2014 23rd Australian Software Engineering Conference

Correctness by Construction with Logic-Labeled Finite-State Machines – Comparison with *Event-B*

Vladimir Estivill-Castro
School of Information and
Communication Technology
Griffith University
Nathan, Brisbane 4111
Australia
Email: v.estivill-castro@griffith.edu.au

René Hexel
School of Information and
Communication Technology
Griffith University
Nathan, Brisbane 4111
Australia
Email: r.hexel@griffith.edu.au

***Abstract*—Formal methods have seen emergent success recently with the deployment of *Event-B*. However, *Event-B* explicitly postulates that models there are not executable. This seems to contradict the parallel emergence of model-driven development (MDD). We show here that logic-labeled finite-state machines (LLFSMs) are effective in carrying out the "correct from construction" agenda of formal methods such as *Event-B* and simultaneously achieve the aims of MDD. As a result, we obtain models that are directly interpretable, compilable, and executable enabling traceability, transparency and rapid maintainability; while at the same time enabling simulation, validation and formal verification with model checking. Moreover, the *Event-B* capacity to develop *closed models* is also very natural with arrangements of LLFSMs; and therefore further safety analysis such as failure-mode effects analysis (FMEA) can be performed. We demonstrate this with two well-known examples in the literature.**

I. Introduction

It is now widely accepted that, while UML has become the most extensively used software and systems modelling language [1], it also has become impractically large [2], [3]. Thus the question arises: what are the truly useful parts of UML? Several studies have consistently found that *class diagrams* and *state charts* repeatedly score equally high in usage in many categories of systems as well as tools, books, and tutorials [4], [5]. In most cases, they are used 100% of the time, while other UML constructs score lower. While *class diagrams* are static descriptions, state charts are dynamic models. With the emergence of model-driven development, class-diagrams directly correspond to the determination of attributes and signature definitions that constitute the properties of classes. Modulo object-oriented concepts (that is, methods, inheritance, aggregation, and other artefacts such as singletons, constructors, destructors, setters and getters), class diagrams are arguably entity-relationship diagrams and their translation from model to implementation is fully automated.

However, for the specification of behaviours, model-driven development (MDD) with state-charts has advanced, but has faced many challenges. We argue that much of the challenges can be attributed to the event-driven nature of the UML approach to state-charts [6]. We propose the succinct and small modelling language of logic-labeled state-charts whose transitions are not labeled by events, but by logical expressions. Earlier, we have shown that logic-based state charts have a clearer semantics, are much more succinct modelling tools, suitable for formal verification with model checking technology [6]. Logic-labeled finite-state machine (LLFSMs) can effectively be used concurrently while removing race conditions, and are applicable to many case studies that previously have heavily attracted the attention of the software engineering community when it comes to modelling and correctness. Moreover, they can be interpreted, simulated, compiled and thus produce formal and efficiently executable descriptions of behaviour that fulfill the aims of MDD [7].

Other formal methods and techniques, in particular *Event-B* [8] and UML-B, have demonstrated the important notion that modelling software is essential for software development. How do LLFSMs compare with *Event-B*? We show here that their event-driven nature results again in significantly more verbose and laborious modeling because of their handling of unpredictable events and their approach to concurrency.

While the world may be non-deterministic (we have little control on *when* different events happen in the environment in which our software executes), we certainly have control on when our software should execute what, in what order, and for how long. We suggest that letting each of our software components treat any other software component of the same system as another agent in the worlds is inadequate. Our software components have the ability of proper coordination, as they are parts of the same system. We have already demonstrated that this can significantly reduce the search space of model-checkers as the space considered is not the Cartesian product of the software components, but can be much smaller [9].

We show here that logic-labeled finite-state machines are effective in carrying out the "correct from construction" agenda of formal methods such as *Event-B* while simultaneously achieving the aims of MDD. As a result, we obtain models that are directly interpretable, compilable and executable enabling traceability, transparency and rapid maintainability. Such LLFSMs models can be simulated, validated, and formally verified with model checking tools such as `NuSMV` [10], [11]. Moreover, the capacity of *Event-B* to develop *closed models* is also very natural with arrangements of LLFSMs; and therefore further safety analysis such as Failure Mode and Effects Analysis (FMEA) can be performed [12]. We use two

978-1-4799-3149-1/14 $31.00 © 2014 IEEE
DOI 10.1109/ASWEC.2014.20

TABLE I. CAR-BRIDGE REQUIREMENTS.

Req.	Description
FUN-1	The system is controlling cars on a bridge connecting the mainland to an island.
ENV-1	The system is equipped with two traffic lights with two colours: green and red.
ENV-2	The traffic lights control the entrance to the bridge at both ends.
ENV-3	Cars are not supposed to pass on a red traffic light, only one a green one.
ENV-4	The system is equipped with four sensors with two states: on and off
ENV-5	The sensors are used to detect the presence of a car entering or leaving the bridge: "on" means a car is willing to enter the bridge or leaving it.
FUN-2a	The number of cars on the bridge is limited but cannot be negative.
FUN-2b	The number of cars on the island is limited but cannot be negative.
FUN-3	The bridge is one-way with the direction switched by the traffic lights.
FUN-4	The system runs indefinitely. Cars can always leave the compound, but only enter if not full.

well-known examples from the literature to demonstrate this.

