma}$ to provides the value of the given expression. This function returns Ω if an error occurs during the evaluation.

The function $[\![\ ]\!]_{pid,\Sigma}$ from *Exprs* to $Integers \cup \{\Omega\}$ is defined as follows. Let $\Sigma = \langle X, g\rangle$. If there is no element x of X with $x.2$ equals *pid* then $[\![\ ]\!]_{pid,\Sigma}$ is a constant function with value Ω.

Otherwise let x be the element $\langle Name, pid, pc, locals, lChansOwned\rangle$. Now $[\![expr]\!]_{pid,\Sigma}$ is given by structural induction below. For readability, we omit the subscripts for $[\![\ ]\!]$ below.

- If $expr \equiv const$ then $[\![expr]\!] = const$.
- If $expr \equiv timeout$ then $[\![expr]\!] = g.timeout$.
- If $expr \equiv var$ then $[\![expr]\!] = RightVal(var, pid, \Sigma)$.

- If $expr = \mathrm{LEN}\, var$ and $Ischan(var, pid, \Sigma) = \mathit{ff}$ then $[\![expr]\!] = \Omega$.

- If $expr = \mathrm{LEN}\, var$ and $Ischan(var, pid, \Sigma) = tt$ and $[\![var]\!] = 0$ then $[\![expr]\!] = \Omega$.

- If $expr = \mathrm{LEN}\, var$ and $Ischan(var, pid, \Sigma) = tt$ and $[\![var]\!] \neq 0$ and $ChContMap([\![var]\!])$ is undefined then $[\![expr]\!] = \Omega$

- If $expr = \mathrm{LEN}\, var$ and $Ischan(var, pid, \Sigma) = tt$ and $[\![var]\!] \neq 0$ and $ChContMap([\![var]\!])$ is well defined then $[\![expr]\!] = |ChContMap([\![var]\!])|$.

- If $expr = var?[convar]$ and $(Ischan(var) = \mathit{ff}$ or $[\![\mathrm{LEN}\, var]\!] = \Omega)$ then $[\![expr]\!] = \Omega$

- If $(expr = var?[convar]$ and $[\![\mathrm{LEN}\, var]\!] = 0$ then $[\![expr]\!] = 0$

- If $expr = var?[convar]$ and $Ischan(var) = tt$ and $[\![\mathrm{LEN}\, var]\!] \notin \{0, \Omega\}$ and $compatible(first(var_1, pid, \Sigma), [\![convar]\!], pid, \Sigma) = tt$ then $[\![expr]\!] = 1$

- If $expr = var?[convar]$ and $Ischan(var) = tt$ and $[\![\mathrm{LEN}\, var]\!] \notin \{0, \Omega\}$ and $compatible(first(var_1, pid, \Sigma), [\![convar]\!], pid, \Sigma) = \mathit{ff}$ then $[\![expr]\!] = 0$

- If $expr = (expr_1)$ then $[\![expr]\!] = [\![expr_1]\!]$

- If $expr = \mathrm{uop}\ expr_1$ then $[\![expr]\!] = [\![uop]\!]([\![expr_1]\!])$.

  If $expr = expr_1\ \mathrm{bop}\ expr_2$ then $[\![expr]\!] = [\![bop]\!]([\![expr_1]\!], [\![expr_2]\!])$.

first is a partial function defined from $Vars \times MaxNumProcSet \times Gstates$ to the set *Integers* as follows. $first(var, pid, \Sigma) = ChContMap([\![var]\!]).1.2$.

2.6.4 Execution Errors, Failure

As an example of an execution error, consider the statement $x = 5$, where x is channel variable. The function *WillFail* formally describes these situations.

WillFail is a partial function defined from $PLabels \times MaxNumProcSet \times Gstates$ to the set of truth values $\{\ tt, \mathit{ff}\ \}$ as follows.

- $WillFail(expr, pid, \Sigma)$ is tt if $[\![expr]\!]$ is Ω and ff otherwise.

- $WillFail(var = expr, pid, \Sigma)$ is tt if ($[\![expr]\!]$ is Ω) or (exactly one of $Ischan(var)$ and $Ischan(expr)$ is tt). Otherwise $WillFail(var = expr, pid, \Sigma)$ is ff.

- $WillFail(var!expr, pid, \Sigma)$ is tt if $[\![expr]\!]$ is Ω or ($Ischan(var)$ is ff) or ($Ischan(var)$ is $tt \wedge ChContMap(ChanId(var, pid, \Sigma))$ is undefined).

  Otherwise $WillFail(var!expr, pid, \Sigma)$ is ff.

- $WillFail(var?convar, pid, \Sigma)$ is tt if ($[\![convar]\!]$ is Ω) or ($Ischan(var)$ is ff) or ($Ischan(var)$ is tt and $ChanId(var, pid, \Sigma) = 0$)

  or ($Ischan(var)$ is $tt \wedge ChContMap(ChanId(var, pid, \Sigma))$ is undefined.

  Otherwise $WillFail(var?convar, pid, \Sigma)$ is ff.

