Simulation and Verification of [Dys]functional Behavior Models: Model Checking for SE

Charlotte SEIDNER & Jean-Philippe LERAT
SODIUS
6 rue de Cornouaille BP 91941
44319 Nantes Cedex 3, France
{cseidner, jplerat}@sodius.com
+33 2 28 23 60 60

Olivier H. ROUX
IRCCYN
1 rue de la Noë BP 92101
44321 Nantes Cedex 3, France
Olivier-h.roux@irccyn.ec-nantes.fr
+33 2 40 37 69 76

Copyright © 2010 by Seidner, Lerat & Roux. Published and used by INCOSE with permission.

Abstract. Verification is a key process in the dependability engineering of complex systems. As we have shown in earlier works, formal verification techniques such as *model checking* can be efficiently used in a Systems Engineering (SE) context, despite their inherent complexity. Considering the widely used Enhanced Function Flow Block Diagrams (EFFBDs), we have indeed developed a formal simulation and verification tool for these functional behavior models. Moreover, great care has been taken to conceal the processing complexity from the tool end-user.

In this paper, we present our latest developments as well as an extension of both method and tool to the case of dysfunctional models, to take into account failures affecting the model elements. By addressing both *fault removal* and *fault forecasting* problems with formal methods, we thus hope to improve the dependability analysis practice in SE.

Introduction: the why's and how's of formal verification

Verification actions are a major tool of dependability assessment and as such have always been a preeminent part of Systems Engineering processes (IEEE, 2005). Given today's trend to develop ever larger and more complex systems, verification has become a key element in the system design, all through its lifecycle.

A common practice for assessing the system safety is to perform either *tests* on the actual system or *simulations* on a behavioral model. However, this analysis cannot be exhaustive, even on "reasonably sized" systems, and carries the risk of missing potentially safety-critical situations in which the system is liable to endanger itself or its environment. To overcome this limitation, the designer may use a *formal method* such as *model checking*, where the identified properties are formally expressed and confronted to a formal model of the system, using efficient algorithms and data structures.

In previous work, we have shown that the inherent complexity of this formal method can be overcome and efficiently used in a SE context (SEIDNER *et al.,* 2008). Considering the widely used EFFBDs (LONG, 1995), we have indeed proposed their formalization and translation into a lower-level language, the time PETRI nets or TPNs (MERLIN, 1974). Using both the properties of our translation and a software tool dedicated to the model checking of behavioral properties on TPNs, we have developed an analysis platform allowing the verification of high-level properties on EFFBDs models. By hiding the underlying models, we have improved the overall efficiency and usability of this formal method by the system designer.

This paper focuses on the latest results, both theoretical and practical, that we have obtained

over the last two years. We have indeed developed a simulation tool which, used in conjunction with our verification tool, helps the designer with the visualization and understanding of the verification results, thus further enhancing the usability of our method. We have also extended our results to take into account the possibility for some model elements to be affected by *failures*. By shifting our work from purely functional architecture analysis to its dysfunctional counterpart, we move from *fault removal* to *fault forecasting*, two major *means* of attaining the dependability of a system (LAPRIE, 1992).

Related works. Over the last decade, the formal verification of high-level models has motivated a number of research work, essentially focused on the formalization of these models. The authors of (ANDRÉ *et al.,* 2007), for instance, have proposed a formal semantics of MARTE[1], a *Unified Modeling Language* (UML) profile dedicated to the description of real-time embedded systems. It is then possible to check the behavior of communicating systems with regards to some protocols (LE TALLEC *et al.,* 2009). Yet, and despite similar efforts, UML semantics is still incomplete (or contradictory) and largely software-oriented.

In this context, the Object Management Group (OMG) and the INCOSE have recently developed the *Systems Modeling Language* (SysML) as a subset of UML, adding such SE-specific notions as requirement management (SysML, 2008). Despite the recentness of the language, some research works on the formal verification of SysML diagrams have already been proposed. The authors of (EVROT *et al.,* 2008) give for instance a method for checking some safety properties on SysML models. However, their method is only semi-formalized and does not address the usability problem of such formal methods in a SE context.

Lastly, the *Architecture Analysis and Design Language* or AADL, described in (SAE, 2009), has been gaining importance over the last years as a powerful modeling tool by supporting the analysis of a number of formal properties such as reliability, deadlock detection, deadline management, etc. (RUGINA 2005; BERTHOMIEU *et al.,* 2009; RENAULT *et al.,* 2009). However, the language still suffers from semantics imprecision, lacks maturity and is still largely real-time and embedded system-oriented.

Outline of the paper. We first give and informal view of the EFFBDs and then describe their formalization and simulation. We then present the verification method and the software tool we have developed over the last two years. The next section proposes a description of the failures that can affect an EFFBD; it also extends the various results obtained so far. We then illustrate these results on the classic level-crossing problem. Finally, we conclude this paper by presenting some outlooks and prospects of our research and development works.

