DEv-PROMELA: an extension of PROMELA for the modelling, simulation and verification of discrete-event systems

Aznam Yacoub*

Aix-Marseille Université,
CNRS, ENSAM,
Université de Toulon,
LSIS UMR 7296,
13397, Marseille, France
Email: aznam.yacoub@lsis.org
*Corresponding author

Maamar el Amine Hamri and Claudia Frydman

Aix-Marseille Université,
CNRS, ENSAM,
Université de Toulon,
LSIS UMR 7296,
13397, Marseille, France
Email: amine.hamri@lsis.org
Email: claudia.frydman@lsis.org

Chungman Seo and Bernard P. Zeigler

RTSync Corp. and Arizona Center for Integrative Modeling and Simulation,
University of Arizona,
Tucson, AZ, USA
Email: cseo@rtsync.com
Email: zeigler@rtsync.com

Abstract: PROMELA is a well-known formalism for the modelling 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 does not consider 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, work to develop a new extension of PROMELA for the modelling of discrete-event systems. The verification of these models is then done by combining formal verification and simulation-based verification using SPIN and the tool DEv-PROMELA Studio, or using any existing DEVS simulators.

Keywords: DEv-PROMELA; simulation; formal verification; verification and validation.

Reference to this paper should be made as follows: Yacoub, A., Hamri, M.A., Claudia, F., Seo, C. and Zeigler, B.P. (2017) 'DEv-PROMELA: an extension of PROMELA for the modelling, simulation and verification of discrete-event systems', *Int. J. Simulation and Process Modelling*, Vol. 12, Nos. 3/4, pp.313–327.

Biographical notes: Aznam Yacoub received his PhD in Computer Science from the Aix-Marseille University. His main research interests include formal verification and discrete-event simulation for the verification and validation of software.

Maamar el Amine Hamri received his PhD in Computer Science from the Aix-Marseille University. He mainly works on DEVS and its extensions G-DEVS and Min-Max DEVS.

Claudia Frydman is a Professor of Computer Science at the Aix-Marseille University. Her main research interests include discrete-event simulation.

Chungman Seo received his PhD from the Department of Electrical and Computer Engineering of the University of Arizona. He is currently a research engineer at RTSync Corp. and developer of the MS4 environment.

Copyright © 2017 Inderscience Enterprises Ltd.

Bernard P. Zeigler is an Emeritus Professor of Electrical and Computer Engineering at the University of Arizona. He is internationally known for his seminal contributions in modelling and simulation theory and has published several books including theory of modelling and simulation. He was named fellow of the IEEE for the discrete event system specification (DEVS) formalism that he invented in 1976. He is currently the Chief Scientist with RTSync Corp., a developer of the MS4 modelling and simulation software based on DEVS.

This paper is a revised and expanded version of a paper entitled 'Towards an extension of promela for the modeling, simulation and verification of discrete-event systems', presented at European Modeling and Simulation Symposium, Bergeggi, 21–23 September 2015.

1 Introduction

Verification and validation (V&V) is becoming a critical point in the understanding and design of large systems like weather events, traffic lights, aircraft autopilot, smart systems, etc. Developing software or hardware without failure and in which people can put their trust is a big challenge for experts and designers. Many research fields have explored solutions to this problem for many years, and two domains have emerged. On the one hand, theory of modelling and simulation (M&S) (Zeigler, 1976) provides an intuitive way to model systems by specifying inputs, outputs, states and a time-advance function expressing the evolution of the model through time, then encapsulating this simple behaviour into black boxes and reusing them to design more complex systems. Even the harder systems can thus be quickly analysed, but the efficiency of simulations strongly depends on the hypothesis made on the system under study and on the played scenarios. Indeed, testing the entire state space appears like impossible, owing to the impossibility for designers to think to all possible cases; or it will take a too long time as simulation can deal with big inputs.

On the other hand, formal methods (FM), especially model-checking (Clarke and Emerson, 1982; Queille and Sifakis, 1982], are well-known to systematically explore the total state space of a model in order to check that it satisfies behavioural properties. However, formal methods can encouter some limits: for example, timed automata (TA) are not able to represent systems with non-linear clock constraints (Alur and Dill, 1994). The main problem is that the formal verification models need to focus on the conceptual aspects of a design that are relevant to the properties one wants to verify for two reasons: this guarantees the efficiency of the model-checking algorithms within a reasonable time when the state space may be huge, and it is shown that many problems are undecidable, resulting that the model-checking cannot be performed in these cases.

The literature thus provides some techniques on combining formal verification and simulation for specific fields like validation of integrated circuit design (Li et al., 2006) or system-level performance analysis (Kunzli et al., 2006), and shows the benefit of combining simulation and formal verification. These approaches propose to enhance V&V process by partitioning the model and using on each part either formal verification or simulation if the first is not applicable. However, there is no guarantee that each part is expressed in the right level of abstraction. These methods generally imply remodelling part of systems which one wants to check, or provide some morphism rules between multiple formalisms to get a verification model from a simulation model. In this context, we propose another methodology to combine simulation and model checking. In our approach, we build a new specification language (called the *target formalism*) upon a verifiable language (called the *source formalism*) by integrating semantics which comes from simulation. This increases the expressiveness of the source formalism without breaking its formal verification capabilities. Then, simulation is used in order to verify and validate some properties that formal verification could not verify and without remodelling the considered part of the system under study. Thereby, each part of a system can be verified and validated at many levels of abstraction. Moreover, thanks to the power of the discrete-event system specification (DEVS) framework, each submodel can be combined into another complex model which can represent the entire system at a level of abstraction. The behaviour of the global model is thus verified and validated by simulation, overcoming the weakness of formal methods applied to complex systems.

One result of our proposed approach is the discrete-event PROMELA (DEv-PROMELA) which we introduced in DEv-PROMELA is based on the well-known process meta language (Holzmann, 1991) and on the Classic DEVS semantics (Zeigler, 1976). It allows accurate modelling of discrete-event systems (DES) in a syntactical way and their verification and validation by using both formal verification and simulation. Another advantage of DEv-PROMELA is that specifying a DES model into a syntactic formalism makes easier the translation from the conceptual model to a computerised simulation model. Indeed, transformation rules can easily be defined and verified between two syntactic formalisms. This advantage is crucial because it also allows a kind of *interoperability* between existing simulators. By using the DEv-PROMELA Studio which integrates the SPIN model checker (Holzmann, 1997) and a DEVS simulator on the one hand, and the MS4 Me environment (Seo et al., 2013) to illustrate the interoperability on the other hand, DEv-PROMELA has been successfully applied in order to model, verify and simulate designs of simple algorithms, but also more complex conceptual models of video games.

This article is built as follow: Section 2 briefly introduces formal verification and simulation especially of timed systems, and talks about existing combining methods. Then, Sections 3.1 and 3.4 shortly introduce our approach combining formal verification and simulation in a same framework, and briefly recall the semantics of DEv-PROMELA, a result of our proposed approach. Section 4 finally talks about few applications of DEv-PROMELA.

2 Related works

2.1 Introduction to formal verification

Formal verification is a well-known verification methodology that "dates back to the origin of computer sciences" (Cousot and Cousot, 2010), and whose the objective is to check whether a model is correct against some formal specifications. Formal verification methods are based on rigorous and mathematical proof techniques. Indeed, formal methods are a set of formal notation and tools that allow a strict and rigourous description of the system under study, with formal semantics and an automatic proof mechanism (Bowen and Hinchey, 1995). Formal methods are divided into two families:

- Automated theorem proving methods (Chang and Lee, 1973; Loveland, 1978; Duffy, 1991) show that a set of statements of a system can be deducted from another set of statements. Formally, we consider Γ, a set of logical properties describing the system (we called them axioms and hypothesis), and ϕ a set of specifications (that we called conjectures). Theorem proving methods try to find a proof that $\Gamma \vdash \phi$, in other words, that we can syntactically deduce specifications from properties of the system.
- Model checking methods (Clarke and Emerson, 1982) show that a system satisfies a set of properties. Formally, we consider M, a model (in the mathematical sense) of the system, and ϕ, a set of logical properties. Model checking methods check whether $M \vDash \phi$: all models M syntactically and semantically satisfy ϕ. In fact, because the system is generally modelled by a finite automaton, model checking tools systematically explore the entire state space of the system model, inducting to the well-known state space explosion problem, which is extensively treated in the literature (Clarke, 2008; Huth and Ryan, 2005; Baier and Katoen, 2008).

