DEv-PROMELA: modeling, verification, and validation of a video game by combining model-checking and simulation

Aznam Yacoub, Maamar El Amine Hamri, Claudia Frydman

SIMULATION: Transactions of The Society for Modeling and Simulation International
XX(X):1–27

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

DEv-PROMELA : Modelling, Verification and Validation of a Video Game by Combining Model-Checking and Simulation

Aznam Yacoub[1], Maâmar El-Amine Hamri[1] and Claudia Frydman[1]

Abstract

Modelling, Verifying and Validating are essential steps in order to build systems and software that do what designers expect. If Formal Verification, and especially Model-Checking, is a popular method for proving correctness of properties, its efficiency depends on the accuracy of the used models, and the quality of abstractions. As a consequence, applying verification techniques on large-scale complex software like video games is hard without strong assumptions and simplifications. Simulation models are generally more accurate than verification models, but it is often like harder to verify them. Combined formalisms that take benefits of both Model-Checking and Discrete-Event Simulation, represent a good deal between these both families, althought strong engineering expertise remains necessary to define the relevant tests and scenarios. This paper proposes an approach to build this kind of formalisms through the example of DEv-PROMELA, which is built by combining DEVS formalism and PROMELA language. Then, it shows how combined formalisms can be used for Modelling, Veryfing and Validating complex software like video games by using both formal-based and simulation-based Verification and Validation.

Keywords

Model Checking, Verification and Validation, Modelling and Simulation, DEVS, PROMELA, DEv-PROMELA

1 Introduction

1.1 Motivation

The need for elaborate techniques for designing and developing systems has been well recognized in recent years. Building reliable systems, hardware or software has become harder, due to the complex behaviours and interactions between their components. This complexity makes harder the understanding of the real behaviours compared to the expected behaviours, and makes harder preventing bugs and defects. Moreover, it is also harder to perform tests directly on the target system, although Verification and Validation (V&V) techniques have deeply changed the last decades [1]. Then, it is well-known that designers must think on representations of systems and software. However, the size of models grows up with the complexity of systems. For this reason, plenty of new V&V concepts, methods and tools have been proposed over the past 20 years to make safer or more efficient V&V of models [2, 3]. These techniques can be classified in two great families.

Formal Verification. On the one hand, Formal Verification (FV), with Formal Methods (FMs) [4, 5], and especially Model Checking (MC) [6–8], represents a family of popular techniques which tend to facilitate a low-level representation of systems under study with automata, and which generally provide execution semantics to make the model more understandable. However, these techniques are generally based on a single specific formal foundation often suitable to represent particular aspects of a system under design [9]. Moreover, a system is generally composed of several components defined by various specifications given from several points of view, which makes their design really tedious. As a consequence, the use of several techniques, tools, notations and formalisms is required to cover a good range of requirements. Engineers are thus enforced to model the same system over and over again, which is expensive and error-prone. Many attempts have been made to integrate several FMs inside a same framework [10, 11] in order to take advantages of their strengths and reduce their weaknesses. In spite of that, the deep problem related to the trade off between speed, accuracy and level of abstraction has not been fully resolved. For example, representing time in the processes of modelling and checking timed systems is always a tough task. An untimed model is generally derived from a timed system [12], but designing a timed system by an untimed model forces to explore behaviours which may not have an interpretation in the real world. Conversely, representing time with accuracy makes the V&V processes heavier because of the growing size of the model [13, 14], and even unreachable in some cases [13, 15]. FMs then need to impose strong constraints to ensure that the models can be verified in finite time. In fact, Model checking techniques

[1]LIS, Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

Corresponding author:
Dr. Aznam Yacoub - Aix Marseille Université - Campus de Saint Jerome - Bat. Polytech - 52 Av. Escadrille Normandie Niemen - 13397 Marseille Cedex 20
Email: aznam.yacoub@lis-lab.fr

and tools (PROMELA, Symbolic Model Checking, etc.) deal with finite, explicit or symbolic, states models to generate and explore all possible paths and to check formally the system properties and assertions [16, 17]. A system with timed behavioural constraints may then be interpreted as refinement of a system without timed constraints. The reason is that at first glance time may be seen as an additional variable in the specifications of the model; this is a misinterpretation. Taking into account the time basis to describe behaviours leads a fortiori to the appearance of new behaviours that do not belong to the untimed specifications (paths to explore from finite state machines).

For instance, consider model checking applied to digital circuits. It is well known that verifications, such as hold violation, flip-flop set up, etc., are based on models in which states are piecewise constant (0 or 1). However, Hamri et al. [18] proposed timed models to design digital circuits based on the boolean algebra and Generalized Discrete Event Specification (GDEVS) [19] formalism showing the appearance of new behaviours (piecewise continuous state functions and new outputs at times when different input signals cross each other) that finite state machines cannot capture. A second example handled in the model checking literature is the Pacman game, as in [20] and [21]. Formal verification of Pacman behaviours is done on an untimed model and on a discretized space. Unfortunately, the proposed formal model for the Pacman game is so abstracted that it does not conform to the final game requirements. For example, the move of entities (pacman and ghosts) depends strongly on time, as shown in [22]. Consequently such paths are uncovered by the formal model.

Modelling and Simulation. On the other hand, the Modelling and Simulation (M&S) theory [23, 24] proposes an elegant and uniform way to understand and design systems. This theory is based on two main separated activities. First, Modelling is the task of making a representation of a system or a software from the point of view of an observer. Modelling answers questions that this observer asks about the system [25]. If this definition meets the one used in FV, the main difference is in the intended purpose of this model. Second, Simulation is then the task of executing the model to generate its behaviour by acting on inputs and parameters of this model [23]. As a consequence, a simulation model is made for validating a particular set of behaviours of a system, while a verification model tries to expose the entire set of computation paths of this system (or at least, the set represented by the model). A simulation model can thus be more complex than a verification model, meaning it can encompass more computational details. A simulation model is thus used when [26, 27]:

- it is impossible or extremely expensive to observe certain processes in the reality, or to interact directly with them;
- the real system has some level of complexity, interaction or interdependence between various components, or pure size that makes it difficult to grasp in its entirety;
- there is no simple analytic model or it is impossible or extremely expensive to validate the mathematical model describing the system.

This assumption introduces the main difference with verification model and simulation model: while a verification model must be as simple as possible in order to be able to generate the entire stateset [28] and to perform verification, a simulation model can deal with the complexity of the interactions between the components by representing what the designer knows about the system, what he is seeing when he stimulates the system, without being constrained by any computational considerations or restrictions. This have an impact on the formalisms used for making simulation models. It is indeed possible to introduce two levels of models: a conceptual model which is the representation of the system under study, and the computerized model (called *simulation* model) [2] which is the realization of the conceptual model. This separation gives its power to simulation formalisms. Indeed, for example, the Discrete Event System Specifications (DEVS) [23] are an algebraic formalism to explicitly represent Discrete-Event Systems (DES), with all considerations about time and data, while the semantics of the abstract simulator gives a unique interpretation to a DEVS model. As a consequence, a DEVS model is then more expressive than an untimed model of a DES. In other words, some verifications and validations can be performed on the conceptual model while some other properties can be verified and validated using the simulation model.

However if this ability to separate conceptual model and simulation model is a strength, it is also a weakness. The power of the simulation-based verification comes from the fact that a simulation is like an empirical experiment. We mean that the simulation is efficient because the played scenarios are well targeted, forcing designers to get more knowledge about the system under study. While the goal of FV techniques is to systematically explore all the behaviours of a system, simulation-based techniques focus on interesting cases. Nevertheless, this also means that simulation-based verification strongly depends on the played scenarios, meaning there is no guarantee that the verification is fully done, like in formal-based verification. In the same manner, simulation is done under a specific set of conditions called the Experimental Frame (EF) [23]. This set of constraints is the same than those under the real system is observed. This adds two difficulties:

- If the hypothesis are incorrect, there is no guarantee about the simulation model, meaning the simulation model could only behave well under the EF, and not under all the non-observed conditions;
- The validation strongly depends on the quality of the implementation; this means the simulation model must be verified and validated against the conceptual model, whose the knowledge depend on the known EF.

Understanding and specifying the context and the EF becomes therefore another challenge [29] in the application of M&S theory. This does not mean the entire state space cannot be checked as with formal verification, but it would be probably more costly; and, 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. This determinism is both a strength and a weakness: from the M&S point of view, this forces again modeller to have precise knowledges about the system; from the FV point of view, this forces the modeller to complexify the model, meaning the model will not focus only on interesting aspects and will encompass irrelevant computational details. Table 1 summarizes the advantages and drawbacks of both formal-based and simulation-based techniques.

1.2 Related work

In order to improve these existing methodologies, some works have been intensively done the last decades on new model-checking algorithms for verification of timed and hybrid models [30, 31], stochastic models [32], improving verification and validation of simulation models [33–35]. Many methodologies combining formal-based verification and simulation like assertion-based approaches [36] have also been proposed. Godefroid [37] says that "model checking can be combined with testing to define a dynamic form of software model checking based on systematic testing", confirming that static and dynamic approaches can be both used for Verification and Validation. Goldberg [38] admits that simulation and formal verification are complementary. The author states that if formal verification proves that a model holds property for all points of a space, the main problem is its unscability. He states also that simulation probes the search space at a subset of points, and "works surprisingly well even though the set of test points (further referred to as the test set) comprises a negligible part of the search space". In fact, combining simulation and verification is an aspect which is included in the definition of model checking [16]. The MC community proposes "testing", "emulation" and "simulation" as a way for analyzing generated counterexamples. In this context, "simulation" (or more precisely execution, or animation of specifications [39]) is similar to run a path of the reachability graph. However, by definition, simulation is "executing a model to generate its behaviour over the time" [24], by acting on inputs and parameters of the model. Errors are determined by finding differences between the simulator output and the output described in the specifications over the time. Then, executing a graph of an untimed model is not really a simulation (in the meaning of the M&S theory) while execution misses two important aspects: the time and the experimental frame.

That is why many other techniques that combines Simulation and Formal Methods have been proposed. Among them, we find methodologies based on morphisms and transformations of models, and techniques that build new formalisms from several other formalisms used in different disciplines. For instance, Abdulhameed et al. [40] propose a methodology to verify and validate SysML specifications by successively translating them into SystemC and UPPAAL models, using a Model-Driven Engineering (MDE) approach. The SystemC model is then simulated for validation purposes, while the UPPAAL model is used for verification purposes. The described methodology was applied on a model of controls of traffic lights. This type of approach is very interesting because they can reach the strength of formal verification and simulation from a common high level specification language. However, the limits of this approach come from the limits of the transformations from SystemC to UPPAAL. Timed Automata (TA) impose some restrictions on time relations and data types. This means that there is no proof that any SysML models can be translated into a SystemC model and an UPPAAL model. In the same manner, Zeigler et al. [41–43] propose to use system morphisms to transform DEVS (resp. model-checking) models into model-checking (resp. DEVS) models.

Dacharry et al. [44] propose another design methodology for control systems by linking simulation using DEVS and formal verification using TA. The high-level specifications are expressed using TA, while the design of the control implementation is described using DEVS formalism. By finding a refinement between the implementation and the specification (i.e. by proving the DEVS model conforms with the TA model), simulation can then be used for validating the implementation. Moreover, a DEVS model is more expressive and enforces fewer constraints about data and time relations. A DEVS model is thus closer to the real system than a verification model. Nevertheless, this approach supposes that it is possible to translate any DEVS models into equivalent TA models, and as the authors stated, this is not possible. Second, because a fully automatic translation from TA to DEVS is not possible, this means that designers must have knowledge of two formalisms to achieve the best of combined simulation and model checking.

He [9] proposes to build a new formalism integrating two other formalisms. On the one hand, PZNets can be seen as an extension of Petri Nets, by adding function definition capabilities; one the other hand, it can be also seen as an extension of Z notations, by adding a new operational semantics with explicit control flow structures. The main advantage of such an approach is that the various aspects of a system can be modelled using a unified formal model which can benefit from a rich set of analysis techniques. Indeed, a PZNet model remains a Petri Net model, meaning that it can be simulated. Moreover, a PZNet model can be analyzed using the Z proof techniques thanks to the defined transformation rules. Thus, a PZNet is suitable for both data and process reasoning.

We proposed [45, 46] to go further in this way and to explore how combining Model-Checking and Simulation for improving V&V by introducing the semantics of simulation formalisms into verification formalisms (at the difference with PZNets which adds Z notations to Petri Nets). In this way, we achieve different goals:

1. Building a unique formalism that allows designing accurate representations of software which before were not fully representable with only one formalism;
2. Keep a model with a clear syntax that focuses on the properties which one wants to verify, without losing the timed semantics;
3. Keep a separation between the conceptual model (which is the verification model) and the simulation model (which is the computerized model);

Many advantages arise from such a methodology:

Table 1. Advantages and Drawbacks of Formal- and Simulation-based V&V

	Advantages	Drawbacks
Formal-based V&V	Explore the full statespace Non-deterministic model Simple analytic model: model is focused on what to be verified/validated	Related to statespace explosion problem (timed models are hard) Related to decidability problem : models must be finite or symbolically finite
Simulation-based V&V	Separation between conceptual and implemented model Interoperability between simulation formalisms, and between simulators is easier (thanks to the separation between the models) Exsisting universal formalism and framework for modelling a large set of systems Deterministic model (well-known specifications) Non-related to complexity (time, infinite data, computations, etc.)	Strongly depends on EF and scenarios No guarantee that the V&V is fully done V&V depends on the quality of the computerized model (the computerized model must also be verified and validated against the conceptual model)
Combined V&V	Separation between conceptual and implemented model Structural static properties can be formally checking exploring the entire statespace Behavioural dynamic properties can be checked using simulation Hierarchical and modular constructions of the models make easier the analysis Makes easier the interoperability between existing tools	Cannot represent all existing systems (the model remains symbolically finite) Remains strongly dependant on experts, requirements, scenarios (parameter space can be fully explored, but it would take huge amount of time)

- A robust V&V of models and systems can be achieved through a method based on the strengths of formal verification and simulation. In this approach, simulation and formal verification complete each other in the V&V processes. Model-Checking is used for verifying and validating some static properties like bounding values, structural deadlocks etc. in an untimed mode, while simulation is used for checking dynamic properties, behavioural deadlocks etc. in a timed mode;
- Simulation formalisms can benefit from a clear syntax that makes easier the implementation of computerized models;
  Unlike timed formal verification, the size of the statespace is subdued, and the expressiveness of the timed model is still preserved;
- What which is checked, verified and validated is more clear: conceptual model, computerized model and real software are clearly separated in this approach.
- The hierarchical construction reduces the complexity of the formal model, and the relevant parameter space is better targeted (analysis of the relevent properties makes easier the choice of the subspace in which the model is tested).

