Metadata of the chapter that will be visualized in SpringerLink

Book Title	Model-Driven Engineering and Software Development	
Series Title		
Chapter Title	Verification and Simulation of Time-Domain Properties for Models of Behaviour	
Copyright Year	2021	
Copyright HolderName	Springer Nature Switzerland AG	
Author	Family Name	**Carrillo**
	Particle	
	Given Name	**Miguel**
	Prefix	
	Suffix	
	Role	
	Division	Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
	Organization	Universidad Nacional Autónoma de México
	Address	Apdo. 20-126, 01000, México D.F., Mexico
	Email	
	ORCID	http://orcid.org/0000-0003-2105-3075
Corresponding Author	Family Name	**Estivill-Castro**
	Particle	
	Given Name	**Vladimir**
	Prefix	
	Suffix	
	Role	
	Division	School of Information and Communication Technology
	Organization	Griffith University
	Address	Brisbane, QLD, 4111, Australia
	Email	v.estivill-castro@griffith.edu.au
	ORCID	http://orcid.org/0000-0001-7775-0780
Author	Family Name	**Rosenblueth**
	Particle	
	Given Name	**David A.**
	Prefix	
	Suffix	
	Role	
	Division	Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
	Organization	Universidad Nacional Autónoma de México
	Address	Apdo. 20-126, 01000, México D.F., Mexico
	Email	
	ORCID	http://orcid.org/0000-0001-8933-8267

Abstract

Modelling and simulation are techniques instrumental in the engineering and design of complex systems. The reason is that both these techniques can anticipate possible failures when corrections are less costly to incorporate. Nevertheless, a correct behaviour is no guarantee, especially with software systems and their ubiquitous modelling notation: state machines. Correctness cannot be guaranteed because semantic gaps result from (1) abstractions in modelling and (2) ambiguities in simulation. Formal verification of a model may thus imply little about the correctness of the implementation. This situation is all the more serious with the emergence of Model-Driven Software Engineering and its penetration in the instrumentation of cyber-physical systems, where verification of time-domain properties of systems is now paramount. We use logic-labelled finite-state machines (LLFSMs), a formalism with a precise semantics. We introduce both model-to-model and model-to-text transformations from LLFSMs to either programming languages or formal-specification languages for model checkers with minimal semantic gaps. We describe a transformation in the Atlas Transformation Language (ATL), producing modules of the NuSMV model checker. The time complexity of this transformation is linear in the total number of states of an arrangement of LLFSMs. The transformation is so faithful that the model checker itself can be used as the execution engine of the LLFSMs models.

Keywords (separated by '-')	Formal verification - Model-to-model transformations - State machines

Verification and Simulation of Time-Domain Properties for Models of Behaviour

Miguel Carrillo[1], Vladimir Estivill-Castro[2](✉), and David A. Rosenblueth[1]

[1] Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas, Universidad Nacional Autónoma de México, Apdo. 20-126, 01000 México D.F., Mexico
[2] School of Information and Communication Technology, Griffith University, Brisbane, QLD 4111, Australia
v.estivill-castro@griffith.edu.au

Abstract. Modelling and simulation are techniques instrumental in the engineering and design of complex systems. The reason is that both these techniques can anticipate possible failures when corrections are less costly to incorporate. Nevertheless, a correct behaviour is no guarantee, especially with software systems and their ubiquitous modelling notation: state machines. Correctness cannot be guaranteed because semantic gaps result from (1) abstractions in modelling and (2) ambiguities in simulation. Formal verification of a model may thus imply little about the correctness of the implementation. This situation is all the more serious with the emergence of Model-Driven Software Engineering and its penetration in the instrumentation of cyber-physical systems, where verification of time-domain properties of systems is now paramount. We use logic-labelled finite-state machines (LLFSMs), a formalism with a precise semantics. We introduce both model-to-model and model-to-text transformations from LLFSMs to either programming languages or formal-specification languages for model checkers with minimal semantic gaps. We describe a transformation in the Atlas Transformation Language (ATL), producing modules of the NuSMV model checker. The time complexity of this transformation is linear in the total number of states of an arrangement of LLFSMs. The transformation is so faithful that the model checker itself can be used as the execution engine of the LLFSMs models.

AQ1

AQ2

Keywords: Formal verification · Model-to-model transformations · State machines

1 Introduction

"*Systems are inherently complex, and tools such as modelling and simulation are needed to provide the means for gaining insight into aspects of their behaviour*" [6].

A previous version of this article appeared in MODELSWARD 2020.

With the emergence of Model-Driven Software Development, there is an expectation that, for complex software systems, high-level designs would automatically translate into implementations. The Unified Modeling Language (UML) is possibly the most frequently used language for modelling the behaviour of software systems as arrangements of state machines, and this includes the internet of things (IoT) devices, embedded systems, and smart things [9]. UML state charts adopted and popularised the *Run To Completion (RTC)* event-driven semantics [20,24,39]. Moreover, UML has elaborated (on top of the RTC) the handling of events for all sorts of other compositions with other behaviour models [25]. Unfortunately, such enlargement of UML's artifacts [13,21] has resulted in the widening of semantic gaps (larger discrepancies between modelling languages and their interpretation; both, in the verification or in the execution) and in more elaborate constructs, that many admit they rarely use. For instance, introductory videos [1] on using Papyrus [26] admit ignoring most of the options and elements offered by the modelling tools. In particular, for state machines, it is suggested [23] that the main use of models is their visualisation aspect, ignoring completely the executability.

Simulations that do not exhibit failure are no guarantee of correct behaviour. Hence, there is significant interest to enable formal verification of models. Moreover, with the emergence of Model-Driven Software Engineering and its penetration in the instrumentation of cyber-physical systems, formal verification of time-domain properties [29] (beyond value-domain properties) of systems is now paramount.

In this paper, we report on model-to-model (M2M) transformations and model-to-text (M2Text) transformations directly producing a Kripke structure for model checking in the SMV language [34], which is the core input language to the NuSMV [14] and nuXmv [12] model checkers. The transformation is so faithful that the model checker itself can be used as the execution engine of the LLFSMs models. That is, we minimise the semantic gaps. Moreover, we use SMV's module notation achieving extremely succinct input files for the model checkers. For example, compared with a former implementation [33] of the generation of the input files for the model checker, instead of over 110,000 lines of code, we now generate fewer than 500.

Our contributions are as follows.

1. We detail M2M transformations that reduce the need for sections with states (see Subsect. 5.1).
2. In Subsect. 5.2, we detail M2M transformations that handle the atomicity of actions in states for the semantics of LLFSMs when translated to SMV.
3. We automatically generate verification properties of the model (see Subsect. 7.1).
4. We demonstrate that the transformations are efficient (see Subsect. 7.3).

At mipal.net.au/downloads.php, we release the code for prototype implementation of these transformations. Figure 1 illustrates the M2Text transformations released with the software. An accompanying video[1] illustrates one case study

[1] www.youtube.com/watch?v=o2Ut5lAsJe8&feature=youtu.be.

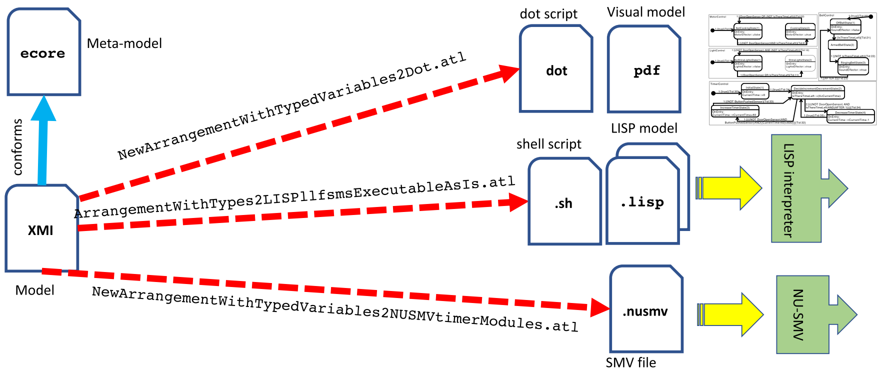

ecore
Meta-model
conforms
XMI
Model
NewArrangementWithTypedVariables2Dot.atl
ArrangementWithTypes2LISPllfsmsExecutableAsIs.atl
NewArrangementWithTypedVariables2NUSMVtimerModules.atl
dot script
dot
Visual model
pdf
shell script
.sh
LISP model
.lisp
LISP interpreter
.nusmv
SMV file
NU-SMV

Fig. 1. ATL model-2-text transformations released at mipal.net.au/downloads.php.

where the model is interpreted by the NuSMV model checker and also by our own interpreter written in LISP. Unlike the conference paper [11], this paper offers a direct focus on verifying time-domain properties. In particular, in Section 6 we include a new case study of the garage door controller [2]. This case study has important time-domain aspects. Our transformation in this version can handle integer variables and predefined constants. Most importantly, the model-2-text transformations produce computation-tree logic (CTL) and linear-time logic (LTL) properties about the scheduling of modules and the proper execution of the arrangement (see Subsect. 7.1). This aspect is a fundamental step in proving the model-2-text transformation is in itself correct. With respect to the prototypes of implementing the transformations, we now release the model-2-model transformation in the ATL transformation language that eliminates the need for sections in states (see Sect. 5.1). We also release EMF-Java tools that use the generated classes from the meta-model to produce a graphical user interface (GUI)-enabled emulator of traces produced by the NuSMV [14] and nuXmv [12] model checkers. We hence establish the minimisation of the semantic gap between the verified model and the execution. Moreover, in Sect. 3, we provide a full mathematical definition of LLFSMs, which directly delivers the corresponding meta-model.

