Proceedings of "The 6th SEAMS-UGM Conference 2011"
Computer, Graph and Combinatorics, pp. 631–646.

A FRAMEWORK FOR AN LTS SEMANTICS FOR PROMELA

Suprapto And Reza Pulungan

Abstract. A high-level specification language PROMELA can be used not only to model interactions that occur in distributed or reactive systems, but also to express requirements of logical correctness about those interactions. Several approaches to a formal semantics for PROMELA have been presented, ranging from the less complete formal semantics to the more complete ones. This paper presents a significantly different approach to provide a formal semantics for PROMELA model, namely by an operational semantics given as a set of Structured Operational Semantics (SOS) rules. The operational semantics of a PROMELA statement with variables and channels is given by a program graph. The program graphs for the processes of a PROMELA model constitute a channel system. Finally, the transition system semantics for channel systems yields a transition system that formalizes the stepwise behavior of the PROMELA model.

Keywords and Phrases: PROMELA, formal semantics, SOS rules, program graphs, channel systems, transition systems.

1. PRELIMINARIES

It is still a challenging problem to build automated tools to verify systems especially reactive ones and to provide simpler formalisms to specify and analyze the system's behavior. Such specification languages should be simple and easy to understand, so that users do not require a steep learning curve in order to be able to use them [1]. Besides, they should be expressive enough to formalize the stepwise behavior of the processes and their interactions. Furthermore, they must be equipped with a formal semantics which renders the intuitive meaning of the language constructs in an unambiguous manner. The objective of this work is to assign to each model built in the specification language PROMELA a (labeled) transition system that can serve as a basis for further automated analysis, e.g., simulation or model checking against temporal logical specifications.

A PROMELA model consists of a finite number of processes to be executed concurrently. PROMELA supports communications over shared variables and message passing along either synchronous or buffered FIFO-channels. The formal semantics of

a PROMELA model can be provided by means of a channel system, which then can be unfolded into a transition system [1]. In PROMELA, the stepwise behavior of the processes is specified using a guarded command language with several features of classical imperative programming languages (variable assignments, conditional and repetitive commands, and sequential composition), communication actions where processes may send and receive messages from the channels, and atomic regions that avoid undesired interleaving [1].

Several researches in formal semantics of PROMELA have been carried out with various approaches [2, 3, 4]. By considering previous researches, the derivation approach of formal semantics of PROMELA presented here consists of three phases of transformation. First, a PROMELA model is transformed into the corresponding program graphs; then the generated program graphs will constitute a channel system. Finally, the transition system semantics for the resulting channel systems then produces a transition system of the PROMELA model that formalizes the operational behavior of that model. The rules of transitions presented here are operational semantics given as a set of Structured Operational Semantics (SOS) rules. SOS rules are a standard way to provide formal semantics for process algebra, and are useful for every kind of operational semantics.

The discussion of LTS semantics of PROMELA with this approach should be considered as an initial version that only covers some small parts of PROMELA features. Therefore, in order to handle more features, an advanced research will be required in the near future. One thing could have been noticed as a contribution in this research is the initial implementation of LTS to explain the behavior of PROMELA model.

1.1. **PROMELA.** PROMELA is a modeling language equipped with communication primitives that facilitate an abstraction of the analyzed systems, suppressing details that are not related to the characteristics being modeled. A model in PROMELA consists of a collection of processes that interact by means of message channels and shared variables. All processes in the model are global objects. In a PROMELA model, initially only one process is executed, while all other processes are executed after a run statement [6]. A process of type *init* must be declared explicitly in every PROMELA model and it can contain run statements of other processes. The init process is comparable to the function *main()* of a standard C program. Processes can also be created by adding *active* in front of the *proctype* declaration as shown in Figure 1.

Line (1) defines a process named *Bug*, a formal parameter x of type *byte* and the body of process is in between and . Line (2) defines the *init* process containing two *run* statements of process *Bug* with different actual parameters; the processes start executing after the *run* statement. Line (3) creates three processes named *Bar*, and immediately the processes are executed.

Communnications among processes are modeled by using message channels that are capable of describing data transfers from one process to another; they can be either buffered or rendezvous. For buffered communications, a channel is declared with the maximum length no less than one, for example, `chan buffname = [N] of byte`, where N is a positive constant that defines the size of the buffer. The policy of a channel communication in messages passing is FIFO (first-in-first-out). PROMELA also allows

```
(1) .....  proctype Bug(byte x) {
                ...
           }
(2) .....  init {
            int pid = run Bug(2);
            run Bug(27);
           }
(3) .....  active[3] proctype Bar() {
                ...
           }
```

FIGURE 1. A general example of PROMELA model

```
(1) ........ chan <name> = [<cap>] of {<t1>, <t2>, ..., <tn>};
(2) ........ chan ch = [1] of {bit};
(3) ........ chan toR = [2] of {int, bit};
(4) ........ chan line[2] = [1] of {mtype, Msg};
```

FIGURE 2. Examples of channel declaration

rendezvous communication, which is implemented as logical expressions of buffered communications, permitting the declaration of zero length channels, for example, `chan rendcomm = [0] of byte`. A channel with zero length means that the channel can pass, but cannot store messages. It implies that message interactions via such rendezvous channels are by definition synchronous. Rendezvous communication is binary, for there are only two processes; i.e., a sender and a receiver can be synchronized in a rendezvous handshake. The declaration syntax and declaration examples of channels are shown in Figure 2.

Line (1) declares a channel named name, with capacity *cap*, and $\langle t_1 \rangle, \ldots, \langle t_n \rangle$ denote the type of the elements that can be transmitted over the channel. Line (2) and (3) are obvious, and line (4) declares an array of channels of size two, where each has capacity of one.