As a result of our work is born the Discrete-Event PROMELA (DEv-PROMELA) formalism [47] which can be seen as a new specification formalism thanks to the combination of DEVS and Process MEta LAnguage (PROMELA) [48] formalisms presented in section 1.3 and section 1.4.

1.3 PROMELA Concepts

PROMELA [48] is a formal verifiable language which allows the specification of concurrent systems and concurrent protocols. The model-checker SPIN makes possible of the validation of properties expressed in Linear Temporal Logic (LTL). We briefly introduces there the main concepts behind PROMELA.

PROMELA primitives. A PROMELA system [28] relies on three main types of objects: *processes*, *data objects* and *messages*.

The components of the system are modelled by a *finite* set of *instances* of *processes*. The latter can communicate with each other thanks to different mechanisms such as *buffered messages*, *shared global variables* or *rendez-vous handshakes*. Each process is a *finite* set of guarded or labelled commands called *instructions*. Each instruction is sequentially executed by each process in an either *synchronous* or *interleaved asynchronous* manner. In other words, at any time t_i, only one instruction is performed by one of the processes, without any assumptions about duration of the execution or timed events. Note that a set of instructions can be labelled as an atomic instruction: in this case, these instructions are considered as a unique instruction. Processes can also be prioritized, meaning that a process with a higher priority will always execute its instructions before other processes (it is interesting to note that the semantics of priority has changed: old PROMELA specifications provided a way to define a ratio between processes, meaning that, for instance, a 10-priority process was 10 times more likely to execute before others).

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:    :: ( y == 4 ) → z = 0;
9:    fi;
10: }
11:
12: active proctype B {
13:   int x = 2, y;
14:   do
15:   :: ( z == 1 ) → x = 2;
16:   :: ( x == 2 ) → y = 4;
17:   :: ( y == 4 ) → z = 0;
18:   od;
19: }
20:
21: ltl {[](z == 1);}
```

Syntactically, a *process* is defined by a `proctype` block of instructions, as given in Program 1.

Instructions are divided into two categories:

- Statements that modify the state of the model by acting on variables: assignments and communication statements. Assignments are always enabled, meaning there can always be executed, while communication statements can be blocking, depending on the size of the involved buffers;
- Statements that act on the flow: selection statements which choose the next state among different branches regarding a guard. If many guards are satisfied, one of them is randomly selected. For instance, in 1, if `x == 2` and `y == 2`, both l.6 and l.7 are enabled. This corresponds to two active paths in the verification graph. If no statement can be satisfied, the control-flow is blocked, meaning there is no next state in the reachability graph.

PROMELA data are represented by classic variables, with a type and an identifier. The type gives the finite size of the variable (Table 2). It can be either scalar values, combinations of scalar values (structs), or finite arrays of scalar and/or structs. Variables can be local, meaning there are defined only in the scope of the process that declares them, or global, meaning they are shared by all the processes.

Table 2. A list of PROMELA basic datatypes.

Type	Size (bits)	Value Range
`bit`, `bool`	1	$[0; 1]$
`byte`	8	$[0; 255]$
`mtype` (constants)	8	$[0; 255]$
`short`	16	$[-2^{15}; 2^{15} - 1]$
`int`	32	$[-2^{31}; 2^{31} - 1]$

Semantics of PROMELA. The semantics of a PROMELA model is given by the verification engine [28, 49]. Each `proctype` defines a finite state automaton $A = (S, T, L, s_0, F_A)$ where:

- S is the set of states that correspond to the possible control points inside the `proctype` block;
- T is the transition system that defines the control flow;
- L is the transition label function that links each transition to a statement that defines the executability and the effect;
- F is the set of final states which are defined for end-state, accept-state and progress-state.

More operationally, a PROMELA process is a tuple

$$P =< pid, lvars, lstates, initial, curstate, trans >$$

where:

- pid is a positive value which identify the process;
- $lvars$ is a finite set of local variables $\{(name, scope, domain, inival, curval)\}$;
- $lstates \subseteq INT$, which defines the identifiers of the local states of the process; $lstates$ hold no information;
- $initial$ is the initial state of the process such that

$$curstate = initial \implies \forall v \in lvars, v.curval = v.inival;$$

- $curstate$ is the current state of the process.
- $trans$ is a the finite set of transitions $\{(tr\_id, source, target, cond, effect, prty, rv)\}$ where $(source, target) \in lstates \times lstates$.

Then, to define a whole PROMELA program, the concept of *System state* is introduced. A system state is a tuple

$$SS =< gvars, procs, chans, exclusive, handshake, timeout, else, slutter >$$

where, in particularly,

- $gvars$ is a finite set of global variables $\{(name, scope, domain, inival, curval)\}$;
- $procs$ is the finite set of processes;
- $chans$ is the finite set of channels.

The semantics engine (Algorithm 2) gives then the meaning of a PROMELA program. Given a current system state s, the semantics engine takes randomly one executable transition from any executable transitions among all the processes. Given a selected couple (p, t), the engine computes the effect of the transition on the current state. If the statement is not a synchronization statement, the engine modifies the current system state and the current state of the selected process. Otherwise, the algorithm tries to find the process that must fulfill the synchronization. If one is found, both processes are updated and the current global state is changed. Otherwise, the system is blocked.

Then, we can consider a PROMELA program also as a finite state automaton in which the set of states is the cartesian product of each process' set of states, and the

ALGORITHM 2: PROMELA Semantics Engine [48]

while there is at least one executable transition from the current global state s
for one $(p,t) \in procs \times p.trans$ {
  $s' = apply(t.effect, s)$
  if $handshake == 0$
  {
  $s = s'$
  $p.currstate = t.target$
  }
  else
  {
  $E' = executable(s')$
  for one $(p',t') \in E'$ {
    $s = apply(t'.effect, s')$
    $p.currstate = t.target$
    $p'.currstate = t'.target$
  }
  $handshake = 0$
  }
}

transition system is a composition of each transition system of each process. We mean that, for two global state s and s', it exists a transition between them only if it exists a local transition that affects one of the local states that composes s and gives its equivalent component in s'.

1.4 DEVS Concepts

As introduced previously, DEVS is an algebraic formalism which allows discrete-event representations of system. In this section, we recall important concepts about Classic DEVS [24].

DEVS primitives. 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. To model such systems and their analysis, Zeigler et al. [23] introduced the DEVS formalism, which can be seen as a generalization of the Moore Machine formalism by associating each state with a lifespan. The Classic 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 $[0; +\infty]$. 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 characterized 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 prioritized;
- The states, 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;

A DEVS model is divided into small pieces called *DEVS atomic model*, and more complex models called *DEVS coupled model*. A coupled model is a coupling of DEVS atomic or coupled model. Each atomic model deals with events that it receives from the environment, or from other models, in order to change its current state. An atomic model thus encompasses the computational properties of the system. When a transition is enabled, an atomic model can emit an event to the other models which it is coupled with. If it is the case, the target models change their respective state according to the emitted event. A coupled model is thus a way to hierarchically compose complex systems.

To each model, the DEVS framework associates either an *abstract simulator* or an *abstract coordinator* in order to build the *simulation* model. The simulator gives the meaning of an atomic model, while the coordinator sets how the events are exchanged between coupled models. The computerized model is then the concrete implementation of simulators and coordinators using a programming language.

Semantics of DEVS. More formally, a DEVS model is thus a coupling of DEVS 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;
- $\delta_{int} : S \to S$ is the internal transition function;
- $\delta_{ext} : Q \times X \to S$ is the external transition function;
- $\lambda : S \to Y$ is the output function;
- $ta : S \to \mathbb{R}^+$ 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.

Then, a DEVS coupled model is defined by

$$M = (X, Y, M, EIC, EOC, IC, Select)$$

where:

- X is the set of input values;
- Y is the set of output values;
- M is the set of components (atomic or coupled models);
- EIC is the external input coupling that connects external inputs to component inputs;
- EOC is the external output coupling that connects component outputs to external outputs;
- IC is the internal coupling that connects component outputs to component inputs (without direct feedback loops);

- Select is the tie-breaking function that chooses the next event from the set of simultaneous events.

The meaning of an atomic DEVS, given by the abstract simulator, 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 $\delta_{ext}(q, x)$.

Given a queue of events sorted by date, the simulation algorithm works as follow:

1. When a coordinator associated to a coupled model processes an internal event, it dispatches it to its imminent child, ie. the one responsible of this event. The simulator associated to this model executes its transition. If it is a coordinator, the event is processed recursively. Otherwise, if it is a simulator, an output event is emitted to the parent coordinator before changing the state of the corresponding atomic model. When the internal event is consummed and all external events caused by this output processed, the time of the next event is computed by choosing the minimum of the next events of all the children;
2. If a coordinator receives an external input event from its parent coordinator, it generates a message to its internal components according to the internal coupling function. Time event are updated.
3. If a coordinator receives an external output event from its imminent child, it may generates an external output event to its parent coordinator according to the external couplung function or internal coupling function;
4. The algorithm is repeated until the event queue is empty.

Based on these two formalisms, we developped DEv-PROMELA presented in this paper as new formalism, and as an illustration of our proposed methodology. The question about the exact position of DEv-PROMELA as an extension of PROMELA or a subclass of DEVS is not raised in this paper. Section 2 reintroduces DEv-PROMELA and the mandatory key concepts to build a combined formalism based both on formal method and simulation in the context of Software Verification and Validation. Unlike previous work, this section shows also that the resulting formalism is usable both by the former checker and the former simulator because of the existance of morphisms that generate equivalent PROMELA specifications and DEVS models from a DEv-PROMELA specification. Through a double bi-simulation relationships, we ensure that DEv-PROMELA models, resulting PROMELA models and resulting DEVS models are equivalent. Then, Section 3 introduces a new unpublished proposed V&V workflow and Software Development Life Cycle (SDLC) using DEv-PROMELA. Indeed, simulation is generally used to develop systems, but we show that it can be combined with formal checking to discover flaws early in a software development. We applied our methodology to develop a video game using DEv-PROMELA and we show how each step of the SDLC is impacted in our proposed workflow in Section 4. Finally, we discuss about the benefits and drawbacks of our approach and its applicability on more complex software and systems.

2 Discrete-Event PROMELA

2.1 Overview

Discrete-Event PROMELA (DEv-PROMELA) [47] is an example of formalism that illustrates our methodology. DEv-PROMELA can be seen as an extension of PROMELA for describing discrete-event models, while it adds new abstract primitives [50, 51] to PROMELA, or as a new discrete-event simulation formalism. More than that, DEv-PROMELA is also a sublclass of DEVS [23]. Proof of the exact position of DEv-PROMELA is outside of the scope of this paper. As previously said, DEv-PROMELA is built upon a verification language (with an operational semantics) in which we add the operational semantics [52, 53] of a simulation language. In other words, the structure of a DEv-PROMELA model can be fully described/abstracted by a PROMELA model (the state-transition graph generated by the semantics of the PROMELA verification engine), while its discrete-event behaviour is described by a DEVS model (the way of going from a state to another given by the semantics of the DEVS abstract simulator). As a consequence,

- a DEv-PROMELA model can be abstracted to a PROMELA model in which the structure is preserved. This means that some qualitative properties like the intrisic existance of a path between two abstract states can be formally checked; time is seen as ordered events and the size of the statespace of models can be subdued (compared to the statespace of a timed model);
- a DEv-PROMELA model can be abstracted to a DEVS model in which the timed behaviour is preserved. This means that some quantitative properties, like properties in which the next state depends on the time elapsed in the current state, can be checked by simulation; time is seen as timed events and discrete-event simulation models can be expressed with a clear syntatic language.

Shorter, DEv-PROMELA is thus a formal language with the syntax of PROMELA. It embbeds both PROMELA and DEVS primitives. The semantics of the simulation model is given by the DEVS abstract simulator.

2.2 Syntax of DEv-PROMELA

DEv-PROMELA is then designed as an extension of PROMELA for the modelling of discrete-event systems. However, in order to model the previously described DEVS primitives, new syntactic elements are obviously needed. PROMELA is a deep and interesting specification language with plenty of elements. Thus, we just present in this section the minimal main modifications that will allow the modelling of discrete-event systems.

A new datatype. For representing infinite and unbounded real values, we introduce a new abstract type called **real**. It is useful for modelling time and infinite data. As any other

PROMELA types, `real` can be used to defined local and global variables.

```
real i, j, k;
```

Real variables can also be used in structures and in arrays, without restriction.

Statements. PROMELA statements define the actions which are done when the system changes its state following a transition. DEv-PROMELA extends them by prefixing each of them with *an event descriptor*, which allows their characterization, i.e. if they are autonomous or reactive statements.

Event descriptors describe the delay between the execution of any previous statement and the prefixed one, or describe an event which will trigger the execution of this statement. Event descriptors are defined as follows:

```
<event stmnt> ::= "[" <timed trans> "]"
<stmnt> | <stmnt>
<timed trans> ::= <clt expr> | <evt
expr> | <clt expr> <op> <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 */
```

Consider the following examples:

1. **[clt**: 3.0 → **emit:newa]** `a = a + 10;`
2. **[evt:newb]** `b = a - b;`
3. **[clt:lifespan(c)** → **emit:newc | evt:newd]** `c = c * d;`

(1) means that the execution of `a = a + 10` is performed 3.0 units of time after the execution of a previous statement. Before executing this statement, an event `newa` is emitted.

(2) means that the statement will be triggered only if the event "newb" is received.

(3) means that the statement `c = c * d` will be triggered either if the elapsed time between the execution of a previous statement and this one is equal to the value lifespan(c) (in this case, "newc" will be outputed) or, if the the event "newd" occurs.

The third example shows one of the main characteristics of DEv-PROMELA: a statement can be executed in different manners, with at most one explicit timed descriptor (defined by `clt` command) and with at most one descriptor per event. Note that the `clt` command is optional. If it is not defined and there is at least one `evt` command, we consider the elapsed time before the execution of the statement is equal to ∞. For convenience, if there is no event descriptor (no `clt` command nor `evt` command), the statement is interpreted as if it is prefixed by **[clt**: 0.0 → **emit:silent]**. The statement is executed without any delay by emitting the default **silent** event. The silent event is an event which does not cause any explicit change in the system, but defined in order to conform to the DEVS formalism.

Selection construct. The selection construct is a control-flow construct that helps to define the structure of the automaton corresponding to the PROMELA program. The original construct has a unique start and stop state. Each option sequence defines an outgoing transition from the start state to the stop state. Thus, the end of each option sequence leads to the end state that follows the construct. Consider the example given in Algorithm 3. The PROMELA resulting structure is composed by three states per path, as shown in Figure 1. When entering the selection construct, the process evaluates each guard, leading to an intermediate state, before executing the option. This mechanism allows PROMELA to separate evaluation of guard and outgoing instructions. If two options are evaluated to true, both paths can be executed in a non-deterministic manner.

ALGORITHM 3: PROMELA conditional structure.