2 Adopting Logic-Labelled Finite-State Machines

Despite the attempts to formulate behaviour models as UML state machines, the interpretation of these models still finds semantic variants, so that even in the value domain, properties submitted for formal verification may or may not hold, depending on the particular semantic variant [3]. This discrepancy is a manifestation of a first type of semantic gap, so that the executable model in simulation runs differently from the code generated from the model. A second type of semantic gap occurs when the input for a model checker is produced from the model using simplifying assumptions to ensure that formal verification is feasible (this is the case of translations that assume the execution will always have no bound on the available time to resolve the current event and therefore does not produce the representation of event queues or other mechanisms implied by the RTC semantics). For instance, STP [4,16] converts STATEMATE's event-driven state-charts into SMV by converting events into Boolean event variables that are `true` for exactly one time step. In effect, the model being verified is some form of logic-labelled finite-state machine which is no longer equivalent to the original RTC semantics. Other researchers have found ambiguities in UML's semantics [36]. For instance, in cases where several transitions are enabled, some researchers have elected to chose randomly which transition shall fire (which casts serious doubts about the semantics), while others prefer to keep a record of the editing of the model (and assigning priorities to transitions on such invisible criteria as the order of inserted transitions while editing the model [36]). In the time domain, verification with UPPAAL [31] (i.e., timed automata) verifies time bounds once the system starts processing an event. Verification with timed automata does not consider the amount of time that the system has to wait for an event to happen. Thus, verification is limited to best-case scenarios, where, for all the configurations of event queues, the current event is not affected by timing deadlines. We attribute the mismatch between UML being formally verifiable and UML being executable to the original goal of UML to be both human centred and a tool for communication between human designers, allowing a significantly loose semantics, with little intention to be executable [13].

A third semantic gap emerges when the constructs by Model-Driven Software Development are translated by programmers or machines into programming languages with significantly different constructs: the verified model would have even less resemblance with the running program:

> "*The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness*" [18]

(also cited by Edmund M. Clarke et al. [15]).

We elect here to use arrangements of logic-labelled finite-state machines (LLFSMs) as the constructs to model behaviour. As a first approximation, we can imagine LLFSMs as UML's state charts where transitions are not labelled by events, but only by guards (without side effects). Thus, each transition is labelled only by a Boolean expression. This simple change has profound implications (a summary appears in Table 1).

with the advantage that the computational power is not lost. More precisely, LLFSMs are extended finite-state machines where the user has the liberty to choose the action language. In addition, they can be executed concurrently, but instead of uncontrolled concurrency, the scheduler is predefined ahead of execution. The result is a sequential executable semantics without the earlier semantic problems of the RTC semantics of the event-driven UML modelling tools.

> "*The more complex a system is, the more important it is to make it as simple as possible. In complex systems, simplicity isn't achieved by coding tricks. It's achieved by rigorous thinking above the code level*" [30].

Table 1. Comparison of behaviour models.

LLFMSs	*UML state charts*
Transitions labelled by logic expressions	Transitions labelled by events
Analogous to time-triggered (explained in text)	RTC semantics
Sequential semantics (but concurrent)	fUML attempts to produce executable semantics
Executable and verifiable models	Semantic gaps reflected in diverse execution models
Scale up by concurrent execution in arrangement	State nesting is one of the fundamental mechanism for composition
Can produce deliberative system	Usually leads to the confusion between reactive system, event-driven system and real-time system
Communication with control/status messages	`send` instantaneous semantics

Moreover, for formal verification, the RTC semantics requires model checkers to consider all possible sequences of arrival of events and all possible states of the queue (or queues) that handle(s) the events. Such combinatorial explosion severely limits the potential for model checking of elaborate models.

Logic-labelled state-machines can be considered an alternative to event-driven models in an analogous fashion as time-triggered architectures are an alternative to event-driven systems. However, when timing issues are critical, a time-trigger architecture is fundamental:

> "*The safety properties of time-triggered architectures can be formally verified (which is extremely difficult with event-triggered systems)*" [22].

Rushby [37,38] showed that time-triggered architectures are resilient and safe in all operating circumstances. Thus, time-triggered architectures guide the design of international safety standards such as IEC 61508 (industrial systems), ISO 26262 (automotive systems), IEC 62304 (medical systems) and IEC 60730 (household goods) [35].

Although LLFSMs can be scheduled as a time-triggered systems, strictly speaking, LLFSMs are not identical to time-triggered systems (see Table 1). The reason is that a "ringlet"s (Definition 8 in Sect. 3) duration is not uniform across different machines. That is, the time it takes to run a scheduler's turn varies depending on (the current state of) the machine in turn. This variation is caused by the fact that some machines may have more statements in their section than others, or that a state may have more transitions than others.

In contrast with both LLFSMs and time-triggered architectures, event-driven systems imply managing queues to store events arriving while handling the current event. Such event-driven architectures exhibit many issues regarding time guarantees [17,28]. But more importantly, in the value domain, all possible orders of events and queue's configurations must be verified (such combinatorial explosion is impractical).

3 Formal Definition of LLFSMs

An *arrangement* of LLFSMs is a sequence of state machines operating in an environment. The order of the state machines in such a sequence defines the execution schedule of such state machines.

Definition 1. *An arrangement $A = (Q, E, W)$ is a triple, where*

1. *Q is a sequence $\langle M_1, M_2, \ldots, M_k\rangle$ of k logic-labelled finite state machines,*
2. *$E = E_s \cup E_e \subseteq \mathcal{V}$ is a set of variables, called external variables, formed of two disjoint sets: the set E_s of sensor (input) variables and the set E_e of effector (output) variables, and*
3. *$W \subseteq \mathcal{V}$ is a set of variables, called shared or whiteboard variables.*

To identify the parts of an arrangement A we use a dot, so $A.Q$ is the sequence of state machines, but if A is understood, we simply write Q.

A *logic-labelled finite state machine* (LLFSM) is a state machine whose transitions are labelled by logic expressions, typically Boolean expressions in an action language. Therefore, as long as there is an effective procedure to evaluate the expressions (even in a multi-valued logic), the formalism applies. (For instance, LLFSMs have been used with defeasible logic [5].) We therefore do not elaborate on the semantics of the logic expressions labelling the transitions, but will refer to them as Boolean expressions.

Definition 2. *A logic-labelled finite-state machine (LLFSM), or machine for short, is a tuple $M = (\mathcal{S}, \mathcal{T}, I, L, \mathcal{F})$, where*

1. *$\mathcal{S}$ is a set of states,*
2. *$\mathcal{T}$ is a partial transition function,*
3. *$L \subseteq \mathcal{V}$ is a set of (local) variables,*
4. *$I \in \mathcal{S}$ is a distinguished initial state, and*
5. *$\mathcal{F} \subseteq \mathcal{S}$ is a set of final or accepting states.*

Definition 3. *The state S of the i-th machine LLFSM in the sequence $A.Q$ is uniquely identified as $M_i.S$.*

States have sections where code of the action language is placed. The sections are of three kinds, in a manner analogous to that of OMT and UML.

Definition 4. *A state S of a machine M_i in an arrangement has the form $M_i.S = (S, Oe_i, Ox_i, Do_i)$, where*

Oe_i, Ox_i, Do_i are respectively called the OnEntry*,* OnExit*, and* Internal *sections of the state. Each section is a set of legal instructions (commands) in some programming language (for example, assignments)*[2] *so that $\lambda(E, W, M_i.L).Oe_i$, $\lambda(E, W, M_i.L).Ox_i$, and $\lambda(E, W, M_i.L).Do_i$ have no free variables (that is, all variables occurring in the instructions of a section of a state are either external variables, whiteboard variables, or the local variables of the machine M_i that holds the state S).*

For convenience, a partial transition function $\mathcal{T}$ of an LLFSM M is represented as a sequence of triplets (S_s, B, S_t), where

1. S_s, S_t are states in $\mathcal{S}$, the states of M, and S_s is the source state while S_t is the target state (S_s and S_t could possibly be the same),
2. B is a Boolean expression in some logic language, so that $\lambda(E, W, M_i.L).B$ has no free variables (that is, all variables occurring in the Boolean expression B are either external variables, whiteboard variables or the local variables of the machine M_i that holds the state S).