With these definitions, we can easily understand why formal methods are powerful methods to check the correctness of a system. Indeed, whatever the family of formal methods, they are based on logics that is the best way to describe properties in an unambiguous manner, and on mathematical formalisms, most of the time state machines, that are also the best way to describe systems in an unambiguous manner. However, it is well-known that these techniques could become very heavy, time and effort consuming because they require advanced mathematical skills and knowledge (Heitmeyer, 1998), and are not very practicable in large and complex systems. Although model checking research is focusing on efficiency and scalability, formal methods are faced with the complexity of systems and data domains (Zervoudakis et al., 2013).

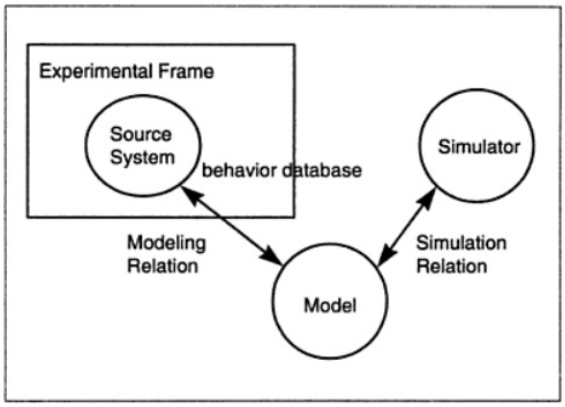
The complexity of systems comes from the complexity of their behaviour and interactions between their internal components, on the one hand, and their environment, on the other hand. Consequently, in order to be efficient and effective, formal methods have to apply some restrictions to their modelling and specifications languages, reducing their expressive capabilities. Consequently, some assumptions can lead to a certain idealisation or rigorous hypothesis which are not representative of the real environment of the system. The obtained model is certainly simpler than the real design, but it also increases the requirement of some expertise to ensure the level of abstraction is appropriate. The problem does not arise from the abstraction process itself, because making models is making abstraction, but the way in which we do abstraction and how we simplify models in order to make them fit the tools' requirement. Abstraction (Clarke et al., 1994) has many advantages: it guarantees, for instance, that the model system is finite (or finitely represents an infinite-state system (To, 2010)], which is a condition for model checking techniques work (Baier and Katoen, 2008). Consequently, this means that a set of problems cannot be checked with formal verification methods, without making strong assumptions. For instance, it is shown in Alur and Dill (1994) that certains classes of timed automata are undecidable and not computable for model checking. Another example is given by Zeigler and Nutaro (2014) for the cyber-physical autonomous cooperative system of systems (CACSoS). The authors state that "despite these drastic simplifications, state space explosion prevents employing more than a handful of uninhabited air vehicles (UAVs) and sensors". Finally, only strong expert knowledge can guarantee that the semantics and restrictions of the model checker tool is enough to make a correct abstraction of the system.

Furthermore, it is relevant to consider how models are built. Abstraction-refinement (A-R) process (Hsieh and Levitan, 1998; Clarke et al., 1999; Grumberg, 2006) is commonly used to approximate more effectively the behaviour of the original system, by iteratively adding details to abstract states, abstract algorithms and types and constants, before verifying the newly obtained model (Draper and Treharne, 1996). However, A-R loop involves a problem: when the abstraction does not satisfy a property, it does not mean that the real model does not satisfy this property. And because refinement involves restrictions, refinement is perhaps representative of only a part of the checked problem. This is especially true when it is difficult to prove that the correctness of a refinement against a property implies the correctness of the real model against this same property. Indeed, it is a vicious circle: the more the model is refined, the less it represents the real system because of restrictions that the modellers add, although it

makes the verification easier. Conversely, the less a model is refined, the more it is imprecise or not verifiable with formal methods (that is the generalisation problem stated by Baier and Katoen, 2008). Finally, according to Baier, "complementary techniques, such as testing, are [thus] needed to find fabrication faults (for hardware) or coding errors (for software)", although testing is not possible all the time.

The latter point leads to a discussion of the interoperability and universality of formal methods. There exists many formal methods, each having an appropriate expressiveness power to describe specific models. This is particularly problematic because experts may have strong knowledge in each formal method, but also because formal methods could return different results for the same design: Owen (2007) shows, for instance, Cadence SMV and NuSMV give two different results for the same requirement and the same input model. In that sense, we can state that each formal method is specialised as regards a type of system. This can be a problem to justify the use of a formal method instead of another.

Figure 1 The basic M&S entities



Source: Zeigler et al. (2000)

2.2 Introduction to discrete-event simulation

M&S domain has been explored since the early 1960s, but was really theorised in Zeigler (1976). This theory tried to make uniform these two notions used extensively in many disciplines like medicine, physics, etc; it also defines a global and universal framework and methodology that is not dependent on the domain of application. As the name suggests, the two key concepts behind M&S are 'Model' and 'Simulation'. The most popular definition of 'model' is perhaps the one given in Minsky (1965): "To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A". Model is then an abstraction, a simplification, a representation of the reality. This definition joins the one given in formal verification domain, even in the mathematical sense. Indeed, a model is a semantic interpretation of a structure. Simulation is thus "executing a model to generate its behaviour" (Zeigler, 1976), by acting on inputs and parameters of the model. Experimental frame (EF) is a set of conditions under which the real system is observed. This notion is also important because it implies a certain abstraction. From this, a model will be valid in a specific EF (i.e. the model generates the same behaviour as the real system in the given conditions) but not in another. This is the main limitation of simulation. Indeed, we can easily deduce that simulation strongly depends on the played scenarios. Unlike formal methods which explore the entire statespace, simulation focuses only on an identified part of the statespace. In other words, doing an exhaustive verification with simulation is hard and maybe impossible.

Another advantage of M&S is that Zeigler defines a unique and universal formalism to describe discrete-event system in a generic manner. DEVS formalism and its extensions allows modelling a full-range of discrete-event systems as simple transition system. All these contribute to make the M&S framework very easy to use for modelling and simulation of many types of systems, like complex natural phenomena (Wainer, 2015).

In the V&V terminology, an analogy can be made between testing and simulation. The weakness of simulation is due to this being an empirical method. Correctness of the simulation essentially comes from the precision of the assumptions made on the EF. Then, simulation-based verification depends on specific scenarios and conditions under which it is tested. This does not mean the entire state space cannot be checked as for formal verification, but it would be probably more costly (and because efficiency of a simulation model can only be evaluated by comparing its outputs with those of the real system for specific inputs). Note that because simulation is evaluated under specific EF, this enforces the notion of determinism; in other words, for the same set of inputs, the model must generate the same behaviour and the same outputs.

2.3 Complementarity between simulation and formal verification

In order to overcome the weakness of formal methods and simulation, the literature explores some techniques which combine these methodologies. Firstly, papers concerning formal verification and model checking clearly consider simulation. According to Baier and Katoen (2008): "In order to improve the quality of the model, a simulation prior to the model checking can take place. Simulation can be used effectively to get rid of the simpler category of modelling errors". Instenberg et al. (2006) note that "UPPAAL provides an integrated simulation tool. It allows the user to examine the dynamic system behavior in a graphical manner either interactively, [...] or randomly to let the system run on its own". And SPIN and NuSMV model checkers also integrate a 'simulator' (Holzmann, 1997; Cimatti et al., 1999]. These simulators are not in the sense of M&S concepts because simulation involves determinism of models; however, they can be seen as the result of the need of "executing formal specifications" for the purpose of validation. PROMELA and SMV are based on an operational semantics that allows the execution, but that is

not the case for all specification languages. Leuschel and Butler (2003) prefers the term "animation of specification" for their ProB model checker.

The need of increasing the credibility of simulation models leads researchers to apply formal methods in M&S (Kuhn et al., 2003) in order to verify simulation models. Many methods to transform certain DEVS subclasses into Timed Automata for purpose of static verification were developed (Dacharry and Giambiasi, 2007b; Saadawi and Wainer, 2009). Other approaches tend to integrate Z into DEVS models (Trojet et al., 2009; Trojet, 2010) also by transformation.