From the description of EFFBDs to their simulation

This section introduces the EFFBDs as the high-level models on which we later propose the verification of some complex behavioral properties. We give here a simplified version of its formal syntax and semantics, as established in (SEIDNER, 2009), and briefly present the simulation tool built from this formal description.

An informal description. System designers often use relatively simple graphical representations to efficiently describe both the functional structure of a system and the data flow exchanges. Among these representations, FFBDs (developed in the late 1950s) and the derived EFFBDs are some of the most widely used in Systems Engineering (LONG, 1995). Indeed, they provide the designer with an easy framework to describe the behavior of complex, distributed, hierarchical, concurrent and communicating systems. More specifically, they describe the *functions* performed by the system and the order in which they are executed.

[1] MARTE stands for *Modeling and Analysis of Real Time and Embedded systems*.

This order is specified through:

- the functions' dynamic *i.e.* their execution duration;
- the control environment, namely the control *constructs* or *structures*;
- the data environment (*items* and *resources*).

While functions are represented by rectangles, control structures are usually denoted by two complementary round *nodes* (see below). A large variety of control structures is available, such as:

- *parallel branches* (AND), including "kill branches" that force the termination of other branches in the parallel structure;
- *selection structures* (OR) and *multi-exit functions*;
- *iterations* (IT) and *loops*[2] (LP).

An EFFBD is organized in a main *scenario*; to support the modeling of hierarchical systems, functions can in turn contain sub-scenarios. In addition, EFFBDs model data flows; in this work, we merged the notions of both *item* and *resource* (distinct in the original formalism) into the concept of *flow*. A flow can be *consumed*, *produced* or *captured* by functions[3], and models such diverse elements as signals, finished products or CPU memory. Graphically, the relation between a function and a flow is represented by a weighted arrow.

Figure 1 shows an example of an EFFBD; the diagram does not correspond to an actual system but rather represents the language main features. To make the reading easier, flow quantities equal to 1 are not mentioned on the arrows modeling their exchange.

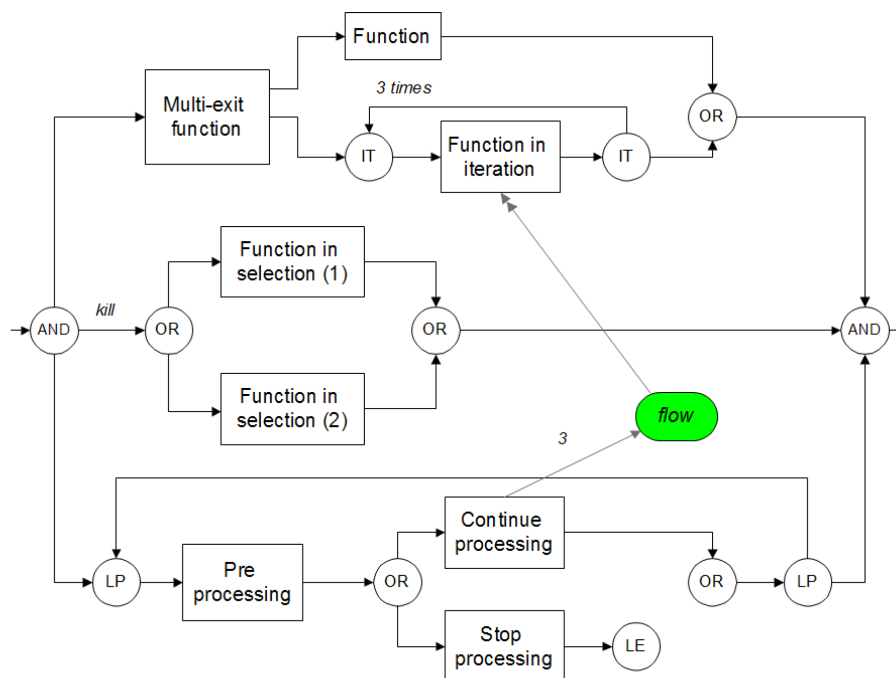

Figure 1: Example of an EFFBD, adapted from (LONG, 1995)

The behavioral semantics of an EFFBD is quite intuitive: informally speaking, it consists of a set of rules describing the course of the *control flow* along the structures, the execution of a function, the production of some flow, etc.

---

[2] Loops are infinite and can only be exited by reaching a *loop exit* (LE) structure.

[3] It can also be *checked*, that is read without being actually consumed by the receiving function.

For instance, the behavior of a parallel structure is the following:

- ❶ once the control flow has reached the opening AND node, the first structure on each parallel branch is activated;
- ❷ once the last structure of each parallel branch has finished its execution[4], the parallel structure is exited and the control flow transferred to the structure following the closing AND node.