Definition 5. *If two transitions $T_b = (S_{s_b}, B_b, S_{t_b})$ and $T_a = (S_{s_a}, B_a, S_{t_a})$ in the transitions $\mathcal{T}$ of an LLFSM M, where*

$$\mathcal{T} = \langle (S_{s_1}, B_1, S_{t_1}), (S_{s_2}, B_2, S_{t_2}), \ldots, (S_{s_t}, B_t, S_{t_t}) \rangle$$

are such that $S_{s_b} = S_{s_a}$ (that is, T_b and T_a have the same source state), and $b < a$ (that is, transition T_b is before T_a), then we say that T_b has precedence *over T_a.*

When transitions share a source state, they are evaluated in sequence in a shared snapshot of the environment. A transition that shares its source state with others fires only if all transitions that precede it do not fire and its labelling expression evaluates to `true`.

Definition 6. *Let $T_a = (S_{s_1}, B_a, S_{t_a})$ be a transition in the transitions $\mathcal{T}$ of an LLFSM M. If $\mathcal{T}$ has transitions $T_1 = (S_{s_1}, B_1, S_{t_1})$, $T_2 = (S_{s_1}, B_2, S_{t_2})$, …, $T_p = (S_{s_1}, B_p, S_{t_p})$ preceding T_a (and thus all have the same source state), then the* Boolean expression for T_a *is*

$$B_A = B_a \wedge \neg(B_1 \vee B_2 \vee \ldots \vee B_p).$$

Definition 7. *A state is* final *when it has no outgoing transition.*

A self-transition (same source and target) is sufficient for a state not to be final. If a self-transition fires, the *OnEntry* is not executed again. Once an LLFSM has reached a final state it will execute its *OnEntry*, its *Internal* once and will be removed from the scheduling in the arrangement. The *OnExit* code of a final state is not executed.

[2] For the ATL transformation here we use as the action language an extension of IMP [42] as statements in this language have direct equivalents in CLISP and SMV. However, implementations of LLFSMs have used Simple-C, `Java`, `C++`, and `Swift`.

3.1 Semantics

Depending on the action language, variables may be typed. This just means that the set $\mathcal{V}$ of variables is partitioned. For our implementation, we have at least two types: Boolean variables, and Integer variables since again these are common types between SMV and CLISP. Specifying the type of a variable corresponds to specifying its domain of values. We assume that all domains have a special value $\perp$ which will be the default value for all variables (and represents that the variable has not been assigned a value) [41].

Definition 8. *Given an arrangement $A = (Q = \langle M_1, \ldots, M_k\rangle, E = E_s \cup E_e, W)$, a* Kripke state *is a valuation of*

1. *the external variables $E = E_s \cup E_e$,*
2. *the whiteboard variables W,*
3. *the local variables L_i, for each machine M_i,*
4. *the current state $pc_i \in \mathcal{S}_i$, for each machine M_i,*
5. *a Boolean variable* `has_fired`$_i$*, for each machine M_i,*
6. *a variable* `turn` $\in \{1, \ldots, k\}$ *that indicates which machine holds the current token of execution (and performs a* ringlet*; that is, executes its current state).*

Valuation for variables means that we have a value q_i (which could be $\perp$) for the variable v_i.

We note that for model checking, systems are categorised as *open* or *closed* [40, Page 88]. Closed systems have no inputs, and are frequently those that are formally verified. But naturally, software operates in interaction with the environment. When dealing with an open system, a corresponding closed system is used by composing the open system with a model of its environment. For the definition of the next Kripke state in computation we adopt the standard convention (of modelling the environment) by allowing the valuation of a sensor variable to remain the same or change non-deterministically to any value in its domain if it is $\perp$, and to remain or change non-deterministically to any value in its domain (except $\perp$) if it is already different from $\perp$. External *sensor* variables will have a subscript s. External *effector* variables will have a subscript e. Whiteboard variables will have a subscript w.

Definition 9. *Given a Kripke state, (that provides (1) a value t for* `turn`*, (2) for each M_i, a current state pc_i, and a value for* `has_fired`$_i$*, and (3) values (or $\perp$) for all external variables, all whiteboard variables, all local variables of each LLFSM), the descendant Kripke states (*`next` *in NuSMV notation) are valuations (new assignment of values to variables) that are obtained as follows. First, all local variables of all other machines besides machine M_t retain their values as the only machine which executes a ringlet whose turn it is. That is,*

$$\texttt{next}(Mj.v_u) \leftarrow \texttt{value}(Mj.v_u)\ \forall j \neq t.$$

Second, new values are calculated for external variables, whiteboard variables and the local variables of machine M_t according to the following cases.

Case $\textbf{has\_fired}_i = \texttt{true}$. *This is the case when a transition fired as the last action of running the ringlet in machine M_t.*

1. *A copy $\mathcal{C} = (E, W, M_t.L)$ (recall $\texttt{turn} = t$) (a snapshot is taken) of the valuation defined by the environment variables, the whiteboard variables and the local variables for M_t.*
2. *The current program counter pc_t (points to a state of M_t) of the current state-machine indicated by turn $= t$, is used to select the statements Oe_{pc_t} of OnEntry section of the t-th machine and Oe_{pc_t} is executed in $\mathcal{C} = (E, W, M_t.L)$ producing a new valuation $\mathcal{C}_1$*
3. *The set of transitions is analysed resulting in two sub-cases.*
   (a) *If the set of transitions of M_t that have source state pc_t is empty, the machine is removed from the arrangement.*
   (b) *Otherwise, the transitions of M_t that have source state pc_t are evaluated in their precedence order in the same valuation $\mathcal{C}_1$. We have two sub-subcases.*
      i *If a transition $T_a = (S_{pc_j}, B, S_t)$ evaluates to* true*, then the OnExit section Ox_{pc_t} is evaluated in the context $\mathcal{C}_1$ producing a new context $\mathcal{C}_2$. The* `next` *value of the variable* $\texttt{has\_fired}_t$ *is set to true.*
      ii *Otherwise, no transition fired. Then, the Internal section Do_{pc_t} is evaluated in the context $\mathcal{C}_1$ to produce a new context $\mathcal{C}_2$.*
4. *The values of the context $\mathcal{C}_2$ are written back to the corresponding external variables, whiteboard variables and the local variables of M_t.*

Case: $\textbf{has\_fired}_i = \texttt{false}$. *The same steps as in the case when* $\texttt{has\_fired}_i = \texttt{true}$ *are performed except that Step 2 that executes the OnEntry is skipped and thus the context for evaluation transitions is the first snapshot $\mathcal{C} = (E, W, M_t.L)$.*
In all cases, the variable `turn` *is updated to* `turn+1` mod `k`*. In the open execution the sensor variables are executed with sensor values. In the closed system version, several new Krikpe states are produced by generating all possible combinations of the external sensor variables.*

3.2 Timed LLFSMs

Timed LLFSMs allow the predicate `after (int)` as part of the Boolean expression of a transition.

The machines are extended so that the information for a Kripke state about the state of an individual machine is extended. For each machine M_i, besides the current state $pc_i \in \mathcal{S}_i$ and $has\_fired_i \in \{\texttt{true}, \texttt{false}\}$, we now have a time-stamp t_i which records the time of entering the state pc_i (and before any action in the OnEntry of pc_i runs).

The semantics of the predicate `after (int)` in a transition B_p with source state S is essentially syntactic sugar for a "less than" comparison between the current time and the time recorded in the time-stamp t_i.

The results should be that, for positive values for the integer argument of `after (int)`, the first time the transition is evaluated, it will evaluate to `false`.

Every subsequent evaluation while in state S will retrieve the current universal time in the executing hardware as *later-time*. If the later-time is at least `int` seconds later than the time-stamp t_i, then the expression `after (int)` evaluates to `true`; otherwise it evaluates to `false`.

Most importantly, any other transition out of S that fires will reset to a new value the time-stamp t_i as part of the *OnEntry* section of the arrival state.

4 Value-Domain Versus Time-Domain

The common use of model checking is to eradicate value-domain failures. A value-domain failure means that an incorrect value is produced [29, Page 139]. A temporal failure, on the other hand, means that a value is computed outside the intended interval of real time. We already referred to Besnard et al. [3] for the discussion of semantic gaps and the ambivalent semantics of UML. We refer to these authors again because of their case study of the level-crossing train (see Fig. 2). In this case study, a train approaches a level crossing, and presses a first entrance sensor (called the far-entrance sensor). This should activate railway signs and road signs. Closer to the crossing, a second entrance sensor (the near-entrance sensor) activates the lowering down of a gate. Once the train passes, it presses an exit sensor and the gate lifts and the signals stop shining.

Besnard et al. [3] perform model checking (of the closed representation of their executable models) in the value domain. In particular, these authors verify deadlock detection (that the model is deadlock free) as well as four system requirements.

Property 1. The gate is closed when the train is in the level crossing.
Property 2. The light of the road-sign is active when the train is in the level crossing.
Property 3. Eventually, the gate will open after being closed.
Property 4. Eventually, the light of the road-sign will turn off after being activated.