```
1: if
2: :: ( x == 2 ) → x = 3;
3: :: ( y == 2 ) → y = 4;
4: fi;
```

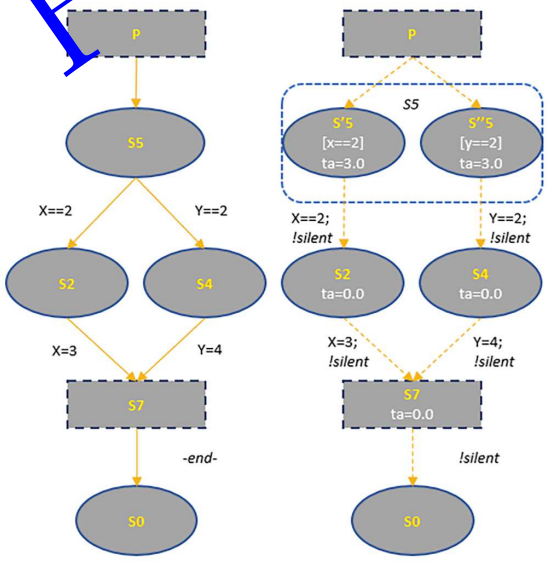

Figure 1. Structure generated by Algorithm 3 (on the left) and Algorithm 4 (on the right).

In a DEv-PROMELA model, such a construct defines transitions between a set of *equivalent* states. The meaning is that, whatever the state for which $x = 2$, only the transition defined by the first option is defined (except for the state $(x = 2, y = 2)$). This transition leads to a state for which $x = 3$. In this case, this means that each option of the construct describes a set of source states verifying at least the guard of the option. Thus, like for assignments, each DEv-PROMELA option is prefixed by an event descriptor. Then, a DEv-PROMELA version of Algorithm 3 is given in Algorithm 4.

ALGORITHM 4: DEv-PROMELA conditional structure.

```
1: if
2:   :: [clt: 3.0 → emit:silent] ( x == 2 ) → [clt: 0.0 → emit:newx] x = 3;
3:   :: [clt: 1.3 → emit:silent] ( y == 2 ) → y = 4;
4: fi;
```

Note that the event descriptor can be placed before the control structure. In this case, this means that all evaluation of each guard will occur after the same delay.

But what happens if multiple guards are verified ? Traditional PROMELA allows non-deterministic behaviours, because the model-checker will systematically explore all possibilities. Thus, for a source state, multiple outgoing transitions can exist. Still, a discrete-event simulation model is deterministic by definition. This could be seen as a restriction, but in fact, it forces designers to fully specify the modelled system. Indeed, what does the system actually do if $x = 2$ and $y = 2$? DEv-PROMELA considers that the options are ordered by time then by order. For example, in Algorithm 4, only the second option is executed. If the time was the same for both options, the first one would be executed.

Repetition construct. The repetetion construct allows the definition of loops. The structure is likely similar to the selection construct, except that the end of each option leads back to the start of the structure. Only the option associated to a `break` statement leads to the end of the structure. As in the case of a conditional structure, the option can be prefixed by an event descriptor.

ALGORITHM 5: DEv-PROMELA loop structure.

```
1: do
2:   :: [clt: 3.0 → emit:newx] ( x == 3 ) → y++;
3:   :: [clt: 1.3 → emit:newy] ( y == 2 ) → break;
4: od;
```

Consider Algorithm 5. If the program is in a state $(x = 3, y = 1)$, the first transition will be triggered after 3.0 units of time, leading to a new state $(x = 3, y = 2)$. The next instruction `y == 2` is then executed after a delay of 1.3 units of time.

Process priority. Priority between processes is another thing that we need to be able to define. Indeed, if two events occur at the same time, we need to know what event must be processed first. This corresponds to the *select* tie-breaking function seen previously. For that, DEv-PROMELA defines a process descriptor using the following grammar:

```
<proctype decl> ::= "[" priority "=" <int> "]" <proctype>
```

The semantics of priority will be defined in the next section.

Clock and timeout. The last syntactic element concerns time handling. PROMELA defines a **timeout** keyword used as an escape for a blocked system (i.e. a system for which there is no more enabled statement), for example when a system has no valid option to progress through a selection construct. In DEv-PROMELA, such a case means that the system is not well specified. In such a case, simulation cannot be performed and will return an error.

Each process is also associated to a virtual local clock which measures the elapsed time since the last event. DEv-PROMELA allows transitions depending on the elpased time. Two convenient instructions, **getElapsedTime** and **getCurrentDate**, enable access to this clock valuation.

2.3 Meaning of DEv-PROMELA

DEv-PROMELA primitives. To keep the equivalence between PROMELA, DEVS and DEv-PROMELA, we associate for each DEv-PROMELA `proctype` block a DEv-PROMELA atomic process. Each atomic process is coupled with the other processes that compose the specifications. The whole system is then viewed as a DEv-PROMELA coupled program. Programs can be coupled to make a more complex program or interact with the environment.

Semantics of a DEv-PROMELA program. Formally, 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. This definition comes from the fact that a DEv-PROMELA program can be simulated by a DEv-PROMELA atomic process. The demonstration is similar to the closure under coupling property 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;
- $\delta_i : Q_f \rightarrow Q_0 \times E$ is the internal transition partial function;
- $\delta_e : Q \times E \rightarrow Q$ is the external transition partial function;
- s_0 is the initial state;
- F is the set of final states.

Moreover, we define:

- $ta : \begin{cases} S_\tau \to \mathbb{R}^+ \\ s \mapsto t_s \end{cases}$ 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 statment 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 $ta(s) = t_s$ where dt is the elpased time from the previous event.

2.4 Relations between DEv-PROMELA and DEVS

The main goal of DEv-PROMELA is to provide another way to enhance modelling, verification and validation of discrete-event systems by using simulation and model checking. To do that, we must demonstrate that a DEv-PROMELA model can be simulated and verified. The first relation can be easily demonstrated by showing that, for a given DEv-PROMELA model, it exists a DEVS model that simulates it.

Proposition 1. *A DEv-PROMELA atomic process P is a DEVS atomic model.*

This demonstration is relatively easy, thanks to the construction of DEv-PROMELA. Consider a DEv-PROMELA process $P = (S, E, \delta_i, \delta_e, s_0, F)$ and a DEVS atomic model $A = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$. P will define the same system as A if and only if :

1. $S_\tau = S$, both models have the same state space;
2. $X \subseteq E$ and $Y \subseteq E$; the sets of inputs and outputs are subsets of the set of events;
3. given s and s' two states such that $s' = \delta_{int}(s)$ and $y = \lambda(s)$; then, it exists s_τ and s'_τ such that $((s'_\tau, 0), y) = \delta_i((s_\tau, ta(s_\tau)))$ and $ta(s) = ta(s_\tau)$;
4. given s and s' two states, and x an input such that $\delta_{ext}(s, x) = s'$; then it exists s_τ and s'_τ such that $s'_\tau = \delta_e(s_\tau, x)$;

Proof. Considering a DEVS atomic model built upon a DEv-PROMELA process model, in which $X = E$, $Y = E$, and $S = S_\tau$. We define δ_{int} and λ such that:

- if $ta(s) \neq \infty$ and $\delta_i(q) = (q', e)$ such that $q = (s, ta(s))$ and $q' = (s', 0)$, then $\delta_{int}(s) = s'$ and $\lambda(s) = e$.
- if $ta(s) = \infty$, $\delta_{int}(s) = s$ and $\lambda(s) = \varnothing$. s is a passive state, then this transition will never be enabled.

We define δ_{ext} as follows:

- if $\delta_e(q, e) = q'$ with $q = (s, dt)$ and $q' = (s', dt')$, then $\delta_{ext}(q, e) = s'$;
- if $q = q'$ and $ta(s) \neq \infty$, then $ta(s') = ta(s) - dt$ and $q' = (s', 0)$. This condition ensures that δ_{ext} is defined for all $(q, e) \in Q \times X$ and time is preserved.

Then, we can show that A simulates P. Considering the transition system $< S', \Lambda, \to >$ where

- $S' = S_\tau \cup S$;
- $\Lambda = (X \cup E) \times (Y \cup E)$;
- $\to = Im(\delta_i) \cup Im(\delta_{int}) \cup Im(\delta_e) \cup Im(\delta_{ext})$;

where $\cup$ denotes the disjoint union operator. Therefore, A simulates P if there is a simulation $R = S' \times S'$ such that for all $(p, q) \in R$ and $l = (x, y) \in \Lambda$, if

$$p \xrightarrow{l} p'$$

then

$$q \xrightarrow{l} q'$$

However, $p \xrightarrow{l} p'$ only if:

1. $(l', y) = \delta_i(p)$, meaning that p' is reached by an internal transition that outputs y. By construction, we know that it exists $(q, q') \in \to$ such that $q' = \delta_{int}(q)$ and $y = \lambda(q)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.
2. $p' = \delta_{int}(p)$ and $y = \lambda(p)$, meaning that p' is reached by an internal transition that outputs y. By construction, we know that it exists $(q, q') \in \to$ such that $(q', y) = \delta_i(q)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.
3. $p' = \delta_e(p, x)$, meaning that p' is reached by an external transition that consumes x. By construction, we know that it exists $(q, q') \in \to$ such that $q' = \delta_{ext}(q, x)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.
4. $p' = \delta_{ext}(p, x)$, meaning that p' is reached by an external transition that consumes x. By construction, we know that it exists $(q, q') \in \to$ such that $q' = \delta_e(q, x)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.

Thus, A simulates P. Symetrically, we can show that for all $(p', q') \in R$, if

$$q \xrightarrow{l} q'$$

then

$$p \xrightarrow{l} p'$$

Thus, P simulates A, meaning P and A are bisimilar.

We can then build a DEVS atomic model that simulates exactly the behaviour of a DEv-PROMELA process.

Proposition 2. *A DEv-PROMELA program P_r is a DEVS atomic model.*

Proof. Given a DEv-PROMELA program P_r with n processes P_1 to P_n. Given $EVENT$, a convenience function such that $EVENT(P_n)$ is the set of events of P_n. Then, we can define a DEVS atomic model $A = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$ which simulates the DEv-PROMELA program:

1. $S_\tau = \cup_{i=1}^{n} S_i \times \prod_{j=1}^{n} G_j \times \prod_{k=1}^{o} C_k$, where S_i are the sets of the states of each process, G_j the sets of the values of the global variables, and C_k the sets of the values of the channels;
2. $X \subseteq \cup^n EVENT(P_n)$ and $Y \subseteq \cup^n EVENT(P_n)$; the sets of inputs and outputs are subsets of the set of events;
3. $\delta_{int} : S_\tau \to S_\tau$. Given $s = (s_1, s_2, ...)$ and $s' = (s'_1, ...)$ and $s'' = (..., s'_2, ...)$ in S_τ. Then:
   - if $\delta_{i_1}(s_1, ta(s_1)) = (s'_1, 0, e)$ where δ_{i_1} is the internal transition function of the process 1, then $\delta_i(s) = s'$;
   - if $\delta_{int}(s) = s'$ and $\lambda_1(s_1) = e$, and $\delta_{e_2}(s_2, dt, e) = (s'_2, dt')$, then $\delta_{int}(s') = s''$ and $ta(s') = 0$; this case describes the internal coupling of DEv-PROMELA processes ;
4. $ta : S_\tau \to \mathbb{R}$. $s = (s_1, s_2, ...)$ and $s' = (s'_1, ...)$, then:
   - if s is the initial state, $ta(s) = min(ta(s_1), ..., ta(s_n))$;
   - if $s' = \delta_{int}(s)$, then $ta(s') = min(ta(s'_1), ta(s_2) - ta(s_1), ..., ta(s_n) - ta(s_1))$;
   - if $s' = \delta_{ext}(q, e)$ and $q = (s, dt)$, then $ta(s') = min(ta(s'_1), ta(s_2) - dt, ..., ta(s_n) - dt))$;
5. $\delta_{ext} : Q \times X \to S_\tau$; Given $s = (s_1, s_2, ...)$ and $s' = (s'_1, ...)$ and $s'' = (..., s'_2, ...)$ in S_τ. If $\delta_{e_1}(s_1, dt, x) = (s'_1, dt')$, then $\delta_{ext}(s, e) = s'$;
6. given $s = (s_1, ...)$, $\lambda(s) = y$ if $\lambda(s_1) = y$;

We must show that A simulates P_r. We denote by $STATESPACE(P_r)$ the total statespace of the DEv-PROMELA program P_r. If A simulates P_r, this means that for each $(p, p') \in STATESPACE(P_r)$ such that $p \to p'$, it exists $(q, q') \in Q$ such that it exists an internal or an external transition to go from q to q'. But, $p \to p'$ if:

1. $p = (s_{p_i}, ..)$, $p' = (s_{p'_i}, ..)$ and $\to = (\delta_{i_p}, ...)$ such that $(s_{p'_i}, y) = \delta_i(s_{p_i})$; by construction, we know that it exists $\delta_{int}(q) = q'$ and $y = \lambda(q)$ that corresponds to the transition $\delta_i(s_{p_i})$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$. Indeed, the next event in P_r is generated by the minimum value of all the future events. And by definition, $ta(q') = min(ta(s'_1), ta(s_2) - ta(s_1), ..., ta(s_n) - ta(s_1))$.
2. $p = (s_{p_i}, ..)$, $p' = (s_{p'_i}, ..)$ and $\to = (\delta_{e_p}, ...)$ such that $(s_{p'_i}) = \delta_e(s_{p_i}, x)$ where x is an internal event generated by any other process of the system. This transition is enabled before the internal transition that has emitted the event is triggered. However, by construction, we know that it exists $\delta_{int}(q) = q'$ and $\varnothing = \lambda(q)$ that corresponds to the transition $\delta_e(s_{p_i})$, and $ta(q) = 0$.
3. $p = (s_{p_i}, ..)$, $p' = (s_{p'_i}, ..)$ and $\to = (\delta_{e_p}, ...)$ such that $(s_{p'_i}) = \delta_e(s_{p_i}, x)$ where x is an external event received by the system. However, by construction, we know that it exists $\delta_{ext}(q, x) = q'$ that corresponds to the transition $\delta_e(s_{p_i}, x)$.

Thus, A simulates P_r.

We can then define a DEVS atomic model which simulates exactly the behaviour of a DEv-PROMELA program. It is interesting to note that in the case of a DEv-PROMELA program without global variables and channels, we can build a DEVS coupled model that simulates the DEv-PROMELA specifications. In that case, the property of closure under coupling gives exactly the DEVS atomic model described above, and in this case, the coupled model is similar to the DEv-PROMELA program.

Proposition 3. *A DEv-PROMELA program P_r is legitimate if the DEVS equivalent model is legitimate.*

Because a DEv-PROMELA program can be simulated by a DEVS model, we can deduce all the properties of the program from the DEVS model. Particularly, a DEv-PROMELA program is legitimate if the DEVS equivalent model is legitimate. For example, if the DEVS model goes into an infinite loop of internal events where time does not advance beyond a certain point, we can deduce that the DEv-PROMELA program has the same behaviour.

2.5 Relation between DEv-PROMELA and PROMELA

Checking a DEv-PROMELA model is possible only if we can at least find an equivalent PROMELA model, meaning the structure expressed by the DEv-PROMELA model can be abstracted to a PROMELA model. We must then prove that it exists at least one PROMELA model which is an abstraction of the given DEv-PROMELA model. We can do that by using the pre-order simulation relationship between models.

Proposition 4. *Given a DEv-PROMELA process model P, it exists a PROMELA process model P' that preserves the structural properties of P.*

Consider a DEv-PROMELA process model P. We get a PROMELA process model P' by removing all the event descriptors and abstracting real data from P. P and P' are two state-transition systems, whose respective entire statespace is denoted S and S'. Thus, P' preserves the structure of P if and only if

$$\forall (s, s') \in S \times S, \exists (t, t') \in S' \times S',$$
$$\delta_i(s) = (s', e) \vee \delta_e(s, e) = s' \Rightarrow t \xrightarrow{l} t'$$

where l is a statement. Look at each type of statement defined previously.

Assignment. A DEv-PROMELA assignment is a statement l with an event descriptor ev that defines one or several transitions between two states s and s'. A PROMELA assignment is a statement l that defines only one transition between two states t and t'. Then, if P' is obtained by removing the event descriptor ev, it exists a $(t, t') \in S' \times S'$ such that $t \xrightarrow{l} t'$.

Selection and repetition constructs. A DEv-PROMELA selection (repetition) construct defines transitions between subsets of S. Given $S_a \subset S$ and $S_b \subset S$ such that S_a verifies a guard, and S_b is the subset of end states related to the selected option. This means $\forall s_a \in S_a, \exists s_b \in S_b, \delta_i(s_a) = (s_b, e) \vee \delta_e(s_a) = (s_b, e)$ by executing the option. Then, if P' is obtained by removing the event descriptor, each couple (s_a, s_b) can be mapped to a (t_a, t_b) such that $t_a \xrightarrow{l} t_b$. Moreover, we can be sure it exists at least one such couple because PROMELA allows non-deterministic behaviours.

Channels. The mechanism of channels is exactly preserved in DEv-PROMELA. Sending and receiving operations link two states s and s'. Only the lifespan and the meaning of transitions are changed. Thus, removing the event descriptor preserves the link between the states.

Definition 1. *We call autonomous instance of a DEv-PROMELA process P any parametrization of the lifespans of states such that the model contains no passive state.*

DEv-PROMELA model allows modelling systems whose next states depend on the time elapsed in the current state. This behaviour can obviously, for example, lead to deadlock in passive state. Because PROMELA does an abstraction of time, these kinds of behaviours cannot be captured or modelled. However, a PROMELA model will be a good abstraction if it covers at least all the parametrizations of the DEv-PROMELA model containing no passive state. In this case, the PROMELA model simulates all the autonomous instances of the DEv-PROMELA model.

Proposition 5. *Given a DEv-PROMELA process model P, and a PROMELA process model P' obtained by removing all the event descriptors. Then, P' simulates all autonomous instances of P.*

As demonstrated, P' preserves all the structural properties of P. The set of autonomous instances of P contains all the possible orders of events. Because P is a process, only a change in conditional/loop structures can lead to different behaviours between instances. However, the PROMELA model P' contains all the possible paths for these structures. As a consequence, P' simulates all the autonomous instances of P.

Proposition 6. *A PROMELA program P'_r got from a DEv-PROMELA program P_r by removing all event descriptors preserves the structural properties of P_r. Moreover, P'_r simulates all autonomous instances of P_r.*

A global state of a DEv-PROMELA program is the cartesian product of the set of states of each process. Thus, at a time t, the next event (and the next statement) is selected by taking the minimum value of the date of the next event of each process. This means that the statespace represented by all autonomous instances contains all the possible permutations between statements. This is exactly the executing graph of the PROMELA model P'_r. Then, it exists a PROMELA model that is an abstraction of a DEv-PROMELA model. This model is obtained by only removing the event descriptors from the source model. Moreover, we can say that the symbolic DEv-PROMELA model simulates the PROMELA model.

2.6 Nature of DEv-PROMELA

As a consequence of the two relationships presented above, the DEv-PROMELA specifications embbed two representations of timed properties and two points of view of a same transition system. As a DEVS simulation model, the DEv-PROMELA model considers time as quantitative (events are considered as timed events). As a PROMELA verification model, it focuses on time as qualitative (events are considered as ordered events). One important question can be raised from this construction: is DEv-PROMELA an extension of PROMELA ? Is DEv-PROMELA a new formalism ? Is DEv-PROMELA an extension of DEVS ?

DEv-PROMELA is a formalism based on the PROMELA formalism for its syntaxic part, and on DEVS for its semantics. While the syntax defines the macro-level of the underlying automaton, the semantics gives details on how transition are triggered at the micro-level. This means that the resulting PROMELA model generated from a DEv-PROMELA model is more abstract than the second one, and contains less timed information. Therefore, purely reasonning on a DEv-PROMELA model is possible, and refining a PROMELA model to obtain a DEv-PROMELA model is easily possible by annotating the former model. However, to prove that DEv-PROMELA is an extension of PROMELA, we need to prove that any PROMELA models can be encompassed in a DEv-PROMELA model. This demonstration is out of the scope of this paper.

On the other hand, a DEv-PROMELA model is strictly equivalent to a DEVS model as shown previously. This means that there is always a DEVS model which exactly behaves as a DEv-PROMELA model. However, we know that all DEVS models in general cannot be represented by a DEv-PROMELA model, because a DEv-PROMELA model is structurally finite (or at least, it exists a finite symbolic DEVS to represent it, due to the structural equivalence with PROMELA). As a result, we can hypothesize that DEv-PROMELA is a subclass of DEVS [54].

2.7 Closure under Coupling

Closure under coupling is an important aspect in hierarchical construction because it ensures that the formalism is well-defined and enables checking for the correctness of coupled models [24, 55]. Basically, closure under coupling gives the assurance that the behaviour of a coupled model can be described by an atomic model, which consequently ensures the validity of the hirarchical construction. Therefore, we have to prove that a DEv-PROMELA model obeys the rules stated in [24] and that a DEv-PROMELA coupled model can be expressed itself as a DEv-PROMELA atomic model.

Definition 2. *Given a DEv-PROMELA program $P = (P_0, P_1, ..., P_n, E_p)$ composed by n DEv-PROMELA processes $P_j = (S_j, E_j, \delta_{i_j}, \delta_{e_j}, s_{0_j}, F_j)$. We define E_p as the event set accepted by the network.*

Proposition 7. *We can define a DEv-PROMELA process $M_p = (S_M, E_p, \delta_{i_M}, \delta_{e_M}, S_{0_M}, F_M)$ which describes the behaviour of P.*

In this case:

- $E_p = \cup E_j$;
- $S_M = \times S_j$ for all j such as P_j is a component of P;
- $F_M = \times F_j$ for all j such as P_j is a component of P;

We define then the remaining equivalent function δ_{i_M} and δ_{e_M}. Given an event $e \in E_p$ and $q = (q_0, .., q_n)$, we have:

$$\delta_{e_M}(q, e) = (q'_0, ...)$$

with $q'_j = \delta_{e_j}(q_j, e)$, and

$$\delta_{i_M}(q) = (\delta^*_{e_M}(q^*, e^*_j), e^*_j)$$

with $q^* = (q_0, ..., \delta^*_{i_j}(q_j), ..., q_n)$ and e^*_j the event resulting from the imminent transition $\delta^*_{i_j}$. This transition is unique by definition: if there are concurrent imminent transitions, the priority function selects the next executed transition. The result is that

$$\delta^*_{e_M}(q, e) = (q_0, ..., q_n)$$

if e is the silent event or if there is no component which defines a such partial function for the imminent resulting event e, or

$$\delta^*_{e_M}(q, e) = (..., \delta_{e_j}(q_j, e), ...)$$

if there is a component which defines a partial function for the imminent resulting event e (we recall here that all DEv-PROMELA processes are intercoupled).

By this construction, any DEv-PROMELA program can be rewritten and simulated by a DEv-PROMELA process. However, this construction is possible only because we rigourously define an equivalence between the syntaxic structures in DEv-PROMELA and PROMELA. Indeed, transition functions are built on equivalence between the PROMELA state-transition structure and the automaton underlying DEv-PROMELA. Therefore, a rigourous demonstration of the closure under coupling should involve a rigourous demonstration of the above construction for each syntaxic structure of PROMELA.

3 Combined V&V Workflow

3.1 Overview

A DEv-PROMELA model can be thus used for generating both verification and simulation model. However, it is important to know what exactly the DEv-PROMELA model represents. As an extension of PROMELA, DEv-PROMELA specifications can be a conceptual model of a software, especially event-driven software, or more precisely its formal specifications. While the verification model derived from the specifications removes the computational properties, it can be used for checking the flow and communications between processes. This means that the computerized verification model will be generally used for validating flow properties (what we call static and structural properties). This verification model can also be used for inspecting and verifying the future implementation in a symbolic way. An analogy can be thus done with static test techniques [56].

On the other hand, as a subclass of DEVS, DEv-PROMELA specifications can be considered as a conceptual model of the computerized simulation model and as a conceptual model of the final software. This model can be used for checking behavioural properties and evaluating data (what we call dynamic and behavioural properties). In particular, it can be used for validating counterexamples given by the model-checker, or confirming the absence of errors by playing scenarios. An analogy can be thus done with dynamic test techniques [56]. Furthermore, while the simulation model is obtained from the DEv-PROMELA specifications model, it means that the verification model is also formal specifications of the simulation model (even the simulator is abstracted). This is important in the case of V&V of simulation models.

Shorter, DEv-PROMELA specifications are formal specifications and conceptual model of simulators (i.e. the implementation of the simulaton model) and of the final software. Anyway, depending on what kind of properties is checked, simulation and formal verification can be used for both verification or validation purposes. As a formal framework, the DEv-PROMELA specifications can be combined with many traditional techniques for a more robust V&V workflow.

3.2 Software Verification and Validation in Software Life Cycle

The literature of Software Engineering, Software Verification and Validation, and Verification and Validation of Simulation Models proposes many workflows explaining the V&V procedures. The first one (figure 2) shows clearly the difference between verification and validation. Verification processes overcome only during the development phases to check if the computerized program fulfills the development requirements. Validation is the process in which the final user tests the final software and checks if it meets its functional specifications. This means the validation steps are done only on the final software.

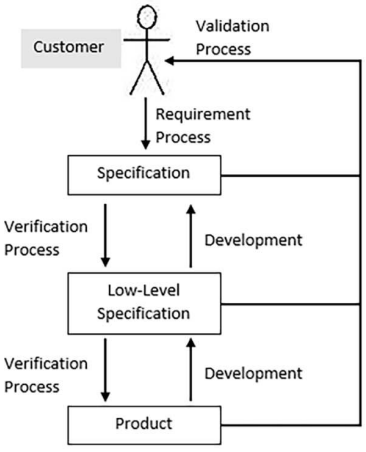

Figure 2. Schema of V&V Process from [57].

Further, if we precisely look at the development cycle (figure 3), we can see that validation is essentialy testing techniques on the final computer program. At each step of

the development cycles corresponds requirements defined at a suitable level of abstraction [58]:

- Unit tests validate each software unit, if they fit for use;
- Integration tests validate each group of combined software unit, if they deliver the expected results;
- System tests (or validation tests) validate the behaviour of the software, in the context of Functional Requirement Specifications and System Requirement Specifications;
- Acceptance tests are formal testing with respect to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

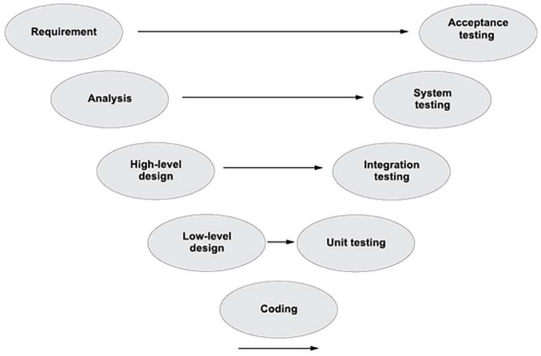

Figure 3. V-model representation from [59].

At each intermediate steps, static verification techniques are also performed (figure 4).

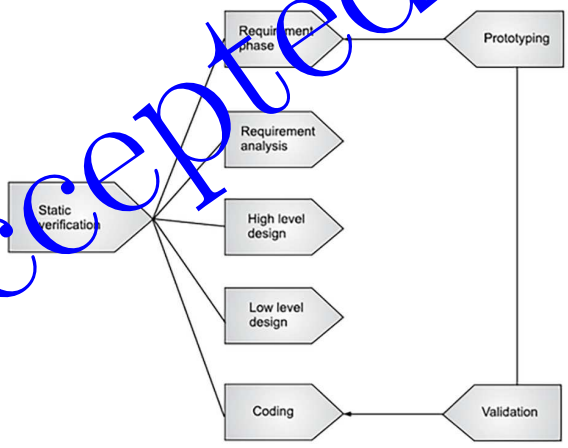

Figure 4. V&V in Software Development Life Cycle proposed by [59].

These techniques evaluate the correctness of the computerized model against the requirements at each step, meaning whether the implementation fulfills the specifications. These techniques essentially concern [60]: code reviews, symbolic execution, data analysis, semantics analysis, etc. As summary, validation a priori concerns only executing the final software, while verification (or testing) is the fact of checking source code, or a model of this code, at each step of the development cycle.

3.3 Verification, Validation and Accreditation of Simulation Models

Verification, Validation and Accreditation of Simulation Models (V&VA) [61] is a specialized V&V procedure applied to simulation models. In fact, simulation models are always developed using observations from an existing system under study or from theoritical assumptions about this system. As a recall, a simulation model is always related to an EF as stated in Section 1. As a consequence, before using a simulation model, ensuring its credibility according to data from real world and from simulation world is necessary. However, a simulation model is also a software program, meaning that Software V&V procedure should also be applied during the development of the simulator.

This feelings is confirmed when we analyze the V&V workflow of simulation models (figure 5).

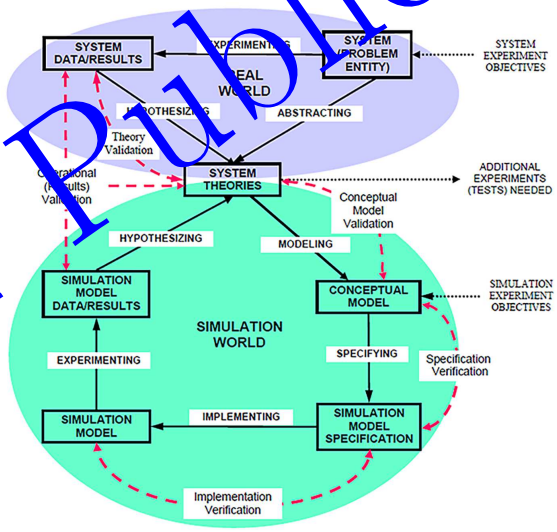

Figure 5. The Sargent Circle for V&V of Simulation Model [2].

In this one, we can clearly see that verification procedures concern only the computerized simulation model and the simulation model specifications. Validation then consists on using a set of test data to ensure the simulator replicates the real system. However, in the same manner than dynamic verification blurs the boundary between verification and validation, we can see that simulation model is indirectly used for validating the conceptual model. Indeed, the simulation model allows inferring new hypothesis on the system theories which is used for building the conceptual model. This means that, if simulation is used on a model of software, it can be used for both verification and validation purposes. In the same manner, while verification is the fact of checking the correctness of an implementation against specifications, model-checking can be used for validating behaviours using an abstraction of a software. This means that model-checking is also a validation activity [62, 63], even if it is performed not on the software itself, but on a model of it.

These notions are very important while designers must exactly know what they are working on. Moreover, DEv-PROMELA adds a new level in the development cycle. For these reasons, we develop a clearer iterative V&V workflow.

3.4 Workflow with Combined V&V

The objective is to make easier the discovery of defects, bugs or flaws in a design, lack of requirements or specifications at early stages of the SDLC, by using combined formal and simulation checking. In the same time, reader shall be conscious that the use of simulation models actually needs an experimental frame which doesn't exist yet in the case of software development. Indeed, our goal is to use simulation in order to help the development of a software. This means that the V&V of the simulation model is performed in a progressive way along to the different development iterations of the target software. That's why we proposed to develop a new combined V&V workflow (Figure 6), based on the V-model and the Sargent's Model. This workflow respects two principles:

- the workflow is defined as a double-iterative cycle in which models and software evolve in alternance.
- the workflow integrates all the steps of the classical V-model: specification phase, analysis phase, design phase, development phase. V&V and evaluation are performed in parallel. If a defect is detected, the next iteration allows fixing the V&V model or the software.

We obtained the V&V workflow proposed in Figure 6 with five important steps.

Specification phase. The specification phase consists on gathering the customer needs and analyzing requirements. Informal specifications are translated into formal specifications, in which technical constraints and formalisms' limitations are added. From requirements and formal specifications, experts can develop a classic software design and translate functional properties in DEv-PROMELA models. This steps allows also experts to design a specific V&V formalism if needed. If it is the case, this formalism should respect the properties of the combined formalisms in order to ensure the resulting models can be both verifiable and simulable. For example, we suppose that DEv-PROMELA can be extended or refined using the hierarchy of simulation formalisms [64]. From this model, a DEVS conceptual model and PROMELA specifications can be automatically extracted. Acceptance tests are also written during this phase. Simulation scenarios and experimental frame for this specific iteration are also defined during this step.

Design phase. The design phase consists on making the architecture of the global software. This design is based on an event-driven architecture, as DEv-PROMELA gives a support for hierarchical event-driven designs. We mean that the DEv-PROMELA model already gives the designer a rigorous frame around which a robust design can be built. Typically, the designer will add all the elements which have a meaning from the semantic point of view, and which are not modelled in the DEv-PROMELA specifications. For example, this step allows the designer to define classes which represent players and for which a DEv-PROMELA model was written. In parallel, integration tests are written.

Model analysis. During this step, the PROMELA model is automatically converted to a SPIN verification model, and the DEVS conceptual model is translated to a DEVS simulation model. Rigourous Computerized Model verification is performed in order to ensure the computerized models are well implemented for their purpose. This verification can be guaranteed through model transformation processes. A cross-checking is then performed: structural and static properties are formally verified and validated using the verification model; behavioural and dynamic properties are validated using simulation. If the cross-checking produces divergent results, counter-examples are generated, and which allow designers to understand the causes and the outcomes of the faulty design. A neww iteration then begins in order to fix the model. Otherwise, classical model verification and validation are performed using the simulation scenarios and requirements specified in step 1. These two complementary verification and validation ensures that both the conceptual and implemented models acts as intended. It corresponds to the operational validation, theory validation, conceptual model validation of the Sargent's model. Depending on the current iteration, experts can decide to start a new iteration to increase the accuracy of the model if the model is not enough refined.

Software generation and implementation. A software implementation is derivated from the DEVS simulation model, called the program. This program is completed with elements from the design phase and which cannot validated using the model, or because the designers decided not to integrate them in the model (readers must be aware that we still remains exposed to decidability, complexity and state-space explosion problems). As explained before, this program relies naturally on an event-driven architecture while the DEVS simulation model is already an event-driven program. Then, classical software verification is performed against the non-functional specifications defined in step 1.

Software validation and testing phase . This step corresponds to the right branch of the V-model. Software validation is performed using simulations scenarios, integration tests and acceptance tests defined in step 1. Especially, designers can compare results obtained during the simulation and results obtained during the testing phase. If divergence is observed, developers can easily detect if errors comes from implementation or from erroneous design (because the design was already validated during the previous iterations and implementation is based on an automatic code generation). If all tests are successfully passed, an artifact is delivered and a new iteration begins in order to implement new functionnalities.

Next iteration. At the next iteration, the previous program artifact can be used as a refined prototype. This prototype is used to gather new datas and make new assumptions in order to refine the specifications and needs (the system theory in the Sargent's model), refine the simulation itself, and perform a verification and validation of the simulation model. This process is repeated until the development of the software is finished.

As a summary, the combined V&V workflow shows exactly the separation between conceptual model, computerized model and final software. This separation allows

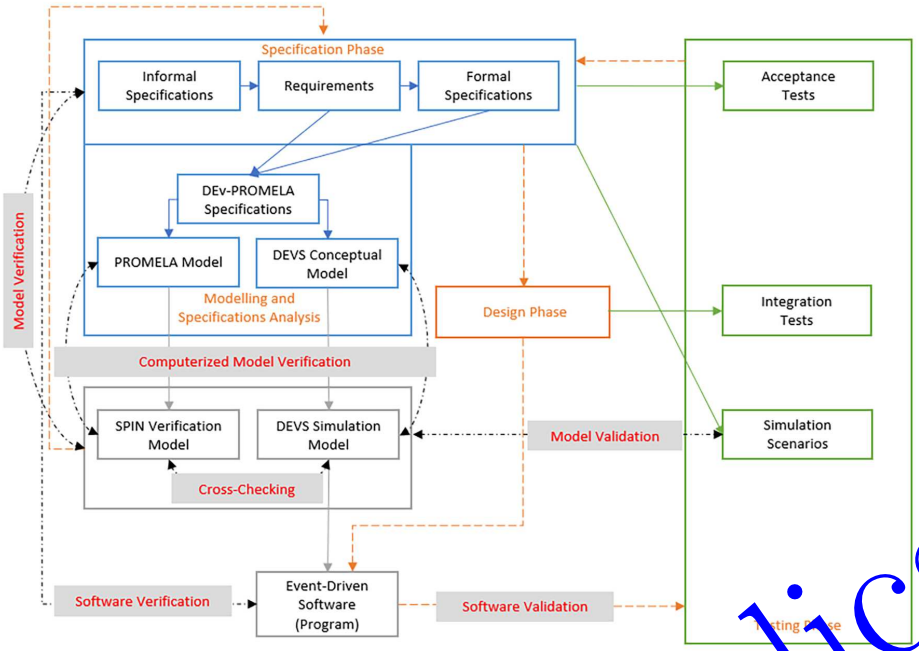

Figure 6. The Combined V&V Iterative Workflow. Orange dashed arrows represent cycles in model development or software development. Black dashed arrows represent checking tasks. Plain arrows represent artifacts generated at each stage of the workflow.

designers to focus on the most important requirements during the analysis and designing phase, and adding the less important computational aspects in the last iterations of the development cycle. Moreover, while the DEv-PROMELA specifications are formal, they can be surely analyzed in a formal fashion using theorem proving for instance. However, this is outside the scope of this paper.

4 Application: Modelling and Building a Video Game

As an illustration of our proposed methodology, we apply it for building a video game, or more precisely a part of the gameplay of a video game. Digital gaming [65, 66] is an interesting emerging field of research, while it gathers many problems of computer sciences. Especially, studies of the structure and development of a game shows that game architectures are composed by hundreds of classes, divided in two categories: functional and non-functional classes. Most of the efforts concern the development of non-functional classes. As a result, a lot of bugs are more related to a misdesign of functional requirements, because we try to adapt these requirements to the non-functional architecture. Video game developers are therefore facing the complexity of game, and it is well-known that they do not expense time in designing and in verification and validation procedures. Indeed, rigourous tests are considered as a loss of time, and only acceptance testing are intensively performed. This leads to many released games with a lot of bugs and defaults.

However, if you look at more precisely the structure of a game [67], we realize that it is in the most of case almost a discrete-event system in which time has a great importance. In fact, digital games are generally designed with a layered approach (figure 7):

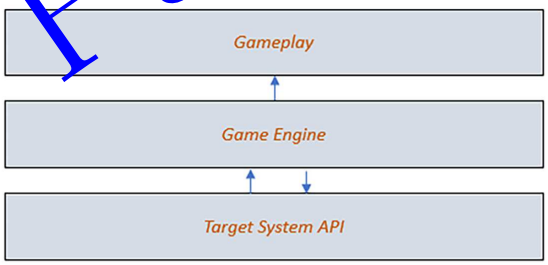

Figure 7. Layered Architecture of a Digital Game.

- The target system API gathers functionalities that allow communications between the engine and the host system;
- The engine is the core of the digital game software. This layer provides generic routines and procedures that perform common operations. Generally, the engine is divided into modules: Graphical Engine handles the rendering process, Physics Engine computes the physical effects of the objects that compose the virtual world, AI handles artificial behaviours, etc. All these modules need to communicate each others using messages. **Modern implementations use threads and events** [68];
- The gameplay layer contains the end-user rules. We can say that this layer is the clothing of a game. Generally, it contains specific procedures and processing. It is common to find event-based implementation through object-oriented pattern (observer pattern, etc.).

Thus, a video game can be seen as a set of processes that communicate each others in an asynchronous manner, and

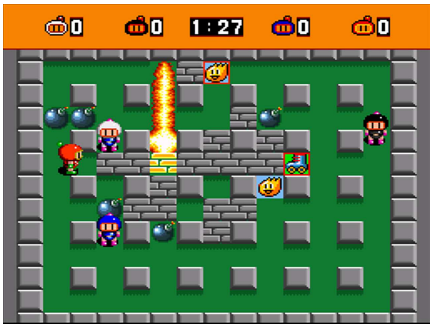

Figure 8. Bomberman, 1983.

using an event-oriented architecture. While our objective is not to deal with the complexity of an entire game, but just illustrating how DEv-PROMELA and the combined V&V approach can be used, we only focus in this section on a small part of the Engine and the Gameplay layers of a well-known game: Bomberman (figure 8). This game, whose the first release dates back to 1983, consists on a small character controlled by the player and which poses bomb on a map. When a bomb explodes, the nearest ennemies die, or new paths of the map are opened. Obviously, the player has many constraints: he cannot bring infinite bombs, he has a finite number of lifes, and he cannot move outside of the map. In this section, we focus on the first development iteration using DEv-PROMELA and our proposed V&V workflow. Readers should understand these steps are repeated for each set of specifications, until the game is fully developed. DUring this iteration, designers decided to focus on part related to character control. This character control involves three subsystems: characters in the gameplay layer, input manager in the engine layer, and the keyboard at the API level. Our iteration focus on these three components.

4.1 Specifications Phase

The first phase consists on analyzing the specifications selected for this iteration. Knowing the architecture of a game and the requirements, we can easily deduce some properties:

- The player controls one character with input coming from the keyboard.
- A character can move in only one direction at time.
- When a player places a bomb, its position is the same than the position of the player.
- When a bomb blows up, all the walls around a finite radius explode.
- The player have a finite number of lifes.
- The game is over when the number of lifes is equal to 0.
- Each bomb blows up after a finite number of seconds.
- The character moves with a constant speed.
- The character cannot move throw a wall.

Using these informal specifications, we can deduce that:

1. Bombs and characters can be considered as asynchronous processes which evolve in parallel. Moreover, we can describe their behaviours with a state machine.
2. Bombs and characters react to and emit events. Indeed, the character moves only when an event enforces it to move. Bombs are created when the player emits a *bomb creation* event, and blows after dt units of time. Explosions involve modifications of the map. As a consequence, we can deduce that transition functions depend on time and events.
3. The map is a finite set of elements. Each element have a position which can be described by a couple $(x, y) \in \mathbb{R} \times \mathbb{R}$.
4. The speed is a constant function. While the game is refreshed at each computed frame, the move equation is descretized and computed following the function:

$$pos(x, y, t + ft) = pos(x, y, t) + speed(x, y) * ft$$

where ft is the elapsed time between two frames. Then, some properties can be expressed using Linear Temporal Logic (LTL). For instance, the fact that a player can move in only one direction at time:

$$\Box\ ((pos_x(t + ft) \neq pos_x(t) \wedge pos_y(t + ft) = pos_y(t)) \vee (pos_x(t + ft) = pos_x(t) \wedge pos_y(t + ft) \neq pos_y(t)))$$

While the positions are computed at each frame, we can store the old positions and the new positions in variables. At the end of the computation, the new state just needs to satisfy this property.
5. In the same way, we can check that a bomb will always blows up with a LTL property:

$$\diamond\ (blows\_up(n))\ \forall n \in D\ (D \subset \mathbb{N})$$

where n is the id of a bomb.

4.2 Modelling Phase

The specifications phase shows that such a game can be pretty modelled using DEv-PROMELA, as each gameplay element can be expressed with discrete-event state machines that communicate each others (Figure 9). In this example, the gameplay layer is seen as a DEv-PROMELA program which receives events from the engine layer. The gameplay layer is a coupling of DEv-PROMELA subprocesses that are interconnected each others. This means that when a process emits an event, this one is transmitted to all the other processes.

Each gameplay element is then represented by a DEv-PROMELA atomic model. For instance, a playable character can be modelled with a state machine as shown in Figure 10. The character stays in the `IDLE` state until it receives an event that implies a move. Then, it stays in this new move state during dt units of time. This amount of time models the time needed to compute two consecutives frames, that is why there is an internal loop transition from each move state to itself. If an `end_move` event is received, the state machine returns to the `IDLE` state. This DEv-PROMELA process thus models a kind of character controller.

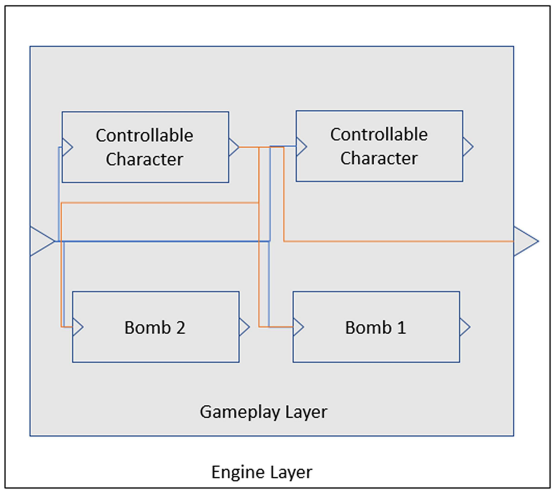

Figure 9. Example of a partial gameplay architecture with DEv-PROMELA. Not all the coupling have been represented.

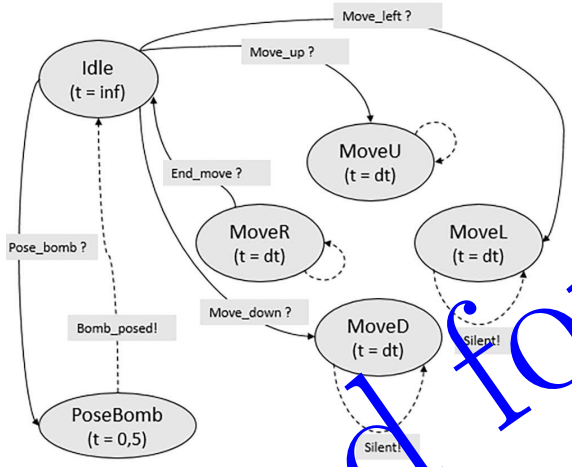

Figure 10. Example of a partial DEv-PROMELA process controlling a playable character. Not all the transitions have been represented. Dashed arrows represent internal transitions, while plain arrows model external transitions.

A syntactic version of the DEv-PROMELA model is given in Algorithm 7. This version is really close to any implementation language. Like said earlier, $cposx, cposx1, cposy, cposy1$ variables represent the position of the character computed at the previous frame and at the current frame. l.11 allows the process to evaluate the global state of the game. If the game is not finished, the state machine acts as expected: it stays in `IDLE` (l.12 to l.18) until an event is received. Otherwise, the move is updated at each dt units of time (l.21 to l.26). Finally, the posing bomb action is modelled (l.17 and l.19). For each gameplay element, a `proctype` block can be written in the same manner. Note that non-prefixed statements means that they correspond to states with a zero lifespan. This enforces the statement to be immediately executed.

At a lower level, we can also model the Input Manager (figure 11). This engine component translates each input keycode generated by the API layer to a an event that the character controller can understand. We can therefore consider the Engine layer as a DEv-PROMELA coupled model which is itself coupled to the DEv-PROMELA models that composed the Gameplay layer. In this design, the Input Manager is just a DEv-PROMELA atomic model as given in figure 12. The interpretation of the model is simple: when the player presses a key, the corresponding event is emitted to the Input Manager that immediately translates it into the corresponding action event. This does not mean that the Input Manager is coded using a State Pattern, but it just represents the fact that a program is a state machine. Therefore, a possible C++ implementation of this Input Manager could be the ones given in Algorithm 6. At this point, reader could thus ask why using DEv-PROMELA to model the Input Manager, while time does not seem to play a role in the implementation. As previously said, this model is not a translation of the source code, even we could also code an Input Manager using a State Pattern. In fact, this model illustrates the fact that DEv-PROMELA allows modeller to express the delay between the moment a key is pressed or released and the moment this event is translated into an event code, by setting the lifespan of the translation states (Left, Right, Up, Down, and Release). This can have an great impact on the other subsystems of the game, while we can therefore model a processing delay which can lead to performance issues in the final game. This is useful in the context of digital games, to introduce laggy scenarios for instance. Another advantage is that an input simulator can be easily implemented, while we just need to couple the input manager model with an input generator model. This generator can be modelled/implemented using any formalisms/language thanks to the DEVS Bus mechanism [23, 24].

ALGORITHM 6: A possible C++ implementation of the InputManager.