II. THE BRIDGE CONTROLLER

We use the controller of the "cars on a bridge" example [8, Chapter 2] to present logic-labeled finite-state machines (LLFSMs) and to illustrate the claims we anticipated earlier in the abstract and the introduction. The requirements of the controller for the system that regulates cars crossing a bridge between the mainland and an island are reproduced in Table I[1] (they closely follow their initial description [8, Page 25-26 and Page 49]). Those requirements labeled FUN correspond to functional requirements, while those labeled ENV correspond to requirements concerned with the environment [8]. This is due to the fact that, in *Event-B*, one models the system and the world surrounding the system.

> "Note that the models we are going to construct will not just describe the control part of our intended system, they will also contain a certain representation of the environment within which the system we build is supposed to behave. In fact, we shall quite often essentially construct *closed models*, which are able to exhibit the actions and reactions taking place between a certain environment and a corresponding, possibly distributed, controller" [8].

We will follow the original notation and development of this example as close as possible, in order to illustrate that our method also delivers correctness by construction. As in the *Event-B* method, we start with a system that observes two simple events. Thus, this is level one of the modelling.

ML_Out : A car has gone out of the mainland onto the compound of the bridge and the island.
ML_In : A car has moved away from the island/bridge compound onto the mainland.

However, we will progress slightly faster that the original presentation. We have refined the requirements, because the original model fails to keep the number of cars in the compound positive. A similar situation would happen in our models. A simpler LLFSM than the one shown in Fig. 1 shows the shortcoming both during simulation or when we produce the corresponding Kripke structure and attempt to verify the properties with a tool such as `NuSMV` [13].

The LLFSMs structure follows the transducer or automata format [14]. They consist of a set S of states, and a transition function $T : S \times E \to S$. There is a distinguished state $s_0 \in S$, named the initial state. In Fig. 1 the set $S = \{$INITIALIZING, INCREASING, DECREASING, CAR_OUT, CAR_IN$\}$, and the initial state is the INITIALIZING state.

[1]Larger tables and figures at `mipal.net.au/aswec2014.pdf`.

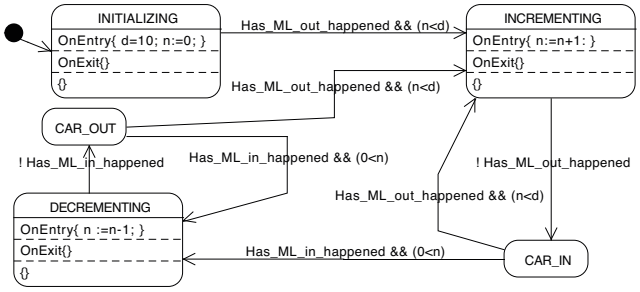


Fig. 1. LLFSM model for level one of the car-bridge.

However, as opposed to UML and *Event-B*, the set E is not a set of *events* but a set of Boolean expressions (in their most general form, our LLFSMs use a decidable logic and allow expressions from such a logic to label the transitions; for example, in many case studies we have used a common sense non-monotonic logic, Decisive Plausible Logic (DPL [15])). That is why they are called *logic-labeled* finite-state machines. Note that UML and *Event-B* have something similar named *guards*. Here, the fact that an event happened in the system is actually incorporated by a predicate. Thus, the predicate `has_ML_Out_happened` will label the transition (rather than the event ML_Out) as the predicate will evaluate to TRUE or FALSE in accordance with the event.

This is because the semantics of LLFSM corresponds to a synchronous execution model rather than an event-driven model. T is usually a *partial function*, that is, there are pairs (s_i, e_t) for which T is not defined; so, T is usually called the transition table. The standard, general description of the semantics for $T(s_i, e_t) = s_j$ is that when the machine is in state $s_i \in S$ and if the expression e_t evaluates to TRUE, the machine will move to the state s_j. However, this requires that, (for any $t \neq s$), if $T(s_i, e_t)$ and $T(s_i, e_s)$ are defined and $T(s_i, e_t) \neq T(s_i, e_s)$, then e_t and e_s never be TRUE simultaneously. The presentation of LLFSMs is simplified by making the projection of T on each state a sequence instead. That is, $T(s_i, e_t) = s_j$ will cause a transition to state s_j if e_t evaluates to TRUE and no previous expression e_s evaluates to TRUE ($\forall s < t$ in the sequence $T(s_i, \cdot)$). Note that while this is simply syntactic sugar, it does make the task of the behaviour designer a lot simpler. In Fig. 1, three transitions have the same label: from INITIALIZING to INCREASING, from CAR_OUT to INCREASING, and from CAR_IN to INCREASING. This is the label

$$\texttt{has\_ML\_Out\_happened} \;\&\&\; n < d.$$

This is `C++` (or `java`) syntax where && stands for the logical operator $\wedge$ (AND). Like in the *Event-B* example, the variable n stands for the number of cars in the compound, while d stands for the capacity. In fact, one observes here already the ambiguity of natural language requirements; the formalisation in *Event-B* shows that d is the total capacity of the bridge and the island.