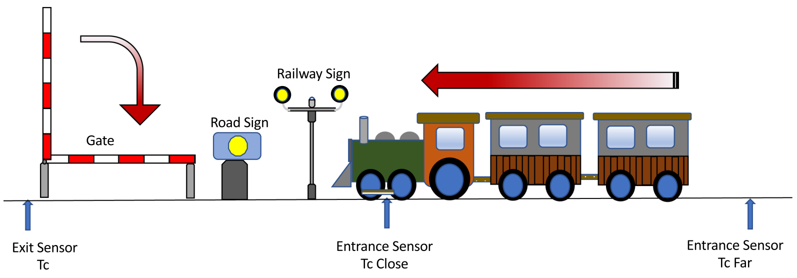


Fig. 2. Scenario of the level-crossing case study [3].

We emphasise that in the work of Besnard et al. [3] there are no time deadlines in the expression and verification of the above properties. A significant contribution of our work is that, in our translation to SMV, we enable integer variables to function as program counters (and a scheduler) for describing the computation progress in the model. Thus, by combining these integer variables and bounded operators of LTL and CTL, we can express and verify properties having explicit time deadlines. Therefore, we can verify the non-existence of temporal failures. For instance, by applying our ATL transformation to the LLFSM model of the level crossing behaviour, we obtain an SMV model where we can verify a variant of Property 4. This variant considers, in addition to the road sign, the train position, and is expressed by the following SMV code.

```
DEFINE
    roadSignIsActive := (RoadSign.pcRoadSign = 2);
    trainIsPassing   := (Train.pcTrain = 4);
LTLSPEC
   G ((roadSignIsActive & trainIsPassing) ->
      (F[0,8] (!trainIsPassing & (F[0,58] roadSignIsInactive))))
```

These bounded LTL formula and specific naming of LLFSMs program counters state that, if the light of the road sign is active and a train is on the level crossing then, after at most eight Kripke transitions, a train is out of the level crossing and then, after a maximum of 58 additional transitions, the road sign light turns off. From here, by a conversion of state transitions to time units, a worst-case execution time estimate of the statements in *OnEntry* sections of the model results in a verified hard real-time bound Property 4.

5 From LLFSMs to SMV Models via ATL Transformations

We now describe M2Text and M2M transformations that we use to transform an arrangement of LLFSMs into an SMV model (a set of SMV modules).

5.1 LLFSM with Non-sectioned States Through an M2M Transformation

We now describe using standard UML diagrams shown in Fig. 3 (constructed with Papyrus [26]) the algorithm that transforms an LLFSM into an LLFSM where states do not have sections. This condition is equivalent to all statements belonging to the OnEntry section. The version of the code released with this chapter includes an ATL M2M prototype of our transformation, implementing the semantics described in Definition 9. This implementation is similar to those that have removed the nesting of states in UML state machines of depth 2 into UMP state diagrams with no nesting and the Promela language of the Spin model checker [10]. In that setting, the interlingua semantics of hierarchical statecharts [19] unfolds a level-2 nesting into essentially a new statechart without nesting but with states given by the Cartesian product of the container statechart and the contained statecharts.

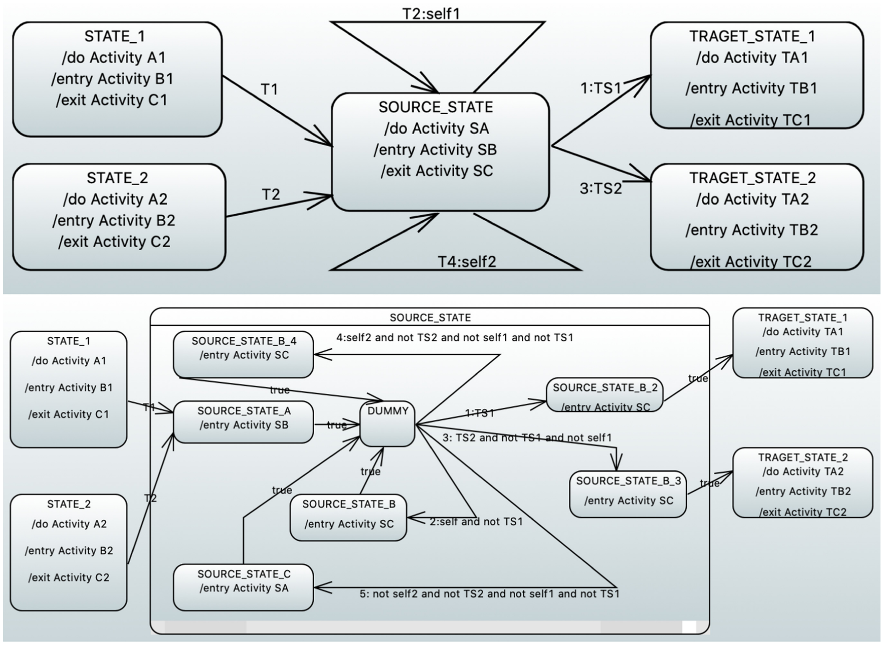
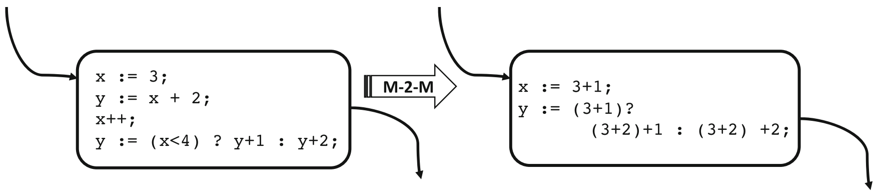


Fig. 3. Schema that defines the M2M transformation that ensures that states do not have sections (or alternatively, all statements are in the OnEntry section).

5.2 M2M Transformation to Handle a State's Section Atomically

Which actions (or code and in which action language) are placed in the sections of the states of LLFSMs is not essential for our discussion. The use of variables and sequences of assignments is a feature common in *extended* state machines, but theoretically, at the cost of having a larger number of states: for every extended state machine there exists an equivalent FSM (or LLFSM) [27].

We also define an M2M transformation that highlights that the semantics of actions (code) in LLFSMs are handled atomically. That is, the intermediate states of the action language inside a state are invisible to all other LLFSMs concurrently executing. Therefore, we can simplify such statements and their translation to SMV's statements. Moreover, this transformation only requires a syntactic analysis of the statements in a state. Figure 4 illustrates the effect of the transformation, which is given by the recursive algorithm in Fig. 5. The algorithm maintains a dictionary that with each variable it associates an expression. When an assignment statement is found for the first time, the variable on the left-hand side (LHS) is inserted as the key in the dictionary with the expression on the right-hand side (RHS) as the associated information. For each new assignment, the RHS has all free occurrences of variables replaced by the values in the dictionary before the LHS is updated with the new expression. Note that the statements are not executed. Instead, they are nested into the resulting expressions.



```
x := 3;
y := x + 2;
x++;
y := (x<4) ? y+1 : y+2;
```

```
x := 3+1;
y := (3+1)?
      (3+2)+1 : (3+2) +2;
```

Fig. 4. Since the statements in a section of a state are executed entirely or not at all in each ringlet, they can be considered atomic.

```
IMP-2-SMV($s$ : IMP-sequence, $d$ : Dictionary) : String
begin
    if $s$.isEmpty() then return‘’else                                  //The empty string
    if $1 == s$.size() or not$s$.tail.inLHS($s$.LHS().variable
    then                        //RHS is converted to SMV and free variables appearing in $d$
                                                  // replaced by corresponding strings in $d$
    return ‘next(’ + $s$.LHS().variable + ‘)=’ + $s$.RHS.to-SMV-subs-free($d$)
    else return IMP-2-SMV($s$.tail(), $d$.update($s$.LHS().variable, $s$.to-SMV($d$))
    fifi
end
```

Fig. 5. Converting an IMP-sequence of statements to SMV.

5.3 Handling Variables

An arrangement of LLFSMs defines a round-robin schedule for the concurrent execution of the state machines in the arrangement (it is possible to use other schedules, but for now we adhere to Definition 9 where `next(turn)`$\leftarrow$`turn+1` mod k). Thus, there are no race conditions; only one state machine is executing its ringlet at a time, so there is no need for mutual exclusion. In our transformation from LLFSMs to modules of SMV, however, some aspects of variables require a special treatment since, in SMV, all variables must belong to a module and must have an explicit value in each state. In the arrangement, all non-local variables are global. A simple solution would be to place all global variables (external and whiteboard variables) in the `main` module in SMV. However, this is not necessary if there is only one LLFSM in the arrangement that is a writer to the variable. Therefore, our M2Text transformation assigns variables in the arrangement to SMV modules as follows:

Variable Is Local to an LLFSM: The variable is declared local in the corresponding SMV module.

Variable Is a Sensor of the Arrangement: The variable is declared global in the `main` SMV module, no LLFSMs writes to this variable. The open SMV model would manage this variable non-deterministically, taking all possible values in the next Kripke state. This represents the possibility that the environment changes the sensor reading at any time.

Variable Is Written by Only One LLFSM: The variable is declared local in the corresponding SMV module.

Variable is written by more than one LLFSM: The variable is declared global in the `main` SMV module, but would behave deterministically in the Kripke states.