```
1: func_type InputManager::handle(int code, int state)
2: {
3:     if(state == RELEASE) return
   action["END_MOVE"];
4:     else if(code == KEY_LEFT) return
   action["MOVE_LEFT"];
5:     else if(state == KEY_RIGHT) return
   action["MOVE_RIGHT"];
6:     else if(state == KEY_UP) return
   action["MOVE_UP"];
7:     else if(state == KEY_DOWN) return
   action["MOVE_DOWN"];
8:     return NULL;
9: }
```

When the modelling of the game subsystems is done, the requirements are then encoded into LTL PROMELA properties. For instance,

- $\diamond\,(state == POSE\_BOMB)$ becomes $ltl\{eventually\ (state == POSE\_BOMB)\}$;
- $ltl\{always\ ((state == POSE\_BOMB)\ implies\ X(state == IDLE))\}$ means that the next state of `POSE_BOMB` is `IDLE`;

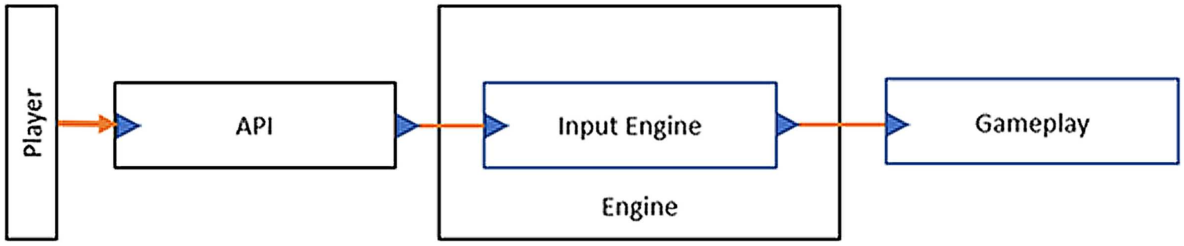

Figure 11. DEv-PROMELA Architecture of the game. Black boxes correspond to DEv-PROMELA coupled model, blue boxes to DEv-PROMELA atomic model, and orange edges to events between the components.

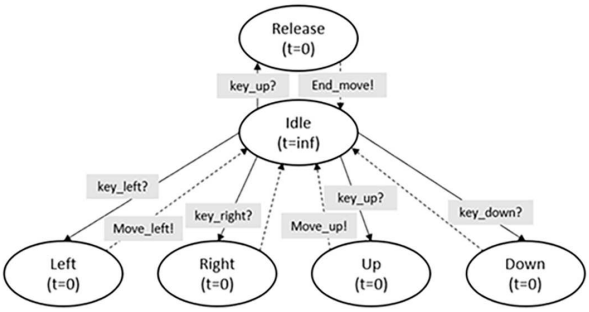

Figure 12. DEv-PROMELA atomic model of the Input Manager.

- *ltl* $\{always(cposx >= 0 \quad and \quad cposx <= mapw) \quad and \quad (cposy >= and \quad cposy \quad <= maph)\}$ checks that the character can't move outside the map.
- etc.

All theses properties follow with the process models in the DEv-PROMELA specifications. When the model is fully written, the computerized verification and simulation models are automatically generated.

4.3 Implementation Phase

Design. During this first iteration, no design phase were needed, allowing designers to focus only on functional requirements. Indeed, non-functional requirements which support the computations are already integrated in the structure which support the simulation. Therefore, our three-layered architecture is implicitely generated by the coupling of the DEv-PROMELA models. While the simulation model is based on OOP, we don't need to overload a new design. However, complementary sequence diagrams or collaborative diagrams can be made if needed.

Verification Model Generation. The generation of the SPIN verification model is done in two steps:

1. PROMELA equivalent specifications are generated from the DEv-PROMELA model. All the meta instructions are removed (Algorithm 8).
2. Then, the model is compiled with the SPIN model-checker to generate the corresponding verification model. Explaning how the model is generated is outside of the scope of this paper, while we let the responsability of this generation to the model-checker. Note that a modification of the SPIN model-checker can be done to take into account the `real` abstract type. For the purpose of this paper, we assume that real values could be abstracted into integer values in the conceptual model.

Simulation Model Generation. Generation of the computerized simulation depends on the simulation library and framework which are used by the developer. Therefore, the DEv-PROMELA model is converted into a DEVS algebraic specification which can then be encoded into any target simulators, for instance, into a XML format, JAVA classes, C++ classes, etc. Concerning OOP implementations of DEVS, many patterns have been proposed [69, 70]. For this paper, we decide to use an implementation close to a game architecture (Figure 14). This architecture is based on an *observer* pattern and a *state* pattern [71]. White classes are a part of a simulation framework which can be embbeded in the Engine Layer. The grey classes are the concrete classes realized in the Gameplay Layer. They embbed the code of the corresponding DEv-PROMELA simulation model.

The DEVS model is therefore a composition of `DEVSAtomic` and `DEVSCoupled` instances. Each instance of them is a finite state automaton which aggregates concrete states. A state corresponds to an atomic statement of the DEv-PROMELA conceptual model. Then, a specific class for each real process is created (`Character` and `Bomb`). These classes inherit from `DEVSAtomic`. This part of this architecture represents the structural part of the model.

Then, the semantics (i.e. the behaviour) is implemented by the classes `DEVSCoordinator`, on the one hand, which is a subclass of `TimeScheduler`, and `DEVSSimulator` on the other hand. The first one is responsible of handling time advance and events. It computes the next minimum event and notifies all the children (given by the composition of `DEVSCoupled` instances). The second one is responsible of generating the corresponding behaviour of each `DEVSAtomic` instance, meaning executed the code inside each concrete DEVS atomic model.

Each LTL property generates also a `DEVSAtomic` subclass. In fact, while LTL properties are Buchi automata, they can be encoded into DEVS models. Instances of these classes are plugged to the global model for monitoring the global behaviour. If a property is violated, a specific error

event is emitted thanks to an assertion or any other exception mechanisms, and stops the simulation.

Verification and Validation using Model Checking. Now both computerized model have been generated, we can use them to check properties. The first property we can verify concerns the flow of the state machine. If we check whether the state $POSE\_BOMB$ can be reached with the Algorithm 7, the model-checker ensures that it cannot be reached. Indeed, there is no statement that leads to this state. In the same manner, we can easily see that there is no test concerning the property that enforces the character to stay into the map. A counterexemple is generated showing that we have a scenario for which the absolute position can be greater than the size of the map. In the next iteration, we modify the model and we add these lines into the algorithm, in replacement of line 21:

```
:: if (cposx - dt * speed != mapw) ->
        cposx1 = cposx;....
```

This time, the checker ensures that the character cannot go out of the map. We check also if we can always reached the `IDLE` state from any move, and the checker ensures that is possible.

Verification and Validation using Simulation. When we complete the verification with simulation, we state that:

1. There exists scenarios in which the character leaves the map. In fact, this come from the fact that the model-checking sees the time as ordered event, meaning it did not take into account the time elapsed between two frames. We can immediately fix this bug by replacing the previous fix with:

   ```
   :: if (cposx - dt * speed <= mapw)
        -> cposx1 = cposx;....
   ```

2. In the same manner, the simulation shows that, if the time scheduler is not well-formed, for example if $dt = \infty$, the model goes into an infinite loop. This loop was not captured by the checker because of the same fact: time is not seen as a quantitative dimension in model-checking.
3. In the same manner, some counterexamples generated by the untimed model can be replayed in the timed model.

Finally, we can deduce that model-checking is useful for verifying and validating the flow and the bounds of values. Discrete-event simulation is useful for checking event properties and confirming the results given by the formal verification. Therefore, we can take benefits from the speed of untimed model-checking coupling the accuracy of simulation analysis.

4.4 Release Phase

Finally, in order to get the final character controller of our game, we need only to replace the DEVSCoordinator root instance by a real-time coordinator which handles the game loop. If the class inherits from the `TimeScheduler` class, the change is immediate according to the OOP best practices. This new class can be verified using traditional verification processes (unit tests and code review). In our specific exemple, this class corresponds to a function which returns immediately at each call. In this way, we reduce the development effort. It is interesting to note that during this first iteration no code was handwritten. Therefore, the code remains unchanged and we are sure that the software fulfills at least the checked properties. The final software then needs to be tested, analyzed, verified and validated using classical techniques (i.e. using approaches that are not model-based).

5 Experimental Evaluation and Comparison

This section introduces the experimental evaluation of the approach proposed in this paper. For this purpose, we compare two versions of the Bomberman. The first one is developped using our model-based engineering approach; the second one is developped using a classical software engineering development V cycle.

The protocole of validation is then divided into multiple steps:

1. Writing the validation test scenarios which will be used for validating the two games;
2. Developping the two versions of the game;
3. Verifying the two programs using classical verification techniques and our approach;
4. Validating the two softwares using classical validation techniques and comparing results with simulation.

5.1 Write validation test cases

While we mainly focus on the development of the character controller and the input manager, we choose a set of fourty validation base tests relative to these both parts. These tests are written during the requirement analysis, and cover safety and liveness properties (Table 6). Particularly, we check for instance that the hero correctly responds to the input events, that the timed responses are always in an acceptable range, and that stressing the game does not create computational errors in the interpolated moves of the entities. These validation tests are then formalized using the linear temporal logic. We also verify by review that the requirements are correctly formalized.

For each timed tests, we also decide to introduce some random parametrizations; for instance, we make vary the speed or the acceleration of the characters, or make change the date of each event occurrence. The total set of validation tests has finally fourty different tests scenarios. Note that we consider the DEVS simulator and the SPIN model-checker as verified and validated, so we do not perform the computerized model verification (Figure 6).

5.2 Pre-validation using DEv-PROMELA

The DEv-PROMELA models of the character controller and input manager are incrementaly built. At the end of each iteration, the model is checked against the fourty properties using both model-checking and simulation like previously presented. The models are refined until they all comply with the requirements. Figure 13 shows the number of fullfilled properties at the end of each iteration. Then, in our example, we need five iterations to ensure all the fourty properties

are correctly checked. Note that this step also allows a way to evaluate correctness, completeness and testability of the requirements. Validation corresponds to evaluation of fourty functional requirements as defined in the game design document:

- Moves of the characters;
- Collision detection;
- Items generations;
- Scoring system;
- Artificial Intelligence system;
- Graphics display and synchronization (use of the correct sprite and animation, verification of computation on variable refresh rate);
- Sound synchronization;
- Input synchronization.

Each iteration corresponds to the implementation of specific groups of specifications, beginning with generic systems at the first iteration, and finishing by the gameplay implementation at the last iteration. In this case, the model is used as follow:

- Model-checking ensures that each instruction is reachable, and there is no deadlock between thread; in this case, model-checking is complementary to unit test;
- Model-checking ensures that the parameters evolves under boundaries (each function respects pre- and post-conditions structurally);
- Simulation allows checking of specific subpart of the parameter space: especially, we can generate long sequences of events, inputs and parameters variations (speed computation to check if long computation doesn't break the deadlock property, refresh rate, resources loading speed, functional parameters evolving outside of the expected boundaries, etc).

Figure 13. Pre-validation results.

5.3 Verification of the derived software and the normal game

Concerning the model-based version of the game, its code is then derivated from the DEv-PROMELA simulation model. Verification is performed using classical methodologies including code review, static analysis using proofs, and unit tests. A DEv-PROMELA model is also regenerated from the code and compared with the results got from the previous steps, using model-checking and simulation. Concerning the normal version of the game, the code is reviewed using classical verification techniques only. The game is also reverse-engineered to produce a DEv-PROMELA model which is formally checked and simulated.

Table 3 summarizes the results between the three methodologies concerning our game. As expected, all the unit tests are successfully passed, including assertion violations. Completeness refers to the fact that the code responds to all the possible inputs defined in the specifications. Consistency refers to the fact that the code cannot produce contradictory behaviours. Unreachable states correspond to line of code which cannot be executed. As expected, classic verification techniques confirms what we expected: the obtained software behave as intended, while it was built from a simulation model.

Table 4 summarizes the results concerning the normal game obtained by classic verification, model-checking and simulation.

First, as expected, we see difference between separate model-checking and simulation. The differences come from the fact that the model-checking cannot capture some time-dependant behaviours as we demonstrated in previous sections. In these cases, simulation gives another interpretation of the false positive errors found by the model-checking. Second, we also see that the DEv-PROMELA Reverse Model allows us to find some inconsistent behaviours, which were not found using only tests. While tests are only focusing on prepared scenarios, model-checking allows us to explore the statespace and check what kind of events can exactly occurs. Simulation then allows us to find false positive and false negative errors among the errors detected by the model-checking.

5.4 Validation of the derived software

Validation consists on testing our model-based game against the fourty test scenarios and comparing the results with the expected values and the values given in the first phase by the simulation. Moreover, we add 2332 random scenarios including random actions. The same evaluation is also done for the normal game. Table 5 summuarizes the results. As we see, concerning the validation tests, our DEv-PROMELA model-based game, like the simulation model, reach 100% of successfully validated test cases. While the model was developed using these validation tests, the results is normal. However, we see a difference between the simulation model and the DEv-PROMELA model-based game concerning the random tests. Even the results is greater that the normal game, 8% of these random scenarios lead to an inconsitent state. In most of the cases, these errors are related to communication between components which were not developed using our approach, for instance, synchronization between graphics component and the input manager. In these cases, simulation allows us to understand what exactly happens in the communication between each component, while formal validation using model-checking allows us to understand what exactly happens in the execution graph.

6 Conclusion and Future Works

As a conclusion, this paper introduced a new way for combining model-checking and discrete-event simulation for combined verification and validation procedures. This promising methodology is based on the building of a new formalism whose the foundations are a verifiable language, to which we add the semantics of a simulation language. As an example, we show how we built DEv-PROMELA from both PROMELA and the DEVS formalism. As a result, we see that formal verification can be used for verification and validation, and that simulation can complete the results obtained by model-checking. Indirectly this approach reduces the size of the checked statespace while formal verification and validation are not applied for checking the same requirements.

We then proposed a new Software Verification and Validation approach by introducing the use of our new modelling language in the developing process. As an illustration of such a methodology, we use DEv-PROMELA for modelling, verifying and validating a part of a video game. Because video games are essentially discrete-event systems, they are well suited for such kind of methods. Firstly, we show how requirements analysis motivates the choice of DEv-PROMELA as formal specifications and modelling language. We model each part of the game using DEv-PROMELA, allowing us to generate two computerized model: a formal verification model and a simulation model. Verification model is used for checking the global flow of the game, for instance to highlight unreachable states, while Simulation model is used for checking the timed behaviour and validating timed properties. Then, by changing the event scheduler into a real-time scheduler, we can obtain an implementation of the final game, which fulfills the verified and validated properties. This approach was already applied to other domains like manufacturing production lines, mobile adhoc network, resource sharing algorithms.

As a model-driven engineering approach, our proposed methodology suffers also from common issues of software engineering. Indeed, our methodology strongly remains dependant on tests scenarios and experts, and the critical key remains the requirement/specification phase. Indeed, we can't completely ensure that all the relevant tests are performed during the cycle. For instance, if a requirement is not specified, there is no chance to check an non-existent property. However, compared to current approaches, the use of formal verification and simulation early in the development allows designers to detect flaws and inconsistencies. Indeed, even if not perfect, the parameter space can be better explored during the investigation phase of the simulation. Additional work using machine learning and cognitive architecture which would help designers to generate relevant test scenarios for their models is interesting way to overcome these difficulties. Test-driven approaches could be also a way to reduce the dependency to scenarios, while the model would be written iteratively according to specific test cases.

Future works can concern improvement of the combination between simulation and model-checking, especially using simulation to narrow the searching statespace. This could make the formal verification faster. In the same way, we can study how modifying the SPIN model checker to take into account the abstract real value. This would prevent false errors returned by the checker and that are caused by the untimed level of abstraction. Another one could concern the study of the properties of the DEVS subclass generated by DEv-PROMELA, as we showed we can extend or reduce it using different simulation formalisms [64]. Especially, we can formally demonstrate that DEv-PROMELA is an extension of PROMELA. Concerning our proposed V&V workflow, we didn't study in this work how we could integrate our methodology in agile methods. Indeed, V&V is a very consuming tasks, therefore it would be intersting to understand how they could be adapted in the case of a development following agile recommendations [72]. Finally, the main improvement is in the use of cognitive architecture and machine learning to overcome the needs of strong expertise. Indeed, research in empirical software engineering and data-driven approaches in machine learning suggests it would be possible to automatically generate relevant scenarios and test cases from an existing database, in order to target specifically incomplete specifications, or focus on specific cases in the experimental frame [29, 73].

Acknowledgements

This work is part of the R&D project "MAGE", from French "Investing for the Future" national program. We thank Bernard P. Zeigler and Chungman Seo for assistance and for comments that greatly improved work and the manuscript.

Author biography

Aznam Yacoub was a member of Laboratoire d'Informatique et des Systemes (LIS), Marseille, France. His research interests include software engineering, artificial intelligence, cognitive science and multimedia.

Maâmar El-Amine Hamri is an Associate Professor at Aix-Marseille University. He is also a member of Laboratoire d'Informatique et des Systemes (LIS), Marseille, France. He has been active for many years in Modeling and Simulation research area.

Claudia Frydman is a full Professor at Aix-Marseille University. She is also a member of the Laboratoire d'Informatique et des Systemes (LIS). She has been active for many years in knowledge management and currently her research is focusing especially on researches on knowledge-based simulation.

References

1. Shah US. An excursion to software development life cycle models: An old to ever-growing models. *SIGSOFT Softw Eng Notes* 2016; 41(1): 1–6.
2. Sargent RG. Simulation model verification and validation. In: *Proceedings of the 23rd Conference on Winter Simulation*, WSC '91, Washington, DC, USA: IEEE Computer Society, 1991. pp. 37–47.
3. Pham H. Software reliability. In: *Wiley Encyclopedia of Electrical and Electronics Engineering*. 1999. DOI:10.1002/047134608X.W6952.

4. Clarke EM and Wing JM. Formal methods: State of the art and future directions. *ACM Computing Survey* 1996; 28(4): 626–643.
5. Woodcock J, Larsen PG, Bicarregui J et al. Formal methods: Practice and experience. *ACM Computing Survey* 2009; 41(4): 19:1–19:36.
6. Clarke EM and Emerson EA. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: *Logic of Programs, Workshop*, Springer-Verlag, 1982. pp. 52–71.
7. Queille JP and Sifakis J. Specification and verification of concurrent systems in cesar. In: *Proceedings of the 5th Colloquium on International Symposium on Programming*, Springer-Verlag, 1982. pp. 337–351.
8. Clarke EM, Grumberg O and Peled DA. *Model Checking*. MIT Press, 1999.
9. He X. PZ nets - a formal method integrating petri nets with z. *Information and Software Technology* 2001; 43(1): 1–18.
10. Nguyen A, Quan T, Nguyen P et al. COMBINE: A tool on combined formal methods for bindingly verification. In: *Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis*, Springer Berlin Heidelberg, 2010. pp. 387–395.
11. Konur S, Fisher M and Schewe S. Combined model checking for temporal, probabilistic, and real-time logics. *Theoretical Computer Science* 2013; 503: 61–88.
12. Holzmann GJ. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., 1991.
13. Alur R and Dill DL. A theory of timed automata. *Theoretical Computer Science* 1994; 126(2): 183–235.
14. Clarke EM. The birth of model checking. In: *25 Years of Model Checking: History, Achievements, Perspectives*. Springer Berlin Heidelberg, 2008. pp. 1–26.
15. Dacharry HP and Giambiasi N. A formal verification approach for devs. In: *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC '07, Society for Computer Simulation International, 2007. pp. 312–319.
16. Baier C and Katoen JP. *Principles of Model Checking*. The MIT Press, 2008.
17. Huth M and Ryan M. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2005.
18. Hamri H, Giambiasi N and Naamane A. Generalized discrete events for accurate modeling and simulation of logic gates. In: *Concepts and Methodologies for Modeling and Simulation*. Springer International Publishing, 2015. pp. 257–272.
19. Giambiasi N, Escude B and Ghosh S. Gdevs: a generalized discrete event specification for accurate modeling of dynamic systems. In: *Proceedings 5th International Symposium on Autonomous Decentralized Systems*, 2001. pp. 464–469. DOI: 10.1109/ISADS.2001.917452.
20. Bill R, Gabmeyer S, Kaufmann P et al. Model checking of ctl-extended ocl specifications. In: *Proceedings of the 7th International Conference on Software Language Engineering*, Springer International Publishing, 2014. pp. 221–240.
21. Gore R. Springsim 2015 - conceptual modeling with alloy, 2015. URL https://github.com/rossgore/alloy-tutorial.
22. Syriani E and Vangheluwe H. Programmed graph rewriting with time for simulation-based design. In: *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, Springer Berlin Heidelberg, 2008. pp. 91–106.
23. Zeigler BP. *Theory of Modeling and Simulation*. John Wiley, 1976.
24. Zeigler BP, Muzy A and Kofman E. *Theory of Modeling and Simulation - Discrete Event and Iterative System Computational Foundations*. 3rd ed. Springer, Academic Press, Inc., 2019.
25. Minsky M. Matter, mind and models. In: *IFIP Congress*, Spartan Books, 1965. pp. 45–50.
26. Banks J and Carson JS II. Introduction to discrete-event simulation. In: *Proceedings of the 18th Conference on Winter Simulation*, WSC '86, New York, NY, USA: ACM, 1986. pp. 17–23.
27. Banks J. *Handbook of simulation: principles, methodology, advances, applications, and practice*. John Wiley & Sons, 1998.
28. Holzmann G. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 2003.
29. Traoré MK and Muzy A. Capturing the dual relationship between simulation models and their context. *Simulation Modelling Practice and Theory* 2006; 14: 126–142.
30. Alur R, Courcoubetis C, Halbwachs N et al. The algorithmic analysis of hybrid systems. *Theoretical computer science* 1995; 138(1): 3–34.
31. Börger E. Abstract state machines: a unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic* 2005; 133(1): 149–171.
32. Katoen JP. Advances in probabilistic model checking. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, 2010. pp. 25–25.
33. Foures D, Albert V and Nketsa A. Simulation validation using the compatibility between simulation model and experimental frame. In: *Proceedings of the 2013 Summer Computer Simulation Conference*, SCSC '13, Vista, CA: Society for Modeling and Simulation International, 2013. pp. 55:1–55:7.
34. Olsen M and Raunak M. A method for quantified confidence of devs validation. In: *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, Society for Computer Simulation International, 2015. pp. 135–142.
35. Zengin A, Köklükaya E and Ekiz H. Verification and validation of the devs models. In: *Proceedings of 2nd International Symposium on Sustainable Development*, 2010. pp. 425–433.
36. Coelho CNJ and Foster HD. *Advanced Formal Verification*. Springer US, 2004.
37. Godefroid P. Combining model checking and testing. Technical report, Microsoft, 2013. URL http://research.microsoft.com/apps/pubs/default.aspx?id=200544.
38. Goldberg E. On bridging simulation and formal verification, 2008.
39. Bicarregui J, Dick J, Matthews B et al. Making the most of formal specification through animation, testing and proof. *Science of Computer Programming* 1997; 29(1-2): 53–78.
40. Abdulhameed A, Hammad A, Mountassir H et al. An approach combining simulation and verification for sysml using systemc and uppaal. In: *8ème conférence francophone sur les architectures logicielles*, Paris, France, 2014.
41. Zeigler BP and Nutaro JJ. 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*, 2014. pp. 350–356.

42. Zeigler BP and Nutaro JJ. Towards a framework for more robust validation and verification of simulation models for systems of systems. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology* 2015; : 1–14.
43. Zeigler BP, Nutaro JJ and Seo C. Combining devs and model-checking: concepts and tools for integrating simulation and analysis. *International Journal of Simulation and Process Modelling* 2017; 12(1): 2–15. DOI:10.1504/IJSPM.2017.082781.
44. Dacharry H and Giambiasi N. Formal verification with timed automata and devs models: a case study. In: *Proceedings of Argentine Symposium on Software Engineering*, 2005. pp. 251–265.
45. Yacoub A, Hamri M, Frydman C et al. Towards an extension of promela for the modeling, simulation and verification of discrete-event systems. In: *Proceedings of the 27th European Modelling and Simulation Symposium (EMSS 2015)*, 2015. pp. 340–348.
46. Yacoub A, Hamri M and Frydman C. 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 '16, ACM, 2016. pp. 245–253.
47. Yacoub A, Hamri M, Frydman C et al. Dev-promela: an extension of promela for the modelling, simulation and verification of discrete-event systems. *International Journal of Simulation and Process Modelling* 2017; 12(3-4): 313–327. DOI:10.1504/IJSPM.2017.085564.
48. Holzmann GJ. The model checker spin. *IEEE Transactions on software engineering* 1997; 23(5): 279–295.
49. Natarajan V and Holzmann GJ. Outline for an operational semantics of promela. In: *Proceedings of the Second SPIN Workshop*, 1996.
50. Caplat G and Sourrouille JL. Model mapping in mda. In: *Workshop in Software Model Engineering (WISME2002)*, 2002.
51. Caplat G and Sourrouille JL. Model mapping using formalism extensions. *IEEE Software* 2005; 22(2): 44–51. DOI:10.1109/MS.2005.45.
52. Plotkin G. A structural approach to operational semantics. *J Log Algebr Program* 2004; 60–61: 17–139. DOI:10.1016/j.jlap.2004.05.001.
53. Kahn G. Natural semantics. In: Brandenburg FJ, Vidal-Naquet G and Wirsing M (eds.) *Proceedings of STACS 87: 4th Annual Symposium on Theoretical Aspects of Computer Science Passau*, Berlin, Heidelberg: Springer Berlin Heidelberg, 1987. pp. 22–39. DOI:10.1007/BFb0039592.
54. Blas MJ, Gonnet SM, Leone HP et al. A conceptual framework to classify the extensions of devs formalism as variants and subclasses. In: *Proceedings of the 2018 Winter Simulation Conference*, WSC '18, IEEE Press, 2018. pp. 560–571.
55. Zeigler BP. Closure under coupling: Concept, proofs, devs recent examples. In: *Proceedings of the 4th ACM International Conference of Computing for Engineering and Sciences*, ICCES'18, New York, NY, USA: ACM, 2018. DOI:10.1145/3213187.3213194.
56. Wallace DR and Fujii RU. Software verification and validation: An overview. *IEEE Softw* 1989; 6(3): 10–17.
57. Ahmad W, Qamar U and Hassan S. Analyzing different validation and verification techniques for safety critical software systems. In: *Software Engineering and Service Science (ICSESS), 6th IEEE International Conference on*, IEEE, 2015. pp. 367–370.
58. Huizinga D and Kolawa A. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.
59. Desai S and Abhishek S. *Software Testing: A Practical Approach*. India: Phi Learning Private Limited, 2012.
60. Adrion WR, Branstad MA and Cherniavsky JC. Validation, verification, and testing of computer software. *ACM Comput Surv* 1982; 14(2): 159–192.
61. Balci O. Verification, validation, and accreditation. In: *1998 Winter Simulation Conference Proceedings (Cat. No.98CH36274)*, 1998. pp. 41–48. DOI:10.1109/WSC.1998.744897.
62. Tran E. Verification/Validation/Certification. Technical report, Carnegie Mellon University, 1999.
63. Gaudel MC. Checking models, proving programs, and testing systems. In: Gogolla M and Wolff B (eds.) *Proceedings of Tests and Proofs: 5th International Conference, TAP 2011*, Zurich, Switzerland: Springer Berlin Heidelberg, 2011. pp. 1–13.
64. Yacoub A, Hamri M and Frydman C. Restricting dev-promela with a hierarchy of simulation formalisms. In: *Proceedings of the Symposium on Theory of Modeling & Simulation*, TMS/DEVS '17, San Diego, CA, USA: Society for Computer Simulation International, 2017. pp. 13:1–13:11.
65. Frans M, Jan VL and Thorsten Q. Disciplinary identity of game scholars: An outline. In: *DiGRA '13 - Proceedings of the 2013 DiGRA International Conference: DeFragging Game Studies*, 2014.
66. Quandt T, Van Looy J, Vogelgesang J et al. Digital games research: A survey study on an emerging field and its prevalent debates. *Journal of Communication* 2015; 65(6): 975–996. DOI:10.1111/jcom.12182.
67. Deloura M. *Game Programming Gems*. Rockland, MA, USA: Charles River Media, Inc., 2000.
68. Epic. Unreal engine. Technical report, 2017. URL https://docs.unrealengine.com/latest/INT/.
69. Hamri M, Messouci R and Frydman C. The state event design pattern. In: *Proceedings of the 19th European Conference on Pattern Languages of Programs*, EuroPLoP '14, New York, NY, USA: ACM, 2014. pp. 15:1–15:14. DOI:10.1145/2721956.2721987.
70. Messouci R. *Conception par patrons des modèles à Evenements Discrets : de la Machine à états finis au DEVS*. PhD Thesis, France, 2017.
71. Gamma E, Helm R, Johnson R et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
72. Huo M, Verner J, Zhu L et al. Software quality and agile methods. In: *Proceedings of the 28th Annual International Computer Software and Applications Conference, COMPSAC'04*, IEEE, 2004. pp. 520–525.
73. Nayrolles M and Hamou-Lhadj A. Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In: *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, ACM, 2018. pp. 153–164.

Appendices

Table 3. Results of Verification of the derived software (in percent).

	Classic Verification	DEv-PROMELA Reverse Model	DEv-PROMELA Initial Model
Unreachable States	0	0	0
Correctness	100	100	100
Consistency	100	100	100
Completeness	100	100	100

Table 4. Results of Verification of the normal software (in percent).

	Classic Verification	DEv-PROMELA Model-Checking	DEv-PROMELA Simulation	DEv-PROMELA Reverse Average
Unreachable States	12	18	19	18.5
Correctness	98	91	96	93.5
Consistency	100	96	98	97
Completeness	100	100	100	100

Table 5. Results of Validation of the both versions of game.

	Pre-Simulation Results	DEv-PROMELA model-based game	Normal game
Number of validated test cases	40	40	37
Number of validated random scenarios	2184	2151	2003
Total	93%	92%	86%

Table 6. Examples of Verification and Validation properties checked by combined methods.

Type	Description	Methodology
Verification	State Reachability: Verify each statement can be executed in any orders without affecting the result	Model-Checking
Verification	Deadlock: Verify the absence of interlock between thread during engine computation	Model-Checking
Validation	Bounded Parameters Verification: Verify the upper and lower boundaries of variables (type verification): score, life, speed, level, map representation Validation: Validate that the parameters evolve under their definition interval	Model-checking and Simulation
Validation	Computing time variation: ensure variation of computational time doesn't change the result of move computation	Model-checking: Computational error (divide by zero, infinite values, cycles on type interval) Simulation: Check if interpolation formula always produce acceptable values for animation

ALGORITHM 7: DEv-PROMELA specifications of the Character Controller.