2.6.5 Channel Manipulation Functions

When a new process is created, all the channels owned by it are entered in the *ChIdMap*. Similary when a process terminates, it local channels are removed from the *ChIdMap*. The functions defined in this subsubsection accomplish this task.

Definition 2.3 *Incorp*(*ChIdMap*, *ChansOwned*, *pid*) *is defined as a partial function f from the set* (*MaxNumProcSet* $\cup \{-1\}) \times$ *Vars to MaxNumChanSet. f has the same value on those elements at which ChIdMap is well defined. Further if $\langle var, \Omega, \Omega\rangle) \in$ ChansOwned then $f(\langle pid, var\rangle)$ is defined as 0. Let $c_1 \cdots c_k$ be the sequence ChansOwned with no elments of the form $\langle var, \Omega, \Omega\rangle$. Let m be the maximum value taken by ChIdMap. Then $f(pid, c_i.1)$ is defined to be $m + i$ for $1 \leq i \leq k$.*

The function *Incorp* augments the information in the *ChIdMap* with that present in *ChansOwned* to produce another function that is of the same type as *ChIdMap*.

Definition 2.4 *IncorpCont*(*ChContMap*, *ChIdMap*, *ChansOwned*, *pid*) *is a partial function f that has the same domain and codomain as that of ChContMap. $f(x)$ is defined as an empty sequence if there exists a $c \in$ ChansOwned such that ChIdMap(pid, c) $= x$ and $x \neq 0$. At other arguments f and ChContMap are identical.*

The following two functions remove information from the *ChIdMap* and *ChContMap*.

Definition 2.5 *RetractId*(*ChIdMap*, *ChansOwned*, *pid*) *is a partial function f from the set* (*MaxNumProcSet* $\cup \{-1\}) \times$ *Vars to MaxNumChanSet. $f(pid, c)$ is undefined if there exists a $x \in$ ChansOwned such that $x.1 = c$. At other arguments, f and ChIdMap are identical.*

Definition 2.6 *RetractCont*(*ChContMap*, *ChIdMap*, *ChansOwned*, *pid*) *is a partial function f that has the same domain and codomain as that of ChContMap. $f(x)$ is undefined if there exists a $c \in$ ChansOwned such that ChIdMap(pid, c) $= x$. At other arguments f and ChContMap are identical.*

3 Preconditions

We refer to the directed labeled edges of a symbolic transition system as transitions. This section gives the definition of a *transition* and a *current transition*. A current transition is *executable* in a global state only if certain conditions are satisfied. This section formalizes these notions.

Definition 3.1 *If S is a symbolic labeled transition system with $S.1 =$ Name and $S.2.3$ contains $\langle s, \alpha, t\rangle$ then $\langle Name, s, \alpha, t\rangle$ will be called a transition.*

Informally, a transition is *current* in a global state Σ if the program counter of some active sequential program points to the source state of the transition.

Definition 3.2 *A* current *transition of a state $\Sigma = \langle X, g\rangle$ is $\langle pid, Name, s, \alpha, t\rangle$ if $\langle Name, s, \alpha, t\rangle$ is a transition and $\langle Name, pid, s, locals, lChansOwned\rangle$ is a member of X for some locals and lChansOwned.*

Next, we define when a current transition is said to be *executable*.

Definition 3.3 *A current transition $\langle Name, pid, s, \alpha, t\rangle$ is* executable *in the state $\Sigma = \langle X, g\rangle$ if all of the following hold.*

1. *$g.Exclusive = \{\}$ or $g.Exclusive = \{pid\}$.*

   Intuitively, this condition states that in order for a current transition to be executable no *other* process may have exclusive control.

2. *WillFail*($\alpha.label, pid, \Sigma) = tt$
   $\Rightarrow g.Handshake = \{\}$.

   If a synchronous communication is in progress, a current transition is not permitted to execute and cause an error.

3. $(\alpha.label \equiv expr) \wedge WillFail(\alpha.label, pid, \Sigma) = ff$
   $\Rightarrow$
   $g.Handshake = \{\} \wedge ([\![expr]\!]_{\Sigma,pid} \neq 0)$.

   An expression is executable if it value is nonzero and if no synchronous communication is in progress

4. $(\alpha.label \equiv var!expr \wedge WillFail(\alpha.label, pid, \Sigma) = ff \wedge capacity(var, pid, \Sigma) \neq 0)$
   $\Rightarrow$
   $g.Handshake = \{\} \wedge [\![LENvar]\!]_{pid,\Sigma} < capacity(var, pid, \Sigma)$.

   The statement that sends a value of the *expr* to the channel *var* is executable only if the the channel *var* is not full. (see Page 99 of [1])

5. $(\alpha.label \equiv var?convar \wedge WillFail(\alpha.label, pid, \Sigma) = ff \wedge capacity(var, pid, \Sigma) \neq 0$
   $\Rightarrow$
   $g.Handshake = \{\} \wedge [\![var?[convar]]\!]_{pid,\Sigma} = 1$.

   Similarly the statement that extracts a value from the channel *var* is executable only if the channel *var* is nonempty (see Page 100 of [1]).

