TOWARDS AN EXTENSION OF PROMELA FOR THE MODELING, SIMULATION AND VERIFICATION OF DISCRETE-EVENT SYSTEMS

Aznam Yacoub, Maamar El Amine Hamri, Claudia Frydman, Chungman Seo, Zeigler Bernard P.

TOWARDS AN EXTENSION OF PROMELA FOR THE MODELING, SIMULATION AND VERIFICATION OF DISCRETE-EVENT SYSTEMS

Aznam YACOUB[(a)], Maamar HAMRI[(a)], Claudia FRYDMAN[(a)] , Chungman SEO[(b)] , Bernard P. ZEIGLER[(b)]

[(a)] Aix-Marseille Université, CNRS, ENSAM, Université de Toulon, LSIS UMR 7296,
13397, Marseille, FRANCE
[(b)] RTSync Corp. and Arizona Center for Integrative Modeling and Simulation, AZ

[(a)]aznam.yacoub@lsis.org, amine.hamri@lsis.org, claudia.frydman@lsis.org
[(b)]cseo@rtsync.com, zeigler@rtsync.com

ABSTRACT

PROMELA is a well-known formalism for the modeling and the verification of concurrent systems. PROMELA deals with high-level specifications. As a result, PROMELA models are expressed in a high-level abstraction which not considers explicit representation of time or events for example. But, the efficiency of the processes of Verification and Validation relies on the accuracy of the models. That is why we propose in this paper preliminary work to a new extension of PROMELA for the modeling of discrete-event systems. The verification of these models is then done by combining formal verification and simulation-based verification.

Keywords: DEv-PROMELA, Simulation, Formal Verification, Verification and Validation

1. INTRODUCTION

Process Meta Language (PROMELA) is a well-known formalism proposed by Holzmann (1991, 2004) for the modeling and the verification of concurrent systems by model-checking. Model-Checking (Clarke and Emerson 1982, Baier and Katoen 2008), and more generally formal verification techniques (Huth and Ryan 2005), represent promising methods of verification. Because they are potentially able to explore all the states of a model, these methods appear as comprehensive and efficient methods. A verification model is focusing on the conceptual aspects of a design that are relevant to the properties one wants to verify. The meaning of that is that a verification model must be as simpler as possible, and PROMELA is designed to encourage the user to make abstraction of the computational aspects of the system under study. This also guarantees the efficiency of the model-checking algorithms. But, by enforcing these restrictions, the expressiveness of PROMELA is reduced. For example, the interactions between the components of a system can strongly depends on the values of data or on their timely coordination. Making strong assumptions about that or a too high-level abstraction can lead to the development of a non-efficient model.

In the opposite, Discrete-Event system Specification (DEVS) formalism (Zeigler 1976) is well-suited for low-level modeling and analysis of real systems. By providing a clean operational semantics, the DEVS formalism allows a clean interpretation of the model elements in the real world. The few restrictions enforced by DEVS increase its expressiveness. The other side of the coin is that it becomes impossible to apply formal proofs on these models. The challenge to introduce the DEVS semantics into a formalism of formal verification like PROMELA then seems to provide some benefits. First, it allows accurate modeling of discrete-event systems (DES) by using a formalism which supports formal verification. In other words, the purpose of such an approach is increasing the expressiveness of PROMELA without breaking its formal verification capabilities. Second, specifying a DES model into a syntactic formalism like PROMELA can make easier the translation from the conceptual model to a computerized simulation model. Indeed, transformation rules can easily be defined and verified between two syntactic formalisms. Third, the verification and validation of the conceptual DES models can be done through two combined methods, formal verification and discrete-event simulation, without needing to use multiple formalisms.

The goal of this paper is to present preliminary work about an extension of PROMELA for modeling of DES. The section 2 is a quick overview of PROMELA. The section 3 presents our extension. Two example is given in section 4 to illustrate the verification process using the MS4Me environment (Seo et al. 2013) and the SPIN (Simple PROMELA Interpreter) model-checker (Holzmann 1991).