Message channels are used to model the transfer of data from one process to another. Similar to the variables of the basic data types, they are declared either locally or globally. The statement *qname!expr* sends the value of expression *expr* via the channel (*qname*), that is, it appends the value of *expr* to the tail of the channel *qname*. The statement *qname?msg*, on the other hand, retrieves a message from the head of the channel *qname*, and assigns it to variable *msg*. In this case, there must be compatibility between the type of the value being stored and the type of variable *msg*. Instead of sending a single value through the channel, it is allowed to send multiple values per message. If the number of parameters to be sent per message exceeds the message channel can store, the redundant parameters will be lost. On the other hand, if the number of parameters to be sent less than the message channel can store, the values of the remaining parameters will be undefined. Similarly, if the receive operation tries to

```
proctype One(chan q1) { chan q2; q1?q2; q2!123 }

proctype Two(chan qforb) { int x;
                           qforb?x;
                           printf(x = %d\n, x) }

init{ chan qname[2] = [1] of { chan };
      chan qforb = [1] of { int };

run One(qname[0]);
    run Two(qforb);
    qname[0] ! qforb
}
```

FIGURE 3. An example of data communication using message channel

retrieve more parameters than are available, the values of the extra parameters will be undefined; on the other hand, if it retrieves fewer than the number of parameters that was sent, the extra values will be lost.

The send operation is executable only when the channel being used is not full. While the receive operation is executable only when the channel (for storing values) is not empty. Figure 3 shows an example that uses some of the mechanisms in data communication using message channel. The process of type One has two channels q_1 and q_2; they are as a parameter and a local channel respectively; while the process of type Two has only one channel *qford* as a parameter. Channel *qford* is not declared as an array and therefore it does not need an index in send operation at the end of the initial process. The value printed by the process of type Two will be 123.

The discussion so far is about asynchronous communications between processes via message channels created in statements such as `chan qname = [N] of { byte }`, where N is a positive constant that defines the buffer size. A channel size of zero, as in `chan port = [0] of { byte }`, defines a rendezvous port that can only pass, but not store, single-byte messages. Message interactions via such rendezvous ports are synchronous, by definition. Figure 4 gives an example to illustrate this situation. The two *run* statements are placed in an atomic sequence to enforce the two processes to start simultaneously. They do not need to terminate simultaneously, and neither complete running before the atomic sequence terminates. Channel name is a global rendezvous port. The two processes synchronously execute their first statement : a handshake on message *msgtype* and a transfer of the value 124 to local variable state. The second statement in process of type XX is not executable, since there is no matching receive operation in process of type YY.

PROMELA allows several general structures of control flow, namely atomic sequences, conditional (*if-statement*), repetition (*do-statement*), and unconditional jumps (*go to*) [7]. The if-statement has a positive number of choices (guards). If there are at least two choices executable, it is executable and the guard is chosen non-deterministically.

```
#define msgtype  33
chan name = [0] of { byte, byte };
byte name;

proctype XX()  {
    name!msgtype(124);
    name!msgtype(121)   }

proctype YY()  {
    byte state;
    name!msgtype(state)  }

    init { atomic { run XX(); run YY() } }
```

FIGURE 4. An example of data communication using message channel

```
if
  :: (n % 2 != 0) -> n = 1
  :: (n >= 0) -> n = n-2
  :: (n % 3 == 0) -> n = 3
  :: else -> skip
fi;
```

FIGURE 5. Example of the modified if-statement structure

Otherwise, it is blocked if there is no choice executable. The structure of if-statement may be modified by replacing choice with else guard as shown in Figure 5. When none of the guards is executable, the else guard become executable. In this example, statement *skip* will be executed when $n\%2 = 0$ *and* $n < 0$ *and* $n\%3 \neq 0$. Hence by adding the else guard, the *if-statement* will never block.

With respect to the choices, a *do-statement* has the same syntax as the *if-statement* does, and behaves in the same way as an *if-statement*. However, instead of ending the statement at the end of the chosen list of statements, a *do-statement* repeats the choice selection. Only one option can be selected for execution at a time, and after the option completes the execution of the structure is repeated. When there is no choice executable the control will be transfered to the end of the loop; the *break* statement which is always executable can be used to exit a *do-statement*.

Besides, in PROMELA there are also some interesting predefined statements, such as *timeout* and *assert*. The *timeout* statement is used to model a special condition that allows a process to abort the waiting for a condition that may never become true, for example an input from an empty channel. The *timeout* keyword is a modeling feature in PROMELA that provides an escape from a hang state. It becomes true only when no other statements within the distributed system is executable. Figure 6 shows

```
proctype keeper()
    {
        do
            :: timeout -> guard!reset
        od
    }
```

FIGURE 6. The example of timeout usage

```
         proctype monitor () {
(1) ............ assert (n <= 3);
                 }

         proctype receiver () {
                     ...
                 toReceiver ? msg;
(2) ............ assert (msg != ERROR);
                     ...
                 }
```

FIGURE 7. The examples of assert statement usage in process

the definition of the process that will send a reset message to a channel named guard whenever the system is blocked.

The *assert* statement, i.e., *assert(any_boolean_condition)* is always executable. If the specified boolean condition holds (true), the statement has no effect, otherwise, the statement will produce an error report during the verification process. The *assert* statement is often used within PROMELA models to check whether certain properties are valid in a state. Figure 7 shows that *assert* statement in line (1) will have no effect whenever the value of variable n is less than or equal to 3, otherwise it will produce an error report during the verification process; and similarly to *assert* statement in line (2).