States model a period of time where actions are being performed. The actions of a state can be classified into three sections. The thread of control executes an **OnEntry** section only upon arrival to the state, while actions in the **OnExit** section are executed only as the machine departs that state. Thus, the actions in these two sections are executed once and only once. The third section is a section for internal actions[2] that are executed only if none of the transitions fire. When the internal actions are completed, the thread of execution returns to evaluate the sequence of expressions that label transitions out of the state and the cycle is repeated. We refer to one pass over the cycle as a *ringlet*. It should not be surprising now that if the actions in the corresponding sections are executable code, the model is executable (and, in fact, we have several prototypes that can interpret or compile the model in Fig. 1). There is one more aspect regarding the execution of LLFSM. An arrangement of these executes on top of a data structure named the *whiteboard*[3]. In fact, when an event happens (e.g. ML_Out), the sensor or device that detects it will modify the *whiteboard variable* `has_ML_Out_happened` and change its value from FALSE to TRUE. Variables that have the scope of one LLFSM are called *local variables*. Whiteboard variables are shared across LLFSMs and are of two types. **Internal variables** are shared by all the LLFSMs in the arrangement (that is, their scope is all the states of the LLFSMs in the arrangement). Finally, **external variables** are variables whose scope goes even beyond the arrangement of the LLFSMs and in embedded systems, are variables that are set by external sensors or are set to activate effectors and actuators. Thus, `has_ML_Out_happened` is an external variable. However, the variables n and d in Fig. 1 are internal variables.

The semantics of LLFSM [6], [7] postulates that the execution of ringlet performs only one **read** operation by which a local copy of external variables is made in the scope of the current LLFSMs. Such a read happens before the execution of any section or the evaluation of any transition-labeling expression. That is, all execution in a ringlet is in the same context that is not modified by any other concurrent LLFSM or any external event (e.g. a new sensor reading). If no transition fires and the internal actions complete, when a new ringlet commences, a new read of the external scope will take place. All writes of external or internal variables by a LLFSM take place immediately in the local context[4]. When the ringlet terminates, the local copy variables of whiteboard variables are reflected into the whiteboard, this is the **write** step.

It is very important to stress again the point made earlier, our software decides when to query if an event has happened, and not the other way around, where an external event causes an interrupt and a transition. It is our software that has the control of when things are executed.

Our executable models fulfil the aims of MDD, in direct contrast with the *Event-B* method.

> "In no way is the model of a program, the program itself. But the model of a program and more generally of a complex computer system, although not executable, allows us to clearly identify the properties of the future system and to prove that they will be present in it." [8, Page 13].
> "Note again that, as with blueprints, the basis is lacking: *our model will thus not in general be executable*" [8, Page 17].

In particular, Fig. 1 can be interpreted or compiled, and executed. Our suite of tools enable to simulate the environment and to supply and modify values for the external variables `has_ML_Out_happened` and `has_ML_In_happened`. However, we can conduct formal verification, as with *Event-B*. For example, we can test that always $n < 0 \wedge n \leq d$, or that d remains constant (that is the compound never overfills). Moreover, we can actually verify that once $n = d-1$, any alternation of the values of `has_ML_Out_happened` (between TRUE and FALSE) will not change the value of n. We also can verify additional properties, e.g. `has_ML_Out_happened` must be set to FALSE before the setting of `has_ML_In_happened` to TRUE causes the value of n to decrease. We achieve all this using our tool that generates the Kripke structure (as source code for `NuSMV`) from the model in Fig. 1. Then, the properties are written in Linear Temporal Logic (LTL) or Computation-Tree Logic (CTL) and formally checked. More importantly, we can also prove deadlock freedom; that is, that under the assumption $0 < d$, the statement $0 < n \ \vee \ n < d$ is TRUE in all future states.

By contrast, at this stage, the *Event-B* approach can only prove $n < 0 \ \vee \ n \leq d$, and that $n \in \mathbb{N}$ as well as freedom from deadlocks.

A. The second level

In the process of progressively developing models that are correct, the next construction level incorporates two more *events* [8]. So we reproduce this level to illustrate the analogous progression with LLFSM.

IL_In : A car has gone out of the bridge onto the island.
IL_Out : A car has moved off the island onto the bridge.

In this case, the software keeps track of the operation with the following internal variables.

`n_b2i` : the number of cars on the bridge heading towards the island.
`n_i` : the number of cars on the island.
`n_b2m` : the number of cars on the bridge heading towards the mainland.