2. QUICK OVERVIEW OF PROMELA

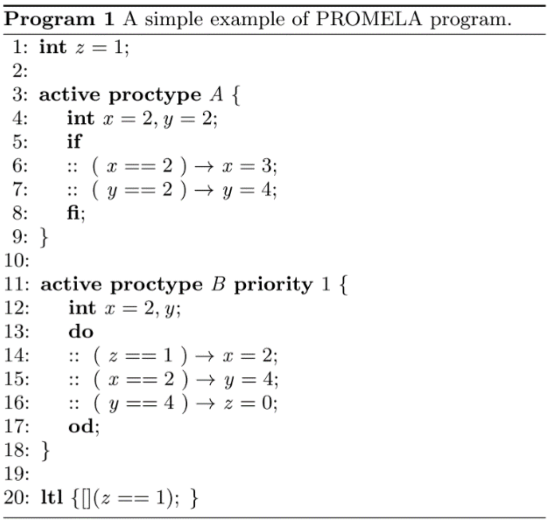
PROMELA is a specification language with a semantics of executability. Its syntax is influenced by the Dijkstra's Guarded Command Language (Dijkstra 1975) and the C programming language defined by Kernighan and Ritchie [1978]. This fact makes that its use is relatively easy compared with other formal methods because its syntax is really close to any implementation languages. We are focusing in this section only on concepts interesting to the scope of this paper. The section 2.1 is about how specifying components of a system in PROMELA. The section 2.2 expose the semantics of PROMELA. We don't deliberately talk about the specification of the properties that one wants to verify on the model in this paper, because we are focusing on how increasing the accuracy of conceptual models.

2.1. System Specifications

A PROMELA system relies on three main types of objects: processes, data objects and messages.

The components of the system are modeled by a finite set of instances of processes. The processes can communicate with each other thanks to different mechanisms: buffered messages, shared global variables or rendez-vous handshakes. Each process is a finite set of guarded commands called instructions. At any time t, only one instruction is executed without any assumptions about its duration. Note that a set of instructions can be labeled as atomic: in this case, these instructions are considered as a unique instruction, like any common atomic operations. Processes can be also prioritized, meaning that a process with a higher priority will always execute its instructions before other processes. Syntactically, a process is defined by a **proctype** block of instructions, as given in Program 1.

Program 1 A simple example of PROMELA program.



```
int z = 1;

active proctype A {
    int x = 2, y = 2;
    if
    :: ( x == 2 ) → x = 3;
    :: ( y == 2 ) → y = 4;
    fi;
}

active proctype B priority 1 {
    int x = 2, y;
    do
    :: ( z == 1 ) → x = 2;
    :: ( x == 2 ) → y = 4;
    :: ( y == 4 ) → z = 0;
    od;
}

ltl {[](z == 1); }
```

Instructions are divided into two categories: statements that modify the state of the system on the one hand, and control-flow instructions on the other hand. Assignments involve local and global variables, whereas communication statements involve global buffered channels. Control flow instructions are classical conditional and loop structures. They allow the selection of the next statement among different branches regarding a guard. Because PROMELA processes are non-deterministic, if several guards are satisfied, the next instruction is randomly selected. If none of them is satisfied, the control flow structure is blocked. PROMELA provides two special guards called **else** and **timeout**: if these guards are present in a control flow structure, and if this one is blocked, these instructions are then executed.

Data in PROMELA is represented by local and shared variables. Local variables are relative to the process only in which they are declared, whereas global variables are shared by all the processes. A variable is characterized by its value and its type, among either all the PROMELA scalar basic datatypes as given in Table 1, or any finite combinations (structures) or finite arrays of these types.

Table 1 : Basic PROMELA Datatypes

Type	Size (bit)	Range of Values
bit, bool	1	[0;1]
byte	8	[0;255]
mtype (constants)	8	[0;255]
short	16	$[-2^{16};2^{16}-1]$
int	32	$[-2^{32};2^{32}-1]$

2.2. PROMELA Semantics

PROMELA centers on a semantics of executability (Natarajan and Holzmann 1997, Holzmann 2004). We can study it at two levels: on the one hand, at the process level, and on the other hand, at the whole program level.

2.2.1. Executability of a Process

A PROMELA process with a set L of statements is a finite state machine P = (Q, T, q_0, F) where

- $Q = \{ q_i = (i, l_1, ..., l_m, g_1, ..., g_n, c_1, ..., c_o) \in \mathbb{N} \times \prod_{i=1}^{m} L_i \times \prod_{j=1}^{n} G_j \times \prod_{k=1}^{o} C_k \}$ is the finite set of states. Each state is defined by its id, the value of each local and global variable, and the value of each channel;
- $T \subset Q \times L \times Q$ is the set of labeled transitions;
- q_0 is the initial state;
- F is the set of final states.

Denote $(q_i,q_j) \in Q^2$ and $l \in L$ (l is an instruction in the process). Then, $t = (q_i,l,q_j) \in T$ iff the process can change its state from q_i to q_j by only executing l. This means that an instruction syntactically denotes two consecutive states. A transition can be executed only if it is enabled:

- l is a non-blocking instruction: an assignment, a conditional instruction with a satisfied guard, or any control-flow atomic instruction (**else**, **skip**, **break**, etc);
- l is an asynchronous message sent over a non-full channel;
- l is an asynchronous message received from a non-empty channel;
- l is an unblocking rendez-vous message.