Another interesting statement in PROMELA is the *unless* statement, with syntax $\{statement_1\}unless\{statement_2\}$. The mechanism of execution of *unless* statement might be explained as follows. The start point of the execution is in $statement_1$, but before each statement in $statement_1$ is executed, enabledness of $statement_2$ is checked. If $statement_2$ is enabled then $statement_1$ is aborted and $statement_2$ is executed, otherwise $statement_1$ is executed. Figure 8 illustrates the use of *unless* statement in a fragment of codes. The result of the statement execution depends on the value of c: if c is equal to 4 then x will be equal to 0. Since then statement $\{x! = 4; x = 1\}$ is not enabled, statement $\{x > 3; x = 0\}$ is executed. In case the value of c is 5 then x is equal to 1, since statement $\{x! = 4; x = 1\}$ is enabled. This means that statement $\{x > 3; x = 0\}$ is aborted and statement $\{x! = 4; x = 1\}$ is executed.

```
byte x = c;
{ x > 3; x = 0 }
      unless
{ x != 4; x = 1 }
```

FIGURE 8. The examples of assert statement usage in process

1.2. **Labeled Transition System.** Labeled Transition System or Transition System (TS) for short is a model utilized to describe the behavior of systems. TS is represented as a directed graph consisting of a set of nodes and a set of edges or arrows; nodes denote states and arrows model transitions (the changes of states). A states describes some information about the system at a certain moment of the system's behavior, whereas transition specifies how the system evolves from one state to another. In the case of a sequential program a transition system describes the sequence of the execution of statements and may involve the changes of the values of some variables and the program counter [1].

There have been many different types of transition systems proposed, however, action names and atomic propositions always denote the transitions (or state changes) and the states respectively in TS. Moreover, action names are used to describe the mechanisms of communication between processes, and the early letters of the Greek alphabet (such as α, β, γ, and so on) are used to denote actions. Besides, atomic propositions formalize temporal characteristics : they express intuitively simple known facts about each state of the system under consideration. The following is the formal definition of transition system [1].

Definition 1.1. ***A Transition System TS*** *is a tuple* $(S, Act, \rightarrow, I, AP, L)$ *where* S *is a set of states,* Act *is a set of actions,* $\rightarrow \subseteq S \times Act \times S$ *is a transition relation,* $I \subseteq S$ *is a set of initial states,* AP *is a set of atomic proposition, and* L *is a labeling function defined as* $L : S \rightarrow 2^{AP}$*. TS is called finite if* $S, Act,$ *and* AP *are all finite.*

1.3. **Program Graph.** A program graph (PG) over a set of typed variables is a digraph (directed graph) whose arrows are labeled with conditions on these variables and actions. The effect of the actions is formalized by means of a mapping : *Effect :* $Act \rightarrow Eval(Var) \times Eval(Var)$, which indicates how the evaluation η of variables is changed by performing an action. The formal definition of program graph is follows.

Definition 1.2. ***A Program Graph PG*** *over set* Var *of typed variables is a tuple* $(Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ *where* Loc *is a set of locations,* Act *is a set of actions,* $Effect : Act \rightarrow Eval(Var) \times Eval(Var)$ *is the effect function,* $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$ *is the conditional transition relation,* $Loc_0 \subseteq Loc$ *is a set of initial locations, and* $g_0 \in Cond(Var)$ *is the initial condition.*

$\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ is shorthand for $(\ell, g, \alpha, \ell') \in \hookrightarrow$. The condition g is called the guard of conditional transition $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$, therefore, if g is tautology then conditional transition would simply be written $\ell \overset{\alpha}{\hookrightarrow} \ell'$. The behavior in $\ell \in Loc$ depends on the current variable evaluation η. A nondeterministic choice is made between all transitions $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$ which

satisfy condition g in evaluation η (i.e., $\eta \models g$). The execution of α changes the variables evaluation according to $\mathit{Effect}(\alpha, .)$. The system changes into ℓ' subsequently, otherwise, the system stop.

Each program graph can be interpreted as a transition system. The underlying transition system of a program graph results from unfolding (or flattening). Its states consist of a control component, i.e., a location of the program graph, together with an evaluation η of the variables. States are thus pairs of the form $\langle \ell, \eta \rangle$. An initial state is initial location that satisfies the initial condition g_0. To formulate properties of the system described by a program graph, the set AP of propositions is comprised of location $\in Loc$, and Boolean conditions for the variables.

Definition 1.3. ***Transition System Semantics of Program Graph*** *The transition system* TS(PG) *of program graph* $PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ *over variables set* Var *is* $(S, Act, \rightarrow, I, AP, L)$ *where*

- $S = Loc \times Eval(Var)$
- $\rightarrow \subseteq S \times Act \times S$ *is defined by the rule*
  $\frac{\ell \overset{g:\alpha}{\hookrightarrow} \ell' \wedge \eta \models g}{\langle \ell, \eta \rangle \overset{\alpha}{\rightarrow} \langle \ell', Effect(\alpha, \eta) \rangle}$
- $I = \left\{ \langle \ell, \eta \rangle | \ell \in Loc_0, \eta \models g_0 \right\}$
- $AP = Loc \cup Cond(Var)$
- $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in Cond(Var) | \eta \models g\}$.

1.4. **Parallelism and Communications.** The mechanism to provide operational models for parallel systems by means of transition systems ranges from simple one where no communication between the participating transition systems takes place, to more advanced (and realistic) schemes where messages can be transfered, either synchronously (by means of handshaking) or asynchronously (by buffer with a positive capacity). Given the operational (stepwise) behavior of the processes that run in parallel with transition systems $TS_1, TS_2, \ldots, TS_n$ respectively, the purpose is to define an operator $\|$ such that :

$$TS = TS_1 \| TS_2 \| \ldots \| TS_n,$$

is a transition system that specifies the behavior of the parallel composition of transition system TS_1 through TS_n. The operator $\|$ is assumed to be commutative and associative, and of course the nature of $\|$ will depend on the kind of communication that is supported. TS_i may again be a transition system that is composed of several transition systems ($TS_i = TS_{i,1} \| TS_{i,2} \| \ldots \| TS_{i,n_i}$).