6. $(\alpha.label \equiv var!expr \wedge WillFail(\alpha.label, pid, \Sigma) = ff \wedge capacity(var, pid, \Sigma) = 0$
   $\Rightarrow$
   $g.Handshake = \{\} \wedge \exists qid, Name', s', t', x, \beta : \; qid \neq pid \wedge \beta.label \equiv var'_1?convar \wedge$
   $[\![var'_1]\!]_{\Sigma,qid} = [\![var]\!]_{\Sigma,pid} \wedge \langle qid, Name', s', \beta, t'\rangle$ *is a current transition of* $\Sigma \wedge$
   $compatible(convar, [\![expr]\!]_{g,pid}, pid, \Sigma)$.

   A synchronous-send statement on a channel *var* is executable only if there is another distinct process that has synchronous-receive statement on the same channel as a current transition. (see Page 100 of [1])

7. $(\alpha.label \equiv var?convar \wedge WillFail(\alpha.label, pid, \Sigma) = ff \wedge capacity(var, pid, \Sigma) = 0$
   $\Rightarrow$
   $\exists qid, const : \; qid \neq pid \wedge g.Handshake = \{\langle qid, [\![var]\!]_{pid,\Sigma}, const\rangle\}$
   $\wedge\; compatible(convar, const, pid, \Sigma)$.

8. $(\alpha.label \equiv asgn) \wedge WillFail(\alpha.label, pid, \Sigma) = ff$
   $\Rightarrow$
   $g.Handshake = \{\}$.

9. $(\alpha.label \equiv run\ Name\ arg)) \wedge WillFail(\alpha.label, pid, \Sigma) = ff$
   $\Rightarrow$
   $g.Handshake = \{\} \wedge g.NrProcs < MaxNumProc - 1$.

   A new process can be created only if the number of currently active processes does not exceed the preset upper bound.

10. $(\alpha.label \equiv end)$
    $\Rightarrow$
    $g.Handshake = \{\} \wedge pid = g.NrProcs$.

    Processes can only terminate in the reverse order of their creation (i.e., a process cannot terminate until all its children have terminated first). If processes are issued pid's sequentially starting from zero, then the only process that can terminate is the one that with a pid equal to $g.NrProcs$. (see Page 96 of [1])

4 Effects

In this section we define how each of the primitive statements transforms the current global state when it is executed.

4.1 Normal Execution Mode

The effect of the execution of a statement depends in part on the "mode" in which it is executed. The following definition presents the effects of statements executed in "normal" mode (i.e., not involving *atomic* or *dstep* sequences).

Definition 4.1 *If* $\langle pid, Name, s, \alpha, t\rangle$ *is* executable *in the state* $\Sigma = \langle X, g\rangle$ *with* $\alpha.mode = normal$ *then* $Effect(\Sigma, \langle pid, Name, s, \alpha, t\rangle)$ *is* $\Sigma' = \langle X', g'\rangle$ *where*

1. $WillFail(\alpha.label, pid, \Sigma) = tt \Rightarrow \Sigma' = \Omega$

   We use the symbol Ω to denote the global error state. The above condition states that the execution of an error-producing statement will transform any state into the error state.

2. $(\alpha.label \equiv expr) \wedge WillFail(\alpha.label, pid, \Sigma) = ff \wedge [\![expr]\!]_{pid,\Sigma} \neq 0$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
   $g' = g[Exclusive = \{\}, dstep = 0]$

   The program counter of the process is updated. No exclusive execution control is obtained or preserved by any statement that is executed in normal mode.

3. $(\alpha.label \equiv var = expr \;\wedge\; WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals', lChansOwned\rangle]$
   $g' = g[Exclusive = \{\}, dstep = 0, globals'/globals, ChIdMap'/ChIdMap]$ *where*

   - $islnumvar(var) = tt$
     $\Rightarrow (locals' = locals[var = [\![expr]\!]_{pid,\Sigma}], globals' = globals, ChIdMap' = ChIdMap)$
   - $isgnumvar(var) = tt$
     $\Rightarrow (globals' = globals[var = [\![expr]\!]_{pid,\Sigma}], locals' = locals, ChIdMap' = ChIdMap)$
   - $ischan(var) = tt$
     $\Rightarrow (locals' = locals, globals' = globals, ChIdMap' = ChIdMap[ChanId(var, pid, \Sigma) = [\![expr]\!]_{pid,\Sigma}])$

   The effect of this statement is to assign the value of the expression to the variable. If the variable is a channel variable then the *ChIdMap* is updated. The scope rules determine if the variable is local or global.

4. $(\alpha.label \equiv var!expr \wedge capacity(var, pid, \Sigma) \neq 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
   $g' = g[Exclusive = \{\}, dstep = 0, ChContMap'/ChContMap]$ *where*
   $ChContMap' =$
   $ChContMap[ChIdMap(pid, var, \Sigma) = ChContMap(ChIdMap(pid, var, \Sigma)) \circ [\![expr]\!]_{pid,\Sigma}]$

   To understand the above condition, recall the meaning of the notation $f[x = y]$ from Subsection 2.1; the value of the expression is appended at the end of the contents of the channel *var*. (See Page 98 of [1]).