If a transition is enabled and executed, the process changes its state from the source state to the target state by applying an appropriate action function. This action function modifies the values of the variables, the content of the channels, etc. If more of one instruction is enabled, the semantics engine randomly selects one of them and executes it.

2.2.2. Executability of a Program

A PROMELA program can be simply seen as the asynchronous product of the automata of each process. Then, a PROMELA program is defined by:

- $S \subseteq \prod_{i=1}^{n} Q_i$ where Q_i is the set of states of the process P_i;
- T is the set of transitions; t = $(s_i,l,s_j) \in$ T if it exists a transition that changes the state of any of the processes by applying l;
- s_0 is the initial state;

A transition t is enabled if its equivalent t_i is enabled in the process p_i. If two or more transitions are enabled, the semantics engine selects one of them at random and executes it (unless there is a defined priority between processes).

3. DISCRETE-EVENT PROMELA

The previous section shortly described the PROMELA semantics. As seen, PROMELA does not define either explicit time representation or events. But as stated by Tripakis (1996), the systems which can be modeled with PROMELA can be characterized as real-time systems. The representation of time becomes thus important to develop a more efficient design. Many timed extensions of PROMELA (Tripakis and Courcoubetis 1996; Bosnacki and Dams 1998; Nabialek et al. 2008) and the algorithms for their verification were studied in the literature. In opposite of these approaches which try to deal with the algorithms of formal verification and with the PROMELA semantics, we propose to integrate the DEVS semantics into PROMELA. This approach has two main advantages. On the one hand, it gives to PROMELA the capability of modeling DES with a lower level of abstraction. The model is more accurate and formal verification can be applied only on interesting paths. Formal verification can also be used to check structural properties on the model. Moreover, while formal verification deals with finite models, discrete-event simulation can be used for simulation-based verification of behavioral properties on infinite models, validation and analysis. On the other hand, modeling DES with a syntactic language close to an implementation language make easier the translation from the conceptual model to the computerized simulation model.

The sections 3.1 and 3.2 will recall concepts around discrete-event systems and the DEVS operational semantics. Then, the section 3.3 will introduce our new extension called Discrete-Event PROMELA (DEv-PROMELA).

3.1. Discrete-Event Concepts

DES are a specific class of timed systems (Zeigler 1976, Zeigler 1984). A DES can be seen as an extension of a Moore Machine, in the sense that the outputs of a DES can depend only on its current state. Zeigler (1976) introduced two important concepts: the association of a lifespan to each state and the hierarchical composition. A DES evolves along the events that it emits or consumes, the distribution of events can be non-linear and time can be represented (through the lifespan concept) by any value of the $\mathbb{R}^{+}$-space. A DES thus relies on the following notions:

1. Each state has a lifespan whose the value is a real. When the lifetime of state expired, the current state changes according to the transition table. An output is then emitted;
2. When an input is consumed by a DES, its current state changes according to the transition table, regardless the current lifetime of the current state;
3. Each event are well-dated. If e1 and e2 are two events, thus e1 and e2 can always be ordered;
4. If two events overcome at the same time, then either they are probably equivalent events (e1 = e2) or they are prioritized; the priority between two events is well-defined (non-deterministic behavior are not permitted);
5. The state trajectory is stable between two successful events;
6. As a result of the previous points, transitions are characterized by their nature: internal transitions correspond to autonomous behaviors while external transitions allow the system to react to any external events.

3.2. Overview of the DEVS Formalism

A DEVS system is built upon DEVS atomic models which can be coupled to get a DEVS coupled model. A DEVS atomic model is then a small simulation unit defined by $A = (X, Y, S, \delta_i, \delta_e, \lambda, ta)$ where

- X is the set of inputs;
- Y is the set of outputs;
- S is the set of sequential states;
- $\delta_e : Q \times X \rightarrow S$ is the external transition function;
- $\delta_i : S \rightarrow S$ is the internal transition function;
- $\lambda : S \rightarrow Y$ is the output function;
- $ta : S \rightarrow \mathbb{R}^{+}$is the time advance function;
- $Q = \{(s, e) \mid s \in S, 0 \leq \mathrm{dt} \leq \mathrm{ta(s)}\}$ is the set of total states.

Taking a DEVS atomic model A in a current state q. If the lifespan of q is ended (i.e. q = (s,ta(s))), the model changes its state to s'= δ_i (s). It emits the output y = λ(s). If A receives an event x, it change its state according to s'= δ_e(q,x). Otherwise, it stays in q, letting the time progress.

A DEVS coupled model is defined by $C = (X, Y, D, \{M\}, EIC, EOC, IC, Select)$ where

- X is the set of inputs;

- Y is the set of outputs;
- D is the name set of components;
- M is the set of atomic components;
- EOC is the set of external output coupling;
- EIC is the set of internal input coupling;
- IC is the set of internal coupling;
- Select is the tie-breaking function to choose the next event from the set of simultaneous events.