Definition 1.4. ***Interleaving of Transition Systems*** *Let* $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$ *i = 1, 2, be two transition systems.* ***The transition system*** $TS_1 ||| TS_2$ *is defined by :*

$$TS_1 ||| TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where the translation relation $\rightarrow$ *is defined by the following rules :*

$$\frac{s_1 \overset{\alpha}{\rightarrow}_1 s_1'}{\langle s_1, s_2 \rangle \overset{\alpha}{\rightarrow} \langle s_1', s_2 \rangle} \text{ and } \frac{s_2 \overset{\alpha}{\rightarrow}_2 s_2'}{\langle s_1, s_2 \rangle \overset{\alpha}{\rightarrow} \langle s_1, s_2' \rangle}$$

and the labeling function is defined by $L(\langle s_1, s_2 \rangle) = L(s_1) \cup L(s_2)$.

1.5. **Communication via Shared Variables.** The interleaving operator $|||$ can be used to model asynchronous concurrency in which the subprocess acts completely independent of each other, i.e., without any form of message passing or contentions on shared variables. The interleaving operator for transition systems is, however, too simplistic for most parallel systems with concurrent or communicating components.

In order to deal with parallel programs with shared variables, an interleaving operator will be defined on the level of program graphs (instead of directly on transition systems). The interleaving of program graphs PG_1 and PG_2 is denoted $PG_1|||PG_2$. The underlying transition system of the resulting program graph $PG_1|||PG_2$, i.e., $TS(PG_1|||PG_2)$ describes a parallel systems whose components communicate via shared variables. In general, $TS(PG_1|||PG_2) \neq TS(PG_1)|||TS(PG_2)$.

Definition 1.5. ***Interleaving of Program Graphs***
Let $PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})$, *for* $i = 1, 2$ *are two program graphs over the variables* Var_i. *Program graph* $PG_1|||PG_2$ *over* $Var_1 \cup Var_2$ *is defined by*

$$PG_1|||PG_2 = (Loc_1 \times Loc_2, Act_1 \uplus Act_2, Effect, \hookrightarrow, Loc_{0,1} \times Loc_{0,2}, g_{0,1} \wedge g_{0,2})$$

where $\hookrightarrow$ *is defined by the rules :*

$$\frac{\ell_1 \overset{g:\alpha}{\hookrightarrow_1} \ell_1'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1', \ell_2 \rangle} \text{ and } \frac{\ell_2 \overset{g:\alpha}{\hookrightarrow_2} \ell_2'}{\langle \ell_1, \ell_2 \rangle \overset{g:\alpha}{\hookrightarrow} \langle \ell_1, \ell_2' \rangle}$$

The program graphs PG_1 and PG_2 have the variables $Var_1 \cap Var_2$ in common. These are the shared (sometimes also called *global*) variables. The variables in $Var_1 \setminus Var_2$ are the local variables of PG_1, and similarly, those in $Var_2 \setminus Var_1$ are the local variables of PG_2.

1.6. **Handshaking.** The term *handshaking* means that concurrent processes that want to interact have to do this in a *synchronous* fashion. Hence, processes can interact only if they are both participating in this interaction at the same time - they *shake-hand* [1].

Definition 1.6. ***Handshaking (Synchronous Message Passing)*** *Let* $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i), i = 1, 2$ *be transition systems and* $H \subseteq Act_1 \cap Act_2$ *with* $\tau \notin H$. *The transition system* $TS_1 \parallel_H TS_2$ *is defined as follow :*
$TS_1 \parallel_H TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times T_2, AP_1 \cup AP_2, L)$ *where* $L(\langle s_1, s_2 \rangle) = L_1(s_1) \cup L_2(s_2)$, *and the transition relation* $\rightarrow$ *is defined by the rules :*

- *interleaving for* $\alpha \notin H$ *:*
  $$\frac{s_1 \overset{\alpha}{\rightarrow_1} s_1'}{\langle s_1, s_2 \rangle \overset{\alpha}{\rightarrow} \langle s_1', s_2 \rangle} \qquad \frac{s_2 \overset{\alpha}{\rightarrow_2} s_2'}{\langle s_1, s_2 \rangle \overset{\alpha}{\rightarrow} \langle s_1, s_2' \rangle}$$
- *handshaking for* $\alpha \in H$ *:*
  $$\frac{s_1 \overset{\alpha}{\rightarrow_1} s_1' \wedge s_2 \overset{\alpha}{\rightarrow_2} s_2'}{\langle s_1, s_2 \rangle \overset{\alpha}{\rightarrow} \langle s_1', s_2' \rangle}$$

Notation : $TS_1||TS_2$ abbreviates $TS_1||_H TS_2$ for $H = Act_1 \cap Act_2$

1.7. **Channel Systems.** Intuitively, a channel system consists of n (data-dependent) processes P_1 trough P_n. Each P_i is specified by a program graph PG_i which is extended with *communication actions*. Transitions of those program graphs are either the usual

conditional transitions (labeled with guards and actions), or one of the communication actions with their respective intuitive meaning :

$c!v$ transmit the value v along channel c,
$c?x$ receive a message via channel c and assign it to variable x.

Let $Comm = c!v, c?x | c \in Chan, v \in dom(c), x \in Var$ with $dom(x) \supseteq dom(c)$ denote the set of communication actions where $Chan$ is a finite set of channels with typical element c.