Because SMV composes modules under synchronous composability, all SMV modules advance simultaneously. Therefore, when producing an SMV module, our M2Text transformation must reflect the LLFSM semantics where a variable is never simultaneously written by more than one LLFSM. We will see in the next section that we use the value of `turn`, which becomes a variable in the `main` module of the SMV result, to ensure that only one SMV module would update the LHS of all its statements using `next`. Moreover, we will use the value of `turn` to ensure that only one SMV module can have an effect on any sort of variable in the model checker.

5.4 The Transformation When There Are No Temporal Transitions

We illustrate the transformation with an arrangement of two LLFSMs that includes Boolean and arithmetic expression as well as Boolean variables and integer variables. Figure 6 displays the visualisation that appears in our new prototype for emulating traces from model checkers. We describe the transformation from LLFSMs that have no sections in states to SMV modules by a series of rules.

Rule 1: There is an SMV module for each LLFSMs M_i in the arrangement. There is an SMV module `main` that holds an integer variable `turn` with domain $\{0, 1, \ldots, k-1\}$ that serves to indicate the only state machine that will affect values of variables. The SMV module `main` ensures the round-robin schedule in the Kripke states with transitions such that, for all $i = 0, \ldots, k-2$, $(\texttt{turn} = i)$ & $(\texttt{next}(\texttt{turn}) = i \bmod k)$; while $(\texttt{turn} = k)$ & $(\texttt{next}(\texttt{turn}) = 0)$.

Rule 2: All transitions of the module M_i (besides `main`) have a test of the form `(turn = i)` which effectively ensures that only the transitions in M_i advance the Kripke state when indeed the value of `turn` is i.

Rule 3: The `main` module instantiates the modules of each M_i in the arrangement for $i = 1, \ldots, k$ with parameters defined by the specification of Sect. 5.3.

For illustration, the corresponding `main` module for Fig. 6 declares the variable `turn` with domain $\{0, 1\}$.

```
MODULE main
VAR turn : 0..1;
MonitorCounter : MonitorCounter(turn , Counter.counter);
Counter : Counter(turn , MonitorCounter.GoDown);
TRANS ((turn = 0) &  (next(turn) = 1 )) | ((turn = 1) &  (next(turn) = 0 ))
```

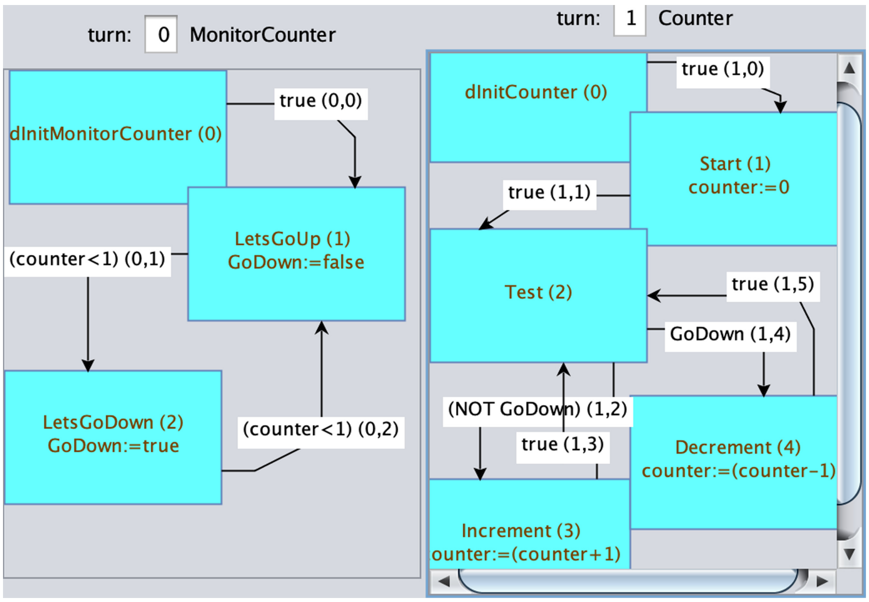


Fig. 6. Arrangement of two state machines with a global (whiteboard) Boolean variable `GoDown` and a global (whiteboard) integer variable `counter`.

This SMV code, produced by our ATL M2Text transformation, shows how the SMV module `main` instantiates the two modules for the two LLFSMs in the arrangement. This piece of code also shows that the SMV module `main` schedules the round-robin cycle through the possible values of the `turn` variable.

Rule 4:
1. The i-th SMV module will correspond to machine M_i.
2. The j-th state in machine M_i will be identified as the j-th state in the corresponding SMV module for M_i for $j = 1$ to the number of states in M_i (and an additional 0-th dummy state will be identified as the initial pseudo-state with a transition labelled with `true` to the initial state of M_i).
3. The current state pc_i (see Definition 8) of machine M_i will be identified by a local variable `pc` in SMV module M_i.
4. Moreover, a transition in M_i, whose source state is the j-th state, will be identified with the tuple i, r, where r is the rank of the transition among the transitions in M_i.

Figure 6 shows the use of these identifiers for machines, states, and transitions in our running example. What follows now is the detailed description of how the transformation ensures the running of a ringlet; that is, the effects of the machine whose turn it is.

Rule 5: Each transition $T_{i,r}$ with source state S in M_i will result in a Boolean SMV expression that will guarantee this transition can only fire when it is the turn of machine M_i and its current state is s. If the Boolean expression for $T_{i,r}$ is $b_{i,r}$ and SMV($b_{i,r}$) is the translation of $b_{i,r}$ from IMP to SMV, then we defined the SMV condition for $T_{i,r}$ by

$$\texttt{cond}T_{i,r} := (\texttt{turn} = i)\ \&\ (pc_i = S)\ \&\ \text{SMV}(b_{i,r}) \tag{1}$$

Rule 6: For transition $T_{i,r}$ to fire, no transition with $q \leq r$ should fire (and in the case of the SMV module this could be because the source state does not match the `pc` or because their Boolean expression is `false`), thus the SMV translation for $T_{i,r}$ corresponds to the conjunction of the negation of all `cond`$T_{i,q}$ with the affirmation of `cond`$T_{i,r}$.

Rule 7: The M_is we are considering have no sections in a state. So when no transition out of a state S fires because the Boolean expressions of all transitions evaluate to `false`, then machine M_i has no effects, and just misses its turn. We represent this in the SMV module for M_i by an SMV expression `condDefault`$_i$ which is the conjunction of all the negations of all the `cond`$T_{i,q}$.

Rule 8: When a transition `cond`$T_{i,r}$ fires, the effects of the statements of the target state are encoded by the corresponding translation of those IMP statements (after the atomicity reduction of Subsect. 4) into SMV statements.

The application of these last rules in our ATL transformation results in the following module for the `MonitorCounter` LLFSM from Fig. 6.

```
MODULE MonitorCounter(turn, counter)
VAR pcMonitorCounter : 0..2;
VAR GoDown : boolean;
INIT (pcMonitorCounter    = 0)
DEFINE
condT00 := (((turn = 0) &  (pcMonitorCounter = 0)) & TRUE);
condT01 := (((turn = 0) & (pcMonitorCounter =  1)) & (counter  < 1));
condT02 := (((turn = 0) &  (pcMonitorCounter = 2)) & (counter  < 1));
condDefault0 := (!(condT00) & !(condT01) & !(condT02));
TRANS
(TRUE &
condT00 & (next(pcMonitorCounter)=1) &  (next(GoDown)=FALSE))
|
(!(condT00) &
condT01 & ((next(pcMonitorCounter)=2) &  (next(GoDown)=TRUE)))
|
(!(condT00) & !(condT01) &
condT02 & ((next(pcMonitorCounter)=1) &  (next(GoDown)= FALSE)))
|
(condDefault0 &
TRUE & (turn=0) & ((next(pcMonitorCounter)=pcMonitorCounter)
        & (next(GoDown)=GoDown)))
|
(condDefault0 & TRUE &  (turn !=0)
        & ((next(pcMonitorCounter)=pcMonitorCounter) & (next(GoDown)=GoDown)))
```

Note that all `condT_` ensure that it is this machine's turn (SMV module) and the current state is the source of the transition. Moreover, with the contrasting Fig. 6, observe that the transition from the state `LetsGoUp` (stated ID 1) to the state `LetsGoDown` (state ID 2) is transition with ID = (0,1) as this is the 0-th machine and this is the second transition, but numbering from 0, its numeral is 1. This transition defines `condT01` in the code above (using the Boolean expression that labels this transition). In Fig. 6, we see that if the transition fires, the effect in the target state is for the local variable (as this is the only module that writes on it) `GoDown` to change to `true`. This happens when not `condT00` and `condT01`, and the code above shows that in this case the next Kripke state will have the variable `GoDown` updated accordingly.

5.5 Temporalised Transitions

We now describe the part of our ATL transformation that handles the particular predicate `after` that enables timed LLFSMs. These timed LLFSMs were probably introduced in robotic systems [8,32] as part of the subsumption architecture under the name of *augmented* fine-state machines [7]. If the Boolean expression of a transition T includes the predicate `after`, we say that it is a *temporalised* transition. To translate the semantics of temporalised transition given in Subsect. 3.2, our ATL transformation defines the following rules.