5. $(\alpha.label \equiv var?const \wedge capacity(var, pid, \Sigma) \neq 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
   $g' = g[Exclusive = \{\}, dstep = 0, ChContMap'/ChContMap]$ *where*
   $ChContMap' =$
   $ChContMap[ChIdMap(pid, var, \Sigma) = cdr(ChContMap(ChIdMap(pid, var, \Sigma)))]$

   Since this transition is executable it is necessarily the case that the head of the channel *var* is *const*.

6. $(\alpha.label \equiv var_1?var_2 \wedge capacity(var_1, pid, \Sigma) \neq 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
$\Rightarrow$
$X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals', lChansOwned\rangle]$
$g' = g[Exclusive = \{\}, globals'/globals, ChIdMap'/ChIdMap, dstep = 0,$
$ChContMap'/ChContMap]$ *where*

   - $islnumvar(var_1) = tt$
   $\Rightarrow (locals' = locals[var_2 = first(var_1, pid, \Sigma)], globals' = globals, ChIdMap' = ChIdMap)$
   - $ischan(var) = tt$
   $\Rightarrow (locals' = locals, globals' = globals, ChIdMap' = ChIdMap[ChanId(var_2, pid, \Sigma) = first(var_1, pid, \Sigma)]$
   - $isgnumvar(var) = tt$
   $\Rightarrow (globals' = globals[var_2 = first(var_1, pid, \Sigma)], locals' = locals, ChIdMap' = ChIdMap)$
   - $ChContMap' =$
   $ChContMap[ChIdMap(pid, var, \Sigma) = cdr(ChContMap(ChIdMap(pid, var, \Sigma)))]$

   The value at the head of the channel var_1 is removed and assigned to the value var_2, just like in a normal PROMELA assignment statement.

7. $(\alpha.label \equiv var!expr \wedge capacity(var, pid, \Sigma) = 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
$\Rightarrow$
$X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
$g' = g[Exclusive = \{\}, dstep = 0, Handshake = \{\langle pid, ChanId(var, pid, \Sigma), [\![expr]\!]_{pid,\Sigma}\rangle\}]$

   The effect of a synchronous-send statement is to store the following values in the mathematical structure *Handshake*: the pid of the sender process, the channel-id of the synchronous channel and the value of the expression being sent. This signifies the initiation of a synchronous (rendezvous hand-shake) communication.

8. $(\alpha.label \equiv var?const \wedge capacity(var, pid, \Sigma) = 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff$
$\wedge\ Handshake = \{\langle qid, [\![var]\!]_{pid,\Sigma}, const'\rangle\})$
$\Rightarrow$
$X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
$g' = g[Exclusive = \{\}, dstep = 0, Handshake = \{\}]$

   Since this transition is executable, it is necessarily the case that $qid \neq pid$ and $const' = const$. One of the effect of this transition is to set structure *Handshake* to $\{\}$, which signifies the completion of the synchronous communication.

9. $(\alpha.label \equiv var_1?var_2 \wedge capacity(var_1, pid, \Sigma) = 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
$\wedge\ Handshake = \{\langle qid, [\![var_1]\!]_{pid,\Sigma}, const\rangle\}$
$\Rightarrow$
$X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals', lChansOwned\rangle]$
$g' = g[Exclusive = \{\}, globals'/globals, dstep = 0, ChIdMap'/ChIdMap]$
where

   - $islnumvar(var_2) = tt$
   $\Rightarrow (locals' = locals[var_2 = const], globals' = globals, ChIdMap' = ChIdMap)$
   - $ischan(var_2) = tt$
   $\Rightarrow (locals' = locals, globals' = globals, ChIdMap' = ChIdMap[ChanId(var_2, pid, \Sigma) = const])$
   - $isgnumvar(var_2) = tt$
   $\Rightarrow (globals' = globals[var_2 = const], locals' = locals, ChIdMap' = ChIdMap)$

The effect of this statement is to assign the value in *Handshake* to the variable var_2 and to make the *Handshake* equal to {} signifying the termination of the synchronous communication.

10. $(\alpha.label \equiv run\ Name_1\ arg) \wedge WillFail(\alpha.label, pid, \Sigma) = ff) \wedge$
$\langle Name_1, Structure_1, Start_1, locals_1, ChansOwned_1, Active_1, Param_1 \rangle$
is a symbolic transition system $\Rightarrow$
$X" = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
$X' = X" \cup \langle Name_1, NrProcs + 1, Start_1, locals_2, ChansOwned_1\rangle$
$locals_2 = locals_1[Param_1 = [\![arg]\!]_{pid,\Sigma}]$
$g' = g[Exclusive = \{\}, NrProcs = NrProcs + 1, dstep = 0,$
$Incorp(ChIdMap, ChansOwned_1, NrProcs + 1)/ChIdMap,$
$IncorpCont(ChContMap, ChIdMap, lChansOwned_1, NrProcs + 1)/ChContMap]$

The effect of this statement is to create a new process with proctype *Name*. The smallest unassigned nonnegative integer is assigned as its process-id; this value equals the old value of *NrProcs*, from before the statement is executed. The formal parameter is instantiated with the value of the actual parameter. *ChIdMap* and *ChContMap* are updated accordingly.