Definition 1.7. ***Channel System***
A program graph over $(Var, Chan)$ *is a tuple* $PG = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ *where*

$$\hookrightarrow \subseteq Loc \times (Cond(Var) \times (Act \cup Comm) \times Loc).$$

A channel system CS *over* $(Var, Chan)$ *consists of program graphs* PG_i *over* $(Var_i, Chan)$ *(for* $1 \leq i \leq n$*) with* $Var = \bigcup_{1 \leq i \leq n} Var_i$*. Channel system is denoted by*

$$CS = [PG_1 | PG_2 | \ldots | PG_n].$$

The transition relation $\hookrightarrow$ of a program graph over $(Var, Chan)$ consists of two types of conditional transitions. Conditional transitions $\ell \overset{g:\alpha}{\hookrightarrow} \ell$' are labeled with guards and actions. These conditional transitions can happen if the guard holds. Alternatively, conditional transitions may be labeled with communication actions. This yields conditional transitions of type $\ell \overset{g:c!v}{\hookrightarrow} \ell$' (for sending v along c) and $\ell \overset{g:c?x}{\hookrightarrow} \ell$' (for receiving a message along c).

Definition 1.8. ***Transition System Semantics of a Channel System*** *Let* $CS = [PG_1 | PG_2 | \ldots | PG_n]$ *be a channel system over* $(Chan, Var)$ *with* $PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{0,i}, g_{0,i})$*, for* $0 < i \leq n$*. The transition system of* CS*, denoted* $TS(CS)$*, is the tuple* $(S, Act, \rightarrow, I, AP, L)$ *where :*

- $S = (Loc_1 \times \ldots \times Loc_n) \times Eval(Var) \times Eval(Chan)$
- $Act = \biguplus_{0<i\leq n} Act_i \biguplus \{\tau\}$
- $\rightarrow$ *is defined by the following rules :*
  - *interleaving for* $\alpha \in Act_i$ *:*
    $$\frac{\ell_i \overset{g:\alpha}{\hookrightarrow} \ell_i' \wedge \eta \models g}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \overset{\alpha}{\rightarrow} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi \rangle}$$
    $\eta' = Effect(\alpha, \eta)$.
  - *asynchronous message passing for* $c \in Chan, cap(c) > 0$ *:*
    - *receive a value along channel* c *and assign it to variable* x *:*
      $$\frac{\ell_i \overset{g:c?x}{\hookrightarrow} \ell_i' \wedge \eta \models g \wedge len(\xi(c)) = k > 0 \wedge \xi(c) = v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \overset{\tau}{\rightarrow} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta', \xi' \rangle}$$
      where $\eta' = \eta[x := v_1]$ *and* $\xi' = \xi[c := v_2 \ldots v_k]$.
    - *transmit value* $v \in dom(c)$ *over channel* c *:*
      $$\frac{\ell_i \overset{g:c!v}{\hookrightarrow} \ell_i' \wedge \eta \models g \wedge len(\xi(c)) = k < cap(c) \wedge \xi(c) = v_1 \ldots v_k}{\langle \ell_1, \ldots, \ell_i, \ldots, \ell_n, \eta, \xi \rangle \overset{\tau}{\rightarrow} \langle \ell_1, \ldots, \ell_i', \ldots, \ell_n, \eta, \xi' \rangle}$$
      where $\xi' = \xi[c := v_1 v_2 \ldots v_k v]$.

– *synchronous message passing over* $c \in Chan, cap(c) = 0$:

$$\frac{\ell_i \overset{g_1 : c?x}{\hookrightarrow} \ell_i' \wedge \eta \models g_1 \wedge \eta \models g_2 \wedge \ell_j \overset{g_2 : c!v}{\hookrightarrow} \ell_j' \wedge i \neq j}{\langle \ell_1,\ldots,\ell_i,\ldots,\ell_j,\ldots,\ell_n,\eta,\xi\rangle \xrightarrow{\tau} \langle \ell_1,\ldots,\ell_i',\ldots,\ell_j',\ldots,\ell_n,\eta',\xi\rangle}$$

where $\eta' = \eta[x := v]$.

- $I = \left\{\langle \ell_1,\ldots,\ell_n,\eta,\xi_0\rangle | \forall 0 < i \leq n.(\ell_i \in Loc_{0,i} \wedge \eta \models g_{0,i})\right\}$
- $AP = \biguplus_{0<i\leq n} Loc_i \biguplus Cond(Var)$
- $L(\langle \ell_1,\ldots,\ell_n,\eta,\xi\rangle) = \{\langle \ell_1,\ldots,\ell_n\} \cup \{g \in Cond(Var) | \eta \models g\}$.

2. TRANSFORMATION

PROMELA is a descriptive language used to model especially concurrent systems. Elements of PROMELA model P mostly consist of a finite number of processes $P_1,\ldots,P_n$ to be executed concurrently. PROMELA supports communication over shared variables and message passing along either synchronous or asynchronous (buffered FIFO-channels). The formal semantic of a PROMELA programs can be provided by means of a channel system, which then can be unfolded into a transition system.