So, $n =$ `n_b2i`$+$`n_i`$+$`n_b2m` and the invariant *Event-B* focuses on is $(\texttt{n\_b2i} == 0) \ \vee \ (\texttt{n\_b2m} == 0)$. That is, enforcing the bridge is one-way (in one direction or the other).

In this case, we can execute the model shown in Fig. 2. In this execution, we can validate the desired properties or we can alternatively produce the corresponding Kripke structure and prove them using `NuSMV`. Each of the corresponding properties need some re-writing. For example, freedom from deadlocks is to prove that in all states

$$\begin{aligned}(\texttt{n\_b2m} > 0) \quad &\vee \quad ((\texttt{n\_b2i} + \texttt{n\_i} < d) \wedge (\texttt{n\_b2m} == 0)) \\ &\vee \quad (\texttt{n\_b2i} > 0) \\ &\vee \quad ((\texttt{n}_\texttt{i} > 0) \wedge (\texttt{n\_b2i} == 0)).\end{aligned}$$

[2]In UML known as the `do` section.

[3]The whiteboard facilitates a data-driven, anonymous publish-subscribe mechanism similar and mappable to ROS topics.

[4]Interestingly, *Event-B* implicitly makes a similar assumption about the atomicity of the statements in the **then-end** block of guards for events; namely, none of these actions can be interrupted by another event.

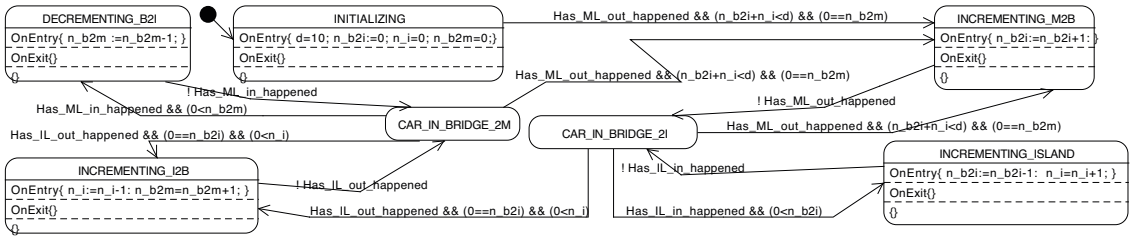
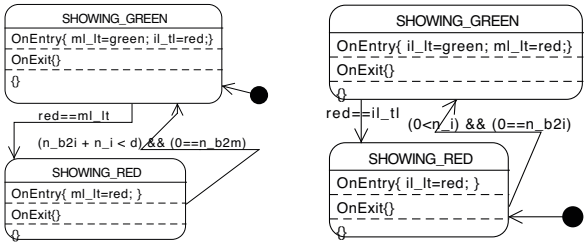
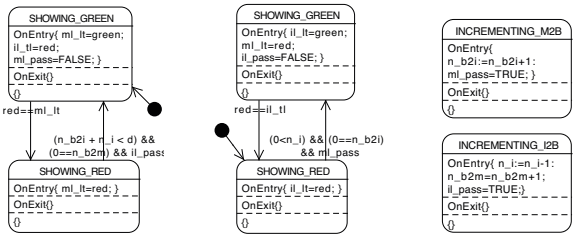


Fig. 2. LLFSM model for level two of the car-bridge.



(a) The mainland light. (b) The island light.

Fig. 4. Lights models with no delay as [8, Sec. 2.6.1 to 2.6.7].



(a) The mainland light. (b) The island light. (c) The replaced states.

Fig. 5. Modification to Fig. 3 to create the forced-alternation model of [8, Sec. 2.8.8].

B. The refinement of introducing the lights

In this case, the system will require two new variables that represent the belief the software has for what colour (GREEN or RED) the corresponding light is showing.

`ml_lt` : The colour of the light on the mainland side.
`il_lt` : The colour of the light on the island side.

We use this example (which is the core introductory example of *Event-B* [8, Chapter 2]) to show the power of LLFSM. Fig. 3 shows our preferred new model at this stage using LLFSM, which, as before, is fully interpretable, can be simulated, and can also be formally verified by generating its Kripke structure as input to `NuSMV`. We would like to stress here the strength of the modelling by LLFSM. One could have modelled the control of the lights as per Fig. 4, corresponding to the *Event-B* model discussed in [8, Sec. 2.6.1 to 2.6.7]; thus LLFSM is fully able to represent that model. But such a model (replacing Fig. 3a and Fig. 3b with Fig. 4) results in a behaviour that (as pointed out in Sec. 2.6.8 of the same source), once there are some cars on the island but none on the bridge, would swap the lights uncontrollably between two states:

1) The mainland traffic light is GREEN and the island traffic light is RED.
2) The mainland traffic light is RED and the island traffic light is GREEN.