What is the problem with existing methodologies? These techniques combining formal verification and simulation like in Abdulhameed et al. (2014) derive the verification model from the simulation model in order to verify it, or derive a verification and a simulation model from a conceptual model. In the second case, the derivative does not guarantee that the obtained models are related to each other (Li et al., 2006). In the first case, the verification is about the simulation model (meaning the implementation of the simulator) and not the initial design. Moreover, it implies to apply some restrictions on the simulation model in order to enable the verification. Thus, there is no guarantee that the verification model represents the initial design.

3 Combining formal verification and simulation with discrete-event PROMELA

3.1 Overview

Our proposed methodology tries to reduce and overcome the previous disadvantages. The main idea is introducing the semantics of a discrete-event simulation language into a verifiable language. A similar approach was recently described in Aliyu et al. (2016). The general approach that we propose is as follows:

- Choose a verifiable formalism (like PROMELA, Timed Automata...) called *the source formalism*. Identify the missing notions among the three main concepts related to discrete-event simulation: event mechanisms, state lifespan, types of transition (internal or external).
- Introduce syntactical elements into the source formalism in order to enable modelling of these notions. Add a new datatype to model infinite values if needed. If the source formalism is based on state machines, make sure that adding these concepts does not change the structure of the underlying automaton.
- Define the new operational semantics based on the *targeted* discrete-event simulation formalism [DEVS, G-DEVS (Giambiasi et al., 2001), temporal sequential machine (Giambiasi, 2009)...].
- Use the obtained formalism to design systems.

Then, we can be sure that we are able to easily define a morphism which translates models expressed in the new formalism into models expressed in the source formalism and which conserve all their structural properties. Indeed, the automaton underlying the new formalism is built upon the automaton underlying the source formalism. Moreover, we are also sure that it exists a simulation model which has the same behaviour than the behaviour of the model expressed in the new formalism. Thus, we can easily define a morphism which transforms models expressed in the new formalism into models expressed in the targeted simulation formalism.

In order to illustrate our approach, we apply it to extend PROMELA.

3.2 Recall about PROMELA

PROMELA is a specification language with an operational semantics and initially designed for the modelling and verification of concurrent protocols, involving synchronous or asynchronous communication between processes. Based on Djikstra's Guarded Command Language, its syntax is close to any imperative language, making their use very easy, compared with others formal methods. Because the language is very complete, we will focus there only on interesting concepts for the scope of this paper.

A PROMELA specification is thus a set of two separate parts: the system specification, on the one hand, which describes the behaviour of the model, and on the other hand, the properties to verify on the model.

3.2.1 System specifications

A PROMELA system is a *finite* set of components: *instances* of *processes*. These instances can communicate with each other thanks to different mechanisms as *buffered messages*, *shared global variables* or *rendez-vous handshakes*. Each instance of each process is modelled by a *finite* set of guarded or labelled command called *statements*. A statement is said non-blocked if the state of the system allows its execution, otherwise it is said blocked. Then, one execution of the specifications, at any time t_i, corresponds to the execution of one among all of non-blocked statements, without any assumption about duration of the statement execution.

Instructions are divided into two categories: statements that modify the system state and control-flow instructions. Statements relative to state changes are assignments and *message* exchange instructions. *Assignment* statements involve local and global variables, whereas communication statements involve buffered channels. It is important to note that, if assignments are always considered as *enabled* statements (i.e., they can be always executed), the instructions relative to channels can be *blocked* if the associated buffered channel is empty or full. *Control-flow* statements are classical conditional and loop instructions. These ones allow selection of the next statement among different branches regarding a guard. Because PROMELA processes are *non-deterministic*, if several guards are satisfied, a random one is selected. If none of them is satisfied, the control-flow structure is blocked. PROMELA

also provides a *timeout* instruction (usable as a guard) which is enabled if all instructions are blocked in the whole system.

Algorithm 1 A simple example of PROMELA program

```
1: int z = 1;
2:
3: active proctype A {
4:     int x = 2, y = 2;
5:     if
6:     :: ( x == 2 ) → x = 3;
7:     :: ( y == 2 ) → y = 4;
8:     fi;
9: }
10:
11: active proctype B {
12:     int x = 2, y;
13:     do
14:     :: ( y == 2 ) → x = 2;
15:     :: ( x == 2 ) → y = 4;
16:     od;
17: }
18:
19: ltl {[](z == 1);}
```

Data in PROMELA are represented by local and shared variables. Local variables are those which are relative to only the process which they belong to, whereas global variables are shared by all processes. A variable is characterised by its value and its type, or any finite combination (structures) or finite arrays of these types. Each PROMELA type represent a finite set of values.

The immediate result is that a PROMELA model can be represented by an underlying finite automaton.

3.2.2 Properties specifications

SPIN supports the verification of linear temporal logic (LTL) properties on the PROMELA models. LTL properties are converted into a *never-claim* process (comparable to any *normal* processes) which don't "participate" to the behaviour of the system. The goal of a never-claim process is only to guarantee the system satisfies the property which is encoded in it. In this sense, a never-claim process acts as a *monitor*. The study of LTL encoding is out-of-scope of this paper. Thus, we recommand the SPIN Reference Manual (Holzmann, 2003) to the interested readers for further informations.

Then, the formal verification of PROMELA specifications intuitively corresponds to the checking of all executable paths against a given property, without any assumptions of duration. It results that the next state of a PROMELA model does not depend on the time elapsed in a previous state. Because it is not possible to model this elapsed time, timed extensions were developed.

3.2.3 Executability of a process

A PROMELA process with a set of statements L is thus a finite state machine $P = (Q, T, s_0, F)$ where

- $Q = \{q_i = (l_1, ..., l_m, g_1, ..., g_n, c_1, ..., c_o) \in \prod_{i=1}^{m} L_i \prod_{j=1}^{n} G_j \prod_{k=1}^{o} C_k\}$, the finite set of states; a state is characterised by the values of each local and shared variables, and channels (the all sets L_i, G_j and C_k)
- $T \subset Q \times L \times Q$, the finite set of transitions labelled by a statement $l \in L$
- $s_0 \in Q$, the initial state of the process
- $F \subset Q$, the finite set of final states of the process.

Denote $(q_i, q_j) \in Q_2$ and $l \in L$ a statement (i.e. an instruction in the PROMELA program). Then, $t \in T$ iff the process can change its state from q_i to q_j by executing l only. In other words, a statement denotes two *consecutive* states and $t = (q_i, l, q_j)$ (we will note $q_i \xrightarrow{l} q_j$).

Given two states (q_i, q_j) and a labelled transition $t = q_i \xrightarrow{l} q_j$, t is *enabled* iff

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

3.2.4 Executability of a program

A PROMELA program with n processes is a subset of the cartesian product of the *state graph* of each process. Thus, a PROMELA Program M is a Kripke (1963) structure $K =< S, s_0, R, L >$ with

- $S \subseteq \prod_{i=1}^{n} Q_i$, where Q_i is the statespace of the process p_i $(1 \le i \le n)$.
- $s_0 = (s_{0_1}, ..., s_{0_n})$ where $\forall i \in [1; n]$, s_{0_i} is the initial state of the process p_i.
- $R \subseteq S \times S$, the set of transitions. By definition, R is *left-total*, meaning $\forall s \in S, \exists s' \in S$ such as $(s, s') \in R$.
- $L: S \rightarrow 2^{AP}$, the labeling function, which gives an interpretation of atomic logic propositions ($p \in AP$) for each state.

Denote $r = (s_m, s_n) \in S^2$. We note q_{m_i}, the state of the process p_i in s_m, and q_{n_i}, the state of the process p_i in s_n. Thus, $r \in R \Rightarrow \exists t \in T_i \setminus t = (q_{m_i}, l_i, q_{n_i})$, where T_i is the set of transitions of the process p_i and $l_i \in L_i$ (here, L_i is the set of statements of the process p_i). By this, we mean r is a transition of a PROMELA program only if it exists a transition t that changes the state of one of the processes composing the program. r is said *enabled* if $\exists t \in T_i$ which is enabled. If at least two transitions are enabled for any state s, thus the system is said *non-deterministic*.