As already mentioned in the previous section, the discussion here will only cover small part of PROMELA features, which primarily concentrates on the basic elements of PROMELA. A basic (element) PROMELA model consists of statements that represent the operational behavior of the processes $P_1, P_2,\ldots,P_n$ together with a Boolean condition on the final values of the program variables. It is then represented as $P = [P_1|P_2|\ldots|P_n]$, where each process P_i is normally built by one or more statement(s). So that, the statements formalize the operational behavior of the process P_i. The main element of the statements are the atomic command (*skip*), variable assignment ($x := expr$), communication activities: reading a value for variable x from channel c ($c?x$) and sending the current value of expression expr over channel c ($c!expr$), conditional commands (*if..fi*), and repetitive commands (*do..od*). The syntax of basic PROMELA statements is shown in Figure 9.

```
stmnt ::= skip | x := expr | c?x | c!expr | stmnt1; stmnt2
        | atomic {assignments}
        | if :: g1 -> stmnt1 ... :: gm -> stmntm  fi
        | do :: g1 -> stmnt1 ... :: gm -> stmntm  od
```

FIGURE 9. Syntax of basic PROMELA-statements

Considering the fact that the PROMELA-statement itself is built by either variables, expressions or channels; before proceeding further discussion about statements it will be wiser to do a brief discussion of them. The variables in a basic PROMELA model P are used to store either global information about system as a whole or information that is local to one specific process P_i, depending on where the variable declaration takes place. They may be in (basic) type of (*bit, Boolean, byte, short, integer* and *channel*). Similarly, data domains for the channels must be specified: they must be also declared

to be synchronous or FIFO-channels of predefined capacity. Furthermore, variable can be formally defined as :

$$(name,\ scope,\ domain,\ inival,\ curval),$$

where *name* is variable name, *scope* is either global or local to a specific process, *domain* is a finite set of integers, *inival* is the initial value of variable, and *curval* is current value of variable. It is assumed that the *expression* used in assignments for variable x are built by constants in set of domain of x $dom(x)$, variable y of the same type as x (or a subtype of x), and operators on $dom(x)$, such as *Boolean connectives* $\wedge$, $\vee$, and $\neg$ for $dom(x) = \{0, 1\}$ and arithmetic operators +, -, $*$, etc. for $dom(x) = \Re$ (set of real numbers). The example of Boolean expressions are *guards* that determine conditions on the values of the variables, ($guards \in Cond(Var)$). In accordance to Figure 8, x is a variable in *Var*, *expr* is an expression, and *chan* is a channel of arbitrary capacity. Type compatibility of variable x and the expression *expr* in assignments $x := expr$ is highly required. Similarly, for the actions $c?x$ and $c!expr$ are required that $dom(c) \subseteq dom(x)$ and that the type of the expression *expr* corresponds to $dom(c)$. The $g_i s$ in both command *if..fi* and *do..od* are *guards*, and $g_i \in Cond(Var)$ by assumption. The body assignments of an atomic region is a nonempty sequential composition of assignments, and it has the form :

$$x_1 := expr_1; x_2 := expr_2; \ldots; x_m := expr_m$$

where $m \geq 1, x_1, \ldots, x_m$ are variables and $expr_1, \ldots, expr_m$ expressions such that the types of x_i and $expr_i$ are compatible. The intuitive meaning of the statements in Figure 8 can be explained as follows. *skip* represents a process that terminates in one step, and it does not affect the variables values neither channels contents. Variable x in assignment $x := expr$ is assigned the value of the expression *expr* given the current variable evaluation. $stmnt_1$; $stmnt_2$ denotes sequential composition, i.e., $stmnt_1$ is executed first and after the execution of $stmnt_1$ terminates, $stmnt_2$ is executed. In basic PROMELA, the concept of atomic region is realized by statement of the form $atomic\{stmnt\}$; the execution of *stmnt* is treated as an atomic step that cannot be interleaved with the activities of other processes. The statements of the form :

$$if :: g_1 \rightarrow stmnt_1 \ldots :: g_m \rightarrow stmnt_m \quad fi$$

stand for a non-deterministic choice between the statement $stmnt_i$ for which the guard g_i is satisfied in the current state, i.e., g_i holds for the current valuation of the variables. However, the *if..fi* command will blocks if all of guard g_is do not hold. This blocking must be seen by other processes that run in parallel that might end the blocking by altering the shared variable values so that one or more of the guards may finally become hold. Similarly, the statements of the form :

$$do :: g_1 \rightarrow stmnt_1 \ldots :: g_m \rightarrow stmnt_m \quad od$$

represent the iterative execution of the non-deterministic choice among the guarded commands $g_i \rightarrow stmnt_i$, where guard g_i holds in the current state. *do..od* loops do not block in a state when all guards are violated, instead the *do..od* loop is aborted.

2.1. **Semantics.** The operational semantics of basic PROMELA statement with variables and channel from *(Var, Chan)* is given by a program graph over *(Var, Chan)*. The

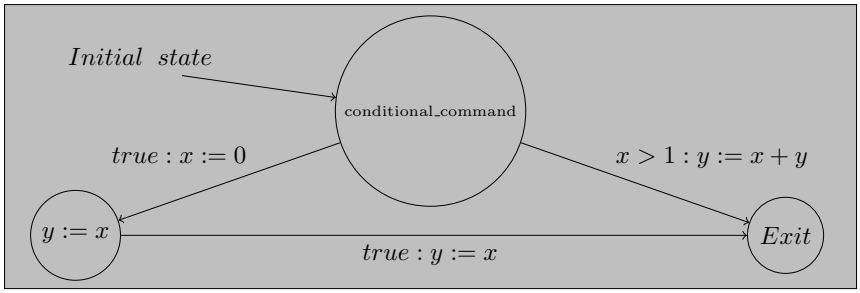

FIGURE 10. Program graph for conditional command *if..fi*

program graphs $PG_1, \ldots, PG_n$ for the processes $P_1, \ldots, P_n$ of a basic PROMELA model $P = [P_1| \ldots |P_n]$ constitute a channel system over *(Var, Chan)*. The transition system semantics for channel systems then results a transition system $TS(P)$ that formalizes the operational behavior of P [1].