Thus, the *Event-B* solution is to force that once one or more cars have crossed from the mainland to the island and the bridge is free, the system sets the mainland traffic light to RED and the island traffic light to GREEN. This is a strange solution, because it forces at least one car on the island to come back to the mainland before more cars can cross over from the mainland. This is an awkward solution. For example, if the capacity is 4 cars, this solution allows two cars to go onto the bridge, then one car to go from the bridge onto the island. While at least one car is still on the bridge, a third car can go onto the bridge. But if the bridge becomes empty (e.g. the two cars have moved to the island), then, despite not exceeding capacity, the fourth car would not be allowed onto the bridge until one of the three cars on the island has left for the mainland. This seems unreasonable as further cars can be forced to wait unnecessarily when there is still capacity left.

To construct this awkward solution, *Event-B* must introduce two Boolean variables (`ml_pass` and `il_pass`), and prove more invariants (an also a variant, showing that the lights will always alternate between the settings above). This constraint, that cars must go in packs and forcibly alternate directions, is simply due to the fact that *Event-B* cannot model execution time (and delays), while LLFSMs can. With LLFSMs, since the delay is controlled by the states for holding the setting of a light, we can use LTL and CTL to verify these properties and establish that each light will always hold its setting.

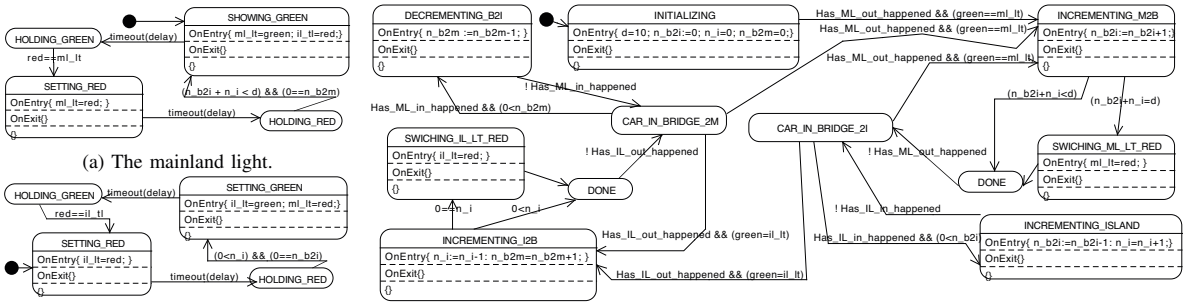
Of course, we can construct an LLFSM model as per *Event-B*'s solution [8, Sec. 2.8.8] as well (see Fig. 5). All models satisfy the condition that the traffic lights are never GREEN simultaneously. That is, in all states, the following will hold:

$$\texttt{ml\_tl} == \text{RED} \ \vee \ \texttt{il\_tl} == \text{RED}. \tag{1}$$

Note that, to avoid reproducing the lengthy LTL formulas and save space, we omit here all the intermediate steps in the Kripke structure our tool generates for the atomic execution of a state in a LLFSM (see the earlier discussion on the **read** and **write** stages of a ringlet execution).

In all cases we can prove deadlock freedom. E.g. for Fig. 5, we must show that in all future states we have

$$\begin{aligned} \texttt{n\_b2i} > 0 \quad & \vee \quad (\texttt{ml\_tl} == \text{RED} \ \wedge \texttt{n\_b2i} + n_i < d \ \wedge \texttt{n\_b2m} == 0 \ \wedge ml\_pass \wedge il\_pass) \\ & \vee \quad (il\_tl == \text{RED} \ \wedge \texttt{n\_b2i} == 0 \ \texttt{n\_i} > 0 \ \wedge ml\_pass \wedge il\_pass) \\ & \vee \quad \texttt{ml\_tl} == \text{GREEN} \\ & \vee \quad \texttt{il\_tl} == \text{GREEN} \\ & \vee \quad \texttt{n\_b2m} > 0. \end{aligned}$$

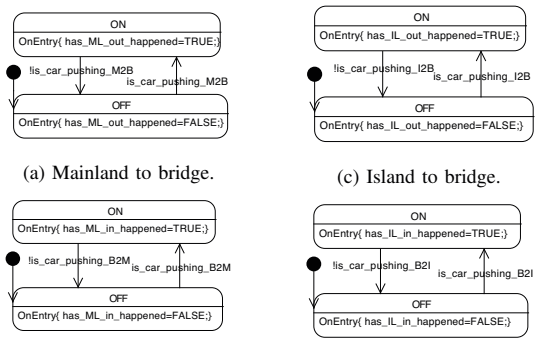


(a) The mainland light.

(b) The island light.

(c) The central controller.

Fig. 3. Versatile model at the level of lights for the car-bridge controller.



(a) Mainland to bridge.

(c) Island to bridge.

(b) Bridge to mainland.

(d) Bridge to island.

Fig. 6. The 4 sensors behaviour models.