Finally, we call *run* any finite or infinite alternating sequence $w \in W$ of states and transitions: $w = s_1 r_1 s_2 r_2 .... r_{n-1} s_n$. Formal verification thus consists to check all runs w against the desired property. Verification methods and algorithms are out-of-scope of this paper, and we suggest (Baier and Katoen, 2008) as reference for the interested reader.

3.3 DEVS concepts

Discrete-event systems (DES) are a specific class of timed systems, whose state changes at various time instants, depending on instant occurrences of events. Thus, a DES evolves along the events that it emits or consumes. For the modelling of such systems and their analysis, Zeigler (1976) introduced the DEVS formalism, which can be seen as a generalisation of the Moore Machine formalism by associating each state with a lifespan. The traditional DEVS thus relies on the following notions:

- Each state is associated with a real number called lifespan. This real number can take its value on [0; +∞]. When the lifetime of a state has expired, the system emits an output and changes its current state according to the transition table.
- When an input is consumed, the state of the system changes according to the transition table, regardless of the current lifetime of the current state.
- As a result of the previous point, transitions can be characterised as internal or external transitions. Internal transitions model autonomous behaviours while external transitions correspond to reactions to any external events.
- Events are well-dated and can be ordered.
- There is no non-deterministic behaviour. If two events occur at the same time, thus either they are equivalent events ($e_1 = e_2$) or they are prioritised.
- The state, input and output trajectories are piecewise segments; the distribution of events can follow any non-linear function, unlike for discrete-time systems in which the time is determined by a linear function of periods;

More formally, a DEVS model is a coupling of DEVS atomic and coupled models. A DEVS atomic model is the smallest simulable unit defined by $A = (X, Y, S, \delta_{int}, \delta_{ext}, \Lambda, ta)$, where

- X is the set of input values
- Y is the set of output values
- S is the set of states
- δ_{int}: $S \rightarrow S$ is the internal transition function
- δ_{ext}: $Q \times X \rightarrow S$ is the external transition function
- Λ: $S \rightarrow Y$ is the output function
- ta: $S \rightarrow \mathbb{R}_+$ is the time advance function
- $Q = \{(s, e) \mid s \in S, e \in [0, ta(s)]\}$ is the total state set; e is the time elapsed since the last transition.

The meaning of a DEVS can easily be depicted as follows. At any time t, the system is in a state s. If no external event occurs, the system stays in s for time $ta(s)$. If the lifetime expires, meaning the elapsed time e from the last event is equal to $ta(s)$, the system outputs the value $\Lambda(s)$ and changes to the state $\delta_{int}(s)$. If an external event x occurs before the expiration time, meaning that the system is in a state $q = (s, e)$ with $e \leq ta(s)$, then the system changes its state to $\delta_{ext}(q, x)$. The event can transit into the coupled model using the previously defined coupling: an external event coming in the system is transmitted to the components using EIC, while an output generated by a component transits using EOC or IC.

3.4 Discrete-event PROMELA

Using our approach, we developed an extension of PROMELA called DEv-PROMELA (Yacoub et al., 2015), which is the result of introducing the semantics of DEVS into PROMELA. We extended the syntax of PROMELA as shown in Yacoub et al. (2016), in order to be able to specify DES concepts and real values. DEv-PROMELA is thus a syntactic language which allows modelling, simulating and verifying discrete-event models by formal verification and simulation. Indeed, the methodology implies that we can encode any DEv-PROMELA model into a DEVS model, which can be implemented and simulated in any DEVS environment. Because it always exists a PROMELA model structurally equivalent to a DEv-PROMELA model, the model checker SPIN can be used to check time-invariant properties. The semantics alignement between DEv-PROMELA and DEVS is not the object of this paper, which is only focusing on the structural relationship between DEv-PROMELA and PROMELA.

3.4.1 DEv-PROMELA syntax

As said in the previous section, some syntactic elements were added to PROMELA in order to model the essential notions of discrete-event models: clocks and types of transitions. Because DEVS allows real valuation, a new data type real was introduced. Each statement of PROMELA

denotes a transition between two states. Then, in order to define types of transitions and types of events (internal or external), events descriptors prefix any statement. The reader is encouraged to read (Yacoub et al., 2016) for more details about the syntactic changes.

3.4.2 DEv-PROMELA semantics

The operational semantics of DEv-PROMELA corresponds to the semantics of DEVS.

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

- $S_\tau = \{s_i = (t_s, i, l_1, ..., l_m, \in \mathbb{N} \times \prod_{i=1}^{m} L_i \times \prod_{j=1}^{n} G_j \times \prod_{k=1}^{o} C_k)\}$ is the set of states. i is the identifier of the state related to the statement l which defines it; the sets L_i (resp. G_j) are the sets of values of each local (resp. global) variable l_i (resp. g_j)
- E is the set of events; E contains at least the silent event denoted ϵ
- δ_i: $Q_f \rightarrow Q_0 \times E$ is the internal transition partial function
- δ_e: $Q \times E \rightarrow Q$ is the external transition partial function
- s_0 is the initial state
- F is the set of final states.

Moreover, we define:

- $ta: \begin{cases} S_\tau \rightarrow \mathbb{R}^+ \\ ta(s) \mapsto t_s \end{cases}$ is the state lifetime function; the lifetime of each state is given by the delay before executing the next statement in the specifications
- $Q = \{q = (s, dt), \forall s \in S\tau \}$ such that $0 \leq dt \leq ta(s)$ is the set of total states; dt denotes the time elapsed in the state s
- $Q_0 = \{q = (s, 0), \forall s \in S_\tau\} \subset Q$
- $Q_f = \{q = (s, ta(s)), \forall s \in S_\tau\} \subset Q$.

Consider a DEv-PROMELA process P in a state s at time t, and the next statement l with its event descriptor. We can admit the process P is in fact in a state $q = (s, t)$ (if t denotes the elapsed time since the last event). If l denotes an internal transition and if $t = ta(s)$, then the statement l is enabled. The event associated with the transition is emitted to all the other processes composing the program, before the transition is triggered, and the next event for the process P is defined by:

$$d_{e'} = getCurrentDate + ta(s')$$

with $((s', 0), e') = \delta_i(s)$. If l denotes an external transition on an event e, then the transition is triggered only if the process receives the event e. In this case, denote t the date of the event e. The next state is given by $q' = \delta_e(q, e)$ with $q' = (s', 0)$. If δ_e is not syntactically defined for (s, e), then the next state is given by $q' = (s, dt)$ and $ta(s) = t_s$.

Semantics of a DEv-PROMELA program. A DEv-PROMELA program P_r is a transition system $T = (S, \Lambda, \rightarrow)$ where S is the Cartesian product of the set of states of each process, the set of global variables and channels that compose the program, Λ is the set of all statements and $\rightarrow$ the set of transitions. Consider a program Pr at time t and two states $s = (s_{p_i}, s_{q_j}, ...)$. and $s' = (s'_{p_i}, s'_{q_j}, ...)$. Then, $s \xrightarrow{l} s'$ with $l \in \Lambda$ if it exists a transition from s_{p_i} to s'_{p_i} from s_{q_j} to s'_{q_j}, ... and if it does not exist any other transition which can be triggered before the date t. In other words, the next event of Pr is the minimum value of all the next events of each process and external events.