Informally, a DEVS coupled model is composed by atomic models. The date of the next event is computed by getting the minimum value among all the dates of all the next events. By processing an event, each component is updated according to its transition functions. If multiple events occur at the same date, the Select function chooses the next event to proceed.

3.3. Building Discrete-Event PROMELA

Introducing a new semantics in PROMELA needs to define new syntactic elements to model the concepts we are adding. The section 3.3.1 introduces these new syntactic elements. The section 3.3.2 and 3.3.3 define the meaning a DEv-PROMELA model.

3.3.1. Syntactic Rules

A new datatype called **real** is introduced. The purpose of real variables is to represent infinite and unbounded real values. Real variables can be local or global, and their declaration does not differ from any scalar variables:

real i,j,k;

Real variables can be combined in complex structures or in arrays, without any restriction.
An implicit local clock is associated to each process. Consequently, like in Timed PROMELA, introducing a specific type for clocks is not needed. Each process is ensured to have a clock. The clock valuation can be accessed through a specific **getCurrentDate** instruction. Clocks can be used to define exactly the date of next events. The purpose of clocks is only to determine the next event by taking the minimum value of the dates of all next events.
Missing concepts of events, state lifespan and type of transitions as viewed in the previous section are then introduced. Syntactically, a state is defined between two instructions in PROMELA. This means we can easily define the lifespan associated to each state and the type (internal or external) of the transitions associated to each instruction. We extend each statement with the following grammar described in the Backus-Naur Form:

```
<proctype decl> ::= "[" priority "=" <int> "]" <proctype>
<event stmnt> ::= "[" <timed trans> "]" <stmnt> | <stmnt>
<timed trans> ::= <clt expr> | <evt expr> |
                                    <clt expr> "|" <evt expr>
<clt expr> ::= "clt:" <real expr> "->emit:" <evt val>
<evt expr> ::= "evt:" <evt val> [ <op> <evt expr> ]
<op> ::= " |"
<evt val> ::= <mtype> | "silent"
<real expr> ::= <real> | "infinity" | /* Any C-function returning a real value */
```

Each statement is prefixed by an event descriptor (in brackets) which defines the lifespan of the source state of the considering transitions and how transitions are triggered. For instance, look at these instructions:

[clt:3.0->emit:newa] a = a + 3; (1)

[evt:newc] d = a + 9; (2)

[clt:lifespan(a)->emit:newa | evt:newb] b = a * 7; (3)

(1) defines a state with a lifetime equal to 3 units of time, and an internal transition will emit the event "newa". By executing this transition, the assignment is executed. (2) defines a passive state (with an infinite lifetime) and an external transition which will be enabled by receiving the event "newc". (3) defines a state with a lifetime equal to "lifespan(a)" and two transitions: an internal one which will emit the event "newa" and an external one which will be enabled by consuming the event "newb". This third example shows how event descriptors can define transitions either as internal (with the clt command), or as external (with the evt command), or as a combination of multiple external events with probably one internal transition. A PROMELA statement can now label several transitions. Note that the default lifespan of each state is equal to infinity if an evt descriptor is defined and if there is no *clt* command. The *clt* command is optional and needed only to denote internal transitions. For convenient purposes, if a statement is not prefixed by an event descriptor, that is interpreted like it is prefixed by *[clt:0 ->emit:silent]*. The statement is executed without any delay by emitting the silent event. Priority descriptors on processes are also added. Indeed, the concept of DEv-PROMELA priority allows the ordering the events coming (resp. generating) from (resp. by) each process. This is a different meaning than the PROMELA initial concept of priority.

3.3.2. Semantics of a DEv-PROMELA Process

A DEv-PROMELA process P with a set L of statements is an automaton $T = (S_\tau, E, \delta_i, \delta_e, s_0, F)$ where

- $S_\tau = \{ s_i = (t_s, i, l_1, ..., l_m, g_1, ..., g_n, c_1, ..., c_o) \in \mathbb{N} \times \prod_{i=1}^{m} L_i \times \prod_{j=1}^{n} G_j \times \prod_{k=1}^{o} C_k \}$ is the finite set of states;
- E is the finite set of events; E contains at least the silent event denoted ε;
- $\delta_i : Q_f \rightarrow Q_0 \times E$ is the partial function that defines transitions labeled by a statement l ∈ L; note that it is a partial function because no internal transition can be defined for passive state.
- $\delta_e : Q \times E \rightarrow Q$ is the partial function that defines external transitions labeled by a statement l ∈ L; note that it is a partial function

because for a state q, there is not necessarily an external transition defined for each event e;
- s_0 is the initial state;
- F is the set of final states.