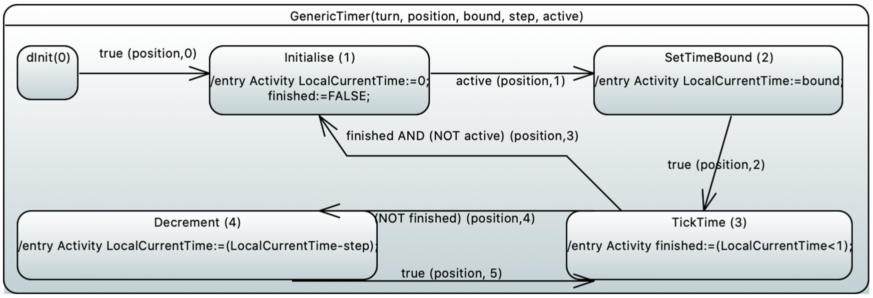

GenericTimer(turn, position, bound, step, active)
dInit(0)
true (position,0)
Initialise (1)
/entry Activity LocalCurrentTime:=0; finished:=FALSE;
active (position,1)
SetTimeBound (2)
/entry Activity LocalCurrentTime:=bound;
finished AND (NOT active) (position,3)
true (position,2)
Decrement (4)
/entry Activity LocalCurrentTime:=(LocalCurrentTime-step);
(NOT finished) (position,4)
TickTime (3)
/entry Activity finished:=(LocalCurrentTime<1);
true (position, 5)

Fig. 7. Chart of the generic timer LLFSM.

Rule 9: For each temporalised transition (a transition whose Boolean expression includes the predicate `after`), there will be an SMV instance of a timer module. For an occurrence of `after`(b) in a transition $T_{i,r}$ with source state S, the corresponding instance of a timer SMV module receives its turn in the round-robin schedule. It also receives as input parameter the bound b and a Boolean variable $M_i\_S\_$`ACTIVE` (recall that $T_{i,r}$ means this is the $(r+1)$-th transition in machine M_i).

Rule 10: For each transition $T_{i,r}$ with source state S, the SMV module corresponding to machine M_i will have the local Boolean variable $M_i\_S\_$`ACTIVE` mentioned before; but also an array of variables $M_i\_S\_$`BOUND` (the array size is equal to the number of temporalised transitions that share S as a source state). It also includes an integer variable $M_i\_S\_$`STEP` that typically has the value 1, but can have a larger value to model the speed of the timers relative to the schedule of LLFSMs in the arrangement.

Rule 11: If an LLFSM M_i has temporalised transitions, its SMV module will have additional parameters, as many as temporalised transitions in M_i, where the corresponding instance of the timer communicates that the time-bound in the corresponding `after` predicate has been reached.

Rule 12: There will be only one SMV module for a timer if any of the LLFSMs in the arrangement has a temporalised transition. The SMV module is the image (by the ATL transformation) of a timer LLFSMs.

```
MODULE Timer(turn, position, bound, step, active)
VAR pcTimer : 0..4;
finished : boolean;
LocalCurrentTime : 0..bound;
INIT (pcTimer=0)
DEFINE
condT50 := (((turn=position) & (pcTimer=0)) & TRUE);
condT51 := (((turn=position) & (pcTimer=1 )) & (active));
condT52 := (((turn=position) & (pcTimer=2)) & TRUE );
condT53 := ((turn=position) & (pcTimer=3)) & ((finished) & (!(active)));
condT54 := (((turn=position) & (pcTimer=3)) & (!(finished)));
condT55 := (((turn=position) & (pcTimer=4)) & TRUE);
condDefault5 := (!(condT50) & !(condT51) & !(condT52) & !(condT53) & !(condT54) & !(condT55));
TRANS
(TRUE & condT50 & (next(pcTimer)=1) & (next(LocalCurrentTime)=0) & (next(finished)=FALSE ) & TRUE)
|
( !(condT50) & condT51 & ((next(pcTimer)=2) & (next(LocalCurrentTime)=bound)
     & (next(finished)=finished) & TRUE)
|
( !(condT50) & !(condT51) & condT52 &
  ((next(pcTimer)=3) & (next(finished)=(LocalCurrentTime<1)) & TRUE
     & (next(LocalCurrentTime)=LocalCurrentTime))
|
( !(condT50) & !(condT51) & !(condT52) & condT53 &
  ((next(pcTimer)=1) & (next(LocalCurrentTime)=0) & (next(finished)=FALSE) & TRUE)
|
( !(condT50) & !(condT51) & !(condT52) & !(condT53) & condT54 &
  ((next(pcTimer)=4) & (next(LocalCurrentTime)=(LocalCurrentTime-step)) & (next(finished)=finished ) & TRUE)
|
( !(condT50) & !(condT51) & !(condT52) & !(condT53) & !(condT54) & condT55 &
  ((next(pcTimer)=3) & (next(finished) = (LocalCurrentTime<1)) & (next(LocalCurrentTime)=LocalCurrentTime))
|
(condDefault5 & TRUE & ((next(pcTimer)=pcTimer) & (next(firedTimer)=1)
   & (next(LocalCurrentTime)=LocalCurrentTime) & (next(finished)=finished)))))))
```

Fig. 8. The SMV module for the generic timer for Fig. 7.

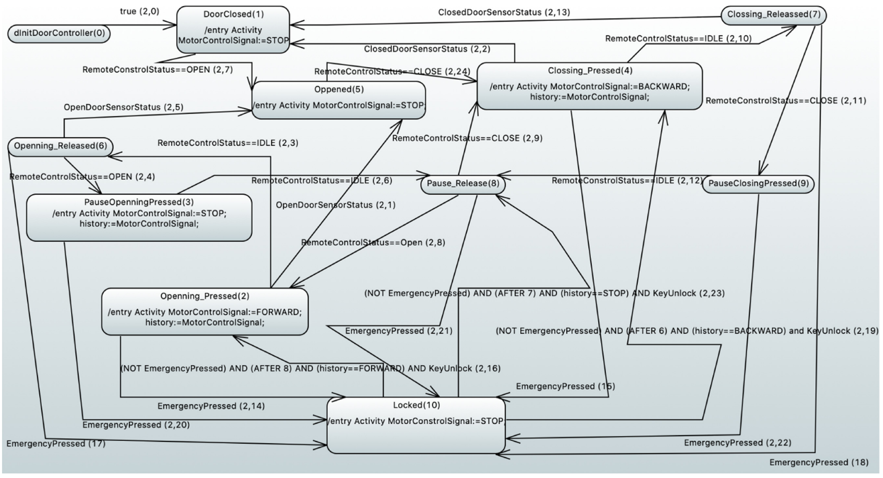

true (2,0)
dInitDoorController(0)
DoorClosed(1)
/entry Activity MotorControlSignal:=STOP
ClosedDoorSensorStatus (2,13)
Clossing_Releassed(7)
RemoteControlStatus==IDLE (2,10)
ClosedDoorSensorStatus (2,2)
RemoteConstrolStatus==OPEN (2,7)
RemoteControlStatus==CLOSE (2,24)
Clossing_Pressed(4)
/entry Activity MotorControlSignal:=BACKWARD; history:=MotorControlSignal;
Oppened(5)
/entry Activity MotorControlSignal:=STOP
RemoteConstrolStatus==CLOSE (2,11)
OpenDoorSensorStatus (2,5)
Openning_Released(6)
RemoteControlStatus==IDLE (2,3)
RemoteControlStatus==CLOSE (2,9)
RemoteControlStatus==OPEN (2,4)
RemoteControlStatus==IDLE (2,6)
Pause_Release(8)
RemoteControlStatus==IDLE (2,12)
PauseClosingPressed(9)
PauseOpenningPressed(3)
/entry Activity MotorControlSignal:=STOP; history:=MotorControlSignal;
OpenDoorSensorStatus (2,1)
RemoteControlStatus==Open (2,8)
Openning_Pressed(2)
/entry Activity MotorControlSignal:=FORWARD; history:=MotorControlSignal;
(NOT EmergencyPressed) AND (AFTER 7) AND (history==STOP) AND KeyUnlock (2,23)
EmergencyPressed (2,21)
(NOT EmergencyPressed) AND (AFTER 6) AND (history==BACKWARD) and KeyUnlock (2,19)
(NOT EmergencyPressed) AND (AFTER 8) AND (history==FORWARD) AND KeyUnlock (2,16)
EmergencyPressed (16)
EmergencyPressed (2,14)
Locked(10)
/entry Activity MotorConstrolSignal:=STOP
EmergencyPressed (2,20)
EmergencyPressed (17)
EmergencyPressed (2,22)
EmergencyPressed (18)

Fig. 9. Chart of the door controller LLFSM in the Garage Door arrangement.

Figure 7 displays the chart of the generic timer and the ATL transformation for it appears in Fig. 8. This illustrates the M2M transformation once more. More importantly, we see that the LLFSMs is parameterised. Since there could be several instances for the same LLFSMs there is a parameter `position` where the instance of the LLFSMs is placed in the arrangement.