3.4.3 Structural relationship between DEv-PROMELA and PROMELA

Denote P a DEv-PROMELA process with L the set of statements. As previously introduced, P is an automaton $A = (S_\tau, E, \delta_i, \delta_e, s_0, F_A)$. Denote P', a PROMELA process represented by a finite state machine $B = (S, T, s'_0, F_B)$. If P' is a structural equivalent to P, that means it exists a morphism M that transforms P to P', such as:

- $\forall s'_i = (l_1, ..., l_n, g_1, ..., g_m) \in S$, $\exists s_i \in S_\tau, s_i = (t_s, l_1, ..., l_n, g_1, ..., g_m)$; we denote $S \subset S_\tau$ by abstraction and ϕ: $S_\tau \rightarrow S$, the abstraction function; (1)
- $\forall t_s \in \mathbb{R}^+, s_i = (t_s, l_1, ..., l_n, g_1, ..., g_m) \in S_\tau \Rightarrow \exists! s'_i \in S$ such as $s'_i = \phi(s_i) = (l_1, ..., l_n, g_1, ..., g_m)$; (2)
- $\forall t = (s'_i, l, s'_j) \in T$, $\exists (s_i, s_j, e) \in S\tau \times S_\tau \times E$ such as $\delta_i(s_i) = (s_j, e)$ or $\delta_e(s_i, e) = s_j$, and $\phi(s_i) = s'_i$ and $\phi(s_j) = s'_j$; (3)
- $(s_i, s_j, e) \in S_\tau \times S_\tau \times E, \delta_i(s_i) = (s_j, e)$ or $\delta_e(s_i, e) = s_j \Rightarrow \exists (s'_i, s'_j)$ $S \times S \setminus \exists t = (s_i, l, s_j) \in T$ and $\phi(s_i) = s'_i$ and $\phi(s_j) = s'_j$; (4)
- $s'_0 = \phi(s_0)$; (5)
- $F' = \phi(F)$. (6)

(1) and (2) mean that we can always make a projection of any state of S_τ to a corresponding state of S. Moreover, all states s of S_τ sharing the same memory state are projected to an unique state s' of S with the same memory representation. ϕ is thus a *surjective function*.

(3) and (4) mean that if it exists a transition function, either internal or external, from any state s_i to any state s_j,

and if s'_i (resp. s'_j) is a projection of s_i (resp. s_j) by ϕ, then it exists a transition from s'_i to s'_j.

(5) and (6) means that initial and final states are preserved by abstraction.

Suppose M is constructed by only removing the event descriptors of P. By this, we mean that we delete the concepts of event, state lifetime and the characterisation of each transition, in other words the notion of internal and external transition. M is thus an time-abstraction function such as (1) and (2) are verified (all timed states are projected to their untimed equivalent depending only on their memory state). Because we don't remove any statement, (3) and (4) are trivially verified. Indeed, even if a DEv-PROMELA (internal or external) transition is defined between two states $q_i = (s_i, dt_i)$ and $q_j = (s_j, dt_j)$ in Q, a such transition exists only if there is a relationship between s_i and s_j. However, a such relationship is defined syntactically (transitions are defined by the statements). Because we don't remove any statements, a such relationship is always existing, and even for the branching structures. For the same reason, (5) and (6) are also true.

Because a PROMELA program is a synchronised product of each automaton that composes it, the graph of DEv-PROMELA program P is included in the one of P'. Indeed, the graph of P' is composed by the all possible permutation between statements, whereas the graph of P is only composed by the permutation of ordered events. That means, given two events e_1 and e_2 associated to two statement l_1 and l_2, if $date(e_1) < date(e_2)$), the graph of P will semantically not take into account the path where l_1 is executed before l_2 (the path may exists, but may be not valid). Thus, the morphism M defined previously is also valid for the entire program.

3.5 Discussions

Before presenting applications of DEv-PROMELA, we must shortly answer two questions: why discrete-event simulation and why PROMELA? For the former question, discrete-event approach has many advantages comparing real-time execution or discrete-time approach. The discrete-event nature assumes that a system is constant between two occurrences of events. Then, the evaluation of the system is needed only when an event occurs. This substantially reduces the statespace compared to discrete-time approaches which generates a state at fixed rate. Moreover, while real-time execution enforces to really wait an amount of time, discrete-event approach allows jumping from a time to another.

Concerning the second question, we would effectively have been able to extend timed automata (TA) and the UPAAL model checker, considering there are already many efforts and a lot of works to encode DEVS into timed automata (Dacharry and Giambiasi, 2005). However, among all the problems we could encouter, the main problem with TA would be about clock constraints. DEVS allows modelling systems with non-constraint clocks, which will be also done in DEv-PROMELA, while TA impose that clocks constraints are linear relations. Then, extending TA would be possible only with the Time Constrained DEVS (TCDEVS) (Dacharry and Giambiasi, 2007a).

4 Applications

4.1 A soccer video game

Numerous works try to improve video games development environments (Ekyalimpa et al., 2014; Choi et al., 2015). In this context, DEv-PROMELA was applied to model, verify, and validate a design of a soccer video game. Indeed, a video game can be seen as a discrete-event system at three levels. At a lower level (the implementation level), a game is a finite automaton in which each instruction is executed after an amount of time (the time needed by the CPU or the GPU to execute an instruction). Thus, an executable statement denotes two states with a defined lifespan. The lifespan can be fixed or can evolve through the time. Moreover, in a higher level (the conceptual level), the design of a game is reactive, because software must react to inputs coming from the player. Each input is an event which is well-dated. If the game is a multiplayer game, events can come from many players. In this case, each player is a component of the system. At a middle level, the game is composed by several components like a graphics engine, a physics engine... which interact with each other. The communications are generally done through mechanisms of events in order to synchronise components. Moreover, we know that a game is just a set of images which are computed with a defined frequency. Thus, each image depends on the time needed to compute it. By extension, we can consider that a new picture is shown when their computation time is done. This corresponds to stay in a state for a defined time, and go to another state in an autonomous manner when the lifespan of the current state is over.

Figure 2 The goalkeeper design error in a soccer video game (see online version for colours)

If we want to verify the implementation, it seems that a verification model is enough. Indeed, we can easily transform an implementation in a PROMELA model to check liveness properties. Indeed, the verification model will check all of the possible executions to check deadlock states, unreachable states... One can argue also we can use timed automata to model time (i.e., the time needed to

compute an image for example). However, if model-checking can check dynamics which depend on time, it is not able to check properties on states that depend themselves on time. For instance, the avatars of a soccer video game move with a defined speed. Because of the discrete nature of software, the new position of the avatar is computed by taking the speed and the time elapsed in a previous state (i.e., the time taken by the CPU/GPU to compute the new image). By making an linear or a polynomial interpolation (depending on the physics rules), the new positions are computed. In other word, each state of the game itself depends not only on the timed dynamics but also on the time elapsed in a previous state. Then, if we want to verify a property related to collision detection, model-checking cannot be applied because the problem become undecidable (Alur and Dill, 1994). In fact, verification languages will enforce strong abstractions which ensure that the problem remain verifiable (for instance, by enforcing that the domains of values are finite). Then, such cases cannot even be modelled. And even designers makes an abstraction of such properties, this one would not be representative of the real implementation.

With DEv-PROMELA, that kind of properties can be express through the DEVS semantics, like show in Algorithm 2. The `clt` event descriptors (l.4, l.7) defines the elapsed time before the execution of the next statement. This corresponds to an autonomous behaviour. The value can be a constant or a variable, or any function evaluated to real. However, we introduce also a reactive transition like in DEVS through the `evt` descriptor (l.12). When the goalkeeper receives an event 'MOVE' from any other components of the game, the instruction l.13 is executed. The new state depends on the elapsed time from the last evaluation, because it is defined by the goalkeeper's new position which is computed according to the time.

Algorithm 2 A DEv-PROMELA model of goalkeeper

```
1:  active proctype goalkeeper () {
2:      gk_state = GK_IDLE;
3:      do
4:       :: [clt : 0.5 → emit : silent]
5:      ( gk_state == GK_DIVE ) →
6:          gk_state = GK_IDLE;
7:       :: [clt : 0.5 → emit : silent]
8:      ( shooted && gk_state != GK_CATCH ) →
9:          gk_state = GK_DIVE;
10:      :: [clt : 0.5 → emit : silent]
11:      ( gk_state == GK_IDLE ) →
12:      :: [evt : MOVE] else →
13:              sk_position = sk_position −
14:          speed ∗ getElapsedTime;
15: ...
```

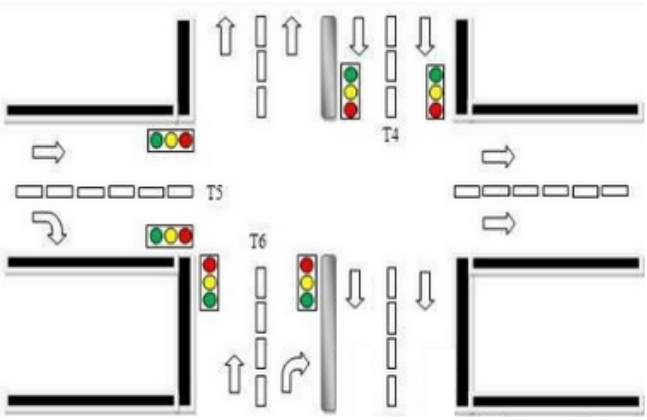
The verification is then done in two steps. Our tool DEv-PROMELA Studio automatically transforms the DEv-PROMELA specification into a PROMELA equivalent model by removing the event descriptors. The obtained model is structurally equivalent to the initial model and the classic model-checking can be used to check structural deadlocks and unreachable states (because the structure of software does not depend on time). For instance, the PROMELA model ensures that the score is always a positive value. For behavioural properties (like collision, speed, constraints on position, deadlocks due to absence of events or infinite-life states...), the model is simulated. A simulation-based verification is then performed by checking several scenarios. Scenarios can be randomly generated or defined by the designer.