We also define:

- $ta\colon \begin{cases} S_\tau \to \mathbb{R}^+ \\ ta(s) \mapsto t_s \end{cases}$ is the lifespan function;
- $Q = \{\, q = (s, dt), s \in S_\tau$ and $0 \le \mathrm{dt} \le \mathrm{ta(s)} \,\}$ is the set of total states;
- $Q_0 = \{\, q = (s, 0), s \in S_\tau \,\}$
- $Q_f = \{\, q = (s, ta(s)), s \in S_\tau \,\}$

Take s a state, and l a statement with an event descriptor. If l denotes an internal transition, l is enabled and δ_i(s) is triggered only if the value of the current clock of the process is equal to the lifetime ta(s). The event associated to the transition is emitted and the next event date for the process is computed by:

$$d_e = \text{getCurrentDate} + \text{ta(s')} \text{ if (s'; e)} = \delta_i\text{(s)}.$$

If l denotes an external transition on an event e, l is enabled if the process receives the event e. If the next event of the current process is at the same date than another external event, the priority statement resolves the conflict. If ta(s) = ∞, all of the statements related to s are blocked. Only external transitions can be enabled in this case. Because there is at most one internal transition per state and because there is a priority statement, a DEv-PROMELA process is deterministic.
What about the non-determinism expressed in the PROMELA control-flow structures ?

Program 2 Simple condition structure in Classic PROMELA.

```
1: int $x, y = 2$;
2:
3: if
4: :: ( $x == 2$ ) → $x = 3$;
5: :: ( $y == 2$ ) → $y = 4$;
6: fi;
```

Take a look at the Program 2. In the meaning of PROMELA, both guards are satisfied, meaning one of them is non-deterministically chosen. From another point of view, this means the model checker will verify the both paths, and this is the purpose of a good low-level verification model. But, if we take a look at the state space (Figure 3), we can clearly note that there is three cases. Two of them are unambiguous; concerning the intersection, the behavior is not well-defined. Transposed to an event system, this model would describe two situations:

1. Denotes e_1 and e_2 two different events that overcome at the same date. The verification model does not care about which event is consumed at first; the two orders will be checked;
2. The lifespan of the state (x=2, y=2) is ended, but two behaviors are possible. In this case, the verification model is not well-defined and effort are put into the verification of paths that are maybe meaningless for the real system.

We mean that the non-determinism involved by the PROMELA control-flow structures doesn't come necessarily from the non-deterministic nature of the considered program, or from a high-level abstraction, but can be involved by incomplete specifications.

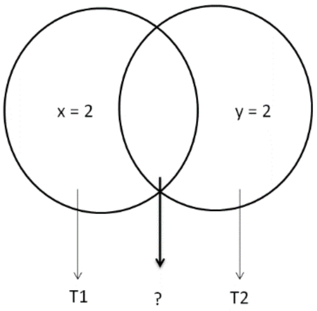


Figure 1 : State space of the Algorithm 2. T1 denotes the transition labelled by the statement line 4, and T2 the transition labelled by the statement line 5.

Enforcing determinism allows a unique, clear and unambiguous interpretation of the model in the real world, even non-determinism is really convenient for the verification of high-level models.
Taking into account these considerations, the Program 3 introduces a DEv-PROMELA version of the Program 2.

Program 3 Simple condition structure in DE-PROMELA.

```
1: int $x, y = 2$;
2: real $t1, t2$;
3:
4: if
5: :: [$clt : t1 \to emit : changex$] ( $x == 2$ ) → $x = 3$;
6: :: [$clt : t2 \to emit : changey$] ( $y == 2$ ) → $y = 4$;
7: fi;
```

This program can be interpreted in three manner:

1. if $t_1 \neq t_2$, there is not ambiguity: the state concerned by the branches are not the same. Indeed, if t_1 is lesser than t_2, T1 will be always triggered in first. If t_2 is lesser than t_1, T2 will be always triggered in first;
2. if t1 = t2, this model is non-deterministic. Determinism is then enforced by always executing the first enabled branch T1 in the case (x=2, y=2).

This property is applicable to any branching structures.

3.3.3. Semantics of a DEv-PROMELA Program

A DEv-PROMELA program is a product of the automata of each process that composed the program, as in PROMELA. The difference is that, while in PROMELA the sequences of the states are interleaved (i.e. the final graph is composed by all the possible permutations of the statements), DEv-PROMELA takes into account only the execution paths given by the order of the events. Denote e_1 and e_2, two events of the DEv-PROMELA system composed by two processes P1 and P2. If date(e_1) < date(e_2), then the statement associated to e1 will be executed in first. Both processes update their state according to the action function and computes their local next event e_1' and e_2'. The next event date is then given by min(e_1' ; e_2').