11. $(\alpha.label \equiv end) \wedge WillFail(\alpha.label, pid, \Sigma) = ff) \wedge$
$\Rightarrow$
$X' = X \setminus \{\langle Name, pid, t, locals, lChansOwned\rangle\}$
$g' = g[RetractId(ChIdMap, lChansOwned, pid)/ChIdMap,$
$RetractCont(ChContMap, ChIdMap, lChansOwned, pid)/ChContMap,$
$dstep = 0, NrProcs = NrProcs - 1]$

4.2 Atomic and Deterministic Execution Mode

In general, when a process executes a primitive statement in *atomic* or *deterministic* mode it retains the exclusive right to execute also the next statement (it "retains control"). Otherwise, the effect of the execution of statements in these two modes is same as that in normal mode. To indicate this, in the following definition we set $Exclusive = \{pid\}$ besides the other conditions that are presented in the Definition 4.1. There is only one statement that when executed in *atomic* mode does *not* retain control: the synchronous send statement. Furthermore, it is an error to execute synchronous-send and synchronous-receive in *deterministic* mode.

The parts of the following definition that differ from the corresponding parts in Definition 4.1 are indicated in bold.

Definition 4.2 *If* $\langle pid, Name, s, \alpha, t\rangle$ *is* executable *in the state* $\Sigma = \langle X, g\rangle$ *with* $\alpha.mode \in \{atomic, deterministic\}$ *then* $Effect(\Sigma, \langle pid, Name, s, \alpha, t\rangle)$ *is* $\Sigma' = \langle X', g'\rangle$ *where*

1. $WillFail(\alpha.label, pid, \Sigma) = tt \Rightarrow \Sigma' = \Omega.$

2. $(\alpha.label \equiv expr) \wedge WillFail(\alpha.label, pid, \Sigma) = ff \wedge [\![expr]\!]_{pid,\Sigma} \neq 0$
$\Rightarrow$
$X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
$g' = g[\mathbf{Exclusive} = \{pid\}, \mathbf{dstep} = dstep']$
$\mathbf{dstep}' = 1$ *if* $\alpha.label = deterministic$ *and 0 otherwise* .

3. $(\alpha.label \equiv var = expr \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
$\Rightarrow$
$X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals', lChansOwned\rangle]$
$g' = g[\mathbf{Exclusive} = \{pid\}, globals'/globals, ChIdMap'/ChIdMap, \mathbf{dstep}'/\mathbf{dstep}]$
where $dstep' = 1$ *if* $\alpha.label = deterministic$ *and* 0 *otherwise.*
   - $islnumvar(var) = tt$
   $\Rightarrow (locals' = locals[var = [\![expr]\!]_{pid,\Sigma}], globals' = globals, ChIdMap' = ChIdMap)$

- $ischan(var) = tt$
  $\Rightarrow (locals' = locals, globals' = globals, ChIdMap' = ChIdMap[ChanId(var, pid, \Sigma = [\![expr]\!]_{pid,\Sigma}])$
- $isgnumvar(var) = tt$
  $\Rightarrow (globals' = globals[var = [\![expr]\!]_{pid,\Sigma}], locals' = locals, ChIdMap' = ChIdMap)$

4. $(\alpha.label \equiv var!expr \wedge capacity(var, pid, \Sigma) \neq 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
   $g' = g[\textbf{Exclusive} = \{pid\}, ChContMap'/ChContMap, dstep/dstep']$
   $dstep' = 1$ *if* $\alpha.label = deterministic$ *and* 0 *otherwise. where*

   $ChContMap' =$
   $ChContMap[ChIdMap(pid, var, \Sigma) = ChContMap(ChIdMap(pid, var, \Sigma)) \circ [\![expr]\!]_{pid,\Sigma}]$.

5. $(\alpha.label \equiv var?const \wedge capacity(var, pid, \Sigma) \neq 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
   $g' = g[\textbf{Exclusive} = \{pid\}, ChContMap'/ChContMap, \textbf{dstep}/\textbf{dstep}']$ *where*

   - $\textbf{dstep}' = 1$ *if* $\alpha.label = deterministic$ *and* 0 *otherwise.*
   - $ChContMap' =$
     $ChContMap[ChIdMap(pid, var, \Sigma) = cdr(ChContMap(ChIdMap(pid, var, \Sigma)))]$.