Likewise, the behavior of a function, once enabled by the control flow, is the following:

- ❶ [if needed] wait for *all* the input flows to become available;
- ❷ [if needed] synchronously consume the input flows;
- ❸ start the execution;

After a certain duration, described by an interval:

- ❹ end the execution;
- ❺ [if needed] synchronously produce the output flows;
- ❻ transfer the control flow to the next structure.

Formalization of the EFFBDs. To our knowledge, no formal semantics had been given to the EFFBDs before our work, despite their intuitive behavior. Yet, this formalization is an indispensable step to any further formal analysis.

Regarding its syntax, an EFFBD can be seen as a set of:

- *nodes* modeling the control structures (including the functions);
- *flows* linked to the function nodes by weighted arcs denoting exchanged quantities;
- *intervals* giving for each function its minimum and maximum *execution duration*.

The behavior of an EFFBD is then described as a succession of *states*; each state represents:

- the *activity* of each node: it can be either **`inactive`**, **`enabled`**, **`executed`**[5] or, for functions only, **`executing`**;
- the *level* of each flow, *i.e.* the quantity of flow available in this state;
- the *time* elapsed since the beginning of the execution of each function.

Since time elapsing is here a continuous process, each given model corresponds to an infinity of possible executions. The set of all possible executions can be symbolically represented by a very low-level language known as a Timed Transition System or TTS (HENZINGER *et al.,* 1991).

More formally, the semantics of an EFFBD $\mathcal{E}$ is a tuple $(\mathcal{S}, s_0, \mathcal{N}, \rightarrow)$ where:

- $\mathcal{S}$ is the set of *states*, as described above;
- s_0 is the *initial state*;
- $\mathcal{N}$ is the set of *nodes*;
- $\rightarrow$ is the *transition relation* that formalizes the semantics rules described above by declaring how to proceed from a state to the following.

From this description, a complete version of which can be found in (SEIDNER, 2009), we were able to define the class of *bounded EFFBDs* for which the flow levels always stay finite. The practical implication of the boundedness of an EFFBD will be discussed in the next section. The formalization of the language was also the first step in the definition and implementation of a simulation tool, described in the following paragraph.

---

[4] Note that such a behavior may induce synchronization-waiting states.

[5] An **`executed`** node is located at the end of a parallel branch and is waiting for the completion of the other parallel branches.

An EFFBD simulation tool. As we have recalled in the introduction, simulating a system model cannot be, by itself, sufficient proof of its correct behavior. However, it provides the system designer with a fair insight upon its behavior, which is particularly needed in the early design phases of a complex system.

Therefore, we have implemented the formal semantics mentioned above into a simulation tool. This tool is included in MDWORKBENCH®[6], a design and analysis platform based on Eclipse®. A specific version has been developed for the French Department of Defense and is currently being deployed and evaluated. The global running of the simulation tool is illustrated on Figure 2.

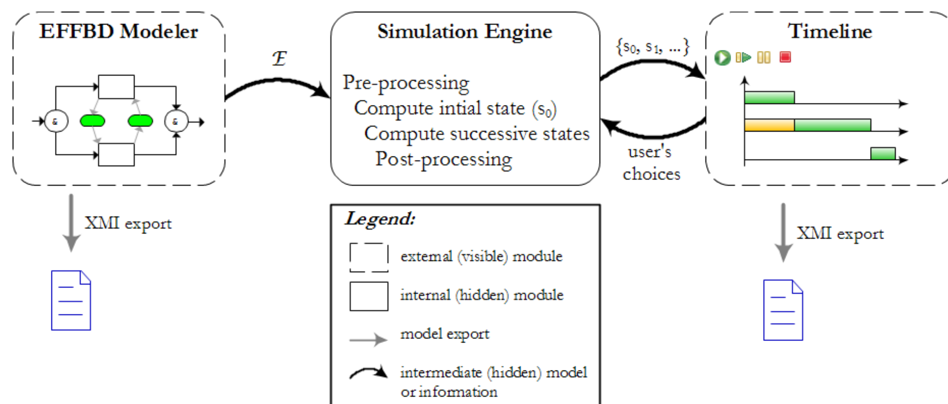

Figure 2: Description of the simulation tool

The left-most box represents the model editor (the dashes indicate that the designer has access to this module); it allows the user to describe the system, mostly graphically. The model $\mathcal{E}$ is then transferred to the simulation engine (center box) to compute a series $\{s_0, s_1, \ldots\}$ of states representing a possible execution of the model. This *run* is then displayed as a *timeline* or *chronogram* in the corresponding interface (right-most box). If needed, the user can perform on-the-fly *choices* to resolve flow conflicts, multiple choices in selection branches, etc. Moreover, the simulation can be played step-by-step or in a unique run. Finally, the simulation can be exported as an XMI[7] file so it can be later analyzed or printed.

Formal verification of EFFBD models

In this section, we present the main results and tools used to perform the formal verification of complex behavioral properties expressed on EFFBDs. The complete verification process, including the transformation into TPNs, was described in (SEIDNER *et al.,* 2008); in this paper, we focus on the system engineer's point of view by describing the global verification principles, the main property classes and finally our verification software tool.