If date(e_1) = date(e_2), priority as defined in the previous section determines the next event between both events. Non-determinism is implicitly resolved for a precise instance: this does not mean that non-determinism doesn't exist in the global system. Because order of events can depend on the execution context, multiple branches could exist for a given global state. A verification algorithm would then have to check any possible orders of events. But in opposite of PROMELA, for a given order of events, one and only one path can be walked through.

It is also important to note that if the value of a global variable is changed, an implicit event is emitted. Other processes change their respective state. Their respective clock is however preserved, meaning their respective next event remains unchanged.

4. APPLICATIONS

We used our extension to model two examples of systems. First is the well-known Fischer's Mutual Exclusion Protocol. Second concerns the Alternate Bit Protocol with Finite Queue and Infinite Queue. Verification of properties were done through combined formal verification using the classic SPIN model-checker, and simulation-based verification using the MS4Me Environment. Because the DEv-PROMELA semantics is the same than the one of DEVS, the translation from DEv-PROMELA specifications to DNL format were easy.

4.1. Fischer's Mutual Exclusion Protocol

The Fischer's Mutual Exclusion Protocol (Abadi and Lamport 1994) looks like a very simple algorithm (Algorithm 4) for handling mutual exclusion: firstly, the process p checks whether another process either is already or wants to enter the critical section (line 4). If it is the case, the process stays in an active wait. Then, the process p declares his willing to enter the critical section before entering a sleep mode. When it wakes up, it checks whether another process is entered the critical section during its sleeping. If it is the case, the protocol restarts from the beginning, else the process can enter the critical section. Fisher's protocol can be seen as a timed system, and also a timed event system.

Using DEv-PROMELA, it is easy to obtain a verification and a simulation model (Program 6) of this algorithm. Because of the system can be considered as closed, almost of the transitions correspond to an autonomous behaviors which depend only on the current state. Priority ensures that each process will have the hand in turn. When a process is sleeping, it is blocked until the delay is exactly expired or until another process changes the value of the *id* variable. This is modeled in l.16: the model stays in this state during *delay* units of time or until it receives the *changepid* event. In other words, the next event for a process will be at getCurrentDate+*delay* after the execution of line 14.

Algorithm 4 Fischer's Mutual Exclusion Protocol (1984).

```
 1: while true do
 2: begin
 3:     /* non-critical section */
 4:     L: if $id \neq 0$ then goto L;
 5:     $id := i$;
 6:     pause($delay$);
 7:     if $id \neq i$ then goto L;
 8:     /* critical section */
 9:     $id := 0$;
10: end
```

If both processes wake up at the same time, priority will decide of the next event to proceed. Remember that the execution of this DEv-PROMELA program is deterministic.

The structural preservation property of DEv-PROMELA is out-of-scope of this paper. This property allows the generation of an untimed PROMELA verification model which has the same structural properties than the DEv-PROMELA model (Program 5). Thanks to that model, two properties can be easily verified:

1. "Two processes cannot be in critical section in same time." (□ ! ($c_i \wedge c_j$))"
2. "Is there cases in which two processes cannot be in critical section in same time ?" (◊ ! ($c_i \wedge c_j$))

While the property 2 is well-verified by the model, the SPIN model-checker returns a violation error as expected for the property 1. Indeed, a process P can execute the l.11 and l.14, then a process Q can execute the l.11 and l.14 and enters the critical section while P is always in it. This is well-known that the timed constraint *delay* > C (where C is the longest time that a process may take to perform a step while trying to enter the critical section) must be met to guarantee the mutual exclusion by this protocol. This only result could enforce the designers to design solutions to prevent such an error.

But if the system is well specified, for example if the duration taken to execute each instruction is quantified, a simulation-based verification can bring another interpretation. Remember that the DEv-PROMELA semantics is the same than the DEVS semantics, meaning a DEv-PROMELA model is a DEVS model. As a consequent, a DEv-PROMELA model is also a simulation model. We encode the Program 6 into a DNL format given in Appendix A by translation; then, we simulate it into the MS4 Me Environment (Figure 2). A simulation-based verification is then performed. In the context in which two processes exactly acts in the same manner and with the semantics of Classic DEVS, the analysis reveals that whatever the value of the delay, two processes can't be in the critical section at the same time. Property 1 is thus verified by this DEv-PROMELA model. How interpreting the difference between the results of formal verification and simulation ? On the one hand, the formal verification gives first information about the correctness of the model, on a high-level abstraction. For example, the formal verification ensures the absence of deadlocks. It can help designer to fix the design before considering other forms of testing. On the other hand, the simulation-based verification gives more precise information to the designer to understand what really happened. Because the DEv-PROMELA model is at lower level of abstraction than the PROMELA model, the designers can decide that the error found by the model checker is outside the scope of the model or not. Another advantage is that discrete-event simulation can be also used for validation at the same time. Designer can then get information about the validity of this model against its intended use.