6. $(\alpha.label \equiv var_1?var_2 \wedge capacity(var_1, pid, \Sigma) \neq 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals', lChansOwned\rangle]$
   $g' = g[\textbf{Exclusive} = \{pid\}, globals'/globals, ChIdMap'/ChIdMap,$
   $ChContMap'/ChContMap, \textbf{dstep}/\textbf{dstep}']$ *where*

   - $\textbf{dstep}' = 1$ *if* $\alpha.label = deterministic$ *and* 0 *otherwise.*
   - $islnumvar(var_1) = tt$
     $\Rightarrow (locals' = locals[var_2 = first(var_1, pid, \Sigma)], globals' = globals, ChIdMap' = ChIdMap)$
   - $ischan(var) = tt$
     $\Rightarrow (locals' = locals, globals' = globals, ChIdMap' = ChIdMap[ChanId(var_2, pid, \Sigma) = first(var_1, pid, \Sigma)]$
   - $isgnumvar(var) = tt$
     $(globals' = globals[var_2 = first(var_1, pid, \Sigma)], locals' = locals, ChIdMap' = ChIdMap)$
   - $ChContMap' =$
     $ChContMap[ChIdMap(pid, var, \Sigma) = cdr(ChContMap(ChIdMap(pid, var, \Sigma)))]$

7. $(\alpha.label \equiv var!expr \wedge capacity(var, pid, \Sigma) = 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
   $g' = g[Exclusive = \{\}, Handshake = \{\langle pid, ChanId(var, pid, \Sigma), [\![expr]\!]_{pid,\Sigma}\rangle\}]$

   Note that the execution of a synchronous *send* statement in *atomic* mode causes the process to lose the right to execute the next statement (that right passes to the receiving process).

8. $(\alpha.label \equiv var?const \wedge capacity(var, pid, \Sigma) = 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
   $\wedge\ Handshake = \{\langle qid, [\![var_1]\!]_{pid,\Sigma}, const\rangle\}$
   $\Rightarrow$
   $X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
   $g' = g[\textbf{Exclusive} = \{pid\}, Handshake = \{\}]$

9. $(\alpha.label \equiv var_1?var_2 \wedge capacity(var_1, pid, \Sigma) = 0 \wedge WillFail(\alpha.label, pid, \Sigma) = ff)$
$\wedge$ $Handshake = \{\langle qid, [\![var_1]\!]_{pid,\Sigma}, const\rangle\}$
$\Rightarrow$
$X' = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals', lChansOwned\rangle]$
$g' = g[\mathbf{Exclusive} = \{pid\}, globals'/globals, ChIdMap'/ChIdMap]$
where

   - $islnumvar(var_2) = tt$
     $\Rightarrow (locals' = locals[var_2 = const], globals' = globals, ChIdMap' = ChIdMap)$
   - $ischan(var_2) = tt$
     $\Rightarrow (locals' = locals, globals' = globals, ChIdMap' = ChIdMap[ChanId(var_2, pid, \Sigma) = const]$
   - $isgnumvar(var_2) = tt$
     $\Rightarrow (globals' = globals[var_2 = const], locals' = locals, ChIdMap' = ChIdMap)$

10. $(\alpha.label \equiv run\ Name_1\ arg)) \wedge WillFail(\alpha.label, pid, \Sigma) = ff) \wedge$
$\langle Name_1, Structure_1, Start_1, locals_1, ChansOwned_1, Active_1, Param_1\rangle$
is a symbolic transition system $\Rightarrow$
$X" = X[\langle Name, pid, t, locals, lChansOwned\rangle / \langle Name, pid, s, locals, lChansOwned\rangle]$
$X' = X'' \cup \langle Name_1, NrProcs + 1, Start_1, locals_2, ChansOwned_1\rangle$
$locals_2 = locals_1[Param_1 = [\![arg]\!]_{pid,\Sigma}]$
$g' = g[\mathbf{Exclusive} = \{pid\}, NrProcs = NrProcs + 1,$
$Incorp(ChIdMap, ChansOwned_1, NrProcs + 1)/ChIdMap,$
$IncorpCont(ChContMap, ChIdMap, lChansOwned_1, NrProcs + 1)/ChContMap,$
$\mathbf{dstep}'/\mathbf{dstep}]$
where $\mathbf{dstep}' = 1$ *if* $\alpha.label = deterministic$ *and* 0 *otherwise.*

11. $(\alpha.label \equiv end) \wedge WillFail(\alpha.label, pid, \Sigma) = ff) \wedge$
$\Rightarrow$
$X' = X \setminus \{\langle Name, pid, t, locals, lChansOwned\rangle\}$
$g' = g[RetractId(ChIdMap, lChansOwned, pid)/ChIdMap,$
$RetractCont(ChContMap, ChIdMap, lChansOwned, pid)/ChContMap, NrProcs = NrProcs - 1]$

12. $((\alpha.label \equiv var!expr) \vee (\alpha.label \equiv var?convar) \wedge capacity(var, pid, \Sigma) = 0 \wedge \alpha.mode = deterministic$
$\Rightarrow$
$\Sigma' = \Omega$

    Any attempt to perform synchronous communication (rendezvous handshaking) in deterministic mode fails.

5 Executions

In this section, we define the executions of a PROMELA program. So far we have not taken into account the priorities of the transitions into considerations. We define an executable transition to be *selectable* if there exists no other executable statement with strictly higher priority. The following definition formalizes this statement.