General principles. From the formal semantics of the EFFBDs, it would be possible to design and develop a model checker that directly analyzes the high-level models. However, we chose to propose instead a translation of the EFFBDs into a lower-level model, the TPNs. This step indeed allowed us to take advantage of the numerous works that have been carried

[6] See http://www.mdworkbench.com for further details.

[7] XMI is the OMG *XML Metadata Interchange* standard used for exchanging metadata information via the Extensible Markup Language (XML).

out on the TPNs and most notably of the ROMÉO model checker[8], a software tool designed to check complex quantitative temporal properties on TPNs.

We designed a structural translation which creates elementary TPNs for each node and flow, then connects the patterns together. We have proved that the behavior of an EFFBD and its corresponding TPN are *equivalent* with regards to the *strong temporal bisimulation* relation. In other words, any timed behavioral analysis (including model checking) can be performed on an EFFBD or on its TPN counterpart *without information loss*, which justifies our approach.

Logical property classes. Before illustrating this fundamental result with our verification tool, we first need to describe the safety and vivacity properties that can be expressed and checked by it[9]. These properties are expressed using the Timed Computation Tree Logic or TCTL (ALUR *et al.,* 1990), a powerful yet complex formalism.

To allow for an efficient use of this logic by the system designer, we have chosen to limit the logic expressivity by providing six high-level property classes, described and manipulated in *natural language*. Should the need arise, other property class could be added, provided that it can indeed be expressed with the TCTL logic.

The existing classes are presented in Table 1 along with their parameters and meaning.

Table 1: Behavioral property classes

	Property class	Parameters		Meaning
I	Completing execution	S	main scenario	*"S always reaches its final state"*
II	Timed completing execution	S t	main scenario temporal bound	*"S always reaches its final state in less than t time units"*
III[10]	Strict temporal decomposition	F t	decomposed function temporal bound	*"The sub-scenario associated with F is always executed in less than t time units"*
IV	Mutual exclusion	F_i	set of functions	*"Only one F_i function, at most, can be executed at any given time"*
V	Bounded response	F_1 F_2 t	triggering function triggered function temporal bound	*"Executing F_1 leads to the execution of F_2 in less than t time units"*
VI	Flow boundedness	f m	flow maximum level	*"At any time, the level of flow f is lower or equal to m"*

For instance, a class I property allows the detection of a *system blockage*, most likely due to a function waiting for an input flow that will never be produced. A class VI property can test the *k-boundedness* of the model (where *k,* a positive integer, is specified by the system engineer). Such a property is particularly helpful when dimensioning the system since it states that no flow level is greater than *k*. Moreover, an unbounded EFFBD can be the consequence of a badly designed model (as most systems are physical and intrinsically bounded) and the non verification of the k-boundedness should alert the architect on a flaw in the system design.

From a theoretical point of view, we have shown that the model checking of these properties is a *decidable* problem for bounded EFFBDs and that it belongs to the class of PSPACE-complete problems[11].

---

[8] The software tool is available at http://romeo.rts-software.org

[9] A *safety* property means that a "bad situation" will *never* occur; a *vivacity* property that a "good situation" will *eventually* occur.

[10] This property is actually a generalization of the class II property.

[11] For further details on decidability and complexity problems, see for instance (Garey *et al.,* 1979).

An EFFBD verification tool. As mentioned above, we have implemented these results in MDWORKBENCH®, as illustrated by Figure 3 (the legend is the same as in Figure 2).

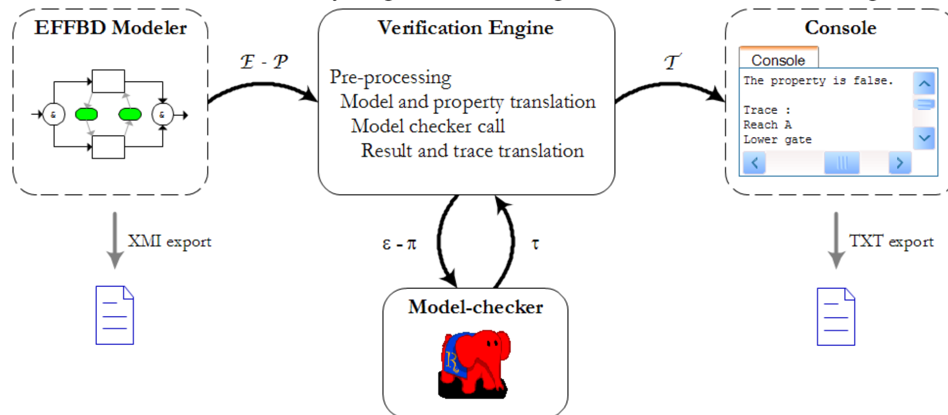

Figure 3: Description of the verification tool

As earlier, the left-most box represents the model editor; the model $\mathcal{E}$ and the property $\mathcal{P}$ are transferred to the verification engine to be transformed into equivalent, low-level model and property ε and π. These elements are then analyzed by the ROMÉO model checker (bottom) which computes the truth value of π. If the property is not true, it provides a run or *trace* τ as a sequence of TPN *transitions* leading to a state contradicting π.