To illustrate these rules and the timer, we review a case study of a garage door [2]. The arrangement includes LLFSMs for a sensor that detects when the door is fully open as well as another for when the door is fully closed. A motor that can be pulling forwards (opening), backwards (closing) or halted (holding the door in position). The remote control has three states: idle, commanding a stop, or commanding a close. An emergency button can be pressed or released, while a key is locked or unlocked. For reasons of space, we focus only on the LLFSM for the door controller (Fig. 9). The behaviour is not trivial. Pressing the emergency button halts and locks the door always, and to unlock the garage door, the emergency button must be released and the key unlocked. Such release returns the door to what it was doing at the time of the emergency (for instance, if it was closing, it resumes closing). Also, users can start opening the door by pressing the open button on their remote control. Users can stop the opening by pressing the open button again, and the motor stops. This implies the state machine of the remote control notices the release of the opening button in the remote and the second pressing. Pressing the close button will close the door if it is (partially or completely) open. Closing can be interrupted: by pressing the close button again, the motor stops. Before we pay attention to the time-domain verification with this example, we highlight that the model can use predefined constants for the control signal and the status signal of the motor (such strings as STOP, FORWARD and BACKWARD) and for the control signal and status signal of the remote control (strings such as IDLE, OPEN, CLOSE). These enumerated constants are now generated in a dedicated module of our M2Text ATL transformation to SMV. We also highlight that we have several integer variables in our examples of this paper. The garage-door example also illustrates the capacity to handle a history mechanism in an integer variable.

The LLFSM of Fig. 9 is illustrative of three temporal transitions with the same source state. Therefore, our M2Text transformation produces three instances of the timer form Fig. 7 in the `main` module of the SMV file.

```
Timer_DoorController_Locked_Opening_Pressed_16:
   Timer(turn,5, 8, 1,DoorController.DoorController_Locked_ACTIVE);
Timer_DoorController_Locked_Closing_Pressed_19:
   Timer(turn,6, 6, 1,DoorController.DoorController_Locked_ACTIVE);
Timer_DoorController_Locked_PauseReleased_23:
   Timer(turn,7, 7, 1,DoorController.DoorController_Locked_ACTIVE);
```

6 Verification in the Time Domain

The garage-door example provides several interest requirements whose verification is required in the time domain. We present here some examples where we can bound the delay in the system's reaction. We use an LTL formulation, and define the following SMV-terms for ease of formulation of the properties.

```
DEFINE -- Abbreviations:
 doorIsClosed := !OpenSensor.OpenDoorSensorStatus;
 OpenButtonPressed    := (RemoteControl.RemoteControlStatus = 1);
 MotorSpinsForward    := (Motor.MotorStatus = 1); MotorSpinsBackwards := (Motor.MotorStatus = 2);
 MotorIsStopped       := (Motor.MotorStatus = 0);
 CloseButtonPressed   := (RemoteControl.RemoteControlStatus = 2);
 DoorIsOpening        := (MotorSpinsForward);
 DoorIsClosing        := (MotorSpinsBackwards);
 DoorIsMoving         := (DoorIsOpening | DoorIsClosing);
 DoorIsStopped        := (! DoorIsMoving);
```

Requirement 1: If the door is closed, and the open button on the remote control is being pressed, then the motor will begin to spin forward.

```
X( (doorIsClosed         -- Excepting the first state, if door is closed
 & G[0,7]                -- and, during 7 transitions (a round of turns),
   OpenButtonPressed)->  -- the "open" button on the remote is being pressed,
 F[0,8]                  -- then: in a future state, after at most 8 transitions,
 MotorSpinsForward);     -- the motor will begin to move forward to open the door.
```

Requirement 2: If the door is closing, and the "close" button on the remote control is pressed again, then the door will stop.

```
X( (MotorSpinsBackward
-- Excepting the first state, if the motor is spinning backwards (to close the door),
 & G[0,4]                  -- and, during 4 transitions,
   CloseButtonPressed)->   -- the "close" button on the remote control is being pressed again,
 F[0,8]                    -- then: in a future state, after at most 8 transitions:
 MotorIsStopped);          -- the motor will stop (and the door stops closing)
```

Requirement 3: While the door is moving, pressing the emergency button results in an immediate halt of the door.

```
X( (DoorIsMoving           -- Excepting the first state, if the door is in movement,
  & EmergencyPressed)->    -- and the "emergency" button on the remote control is pressed, then:
  (F[0,7]                  -- in a future state, after at most 7 transitions (a round of turns):
  DoorIsStopped) )         -- the door is stopped (1)
```

We emphasise that these properties show that within a round of turns (that is, each LLFSM receives a turn), the system reacts accordingly.

Naturally, there are reciprocal properties for ensuring that once the emergency button is pressed, the system does not resume immediately, but a minimum amount of time is to occur. This behaviour is what the temporalised transitions are meant to enforce in the model. The formulation of the LTL properties (analogous versions in CTL exist for these requirements) depends on what the door was previously doing at the time the emergency button was pressed. We present one version of these properties.

Requirement 4: If the garage is locked (while stopped) because the emergency button was pressed, it must stay there some specified amount of time before it resumes the movement it was performing when the emergency button was pressed.

```
X( G(                                  -- Excepting the first state, in all future states:
  (DoorControllerAtLockedState         -- if the DoorController is at state "Locked" (10),
  & TimerFinishedPause                 -- and the timer for emergency when door stopped finished
  & (G[0,7] DoorController.KeyUnlock)-- and the key stays unlocked for at least 7 transitions
  & (G[0,7] !EmergencyPressed)
-- and the emergency button stays unpressed for at least 7 transitions
  & EmergencyWhenStopped)->
-- and the emergency button was pressed when the door was stopped, then:
   (F[0,8]                             -- in a future state, after at most 8 transitions:
    DoorControllerAtPauseState) ))     -- the DoorController will be at state "PauseReleased" (8)
```

For this property we use the timer instances that are produced by the M2Text transformation. And in particular, their local variable indicated they have completed the time counting.

```
DEFINE
 TimerFinishedPause := Timer_DoorController_Locked_PauseReleased_23.finished;
 DoorControllerAtLockedState := (DoorController.pcDoorController = 10);
 DoorControllerAtPauseState := (DoorController.pcDoorController = 8);
```

7 Formal Verification of the SMV Output and Trace Emulation

7.1 Verification of the M2Text SMV Output

One important aspect that we add in this paper is that the M2Text transformation from an arrangement of LLFSMs is not only the corresponding set of SMV modules. We also automatically generate properties that formally verify the correctness of the transformation, and in particular of the scheduling. These properties are generic, but not exactly the same, they depend on the arrangement. For instance, if there are four LLFSMs in the arrangement, the ATL transformation adds the property

```
CTLSPEC
AG ((EF(turn=0)) & (EF(turn=1)) & (EF(turn=2)) & (EF(turn=3)))
```

This property ensures the global condition that from any point in the execution of the arrangement each LLFSM will have its turn. Similarly, for each possible value of the `turn`, we must have the next value in one more (modulo the number of LLFSMs in the arrangement) in the Kripke structure ensuring the round-robin scheduling. For instance, in an arrangement with four LLFSMs the following code is automatically generated.

```
LTLSPEC
G (  (turn=0 -> X(turn=1)) & (turn=1 -> X(turn=2))
   & (turn=2 -> X(turn=3)) & (turn=3 -> X(turn=0)))
```

7.2 Trace Emulation

To further illustrate the minimisation of the semantic gaps discussed earlier, we have used the EMF generated Java classes for our meta-model for LLFSMs to produce a tool that enables reading a model (an XMI file) as well as reading a counterexample's trace (the output of a verification exercise where the property is `false`). The model designer can visualise the execution of the trace in the arrangement of LLFSMs resulting in a more transparent interpretation of the trace and the revision of the behaviour model. The link to a video showing the emulator working on a trace of garage-door example is available at mipal.net.au/downloads.php.

This emulation minimises the semantic gap because it is the model checker that has generated the execution trace. That is, the execution is exactly the execution of the behaviour in the model checker. The trace emulated by our tool is directly visualised in the graphical representation of the XMI file (the model). There are no simplifying assumptions on the model, or any of its constructs.

7.3 Complexity

The size of the resulting SMV file is linear in the number k of LLFSMs in the arrangement since exactly $k + 1$ modules are produced if there are no temporalised transitions and no symbolic constants. One SMV module appears in the output file for each LLFSM and on `main` module. At most one additional module is produced if there are symbolic constants (enumerated types). At most one generic module for the timer is produced if there are t temporalised constants (and t-instances of the timer, one for each temporalised transition in `main`). The size of the SMV file is quadratic in the largest transition out-degree of a state (but this is usually bounded by a small constant). The number of transitions out of a state plays a role because, as specified in Rule 6, for each transition T_i, r, we must explicitly represent that, T_i, r fires when it is true and all previous transitions T_i, s (with $s < r$) have not fired.