```
1: ...
2: bool game_finished = false;
3: mtype {IDLE, MOVE_LEFT, MOVE_RIGHT, MOVE_UP, MOVE_DOWN, POSE_BOMB}
4:
5: active proctype Character ( real dt ){
6:    real cposx = 0, cposy = 0, cposx1 = 0, cposy1 = 0;
7:    real blowsup_time = 0.5;
8:    int state = IDLE;
9:    do
10:   :: [clt : 0.0 → silent] ( game_finised == false ) →
11:      if
12:      :: [evt : MOVE_LEFT] ( state == IDLE ) state = MOVE_LEFT;
13:      :: [evt : MOVE_RIGHT] ( state == IDLE ) state = MOVE_RIGHT;
14:      :: [evt : MOVE_UP] ( state == IDLE ) state = MOVE_UP;
15:      :: [evt : MOVE_DOWN] ( state == IDLE ) state = MOVE_DOWN;
16:      :: [evt : POSE_BOMB] ( state == IDLE ) state = POSE_BOMB;
17:      :: [evt : END_MOVE] ( state ≠ POSE_BOMB ) → state = IDLE;
18:      :: [clt : 0.5 → emit : bomb_posed] ( state == POSE_BOMB ) → state = IDLE;
19:           run Bomb(blowsup_time);
20:      :: [clt : dt → emit : silent] ( state == MOVE_LEFT ) →
21:           cposx1 = cposx;
22:           cposx = cposx − dt ∗ speed;
23:      :: [clt : dt → emit : silent] ( state == MOVE_RIGHT ) → ...
24:      ...
25:      fi;
26:   od
27: }
28: ...
```

ALGORITHM 8: PROMELA specifications of the Character Controller.