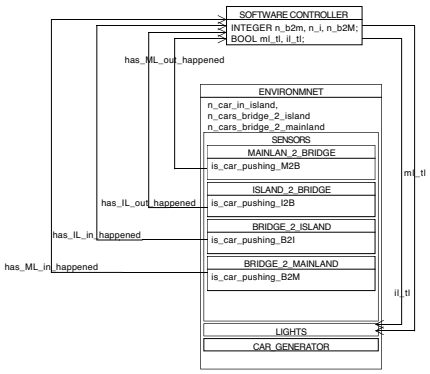


Fig. 7. The communication channels between the environment and the software controller in Fig. 3.

For our original model (Fig. 3), not only can we prove freedom from deadlocks, but also statements such as "after the last car leaves the bridge heading for the island, the island traffic light will be GREEN for the `delay` period or a car will enter the bridge from the island". Similarly, we can prove that "necessarily, after the mainland traffic light has been GREEN for `delay`, the mainland traffic light will turn RED and the island traffic light will turn GREEN".

C. The closed world model

The *Event-B* approach and the LLFSM approach "make clearer the separation between the *software controller* and the *physical environment*" [8, Page 89]. *Event-B* models the sensors, the cars and the traffic-lights as it builds "a *closed model* corresponding to the complete mathematical simulation of the pair formed by the software controller and the environment" [8, Page 89]. Thus, *Event-B* forces the behaviour of the environment and formally checks that "the controller works in harmony with the environment" [8, Page 89]. For LLFSMs, Fig. 3 corresponds to the software controller. LLFSMs can also model the behaviour of all the deterministic elements of the environment. So, in analogy to the presentation of *Event-B* [8, Sec. 2.7], we also present here the case of the four sensors in the car bridge example. We emphasise that these are environment components placed on the bridge, the mainland, or the island to detect a car going onto the bridge from the island or from the mainland, as well as detecting a car going out of the bridge to the island or to the mainland. Fig. 6 shows the LLFSMs for the four sensors. All sensors can be in one of two states, ON or OFF. When a car is triggering the sensor, the sensor is ON, else it is OFF. As per the *Event-B* presentation, we also skip modelling the lights themselves, since it suffices to present the model of the communication channels (Fig. 7). Fig. 7 is analogous to [8, Fig. 2.11], but provides more details on the source and target of the communication channels. We highlight the terminology used, as with *Event-B*. Thus, Table II details the meaning of the variables used. Controller variables, input channels, and output channels are variables in our architecture, in particular, on the whiteboard. The controller variables capture in software what the controller believes about the environment. For example, the variable `n_i` holds what the software believes to be the number of cars on the island. Environment variables are in italics, representing the actual numbers in a particular scenario. The system has little or no control over them. This is an important difference between LLFSMs and *Event-B*. The latter uses a model of control over environment variables (making sure the environment plays fair). So, in *Event-B*, a driver will never run a red light, for example. In LLFSM, we assume no control over these events in the environment and thus, we do not impose these constraints as invariants. In the context of Fig. 7, the behaviour of the cars in the environment is free (although such behaviour could also be modelled, for example, if consequences of certain behaviour are to be examined). We leave this as a non-deterministic aspect that is captured in the Kripke structure in `NuSMV` reacting to the environment changing the corresponding external variables at any time in any way. We believe this to be more interesting, as we can now also verify the behaviour of the controller when the environment does not play fair.

TABLE II. THE TYPES OF VARIABLES USED.

Input channels	has_ML_Out_happened, has_ML_In_happened, has_IL_Out_happened, has_IL_Out_happened
Controller	n_i, n_b2i, n_b2m
Output channels	ml_tl, il_tl
Environments	$n\_car\_in\_island$, $n\_car\_bridge\_2\_island$, $n\_car\_bridge\_2\_mainland$

One further advantage of modelling the controller and elements of the environment is that we can perform fault-injection analysis and generate FMEA tables [10], [12], [9], [16]. For example, we can evaluate the impact of a change in the controller (or an element of the environment) of all the properties and invariants already proven when the system is correct. For example, we can modify the transition-guarding expressions in Fig. 3 or the statements inside the states. The effects of such a fault-injection in the model can then be observed by running the model-checker NuSMV again, but with this faulty model as input. Similarly, a bad sensor can be produced by injecting a fault into one or the models of Fig. 6, and running NuSMV on the fault-injected model.

To conclude the comparison between *Event-B* and LLFSMs for this case, we emphasise what *Event-B* postulates as a requirement: "The controller must be fast enough so as to be able to treat all the information coming from the environment" that must be "checked" [8, Page 95]. The tools provided by *Event-B* cannot establish that the system is fast enough, this becomes an assumption. What is proven, is that, if the environment behaves as modelled "the controller, although working with slightly time-shifted information concerning n_b2i, n_i and n_b2m, nevertheless maintains the basic properties of the physical numbers of cars $n\_car\_in\_island$, $n\_car\_bridge\_2\_island$, and $n\_car\_bridge\_2\_mainland$". With LLFSM, we can establish formal claims regarding the minimum number M of state transitions required for the software controller to carry out the action in response to the stimulus from the environment [11]. These formally verified statements are of the form, *if the sensor is pushed for and held for M Kripke-state transitions, then the system would have necessarily updated the output channel X*. In this way, we establish that the controller is fast enough to handle the inputs from the environment.