8 Final Remarks

Verification in the time domain aims at eradication of time-domain failures. For many cyber-physical systems, it is insufficient to verify that no incorrect value is computed; it is also necessary that the correct value be computed by the required deadline. While event-driven programming has been extremely productive to develop GUI-based applications, this setting has the luxury that (1) human users can usually wait (although many users have noticed occasions when the system becomes less responsive) and (2) human users can hardly generate a shower of events. We have presented here efficient ATL-M2Text transformations that enable time-domain verification of behaviour models. Moreover, we have argued that these transformations support the spirit of model-driven engineering, because the model is executed in an unambiguous semantics. The transformations are so loyal to the model checker itself could be used as the interpreter. Our EMF application is the ultimate illustration of the minimisation of the semantic gap. This application uses the `ecore` generated classes for the meta-model of LLFSMs on one hand, and the trace of the SMV-enabled model checker on the other. This is only possible because there is no semantic gap between how the model checker simulates an arrangement of LLFSMs with no subsection for the states and the semantics of LLFSMs models.

Naturally, we are not arguing that models of behaviour be executed by interpreting them with a model checker. This would imply inefficiencies at run-time and potential limitations to the actions language of the behaviour models. However, the clear and small semantics of LLFSMs facilitates the implementation of compilers that are also loyal to the controlled concurrency of arrangements of LLFSMs and can provide modern constructs of programming (for instance object-orientation). This work here opens the door to new ideas. For instance, test-driven-development suggest building a suite of test that lives along with the development of a system, from requirements engineering to evolution and maintenance. The suite ensures that new features do not incur in regressions. We envisage that we could also have verification-driven development of behaviour models, were the requirements are codified for a model checker and also have a parallel life with the implementation and maintenance.

References

1. Alhaj, M.: UML modeling using Eclipse Papyrus (2018). https://www.youtube.com/watch?v=aMiqJXWfAtQ. Accessed 26 May 2020
2. André, P., El Amin Tebib, M.: Refining automation system control with MDE. In: Hammoudi, S., Ferreira Pires, L., Selic, B. (eds.) Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, pp. 425–432. SCITEPRESS (2020). https://doi.org/10.5220/0009147804250432
3. Besnard, V., Brun, M., Jouault, F., Teodorov, C., Dhaussy, P.: Unified LTL verification and embedded execution of UML models. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, pp. 112–122. ACM, New York (2018). https://doi.org/10.1145/3239372.3239395
4. Bhaduri, P., Ramesh, S.: Model checking of statechart models: Survey and research directions (2004)
5. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Requirements engineering via non-monotonic logics and state diagrams. In: Maciaszek, L.A., Loucopoulos, P. (eds.) ENASE 2010. CCIS, vol. 230, pp. 121–135. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23391-3_9
6. Birta, L.G., Arbez, G.: Modelling and Simulation – Exploring Dynamic System Behaviour. Springer, Heidelberg (2019)
7. Brooks, R.: A robust layered control system for a mobile robot. IEEE J. Robot. Autom. **2**(1), 14–23 (1986). https://doi.org/10.1109/JRA.1986.1087032
8. Brooks, R.: The behavior language; user's guide. Technical report AIM-1227, Massachusetts Institute of Technology - MIT, Artificial Intelligence Lab Publications, Department of Electronics and Computer Science (1990)
9. Bryce, C.R., Kuhn, R.: Software testing [guest editors' introduction]. IEEE Comput. **47**(2), 21–22 (2014)
10. Caltais, G., Leue, S., Singh, H.: Correctness of an ATL model transformation from sysml state machine diagrams to promela. In: Hammoudi, S., Ferreira Pires, L., Selic, B. (eds.) Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD, pp. 360–372. SCITEPRESS (2020). https://doi.org/10.5220/0008968303600372
11. Carrillo, M., Estivill-Castro, V., Rosenblueth, D.A.: Model-to-model transformations for efficient time-domain verification of concurrent models by NuSMV modules. In: Hammoudi, S., Ferreira Pires, L., Selic, B. (eds.) Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, pp. 287–298. SCITEPRESS (2020). https://doi.org/10.5220/0008910202870298
12. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9
13. Ciccozzi, F., Malavolta, I., Selic, B.: Execution of UML models: a systematic review of research and practice. Softw. Syst. Modeling **18**(3), 2313–2360 (2018). https://doi.org/10.1007/s10270-018-0675-4

14. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NUSMV: a new symbolic model checker. Int. J. Softw. Tools Technol. Transf. **2**(4), 410–425 (2000). https://doi.org/10.1007/s100090050046
15. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 1–26. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_1
16. Clarke, E., Heinle, W.: Modular translation of statecharts to SMV. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburg, PA 15213 (2000). Sponsored by General Motors Corp
17. Damm, W., Jonsson, B.: Eliminating queues from RT UML model representations. In: Damm, W., Olderog, E.R. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 375–393. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45739-9_22
18. Dijkstra, E.W.: The humble programmer. Commun. ACM **15**(10), 859–866 (1972). https://doi.org/10.1145/355604.361591
19. Drusinsky, D.: Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design. Runtime Monitoring and Execution-based Model Checking. Newnes, Newton, MA, USA (2006)
20. Eriksson, H.E., Penker, M., Lyons, B., Fado, D.: UML 2 Toolkit. Wiley, Hoboken (2003)
21. Evans, A., Bruel, J.M., France, R., Lano, K., Rumpe, B.: Making UML precise. In: Andrade, L., Moreira, A., Deshpande, A., Kent, S. (eds.) OOPSLA 1998 Workshop on "Formalizing UML. Why and How?", October 1998. www.se-rwth.de/publications
22. Furrer, F.: Future-Proof Software-Systems: A Sustainable Evolution Strategy. Springer, Berlin (2019). https://doi.org/10.1007/978-3-658-19938-8
23. Grischa, L.: Papyrus 2.0: State machine diagrams (2016). www.youtube.com/watch?v=xEC8bQ27lBk. Accessed 26 May 2020
24. Group, T.O.M.: Precise Semantics of UML State Machines (PSSM). OMG, May 2019
25. Group, T.O.M.: Precise Semantics of UML Structure (PSCS). OMG, June 2019
26. Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, e., Dhouib, S., Gérard, S.: Executable modeling with fUML and Alf in Papyrus: tooling and experiments. In: Mayerhofer, T., Langer, P., Seidewitz, E., Gray, J. (eds.) Proceedings of the 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015). CEUR Workshop Proceedings, vol. 1560, pp. 3–8. CEUR-WS.org (2015)
27. Kang, I., Lee, I.: A state minimization algorithm for communicating state machines with arbitrary data space. Technical report MS-CIS-93-07, Department of Computer & Information Science, University of Pennsylvania, January 1993
28. Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In: Damm, W., Olderog, E.R. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 395–414. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45739-9_23
29. Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications, 2nd edn. Springer, Heidelberg (2011). https://doi.org/10.1007/978-1-4419-8237-7
30. Lamport, L.: The TLA$^+$ home page, 6th December 2018. lamport.azurewebsites.net/tla/tla.html. Accessed 20 Apr 2020

31. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997). https://doi.org/10.1007/s100090050010, https://doi.org/10.1007/s100090050010
32. Mataric, M.: Integration of representation into goal-driven behavior-based robots. IEEE Trans. Robot. Autom. **8**(3), 304–312 (1992). https://doi.org/10.1109/70.143349
33. McColl, C., Estivill-Castro, V. Hexel, R.: An OO and functional framework for versatile semantics of logic-labelled finite state machines. In: Lavazza, L. (ed.) ICSEA : The Twelfth International Conference on Software Engineering Advances, pp. 238–243. Int. Academy, Research, and Industry Association (IARIA), Curran, 8th–12th October 2017
34. McMillan, K.L.: Symbolic Model Checking – An approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, United States, May 1992. cMU-CS-92-131
35. Obermaisser, R., Kopetz, H.: Chapter 3: properties of time-triggered communication systems. In: Obermaisser, R. (ed.) Time-Triggered Communication. CRC Press Inc., USA (2011)
36. Pham, V.C., Radermacher, A., Gérard, S., Li, S.: A framework for UML-based component-based design and code generation for reactive systems. In: Pires, L.F., Hammoudi, S., Selic, B. (eds.) MODELSWARD 2017. CCIS, vol. 880, pp. 300–327. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94764-8_13
37. Rushby, J.M.: Systematic formal verification for fault-tolerant time-triggered algorithms. IEEE Trans. Softw. Eng. **25**(5), 651–660 (1999). https://doi.org/10.1109/32.815324
38. Rushby, J.: Bus architectures for safety-critical embedded systems. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 306–323. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45449-7_22
39. Samek, M.: Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems, 2nd edn. Newnes, Newton (2008)
40. Seshia, S.A., Sharygina, N., Tripakis, S.: Modeling for verification. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 1–26. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_1
41. Weise, C.: An incremental formal semantics for PROMELA. In: Proceedings of the Third SPIN Workshop, SPIN 1997 (1997)
42. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge (1993)

Author Queries

Chapter 10

Query Refs.	**Details Required**	**Author's response**
AQ1	This is to inform you that corresponding author has been identified as per the information available in the Copyright form.	
AQ2	Please confirm if the corresponding author email address are correctly identified. Amend if necessary.	