4.2 A crossroad

Another application of DEv-PROMELA is the classical crossroad problem. We consider a crossroad with four traffic lights. Each traffic lights stays in a state (red, yellow, green) for a defined time, and only two of them can be green at the same time. We want to verify that a controller software respects this constraint.

The problem is verifiable with timed automata (Alur and Dill, 1994) and timed extensions of PROMELA if the clock of each traffic light evolves linearly. But in the case in which the time elapsed in each state is a nonlinear combination of all clocks, the problem becomes undecidable. DEv-PROMELA overcomes this problem thanks to simulation. Like for the video game example, the model-checking will help designer to see problems related to the structure of the underlying automaton, while simulation will allows verification of timed properties (because non-linear clock evolutions are not a problem in simulation). In this example, we encode the DEv-PROMELA model into a DEVS model which we simulate into the MS4 Me Environment (Seo et al., 2013).

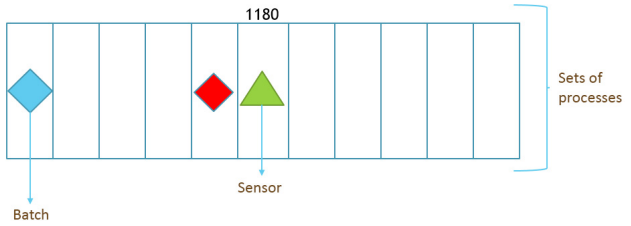
Figure 3 The crossroad problem (see online version for colours)



4.3 A manufacture chain

Manufacturing problems were classical and already intensively studied in the literature (Detty and Yingling, 2000; Longo et al., 2012; Tajini et al., 2014). This problem consists on a set of thousand processes which are executed on several batches of products. At each stage of the production, a controller performs some operations which take an amount of time.

Figure 4 The manufacture chain. Each square represent a step or a process/operation (see online version for colours)



When an operation is finished, a signal is sent to the controller which leads the batch to the next operation. Operations are just encoded with Bash-like instructions, like shown in Figure 5. In order to show the potential of our proposed methodology, we model a chain with only two operations and one sensor, successively using PROMELA, Real-Time PROMELA (Tripakis and Courcoubetis, 1996) and DEv-PROMELA. Then, we check if there is any unreachable state and that, in any cases, the attribute `batch attr1615` is equal to 0 at end of the manufacturing process when attribute 1001 is equal to 0. While the PROMELA and RT-PROMELA models are checked using model-checking, the DEv-PROMELA model is verified using model-checking and discrete-event simulation.

Figure 5 Exsmple of the operation 1180

```
REWORK
BEGIN
START
LOG EVENT PRPMARKLAS FOR LASER
IF ATTRIBUTE (1001) OF LOT EQ LAS01 THEN GO TO 8511
LOG EVENT PRMCHECWID FOR MEASURE
IF ATTRIBUTE (1615) OF LOT NE 0 THEN GO TO 9000
LOG EVENT PRMD0LAS01D FOR MEASURE
LOG EVENT MSLD0LAS01D FOR LASER
SET ATTRIBUTE (1615) OF LOT TO 0 NODISPLAY
END
```

The translation from the Bash instruction to PROMELA is very easy. Each operation is encoded using a PROMELA process.

Batches' attributes are modelled using integers variables batch_*. The sensor is modelled using a process that randomly changes the values of attributes. Communications between sensors and processes are modelled using channels. With PROMELA, quantitative delays are totally abstracted, because they cannot be modelled, while RT-PROMELA allows modelling of time using clock, as shown in Algorithm 5.

The DEv-PROMELA model 4 is slightly different.

Because DEv-PROMELA is based on DEVS, the event mechanism is directly expressed by the transitions, and each event are well-dated. Moreover, even the sensor can be also modelled using DEv-PROMELA, we decided to use a classic DEVS model to represent it. Then, the model-checking is only applied of the model of operations. Then, by coupling the DEv-PROMELA model and the DEVS model of the sensor, we check by simulation the potential deadlocks, while the safety property concerning the attribute 1,615 is always checked using model-checking. Results are presented in Table 2.

Models generation. Two subsequent models are generated from the DEv-PROMELA models. The formal verification model is generated using Classic PROMELA by removing the event descriptors, and using the mapping previously defined. The obtained untimed model represents all the possible event permutations, but does not include the time. This level of abstraction is enough to verify structural properties or untimed properties like the one that we want to verify: "Is the attribute 1615 equal to 0 at the end of the manufacturing process?".

The second model is a DEVS abstract model independent of the target simulator. This model has the same semantics than those of the DEv-PROMELA model. This DEVS abstract model is then implemented in a DEVS simulation model [in this case, in the MS4 Me Environment (Seo et al., 2013)] and simulated to check timed properties.

Table 1 Results of crossroad checking

	States	*Transitions*	*Remarks*
DT-PROMELA	1,251	2,759	
Timed PROMELA	927	1,603	Only linear relations between clocks
DEv-PROMELA	422	834	Simulation allows using of arbitrary clocks

Algorithm 3 PROMELA model of the operation 1180

```
1: active proctype op1180 () {
2:     event!PRPMARKLAS;
3:     event?code; // Log Event PRPMARKLAS
4:     if
5:      :: batch_attr1001 == 0 → goto STEP8511;
6:      :: else → skip;
7:     fi;
8:     event!PRMCHECWID;
9:     event?code; // Log Event PRMCHECWID
10: STEP8511: if
11:      :: batch_attr1615 ! = 0 → goto STEP9000;
12:      :: else → skip;
13:     fi;
14:     event!PRMD0LAS01D;
15:     event?code; // Log Event PRMD0LAS01D
16:     event!MSLD0LAS01D;
17:     event?code; // Log Event MSLD0LAS01D
18: STEP9000: batch_attr1615 = 0;
19: ...
20: }
```

Algorithm 4 DEv-PROMELA model of the operation 1180

```
1: active proctype op1180 () {
2:     [clt : 0.28 → emit : PRPMARKLAS]
3:     skip; // Log Event PRPMARKLAS
4:     if
5:      :: [evt : laser] batch_attr1001 == 0 → goto STEP8511;
6:      :: [evt : laser] else → skip;
7:     fi;
8:     [clt : 0.27 → emit : PRMCHECWID];
9:     skip; // Log Event PRMCHECWID
10: STEP8511: if
11: ...
12: }
```

Results. As expected, clocks in RT-PROMELA generate a huge statespace compared with PROMELA and DEv-PROMELA. Indeed, both model generated by PROMELA and DEv-PROMELA for formal verification are untimed model. In the three cases, model-checking was able to detect that the event `PRMCHECWID` is never generated by the model. But, the main difference in this case is about the deadlock found by the checkers. In the case of PROMELA and RT-PROMELA, no deadlock were detected in the model, while DEv-PROMELA allows us to detect some deadlocks. In fact, the problem comes from the model of the sensor. The level of abstraction of time used in PROMELA and RT-PROMELA does not allow modelling cases in which the sensor does not send the `laser` event. But, the discrete-event simulation shows that it exists some cases in which this event is never generated. In these cases, the model stays locked, and the operation cannot progress. It is important to note that if simulation could detect the deadlock, it was mainly because scenarios were well-chosen. Indeed, simulation-based verification depends on the played scenarios.