Finally, this trace is translated back by the verification tool into a high-level trace $\mathcal{T}$, which is a sequence of EFFBD functions executions, ending on a state contradicting *P*. This result is then displayed in a console (right-most box) and can be exported as a text file. It should be noted that the complexity of the treatment is completely hidden from the system designer, who only manipulates the high-level objects.

From functional to dysfunctional models

The previous sections have described the system from a *functional* point of view only; the verification of such models is indeed a major process in *fault removal*, one of the mean of attaining the dependability of a system. This dependability can also be reached through *fault forecasting*, where faults or *failures* are injected into the model and analysis are performed on the *dysfunctional* architecture.

In this section, we first propose a description of the failures that can affect either the functions or the flows of an EFFBD. We then extend the behavioral properties defined earlier as well as the main results presented above.

Failures description. The failures we chose to model are largely inspired by the usual *failures modes*, described in such methods as the *Failure Modes and Effects Analysis* (FMEA). Failures can affect a function or a flow and are either *permanent* or *transient*. In the latter case, an occurrence probability is added to the failure description. We consider seven failure types; the five first affect a function:

- ***(i)*** *No activation:* when reached by the control flow, the function is neither enabled nor executed; the control flow is not transferred to the next structure.
- ***(ii)*** *Infinite duration:* once the execution has started, it cannot end.
- ***(iii)*** *Modified duration:* the execution starts and ends normally, except its duration now belongs to a new interval (described by the failure parameters).
- ***(iv)*** *No outputs:* at the end of its execution, the function transfers the control flow without

producing its output flows.

(v) No control transmission: at the end of its execution, the function produces its output flows but does not transfer the control flow.

A type ***(i)*** failure corresponds to the *"no start"* generic failure mode while type ***(ii)*** is a *"no end"* failure mode.

Two failure types can affect the flows:

(vi) Modified initial level: the initial level of the flow is given by a new value.

(vii) Modified produced quantity: the flow quantity produced by the function is given by a new value.

A type ***(vii)*** failure is actually a refinement of type ***(iv)*** and affects a function-flow couple.

Extending the properties. We have extended the formal syntax and semantics of the EFFBDs to take the failures into account. In particular, a function can now be in the `failed` activity. We have also described four more high-level property classes, given in Table 2. In the following, we assume that all failures are of type ***(i)*** and that they affect functions F and F_1 to F_n.

Table 2: Extension of the properties

	Property type	Parameters		Meaning
VII	Reliability	F	function	*"F never fails"*
VIII	Recovery	F t	function temporal bound	*"F never stays in the* `failed` *activity more than t time units in a row"*
IX	Bounded response after failure	F, G t	functions temporal bound	*"The failure of F always triggers the execution of G in less than t time units"*
X	Failure mutual exclusion	F_i	set of functions	*"Only one function F_i, at the most, can fail at the same time"*

For instance, a class X property can test the robustness of a two-branch redundant architecture, where at last one redundant branch must not fail.

Main results. From the various formal descriptions we developed, we were able to extend the elementary patterns to translate the dysfunctional EFFBDs into TPNs. We were also able to extend the simulation and verification tools; the model checking of the properties is still a decidable and PSPACE-complete problem for bounded EFFBDs. Additionally, the simulation tool allows some failure *traceability* since any failed function execution or flow production can be graphically marked and linked to the description of the failure (see below).

The level-crossing problem

We have chosen to illustrate these results on the classical level-crossing problem. This section voluntarily presents a simplified version of the problem: indeed, we want to hint with this example at the expressivity and power of the method, while keeping a discourse as clear as possible.

We first discuss the system and its modeling, including two function failures, and then describe two safety and vivacity properties. Finally, we show the simulation and verification results obtained on the dysfunctional model.

System and model description. We focus here on the common half-barrier level-crossing (LC). It consists of two half-barriers, bells and red flashing lights on either side of the road. The gates are surrounded by the *short zone* (see Figure 4) which is preceded by a longer *approach warning zone*. A pedal located at point A signals any approaching train; another

pedal in E detects the last train carriage and commands the reset of the LC. A fuller description of the system and modeling hypothesis can be found in (SEIDNER, 2009).