The program graph associated with a basic PROMELA statement *stmnt* formalizes the control flow of *stmnt* execution. It means that the sub-statements play the role of the locations. For instance, in modeling termination, a special location exit is used. Thus in a program graph, any guarded command $g \rightarrow stmnt$ corresponds to an edge with the label $g : \alpha$ where α represents the first action of *stmnt*. For example, consider the statement :

```
conditional_command = if
                          :: x > 1 -> y := x + y
                          :: true  -> x := 0; y := x
                      fi
```

The program graph associated with this command may be explained as follows. *conditional_command* is viewed as an initial location of the program graph; from this location there are two outgoing edges : one with the guard $x > 1$ and action $y := x + y$ leading to (location) *exit*, and the other edge with the guard *true* and action $x := 0$ resulting in the location for the statement $y := x$. Since $y := x$ is deterministic there is a single edge with guard *true* and action $y := x$ leading to location *exit*. Figure 10 shows the program graph for conditional command *if..fi*.

2.2. **Substatement.** The set of sub-statements of a basic PROMELA statement *stmnt* is defined recursively. For statement $stmnt \in \{skip, x := expr, c?x, c!expr\}$ the set of sub-statements is $sub(stmnt) = \{stmnt, exit\}$. For example, $sub(x := expr) = \{x := expr, exit\}$, $sub(c?x) = \{c?x, exit\}$, etc. For sequential composition :
$sub(stmnt_1; stmnt_2) = stmnt; stmnt_2 | stmnt \in sub(stmnt_1)\{exit\} \cup sub(stmnt_2)$.
For example, $sub(x := expr; skip) = \{x := expr; skip, skip, exit\}$ where $\{stmnt; stmnt_2 | stmnt \in sub(stmnt_1) \backslash \{exit\}\}$ is $\{x := expr; skip\}$ and $sub(stmnt_2) = sub(skip) = \{skip, exit\}$.

For conditional commands, the set of sub-statement is defined as the set consisting of the *if..fi* statement itself and sub-statements of its guarded commands. Then, its sub-statements is defined as :

$sub(conditional\_command) = \{conditional\_command\} \cup sub(stmnt_1) \cup \ldots \cup sub(stmnt_n)$.

For example, $sub(if :: x > 1 \rightarrow y := x + y \quad :: true \rightarrow x := 0; y := x \quad fi) = \{if :: x > 1 \rightarrow y := x + y \quad :: true \rightarrow x := 0; y := x \quad fi, \quad y := x + y, \quad x := 0; y := x, \quad y := x, \quad exit\}$.

The sub-statements of loop command ($loop = do \quad :: g_1 \rightarrow stmnt_1 \ldots :: g_n \rightarrow stmnt_n \quad od$) is defined as : $sub(loop) = \{loop, exit\} \cup \{stmnt; loop | stmnt \cup sub(stmnt_1)\ \{exit\}\} \cup \ldots \cup \{stmnt; loop | stmnt \cup sub(stmnt_n) \backslash \{exit\}\}$. For example, $sub(do :: x > 1 \rightarrow y := x + y \quad :: y < x \rightarrow x := 0; y := x \quad od) = \{do :: x > 1 \rightarrow y := x + y \quad :: y < x \rightarrow x := 0; y := x \quad od, \quad y := x + y; \quad do :: x > 1 \rightarrow y := x + y \quad :: y < x \rightarrow x := 0; y := x \quad od, x := 0; y := x; \quad do :: x > 1 \rightarrow y := x + y \quad :: y < x \rightarrow x := 0; y := x \quad od, y := x; do :: x > 1 \rightarrow y := x + y :: y < x \rightarrow x := 0; y := x \quad od, exit\}$.

For atomic regions $atomic\{stmnt\}$, the sub-statement is defined as:

$$sub(atomic\{stmnt\}) = \{atomic\{stmnt\}, exit\}.$$

Then, for example the sub-statements of $atomic\{b_1 := true; x := 2\}$ is $sub(atomic\ \{b_1 := true; x := 2\}) = \{atomic\{b_1 := true; x := 2\}, exit\}$.

3. INFERENCE RULES

The inference rules for the atomic commands, such as *skip, assignment, communication actions,* and *sequential composition, conditional* and *repetitive commands* give rise to the edges of a large program graph in which the set of locations agrees with the set of basic PROMELA statements [1]. Thus, the edges have the form :

$$stmnt \xrightarrow{g:\alpha} stmnt' \text{ or } stmnt \xrightarrow{g:comm} stmnt'$$

where *stmnt* is a basic PROMELA statement, *stmnt* is sub-statement of *stmnt*, g is a guard, a is an action, and *comm* is a communication actions $c?x$ or $c!expr$. The subgraph consisting of the sub-statements of P_i then results in the program graph PG_i of process P_i as a component of the model P.

(1) The semantics of *skip* is given by a single axiom formalizing that the execution of *skip* terminates in one step without affecting the variables.

$$\frac{}{skip \xrightarrow{true:id} exit}$$

where *id* denotes an action that does not change the values of the variables, i.e., for all variable evaluations η, $Effect(id, \eta) = \eta$.

(2) Similarly, the execution of a statement consisting of an assignment $x := expr$ has trivial guard (true) and terminates in one step.

$$\frac{}{x := expr \xrightarrow{true:assign(x,expr)} exit}$$

where $assign(x, expr)$ denotes the action that changes the value of variable x according to the assignment $x := expr$ and does not affect the other variables, i.e., for all variable evaluation η ($\eta \in Eval(Var)$) and $y \in Var$ then $Effect(assign(x, expr), \eta)(y) = \eta(y)$. If $y \in x$ and $Effect(assign(x, expr), \eta)(x)$ is the value of $expr$ when evaluated over η.