Table 2 Results of chain checking

	States	*Transitions*	*Time*	*Memory*	*Results*
PROMELA	315	558	0.2	128,653	4 unreachable states, no deadlock
RT-PROMELA	945	1,697	0.5	256,686	4 unreachable states, no deadlock
DEv-PROMELA	128	353	0.09	100,302	2 unreachable states, no deadlock found by model checking, deadlocks found in simulation

The other important aspect of DEv-PROMELA is that we can combine different discrete-event simulation formalisms for the simulation-based verification. Indeed, as we said previously, even we could model the sensor using DEv-PROMELA, it is possible to use Petri Nets or any others discrete-event simulation formalisms to model the sensor and combine it with the DEv-PROMELA model (thanks to the DEVS Bus concept). However, in this case, model-checking is like a supportive method to the simulation-based verification.

Algorithm 5 RT-PROMELA model of the operation 1180

```
1: int batch_attr1615;
2: int batch_attr1001;
3: chan event = [1] of byte;
4: byte code;
5:
6: active proctype op1180 () {
7:     when {y ≤ 28 } reset { y } event!PRPMARKLAS;
8:     event?code; // Log Event PRPMARKLAS
9:     if
10:     :: batch_attr1001 == 0 → goto STEP8511;
11:     :: else → skip;
12:    fi;
13:    event!PRMCHECWID;
14: ...
15: }
```

5 Discussion about high-level verification, validation and test techniques

In brief summary, the literature about verification, validation and test (V&VT) expresses the necessity of improving modelling, implementation, verification and validation techniques, since new systems are increasingly complex. However, it is also admitted that V&VT best practices must focus on high-level specifications and abstractions. For instance, new model-based programming paradigms emerge for the implementation of event-driven systems (Milicevic et al., 2013). The main goal of such techniques is to reduce the number of errors due to the gap between abstraction and implementation. These methodologies thus focus on separating the problems related to the behaviour of the design on the one hand, from the problems related to the implementation on the other hand (low-level messages, queue, data structures...). In other words, these works are not focusing on how improving existing V&V techniques, but how encouraging developers to focus on specifying the core part of the program under implementation, i.e., the event model. DEv-PROMELA is in this spirit by introducing these event-driven aspects in a verification language. However, we think that, if operational details (for example, how real values are implemented into the program) are not relevant for a design and its verification, data abstraction is a source of losing accuracy.

Other works (Vakili and Day, 2012) state that, if high-level declarative specification languages like alloy (Jackson, 2006) are really suitable for high-level modelling of transition systems (i.e., the structure), their lack of operational semantics enforce designers to translate models to other tools in order to carry out full model-checking. Vakili and Day (2012) also state that

> "However, if we wish to provide analysis support for these models to increase their quality and utility, we must be able to express the models precisely."

This statement shows that the models need to embed details about how they will behave, independently of their implementation, but also with respect to their implementation. Especially, the need of encoding temporal logic into declarative specifications shows that untimed models are insufficient to represent systems with accuracy. DEv-PROMELA goes in this direction by adding timed aspects into untimed models. However, DEv-PROMELA goes further by not only focusing on the ordering of event, but also on the relation between the elapsed time, states and properties, by using DEVS and not only temporal logic.

Finally, DEv-PROMELA is a part of this trend that combines model-checking and simulation for enhancing formal verification and formal validation (Gore et al., 2014). However, while these works focus on the conceptual modelling at the Subject Matter Expert (SME) level, DEv-PROMELA is designed to be used at different specification levels. We mean that our goal is not to make easier the conceptual modelling for a SME. We improve the model-checking results by combining formal verification and simulation. We are not trying to define a new Domain-Specific Language (DSL) that can be then translated into verifiable and simulable specifications, or explaining how a DSL can be translated into a verification model (like ConceVE). However, DEv-PROMELA is designed in order to add more details about time and events in an untimed verifiable model. This new model can be then formally verified or simulated for verifying and validating structural properties on the one hand, and behavioural properties in the other hand. This is done by automatically translating the model into a simulation model in order to analyse its behaviour or into a verifiable model to check non-timed properties. In this way, new verification capabilities are added to a verification language, while the model is kept simple and focused on its core aspects.

6 Conclusions

We described in this paper some work to develop a new approach which really combines discrete-event simulation and formal verification for V&V purposes. The main goal is to add new verification capabilities to existing formal verification languages. The approach consists to extend a verifiable language (called the source formalism) by adding the semantics of the DEVS abstract simulator. Consequently, this ensures that for any model expressed in the extended formalism, it exists an abstraction which can be verified using formal methods. Moreover, it can also be simulated in order to verify timed and behavioural properties which can not be verify with formal methods. A result of this approach is DEv-PROMELA, which is an extension of PROMELA for the modelling 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 behavioural properties. It is interesting to note that a DEv-PROMELA model is a syntactic model. This means that a DEv-PROMELA model can be easily automatically translated and implemented in any DEVS simulator which supports Classic DEVS. As an illustration of this capabilites, we use the MS4 Me environment to perform some simulation-based verifications. DEv-PROMELA was successfully applied to verify and validate designs of video games.

We also applied it on a model a manufacture chain, and compared the result with classic model-checking. The DEv-PROMELA model was able to reduce the statespace for the verification of time-invariant properties, while simulation-based verification was able to detect deadlocks that verification models was not able to detect.

One of the major drawbacks of this approach is inherent to the simulation: simulation-based verifications strongly depends on the verified scenarios. Identifying scenarios is a challenge for designers (Banks, 1984) and depends on the applications domains. Several simulations and coverage tests must be performed like in Li et al. (2006) to ensure that the main scenarios were tested. Finally, by integrating discrete-event simulation and formal verification in a same specification language, DEv-PROMELA reduces the risk of errors due to transformations and morphisms.

As future works, we can also explore a way to use simulation in order to speed-up model-checking. Indeed, simulation could allow designers to identify execution paths which have a real meaning in the real world. Then it could be useful in order to limit the statespace explosion problem.

Acknowledgements

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

References

Abdulhameed, A., Hammad, A., Mountassir, H. and Tatibouet, B. (2014) 'An approach combining simulation and verification for sysml using systemc and uppaal', in *CAL 2014, 8`eme conférence francophone sur les architectures logicielles*.

Aliyu, H.O., Maıga, O. and Traoré, M.K. (2016) 'The high level language for system specification: a model-driven approach to systems engineering', *International Journal of Modeling, Simulation, and Scientific Computing*, Vol. 07, No. 01, 1641003.

Alur, R. and Dill, D.L. (1994) 'A theory of timed automata', *Theoretical Computer Science*, Vol. 126, No. 2, pp.183–235.

Baier, C. and Katoen, J-P. (2008) *Principles of Model Checking*, The MIT Press, London, England.

Banks, J. (1984) *Discrete-event System Simulation*, Prentice-Hall, New Jersey, USA.

Bowen, J.P. and Hinchey, M.G. (1995) *Applications of Formal Methods*, Prentice Hall PTR, New Jersey, USA.

Chang, C-L. and Lee, R.C-T. (1973) *Symbolic Logic and Mechanical Theorem Proving*, 1st ed., Academic Press, Inc., Orlando, FL, USA.

Choi, C., Seok, M-G., Choi, S.H., Kim, T.G. and Kim, S. (2015) 'Military serious game federation development and execution process based on interoperation between game application and constructive simulators', *International Journal of Simulation and Process Modelling*, Vol. 10, No. 2, pp.103–116.

Cimatti, A., Clarke, E.M., Giunchiglia, F. and Roveri, M. (1999) 'Nusmv: a new symbolic model verifier', in *Proceedings of the 11th International Conference on Computer Aided Verification*, *CAV '99*, pp.495–499, London, UK, Springer-Verlag.

Clarke, E.M. (2008) 'The birth of model checking', in *25 Years of Model Checking: History, Achievements, Perspectives*, pp.1–26, Springer Berlin Heidelberg, Berlin.

Clarke, E.M. and Emerson, E.A. (1982) 'Design and synthesis of synchronization skeletons using branching-time temporal logic', in *Logic of Programs, Workshop*, pp.52–71, Springer-Verlag.

Clarke, E.M., Grumberg, O. and Long, D.E. (1994) 'Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, September, Vol. 16, No. 5, pp.1512–1542.

Clarke, E.M., Grumberg, O. and Peled, D.A. (1999) *Model Checking*, MIT Press, Massachusetts.