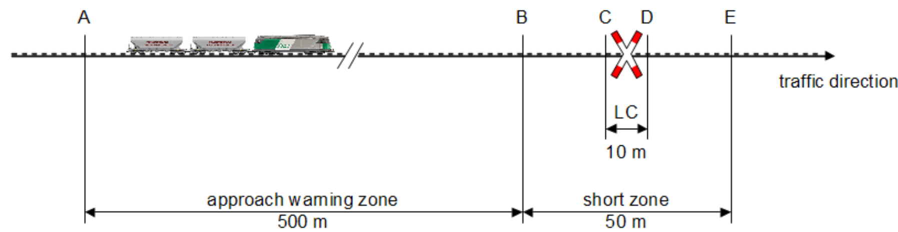

Figure 4: Schematic plan of the railroad level-crossing

Once the pedal in A is activated, the events are the following:

- ❶ flash the lights and start ringing the bells;
- ❷ after 5 seconds, lower the gates;
- ❸ once the gates are down (after 5 to 10 seconds), stop the bells;
- ❹ once the pedal in E is activated, raise the gate;
- ❺ once the gates are up (after 10 to 15 seconds), turn the lights off.

We assume here that the trains are 190 meters long and that they can travel at a speed (not necessarily constant) comprised between 10 m/s and 20 m/s (about 35km/hr or 20 mph and 70 km/hr or 45 mph). In addition, the zone between points A and E is a one-way single track. More precisely, we suppose that the track is looping, that only one train is on it and that it takes at least 2 minutes for the train to get back from E to A. It is therefore unnecessary to model any LC access protection system.

Finally, we suppose that the road users are respectful of the traffic regulations. Therefore, we only have to model the train (which can be seen as the LC environment), the controller and the gates. The resulting model is illustrated by Figure 5; when not equal to [0, 0], the duration intervals are directly mentioned on the functions. The time unit is the second and, to make the reading easier, the flashing light control is not modeled.

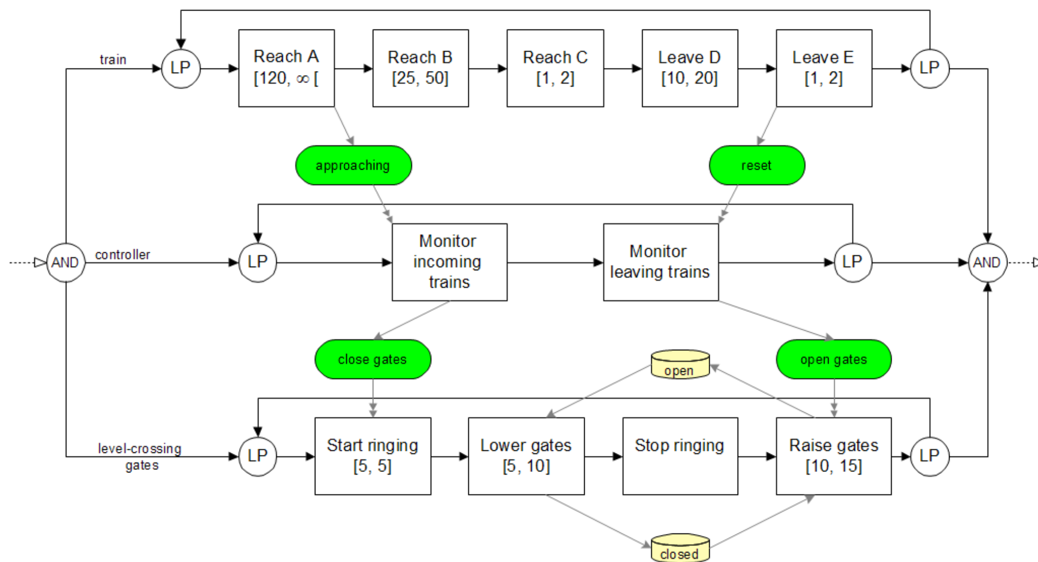

Figure 5: EFFBD model of the level-crossing problem

We have adopted a classical parallel architecture where each branch represents one "organ"

of the system or its environment[12]. Each branch communicates and is synchronized with the others through flows. For instance, the approach of a train (through the activation of the pedal in A) is modeled by the flow **`approaching`**, produced by **`Reach A`** and triggering the execution of **`Monitor incoming trains`**. Note the **`open`** and **`closed`** flows: they describe the aperture state of the gates (as the gates are initially raised, the level of **`open`** is initially 1 while the initial level of **`closed`** is 0). Note also that the train is on the level-crossing exactly when the function **`Leave D`** is executing.

Finally, we suppose that two failures can affect the system:

F1. "Due to a wrong modeling, the gates always take 30% more time than expected to reach the low position"

F2. "Trains have a 1% chance of staying blocked on the level-crossing"

We thus add the following failures:

- F1 is a permanent type ***(iii)*** failure affecting function **`Lower gates`** and is characterized by the new temporal interval [7; 13];
- F2 is a transient type ***(ii)*** failure affecting function **`Leave D`**, with a 0.01 probability of occurrence.