III. THE CAR-WINDOW CONTROLLER

We now proceed to illustrate the iterative refinement of LLFSMs with an example that expands on the power of labeling the transitions with logic expressions. The *power window controller with obstacle detection capability* (PWC-OD)[5] is a common case study of software modelling for the car industry [17]. It consists of a passenger-side window that can go up or down using two switches, one for the driver and one for the passenger. Each switch is in one of 3 states: indicating-Up, indicating-Down, or IDLE. There is a sensor to detect whether the window is fully up or fully down, while another sensor detects if the up motion is being impeded by an obstacle. We describe the requirements in Table III. We note that the original version by MathWorks indicated that the driver's controller takes precedence over the passenger's controller. While the original source indicates this

[5]http://www.mathworks.com.au/help/simulink/examples/simulink-power-window-controller-specification.html

TABLE III. Car_Window_PWC-OD REQUIREMENTS.

Req.	Description
R 1	Both driver and passenger can control glass door movements using their own up/down switches.
R 2	When the glass is at the top position then the up command will not have any effect.
R 3	When the glass is at the bottom position then the down command will not have any effect.
R 4	A driver command has higher priority over a passenger command; when both up and down switches are pressed (by driver or passenger) at the same time, with contradictory signals, the driver's command is the one the system responds to.
R 5	When the window is moving up, and an obstacle is detected, the glass moves down for a prescribed duration, or until the lower position is reached, whichever happens first. During this time, commands from the driver or the passenger are ignored.
R 6	If an up button is pressed and released before a threshold time limit, then it is interpreted as an auto-up command, and the window rolls up to its top limit; however, if the button is pressed for more than the threshold value, then the glass moves up step by step till the button is released or the top limit is reached; similar behaviour occurs when the down button is pressed.

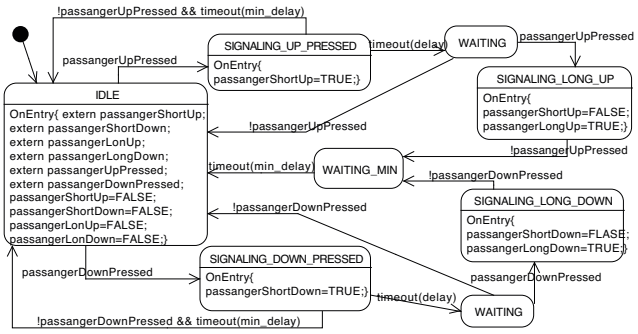


Fig. 8. The LLFSM for the button of the passenger.

is an override, others suggest that contradictory directions halt the movement [17]. This again illustrates how easy is for informal requirements to be incomplete or inconsistent. For our purposes, requirements as per Table III shall suffice.

With the introduction to arrangements of LLFSMs, it should not be very surprising how the software controller handles the sensors. Fig. 8 shows the software controller receiving input channels passangerUpPressed or passangerDownPressed corresponding to the control button on the passenger side. The LLFSM in Fig. 8 uses these signals to distinguish between a long and a short press. Accordingly, one of passangerShortUp, passangerShortDown, passangerLongDown, or passangerLongUp is set on the whiteboard; once the button is released, all these variables are reset. The constant delay is the lower bound for considering the signal as a long press. The constant min_delay is very small, and ensures that the arrangement of LLFSMs executes a ringlet in each machine so even a very short button press (on the order of microseconds) is detected by the system as a short press. An analogous LLFSM handles the drivers button (just replacing passenger with driver); due to its simplicity and for reasons of space, we omit it here.

The software controller for the car window has a third LLFSM that controls the motor (and uses the signals of the two previous LLFSMs as inputs, in addition to the external variables atTop and atBottom that correspond to the sensor signals indicating whether the window has reached the top or the bottom). Finally, there is the external variable obstacleFound that is TRUE when the obstacle sensor detects an obstruction.

Fig. 10 presents the third and last component of the software controller for this case study. For those famil-

```
name{SHALLGOUP}.
input{passangerLongUp}. input{passangerShortUp}.
input{driverLongUp}.    input{driverShortUp}.
input{driverLongDown}.  input{driverShortDown}.
input{obstacleFound}.
input{atTop}.

UP0: {} => ~shallGoUp.

UP1: passangerLongUp => shallGoUp.  UP1>UP0.
UP2: passangerShortUp => shallGoUp. UP2>UP0.
UP3: driverLongUp => shallGoUp.     UP3>UP0.
UP4: driverShortUp => shallGoUp.    UP4>UP0.

UP5: driverLongDown => ~shallGoUp.  UP5>UP1. UP5>UP2.
UP6: driverShortDown => ~shallGoUp. UP6>UP1. UP6>UP2.
UP7: obstacleFound => ~shallGoUp.   UP7>UP1. UP7>UP2. UP7>UP3. UP7>UP4.
UP8: atTop => ~shallGoUp.           UP8>UP1. UP8>UP2. UP8>UP3. UP8>UP4.

output{b shallGoUp,"shallGoUp"}.
```