4.2. Alternating Bit Protocol

We also use DEv-PROMELA to model the Alternating Bit Protocol (APB) (Bartlett et al. 1969), which has been used to illustrate the analysis capabilities of PROMELA (Holzmann 2004). The only things we have to do is to model the time T to transmit a bit, and the time T_1 up to that we can consider a message is lost. The obtained model does not really differ from the PROMELA one. But what happened if we want take into account a queue which stores messages before sending them ? As shown by Zeigler and Nutaro (2014), the capacity of the queue has an impact on the behavior of the system. Modeling the queue in PROMELA certainly increases the size of the state space, and reduces the efficiency of the formal verification algorithms. But DEv-PROMELA can be used in two manner: as a verification model, assumptions can be verified in idealized conditions (in this case, by abstracting the capacity of the queue or by considering there is no queue), and as a simulation model. Then, the DEv-PROMELA model of the APB can be coupled with any simulation model of queues. The simulation-based verification then extends the formal verification by exploring scenarios outside the scope of the formal verification.

Program 5 Fischer's Mutual Exclusion Protocol in Classic PROMELA.

```
int pid = 0;

active proctype P ( int id ) {
do ::
/* non-critical section */
wait_L:
    if
    :: pid != 0 → goto wait_L;
    :: else → skip;
    fi;
    pid = id;
    if
    :: pid != id → goto wait_L;
    :: else → skip;
    fi;
    /* critical section */
    pid = 0;
od;
}

```

Program 6 Fischer's Mutual Exclusion Protocol in DE-PROMELA.

```
int pid = 0;
mtype = { changepid }

[ priority = id ]
active proctype P ( int id ) {
real delay = 2.0 * id;
do ::
/* non-critical section */
wait_L:
    if
    :: [clt:0.1 → emit:silent] pid != 0 → goto wait_L;
    :: [clt:0.1 → emit:silent] else → skip;
    fi;
    [clt:0.1 → emit:changepid] pid = id;
    if
    :: [clt:delay → emit:silent | evt: changepid]
        pid != id → goto wait_L;
    :: [clt:delay → emit:silent] else → skip;
    fi;
    /* critical section */
    [clt:0.1 → emit:changepid] pid = 0;
od;
}

```

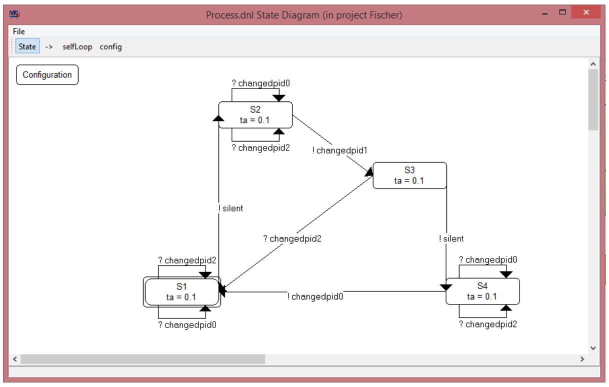


Figure 2 : State Diagram of the DEv-PROMELA model of Fischer's Algorithm in MS4 Me Environment.

5. CONCLUSION AND FUTURE WORK

We described in this paper some preliminary work to develop a new extension of PROMELA for the modeling and simulation of DES. This is done by introducing the DEVS operational semantics into PROMELA, without breaking the verification capabilities of PROMELA. A DEv-PROMELA model is then a verification model and a simulation model which can be verified by combining formal verification for structural properties and simulation-based verification for behavioral properties. The other advantage is that we are providing a syntactic formalism for DEVS modeling. This can help to implement computerized models from conceptual models. This maybe also enables the formal verification of the simulation models by using rewriting rules (for example from DEv-PROMELA to DNL).

The major drawbacks of our methodology is that DEv-PROMELA is limited to the Classic DEVS and its subclasses. Parallel DEVS cannot be modelled with this approach due to the limitations of PROMELA. But it can be interested to study how introducing the semantics of Parallel DEVS into PROMELA. Future work also concerns providing a formal proof that a DEv-PROMELA model is well a DEVS model, and working on limiting the formal verification algorithms only to the paths which are really expressed by the DEv-PROMELA models, without generating all the state space of the PROMELA equivalent.

ACKNOWLEDGMENTS

This work is a part of the R&D project “MAGE”, from French “Investing for the Future” national program.