Properties description. We can now describe the two following behavioral properties:

P1. "To ensure the safety of the LC and its environment (including the road users), the barriers must be closed if a train is on the LC"

P2. "To keep a smooth road traffic, the gates should not stay closed more than 90 seconds in a row"

P1 is a safety property and can be modeled by a type IV property (mutual exclusion) between functions **`Leave D`** and **`Raise gates`**. P2, a vivacity property, can be modeled as a type V property (bounded response) where the triggering function is **`Lower gates`**, the triggered function **`Raise gates`** and the temporal bound 90 seconds.

Main results. In this last paragraph, we focus on the analysis of the dysfunctional model. Moreover, we wish to recall that the properties described above and the model itself are simple enough to be checked "by hand": again, our goal here is to give an insight into the power of the method.

P1 is verified: indeed, the functional model (and the system itself) is designed to enforce the safety property. The model is robust enough to allow for the large imprecision on the duration of **`Lower gates`** caused by F1. Finally, the occurrence of F2, where the train is blocked on the LC (thus blocking the gates in their lower position) only reinforces the property.

On the other hand, the vivacity property is not verified. Indeed, when the train is blocked on the LC due to the occurrence of failure F2, the whole model comes to a deadlock and the gates can no longer be raised, thus invalidating the property. A corresponding trace, given by Table 3, is thus computed by our verification tool. As Reach D never ends, due to the occurrence of F2, it is not part of the trace.

Table 3: An execution trace of the level-crossing model

Ending function
`Reach A`
`Monitor incoming trains`
`Start ringing`
`Lower gates`

[12] Needless to say, the modeling of this system is not unique.

Stop ringing
Reach B
Reach C

A simulation run of the system is illustrated by Figure 6 (to make the reading easier, we shifted the timeline to let is start after 120 seconds). To simplify the reading, no flows except **open** and **closed** are represented; they are marked by blue blocks. The execution of a function is marked by a green block while the wait for some input flow is marked in orange. Note that, during the execution of **Lower gates** (between 160.5 time units and 173.0 time units), the gate is neither open nor closed: it is indeed lowering.

Symbols are added to the blocks to mark the occurrence of a failure. For instance, F1 causes the function **Lower gate** to have a modified duration, which is denoted by a warning sign and a red clock symbol on the right of the execution block. Likewise, the infinite duration of function **Leave D**, caused by the occurrence of F2, is denoted by a ∞ symbol (also mentioned on the date line). Additionally, a contextual menu linking to the failure description can be obtained by leaving the mouse on the block.

The combination of the simulation and verification tools thus helps the system designer to understand the trace results given by the model checker, ultimately enhancing the use of the formal verification method.

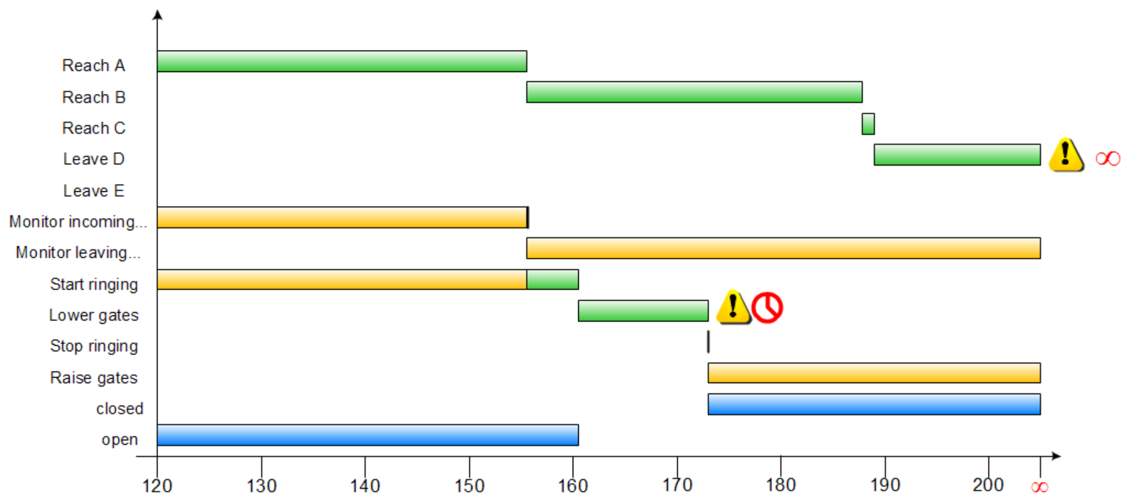

Figure 6: A possible timeline for the level-crossing model

Conclusion and further work

In this paper, we have shown that formal methods such as model checking can be efficiently used in a SE context and can contribute to the global dependability of the system. Despite their initial lack of formal syntax and semantics, we have indeed demonstrated that EFFBDs can support the checking of complex safety and vivacity properties. These results have led to the implementation of a simulation and verification software tool. We have also shown that these tools do not require, from the designer, the knowledge of the underlying models and concepts. Finally, we have taken advantage of the simple and intuitive behavior of the EFFBDs to propose some semantics extensions to describe and analyze the occurrence of failures.