Definition 5.1 *A* selectable *transition of a state* $\Sigma = \langle X, g\rangle$ *is an executable transition of* $\langle pid, Name, s, \alpha, t\rangle$ *such that if* $\langle pid, s, \beta, t'\rangle$ *is any executable transition of* Σ *then* $\alpha.priority > \beta.priority$.

Definition 5.2 *If a transition* t *is selectable in state* Σ *then* $\Sigma \rightarrow_1 Effect(\Sigma, t)$.

Selectable transitions almost correspond to actual execution steps of a program, but there are some steps that have not yet been modeled.

If, for instance, a process becomes blocked while it is executing an atomic sequence, then the process loses its exclusive control (Page 98 of [3]). However, if a process becomes blocked when it is executing a *dstep* sequence (i.e., when it executes in *deterministic* mode), then it fails. We define $\rightarrow_2$-transitions to augment $\rightarrow_1$-transitions with these additional steps.

Definition 5.3 *$\rightarrow_2$ is the smallest binary relation over Gstates satisfying the following conditions.*

1. $\Sigma \rightarrow_1 \Sigma' \Rightarrow \Sigma \rightarrow_2 \Sigma'$.
2. $(\langle X, g\rangle \neq \Omega \wedge \neg(\exists \langle X', g'\rangle : \langle X, g\rangle \rightarrow_1 \langle X', g'\rangle) \wedge \langle X, g\rangle[g.Exclusive = \{\}] \rightarrow_1 \Sigma'' \wedge g.dstep = 0) \Rightarrow \langle X, g\rangle \rightarrow_2 \Sigma''$.
3. $(\langle X, g\rangle \neq \Omega \wedge \neg(\exists \langle X', g'\rangle : \langle X, g\rangle \rightarrow_1 \langle X', g'\rangle) \wedge g.dstep = 1) \Rightarrow \langle X, g\rangle \rightarrow_2 \Omega$.

We have still not captured all the possible execution steps of a PROMELA program. The predefined variable *timeout* is set to be 1 when there are no executable statements in the system (Page 125 of [1]). The executions that result by the setting of *timeout* to 1, augment $\rightarrow_2$-transitions with $\rightarrow_3$-transitions, as defined below.

Definition 5.4 *$\rightarrow_3$ is the smallest binary relation over Gstates satisfying the following conditions.*

1. $\Sigma \rightarrow_2 \Sigma' \Rightarrow \Sigma \rightarrow_3 \Sigma'$.
2. $(\langle X, g\rangle \neq \Omega \wedge \neg(\exists \langle X', g'\rangle : \langle X, g\rangle \rightarrow_1 \langle X', g'\rangle) \wedge \langle X, g\rangle[g.timeout = 1] \rightarrow_2 \langle X'', g''\rangle) \Rightarrow \langle X, g\rangle \rightarrow_3 \langle X'', g''\rangle[g''.timeout = 0]$.

In these semantics we have modeled some conceptual single-step executions as multiple-step executions. For example, synchronous communication is modeled in two steps (a *send* followed by a *receive*). This resulted in the introduction of an "invisible" system state (the one in which the value is already sent but has not yet been received). Similarly, the intermediate states in a *dstep* sequence are invisible. The following definition formalizes this.

Definition 5.5 *A state $\Sigma = \langle X, g\rangle$ is* invisible *if g.Handshake is nonempty or g.Exclusive is nonempty. Any state that is not invisible is visible. The failure state Ω is always* visible. *Further $\Sigma = \langle X, g\rangle$ is* dinvisible *if $g.dstep = 1$.*

It is easy to show that every *dinvisible* state is also *invisible*. According to Page 98 of [3], a dstep sequence is completely deterministic and any nondeterminism encountered may be resolved in an predetermined, but undefined way. The following definition covers this.

Definition 5.6 *$\rightarrow_4$ is* a *largest subset of $\rightarrow_3$ such that for every Σ_d, a dinvisible state the following property holds:*

$$(\Sigma_d \rightarrow_4 \Sigma'_d \wedge \Sigma_d \rightarrow_4 \Sigma''_d) \Rightarrow \Sigma'_d = \Sigma''_d$$

Clearly $\rightarrow_4$ is not unique; this corresponds to the possibility of resolving the nondeterminism in more than one way, The only requirement we have on this relation is that it is the largest subset of $\rightarrow_3$ satisfying the above mentioned property.

We are now finally in a position to define the relation $\rightarrow$ which exactly defines the execution steps of a PROMELA program.

Definition 5.7 *$\rightarrow$ is is the smallest binary relation over visible states defined as follows. If Σ and Σ' are visible global states then*

1. $\Sigma \rightarrow_4 \Sigma' \Rightarrow \Sigma \rightarrow \Sigma'$.
2. *If $\Sigma_1, \cdots, \Sigma_k$ is a finite sequence of invisible states for some $k > 0$ such that $\Sigma \rightarrow \Sigma_1$ and $\Sigma_k \rightarrow \Sigma'$ and $\Sigma_i \rightarrow \Sigma_{i+1}$ for $1 \leq i \leq k$, then $\Sigma \rightarrow \Sigma'$.*
3. *$\Sigma \rightarrow \Sigma$ if there exists an infinite sequence of invisible states $\{\Sigma_i\}$ and $\Sigma \rightarrow \Sigma_0$ and $\Sigma_i \rightarrow \Sigma_{i+1}$ for all $i \in$ Naturals.*

6 Example

In this section, we illustrate by means of an example how a process declaration in textual form is represented as a symbolic labeled transition system.