Cousot, P. and Cousot, R. (2010) 'A gentle introduction to formal verification of computer systems by abstract interpretation', in *Logics and Languages for Reliability and Security*, pp.1–29, IOS Press.

Dacharry, H. and Giambiasi, N. (2005) 'Formal verification with timed automata and devs models: a case study', in *Proceedings of Argentine Symposium on Software Engineering*, pp.251–265.

Dacharry, H.P. and Giambiasi, N. (2007a) 'Discrete event modeling through a multiformalism approach, from a user-oriented perspective', in *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, *SpringSim '07*, pp.207–213, Society for Computer Simulation International, San Diego, CA, USA.

Dacharry, H.P. and Giambiasi, N. (2007b) 'A formal verification approach for devs', in *Proceedings of the 2007 Summer Computer Simulation Conference*, *SCSC '07*, pp.312–319, Society for Computer Simulation International, San Diego, CA, USA.

Detty, R.B and Yingling, J.C. (2000) 'Quantifying benefits of conversion to lean manufacturing with discrete event simulation: a case study', *International Journal of Production Research*, Vol. 38, No. 2, pp.429–445.

Draper, J. and Treharne, H. (1996) 'The refinement of embedded software with the bmethod', in *Northern Formal Methods Workshop*, Springer.

Duffy, D.A. (1991) *Principles of Automated Theorem Proving*, John Wiley & Sons, Inc., New York, NY, USA.

Ekyalimpa, R., AbouRizk, S.M., Mohamed, Y. and Saba, F. (2014) 'A prototype for project management game development using high level architecture', *International Journal of Simulation and Process Modelling*, Vol. 9, No. 3, pp.131–145.

Giambiasi, N. (2009) 'From sequential machines to devs formalism', in *Proceedings of the 2009 Summer Computer Simulation Conference*, *SCSC '09*, pp.216–222, Society for Modeling; Simulation International, Vista, CA,.

Giambiasi, N., Escude, B. and Ghosh, S. (2001) 'Gdevs: a generalized discrete event specification for accurate modeling of dynamic systems', in *Autonomous Decentralized Systems, 2001. Proceedings. 5th International Symposium on*, pp.464–469.

Gore, R., Diallo, S. and Padilla, J. (2014) 'Conceve: conceptual modeling and formal validation for everyone', *ACM Trans. Model. Comput. Simul.*, Vol. 24, No. 2, pp.12:1–12:17.

Grumberg, O. (2006) 'Abstraction and refinement in model checking', in *Formal Methods for Components and Objects*, pp.219–242, Springer.

Heitmeyer, C. (1998) 'On the need for practical formal methods', in *Proceedings of 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp.18–26, Springer Berlin Heidelberg.

Holzmann, G.J. (1991) *Design and Validation of Computer Protocols*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Holzmann, G.J. (1997) 'The model checker spin', *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp.279–295.

Holzmann, G.J. (2003) *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, Boston.

Hsieh, Y-W. and Levitan, S.P. (1998) 'Model abstraction for formal verification', in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp.140–147, IEEE Computer Society.

Huth, M. and Ryan, M. (2005) *Logic in computer science: modelling and reasoning about systems*, Cambridge University Press, Cambridge, UK.

Instenberg, M., Schneider, A., Schnetter, S., Heinkel, U., Guldstrand Larsen, K. and Behrmann, G. (2006) *Formal Methods for Abstract Specifications – A Comparison of Concepts*, Technical report, IEEE Press.

Jackson, D. (2006) *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, Massachusetts.

Kripke, S.A. (1963) 'Semantical considerations on modal logic', *Acta Philosophica Fennica.*

Kuhn, D.R., Craigen, D. and Saaltink, M. (2003) 'Practical application of formal methods in modeling and simulation', in *Proceedings of SCSC03, Summer Simulation Multiconference*, Citeseer.

Kunzli, S., Poletti, F., Benini, L. and Thiele, L. (2006) 'Combining simulation and formal methods for system-level performance analysis', in *Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings*, pp.236–241, European Design and Automation Association.

Leuschel, M. and Butler, M. (2003) 'Prob: a model checker for b', in Keijiro Araki, Stefania Gnesi, and Dino Mandrioli (Eds.): *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pp.855–874, Springer Berlin Heidelberg.

Li, L., Thornton, M. and Szygenda, A. (2006) 'Integrated design validation: combining simulation and formal verification for digital integrated circuits', *Journal of Systemics, Cybernetics and Informatics*, Vol. 4, No. 2, pp.22–30.

Longo, F., Massei, M. and Nicoletti, L. (2012) 'An application of modeling and simulation to support industrial plants design', *International Journal of Modeling, Simulation, and Scientific Computing*, Vol. 03, No. 01,1240001.

Loveland, D.W. (1978) *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*, Sole distributor for the U.S.A. and Canada, Elsevier North-Holland.

Milicevic, A., Jackson, D., Gligoric, M. and Marinov, D. (2013) 'Model-based, event-driven programming paradigm for interactive web applications', in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp.17–36, ACM.

Minsky, M. (1965) 'Matter, mind and models', in *IFIP Congress*, pp.45–50, Spartan Books, London, UK.

Owen, D.R. (2007) *Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy*, PhD thesis.

Queille, J-P. and Sifakis, J. (1982) 'Specification and verification of concurrent systems in cesar', in *Proceedings of the 5th Colloquium on International Symposium on Programming*, pp.337–351, Springer-Verlag.

Saadawi, H. and Wainer, G. (2009) 'Verification of real-time devs models', in *Proceedings of the 2009 Spring Simulation Multiconference*, p.143, Society for Computer Simulation International.

Seo, C., Zeigler, B.P., Coop, R. and Kim, D. (2013) 'Devs modeling and simulation methodology with ms4 me software tool', in *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*, pp.33:1–33:8, Society for Computer Simulation International.

Tajini, R., Elhaq, S.L. and Rachid, A. (2014) 'Modelling methodology for the simulation of the manufacturing systems', *International Journal of Simulation and Process Modelling*, Vol. 9, No. 4, pp.285–305.

To, A.W. (2010) *Model Checking Infinitestate Systems: Generic and Specific Approaches*, PhD thesis.

Tripakis, S. and Courcoubetis, C. (1996) 'Extending promela and spin for real time', in *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, *TACAs '96*, pp.329–348, London, UK, Springer-Verlag.

Trojet, M.W. (2010) *Approche de vérification formelle des modeles DEVS a base du langage Z*, PhD thesis.

Trojet, M.W., Frydman, C. and El-Amine Hamri, M. (2009) 'Practical application of lightweight z in devs framework', in *Proceedings of the 2009 Spring Simulation Multiconference*, p.154, Society for Computer Simulation International.

Vakili, A. and Day, N.A. (2012) 'Temporal logic model checking in alloy', in *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, *ABZ '12*, pp.150–163, Springer-Verlag.

Wainer, G.A. (2015) 'The cell-devs formalism as a method for activity tracking in spatial modelling and simulation', *International Journal of Simulation and Process Modelling*, Vol. 10, No. 1, pp.19–38.

Yacoub, A., Hamri, M. and Frydman, C. (2016) 'Using dev-promela for modelling and verification of software', in *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2016*, Banff, Alberta, Canada, May 15-18, 2016, pp.245–253.

Yacoub, A., Hamri, M., Frydman, C., Seo, C. and Zeigler, B. (2015) 'Towards an extension of promela for the modeling, simulation and verification of discreteevent systems', in *Proceedings of the 27th European Modelling and Simulation Symposium (EMSS 2015)*, pp.340–348, September.

Zeigler, B.P. (1976) *Theory of Modeling and Simulation*, John Wiley, Ann Arbor, Michigan.

Zeigler, B.P. and Nutaro, J. (2014) 'Combining devs and model-checking: using system morphisms for integrating simulation and analysis in model engineering', in *Proceedings of the 26th European Modeling and Simulation Symposium*, pp.350–356.

Zeigler, B.P., Kim, T.G. and Praehofer, H. (2000) *Theory of Modeling and Simulation*, 2nd ed., Academic Press, Inc., San Diego, California.

Zervoudakis, F., Rosenblum, D.S., Elbaum, S. and Finkelstein, A. (2013) 'Cascading verification: An integrated method for domain-specific model checking', in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp.400–410, ACM.