Depending on the feedback we will receive on MDWORKBENCH® (currently tested and deployed industrially), we propose to further extend the semantics of the EFFBDs so as to

model more complex behaviors (such as continuous and asynchronous flow exchanges) as well as new property and failure classes.

References

ALUR R., COURCOUBETIS C. A. and DILL D. L. June 1990. Model-checking for real-time systems. *5th IEEE Symposium on Logic in Computer Science*, p. 414 – 425.

ANDRÉ C., MALLET F. and DE SIMONE R. Sept. 2007. Modeling of immediate vs. delayed data communications: from AADL to UML MARTE. *ECSI Forum on Specification & Design Languages (FDL)*, p. 249 – 254.

BERTHOMIEU B., BODEVEIX J.-P., CHAUDET C., ZILIO S. D., FILALI M. and VERNADAT F. June 2009. Formal verification of AADL specifications in the Topcased environment. *14th Ada-Europe International Conference on Reliable Software Technologies*, LCNS(5570), p. 207 – 221.

EVROT D., PÉTIN J.-F. and MOREL G. 2008. Combining SysML and formals methods for safety requirements verification. *Insight Journal of INCOSE*, 11(3), p. 21 – 22.

GAREY M. R. and JOHNSON D. S. Jan. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.*

HENZINGER T. A., MANNA Z. and PNUELI A. June 1991. Timed transition systems. *Real-Time: Theory in Practice (proceedings of the REX Workshop)*, LCNS(600), p. 226 – 251.

IEEE. Sept. 2005. *IEEE Std™ 1220- 2005: IEEE Standard for Application and Management of the Systems Engineering Process*.

LAPRIE J.-C. 1992. Dependability: basic concepts and terminology. *Dependable computing and fault-tolerant systems*.

LE TALLEC J.-F. and DEANTONI J. Oct. 2009. Multi-level designing approach. *3rd Junior Researcher Workshop on Real-Time Computing (JRWRTC 09)*, p. 23 – 26.

LONG J. July 1995. Relationships between common graphical representations in Systems Engineering. *5th International Symposium of the INCOSE*.

MERLIN P. M. 1974. *A study of recoverability of computing systems*. Ph.D. thesis, University of California, Irvine, CA.

RENAULT X., KORDON F. and HUGUES J. June 2009. Adapting models to model checkers, a case study: Analysing AADL using time or colored Petri nets. *IEEE International Symposium on Rapid System Prototyping*, p. 26 – 33.

RUGINA A.-E. Nov. 2007. Dependability *modeling and evaluation – From AADL to stochastic Petri nets*. Ph.D. thesis, Institut National Polytechnique de Toulouse, France.

SEIDNER C., LERAT J.-P. and ROUX O. H. June 2008. Usability and usefulness of formal verification in a system design process. *18th International Symposium of the INCOSE.*

SEIDNER C. Nov. 2009. *EFFBDs Verification: Model-checking in Systems Engineering.* Ph.D. thesis, Université de Nantes, France.

SAE. Jan. 2009. *AS5506: Architecture Analysis & Design Language (AADL).*

SysML Finalization Task Force. Nov. 2008. *OMG Systems Modeling Language (OMG SysML™) version 1.1.*

Biographies

Charlotte SEIDNER has obtained her Ph.D. degree in Control Theory and Applied Computer Science in 2009 with a dissertation entitled “EFFBDs Verification: Model Checking in Systems Engineering”. She also graduated from the École Centrale of Nantes with a M.S. in Engineering (2005) and with a M.Res. in Control Theory and Computer Science (2006).

Since 2006, she works as an R&D engineer with SODIUS; her research themes include the development of robust tools for the formal verification and simulation of high-level models.

Jean-Philippe LERAT is an engineer in Data Processing, with a specialization in systems engineering and integration. Since 1983, he has been working in several domains, including robotics and telecommunications.

He now runs his own company, SODIUS, devoted to assisting enterprises with the design of systems. He also leads R&D efforts in the field of model transformation and automated code and documents generation for system, software and hardware. He provides worldwide expertise and training to many projects and engineers in various fields like Space industry, Defense, Automotive etc. He is active at an INCOSE level since 1996 and a member of the AFIS.

Olivier H. ROUX is an Assistant Professor (“maître de conférences”) at the University of Nantes; since 2006, he also holds an HDR (accreditation to supervise Ph.D. students). He obtained his Ph.D. degree in 1994, and has held positions at the École Centrale of Nantes before joining the University of Nantes in 1998.

He carries out his research activity at the Research Institute for Communications and Cybernetics in Nantes (IRCCyN). His research themes include verification using model-checking techniques and control issues on timed systems. He has a particular interest in time Petri nets and timed automata as well as in their stopwatch extensions.