```
1: ...
2: bool game_finished = false;
3: mtype {IDLE, MOVE_LEFT, MOVE_RIGHT, MOVE_UP, MOVE_DOWN, POSE_BOMB}
4:
5: active proctype Character ( int dt ){
6:    int cposx = 0, cposy = 0, cposx1 = 0, cposy1 = 0;
7:    int blowsup_time = 1;
8:    int state = IDLE;
9:    do
10:   :: ( game_finised == false ) →
11:      if
12:      :: ( state == IDLE ) state = MOVE_LEFT;
13:      :: ( state == IDLE ) state = MOVE_RIGHT;
14:      :: ( state == IDLE ) state = MOVE_UP;
15:      :: ( state == IDLE ) state = MOVE_DOWN;
16:      :: ( state == IDLE ) state = POSE_BOMB;
17:      :: ( state ≠ POSE_BOMB ) → state = IDLE;
18:      :: ( state == POSE_BOMB ) → state = IDLE;
19:           run Bomb(blowsup_time);
20:      :: ( state == MOVE_LEFT ) →
21:           cposx1 = cposx;
22:           cposx = cposx − dt ∗ speed;
23:      :: ( state == MOVE_RIGHT ) → ...
24:      ...
25:      fi;
26:   od
27: }
28: ...
```

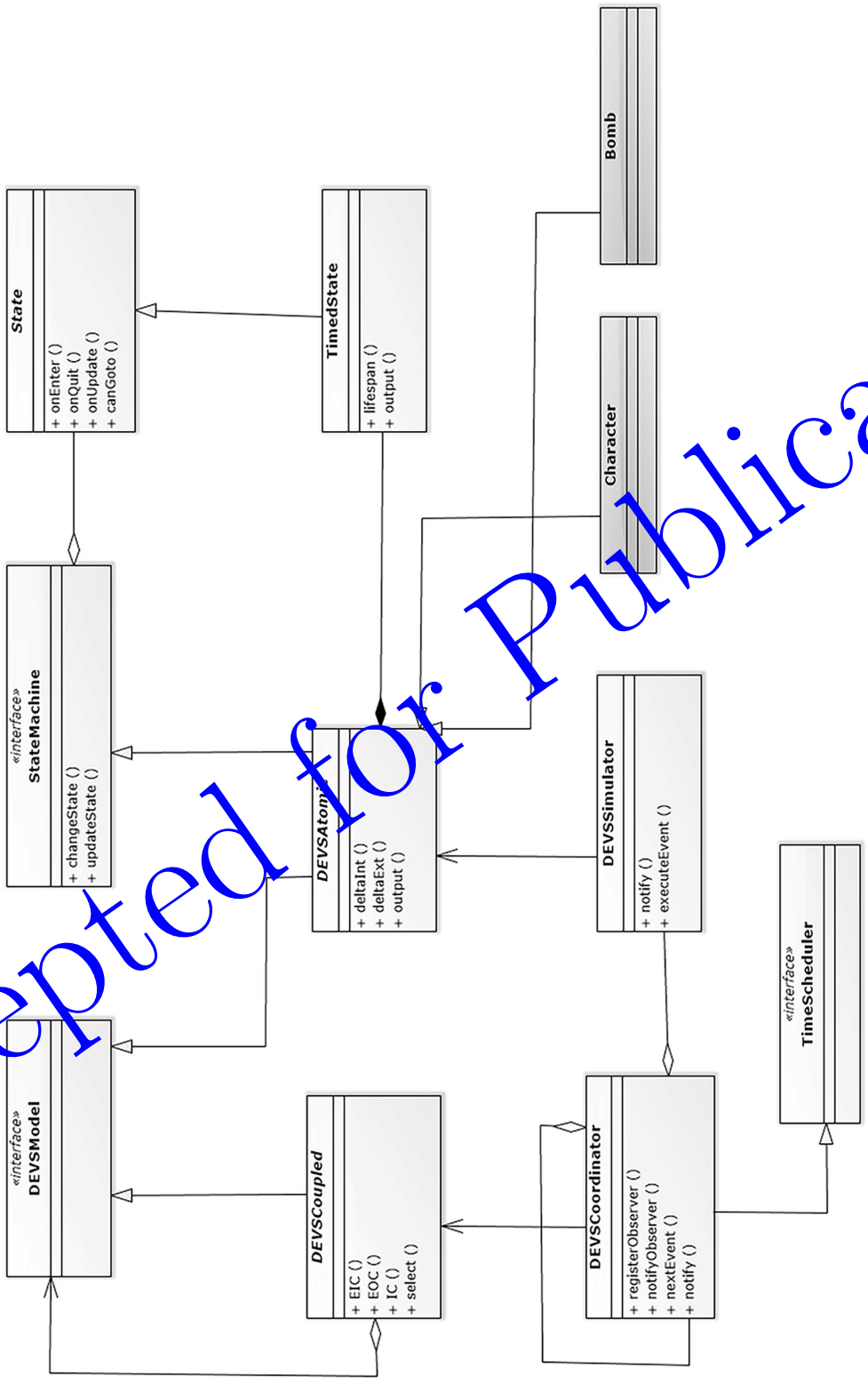

Figure 14. Class Diagram of the Simulation Model.