APPENDIX A: DEV-PROMELA MODEL OF FISCHER’S ALGORITHM ENCODING IN THE DNL FORMAT

```
use delay with type double and default “0.1”!
use id with type int and default "1"!
use pid with type int and default "0"!

generates output on silent!
generates output on changedpid1!
generates output on changedpid0!
accepts input on changedpid0!
accepts input on changedpid2!

to start hold in S1 for time 0.1!
after S1 output silent!
from S1 go to S2!

when in S1 and receive changedpid0 go to S1!
when in S1 and receive changedpid2 go to S1!
external event for S1 with changedpid0
<%pid = 0;%>!
external event for S1 with changedpid2
<%pid = 2;%>!

internal event for S1
<%if(pid==0) holdIn("S2",0.1);
else holdIn("S1",0.1);%>!
hold in S2 for time 0.1!
after S2 output changedpid1!
from S2 go to S3!

when in S2 and receive changedpid0 go to S2!
when in S2 and receive changedpid2 go to S2!
external event for S2 with changedpid0
<%pid = 0;%>!
external event for S2 with changedpid2
<%pid = 2;%>!

internal event for S2
<%pid = id;%>!
hold in S3 for time delay!
when in S3 and receive changedpid0 go to S1!
when in S3 and receive changedpid2 go to S1!

external event for S3 with changedpid0
<%pid = 0; holdIn("S1",0.1);%>!
external event for S3 with changedpid2
<%pid = 2; holdIn("S1",0.1);%>!
after S3 output silent!
from S3 go to S4!

internal event for S3
<%if(pid==id) holdIn("S4",0.1);
else holdIn("S1",0.1);%>!
hold in S4 for time 0.1!
after S4 output changedpid0!
from S4 go to S1!

when in S4 and receive changedpid0 go to S4!
when in S4 and receive changedpid2 go to S4!
external event for S4 with changedpid0
<%pid = 0;%>!
external event for S4 with changedpid2
<%pid = 2;%>!

internal event for S4
<%pid = 0;%>!
```

REFERENCES

Abadi M. and Lamport L., 1994. An old-fashioned Recipe for Real-Time. ACM Transactions on Programming Languages and Systems. Vol. 16, pp1543-1571.

Baier C. and Katoen J.P., 2008. Principles of Model Checking. MIT Press.

Bartlett K.A., Scantlebury R.A., and Wilkinson P.T. 1969. A note on reliable full-duplex transmission over half-duplex lines, Comm. of the ACM, Vol.12, No. 5, pp260-265.

Bosnacki D. and Dams D., 1998. Discrete-Time PROMELA and SPIN. Formal Techniques in Real-Time and Fault-Tolerant Systems, Vol.1486, 307-310. Berlin, Springer Berlin Heidelberg.

Clarke E.M. and Emerson E.A., 1982. Design and Synthesis of Synchronization Skeletons Using

Branching-Time Temporal Logic. Logic of Programs, Workshop, pp52-71. London, UK.

Dijkstra E.W., 1975. Guarded commands, nondeterminacy and formal derivation of programs. Comm. of the ACM. Aug. 1975. Vol. 18, No. 8, pp. 453-457.

Holzmann G.J., 1991. Design and Validation of Computer Protocols. Englewood Cliffs, NJ, Prentice Hall.

Holzmann G.J., 2004. The SPIN Model Checker: Primer and Reference Manual. Readings Massachusetts, Addison-Wesley.

Huth M. and Ryan M., 2005. Logic in Computer Sciences: Modelling and Reasonning about Systems. Cambridge University Press.

Kernighan B.W. and Ritchie D.M., 1978. The C Programming Language. First Edition. Englewood Cliffs, NJ, Prentice Hall.

Nabialek W., Janowska A. and Janowski P., 2008. Translation of Timed PROMELA to Timed Automata with Discrete Data. Journal Fundamata Informaticae, Vol.85, pp409-424. Amsterdam, IOS Press.

Natarajan V. and Holzmann G.J.., 1997. Outline for an Operational Semantics of PROMELA. The SPIN Verification System. DIMACS Series in Discrete Mathematics and Theorical Computer Sciences. AMS, 1997. Vol. 32, pp.133-152.

Seo C., Zeigler B.P, Coop R. and Kim D., 2013. DEVS Modeling and Simulation Methodology with MS4 Me Software. Theory of Modeling and Simulation Symposium, SpringSim MultiConference. San Diego, CA.

Tripakis S. and Courcoubetis C., 1996. Extending PROMELA and SPIN for Real-Time. Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAs, pp329-348. 1996, London, UK.

Zeigler B.P., 1976. Theory of Modeling and Simulation. John Wiley.

Zeigler B.P., 1984. Multifacetted Modelling and Discrete-Event Simulation. San Diego, CA, USA. Academic Press Professional Inc.

Zeigler B.P. and Nutaro J., 2014. Combining DEVS and Model-Checking: Using System Morphisms for Integrating Simulation and Analysis in Model Engineering. Proceedings of the 26th European Modeling and Simulation Symposium, pp350-356. 2014.