(3) For the communication actions $c!expr$ and $c?x$ the following axiom apply :

$$\frac{cap(c) \neq 0}{c?x \xrightarrow{dom(c) \subseteq dom(x):c?x} exit} \text{ and } \frac{len(c) < cap(c)}{c!expr \xrightarrow{dom(Eval(expr)) \subseteq dom(c):c!expr} exit}$$

where $cap(c)$ is maximum capacity of channel c, $len(c)$ is current number of messages in channel c, $dom()$ is set of type, and $Eval(expr)$ is the value of expression $expr$ after evaluated.

(4) For an atomic region $atomic x_1 := expr_1; ...; x_m := expr_m$, their effect is defined as the cumulative effect of the assignments $x_i := expr_i$. It can be defined by the rule:

$$\frac{}{atomic x_1 := expr_1; ...; x_m := expr_m \xrightarrow{true:assign(x,expr)} exit}$$

where $\alpha_0 = id, \alpha_i = Effect(assign(x_i, expr_i), Effect(a_(i-1), \eta))$ for $1 \leq i \leq m$.

(5) There are two defined rules for sequential composition $stmnt_1; stmnt_2$ that distinguish whether $stmnt_1$ terminates in one step. If $stmnt_1$ does not terminate in one step, then the following rule applies :

$$\frac{stmnt_1 \xrightarrow{g:\alpha} stmnt_1' \neq exit}{stmnt_1; stmnt_2 \xrightarrow{g:\alpha} stmnt_1'; stmnt_2}$$

Otherwise, if $stmnt_1$ terminates in one step by executing action a, then the control of $stmnt_1; stmnt_2$ moves to $stmnt_2$ after executing a. The rule is :

$$\frac{stmnt_1 \xrightarrow{g:\alpha} exit}{stmnt_1; stmnt_2 \xrightarrow{g:\alpha} stmnt_2}$$

(6) The effect of a conditional command $conditional\_command = if :: g_1 \rightarrow stmnt_1 \ldots :: g_n \rightarrow stmnt_n \quad fi$ is formalized by the rule :

$$\frac{stmnt_i \xrightarrow{h:\alpha} stmnt_i'}{conditional\_command \xrightarrow{g_i \wedge h:\alpha} stmnt_i'}$$

(7) For repetition command $loop = do :: g_1 \rightarrow stmnt_1 \ldots :: g_n \rightarrow stmnt_n \quad od$ is defined three rules. The first two rules are similar to the rule for conditional command, but the control returns to loop after guarded command execution completes. This corresponds to the following rules :

$$\frac{stmnt_i \xrightarrow{h:\alpha} stmnt_i' \neq exit}{loop \xrightarrow{g_i \wedge h:\alpha} stmnt_i'; loop} \text{ dan } \frac{stmnt_i \xrightarrow{h:\alpha} stmnt_i'}{loop \xrightarrow{g_i \wedge h:\alpha} loop}$$

The third rule applies when none of the guards $g_1, g_2, \ldots, g_n$ holds in the current state :

$$\frac{}{loop \xrightarrow{\neg g_1 \wedge \ldots \wedge \neg g_n} exit}$$

So far, some basic statements of PROMELA (*skip, assignment, communication actions, atomic region, sequential composition, conditional command,* and *repetition command*) have been successfully defined their rules (axioms) for inferencing. However, for the sake of completeness it will be required a large effort to accomplish this work. Therefore, this should be becoming a further consideration.

4. CONCLUDING REMARKS

A new approach to a formal semantics for PROMELA has been presented. This approach first derives channel system from PROMELA model P which consists of a finite number of processes $P_1, P_2, \ldots, P_n$ to be executed concurrently ($P = [P_1|P_2|\ldots|P_n]$). The channel system $CS = [PG_1|PG_2|\ldots|PG_n]$, where PG_i corresponds to process P_i, so that the transition system of CS consists of transition system of PG_i. Each program graph P_i can be interpreted as a transition system, but the underlying transition system of a program graph results from unfolding (flattening). Then the transition system for channel systems yields a transition system $TS(P)$ that formalizes the stepwise (operational) behavior of PROMELA model P. This approach is modular that makes reasoning and understanding the semantics easier. It is more practical and fundamental than the one given in [8]. Therefore, this approach should be more suitable for reasoning about the implementation of a PROMELA interpreter.

References

[1] BAIER, C., AND KATOEN, J. P., *Principles of Model Checking*, The MIT Press, Cambridge, Massachusetts, 2008.

[2] BEVIER, W. R., Toward an Operational Semantics of PROMELA in ACL2, *Proceedings of the Third SPIN Workshop, SPIN97*, 1997.

[3] NATARAJAN, V. AND HOLZMANN, G. J., Outline for an Operational Semantic of PROMELA, *Technical report, Bell Laboratories*, 1993.

[4] RUYS, T., SPIN and Promela Model Checking, *University of Twente, Department of Computer Science, Formal Methods and Tools*, 1993.

[5] SHIN, H., Promela Semantics, *presentation from The SPIN Model Checker by G. J. Holzmann*, 2007.

[6] SPOLETINI, P., Verification of Temporal Logic Specification via Model Checking, *Politecnico Di Milano Dipartimento di Elettronicae Informazione*, 2005.

[7] VIELVOIJE, E., Promela to Java, Using an MDA approach, *TUDelft Software Engineering Research Group Department of Software Technology Faculty EEMCS, Delft University of Technology Delft, the Netherlands*, 2007.

[8] WEISE, C., *An Incremental formal semantics for PROMELA*, Prentice Hall Software Series, Englewood Cliffs, 1991.

SUPRAPTO
Universitas Gadjah Mada.
e-mail: sprapto@ugm.ac.id

REZA PULUNGAN
Universitas Gadjah Mada.
e-mail: pulungan@ugm.ac.id