Consider the following process type which is some arbitrary collection of PROMELA statements.

Example 6.1
```
active proctype foo(int x)
{
chan c1,c2;
chan c3 = [10] of {chan};
int x1 = 1;
int x2;
x1 = 2;
atomic { (x1 == 2) -> x1 = x1 + 1 }
do
::  (x1 > 5) -> x1 = x1 - 5
::  else -> x1 = x1 + 5
od;
skip;
c1?x1 unless c2?x2
}
```

The symbolic labeled transition system corresponding to this program is

$$\langle foo, Structure, q_0, locals, ChansOwned, 1, x\rangle$$

where *Structure* is given in Figure 1. *locals* is the partial function which takes values $0, 0, 1$ at $x, x2, x1$ respectively and undefined at all other arguments. *ChansOwned* is the finite sequence $\langle c1, \Omega, \Omega\rangle, \langle c2, \Omega, \Omega\rangle, \langle c3, 10, chan\rangle$.

The priority of the transition from state q_8 to q_{11} is higher than the one from q_8 to q_9. The *else* statement is translated into the negation of the conjunction of the other guards of the same *if* or *do* statement. It is an error to have *send* or *receive* statements as alternatives to an *else* in a *if* or *do* statement. Statements that transfer control out of an *atomic* into a non-atomic region are executed in *normal mode*.

7 Summary

This report gives an outline for an operational-semantics definition of the verification language PROMELA. We showed how a given PROMELA program, represented as sequence of symbolic labeled transition systems, can be translated into a Kripke structure. This was done by defining primitive statements as *conditional state transformers*. Not defined here is the conversion algorithm, part of SPIN, that translates PROMELA proctype definitions into symbolic labeled transition systems.

Features of the language that contribute to the complexity of the semantics include *mobility* (the ability to pass channel names over channels), and *atomic* and *deterministic* sequences. The first feature, for instance, necessitates the use of channel manipulation functions (Subsection 2.6.5) in the semantics of *run* and *end* statements.

For this outline we have made several simplifying assumptions about PROMELA syntax. For instance, we have not considered

- Correctness properties.
- Array variables.

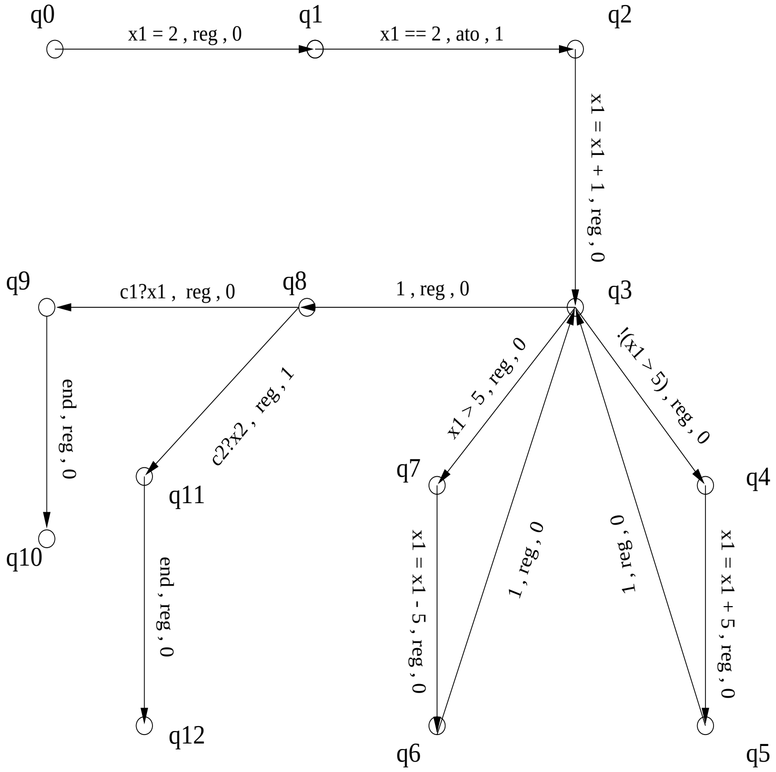

Figure 1: Sample Symbolic Labeled Transition System

- Data structures.
- Channels with more than one message field.
- Run statements with more than one argument.

The semantics of the *run* statement, furthermore, is somewhat different from that presented in Page 95 of [1]. In the semantics given here, the *run* statement cannot be used in composite arithmetic expressions. The semantics definitions can be modified fairly straightforwardly to overcome these restrictions.

References

[1] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[2] G.J. Holzmann. *What's New in* SPIN *Version 2.*, 1994. Online document.

[3] G.J. Holzmann. *The Verification of Concurrent Systems.* Course Notes, To appear.