Fig. 9. DPL coding for the predicate `ShallGoUp`.

iar with the 30 page *Event-B* specification, its simplicity may be surprising. The reason is that only motor control states need to be modelled: Not_Moving, when the motor is OFF, Moving_Down, when moving down in automatic mode, or Moving_Down_Manual, when moving down in manual mode. Analogously for moving up, except that here, `delay_4_obstacle` can be triggered by an obstacle. All the complexity is handled by the logic-labeled transitions. Some of these transitions are immediate. Whenever Moving_Down and `atBottom` becomes TRUE, we shall transition to Not_Moving. Similarly, for going up and `atTop`.

The other transitions are developed by iterative refinement using DPL [15]. We start with the predicate `shallGoUp`. The first level of development of a DPL logic for a transition predicate is as follows. We identify the output, and this is the last line in Fig. 9. We give a name to the theory (this is the first line), and the desirable inputs (these are the declarations with the clause `input`).

The next level is to set up an advisor recommending whether the motor `ShallGoUp` or not. We set up a default rule `UP0`, advising not to go up if we have no information.

A third level of development is to consider what information would make us reverse our earlier decision. If we know that `passangerLongUp` is TRUE, then we should recommend going up. This is rule `UP1` and we indicate that this defeats `UP0`. The next 3 rules (`UP2`, `UP3`, `UP4`) are similar for the passenger or the driver indicating automatic or manual window up. Each of these defeat `UP0`.

The fourth level of iterative refinement analyses if more information suggests we reverse an earlier decision. Because the driver button has priority over the passenger button, knowledge of the driver indicating down is reason to not recommend the motor going up even if the passenger is requesting up. These are rules `UP5` and `UP6`, with each defeating both of the rules where the passenger indicated motor up. Finally, knowledge of an obstacle or the fact that the window has reached the top is evidence to reverse all rules suggesting going up. These are rules `UP7` and `UP8`.

The theory for when the system shall change from the Not_Moving state to Moving_Down is analogous and appears in Fig. 11. Thus, the next stage is to handle when the window should move up manually given that it otherwise would move up automatically; this is the transition labeled `shallGoUpManual` from Moving_Up to

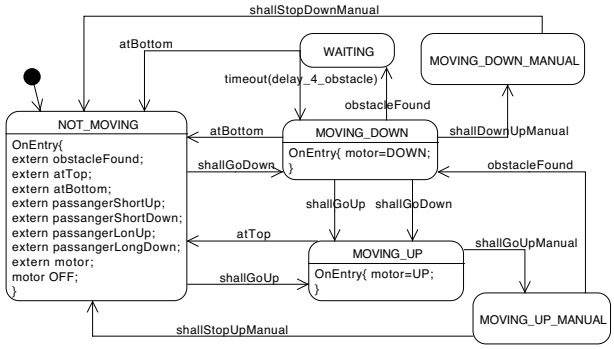


Fig. 10. The LLFSM for the motor of the window.

```
name{SHALLGODOWN}.
input{passangerLongDown}.   input{passangerShortDown}.
input{driverLongUp}.        input{driverShortUp}.
input{driverLongDown}.      input{driverShortDown}.
input{obstacleFound}.
input{atBottom}.

DN0: {} => ~shallGoDown.

DN1:passangerLongDown => shallGoDown.  DN1>DN0.
DN2:passangerShortDown => shallGoDown. DN2>DN0.
DN3:driverLongDown => shallGoDown.     DN3>DN0.
DN4:driverShortDown => shallGoDown.    DN4>DN0.

DN5:driverLongUp => ~shallGoDown.      DN5> DN1. DN5>DN2.
DN6:driverShortUp => ~shallGoDown.     DN6> DN1. DN6>DN2.

DN7:obstacleFound => shallGoDown.      DN7> DN6. DN7>DN5. DN7>DN0.

DN8:atBottom => ~shallGoDown.          DN8> DN7. DN8>DN4. DN8>DN3.
                                       DN8>DN2. DN8>DN1.

output{b shallGoDown,"shallGoDown"}.
```

Fig. 11. DPL coding for the predicate `ShallGoDown`.

Moving_Up_Manual. The logic theory for this transition appears in Fig. 12. The development of this logic is simple. It will pronounce itself about the predicate `shallGoUpManual`, and thus the last line indicating its output. The first line is the name and the theory, and given that the motor is going up we only need information about `passangerLongUp` or `driverLongUp`. If neither of these has been asserted, then rule `MUP0` states it should not be in "manual going up" state. However, if we learn that `passangerLongUp` or `driverLongUp` have been asserted, then we should switch to manual and these are rules `MUP1` and `MU2` that defeat `MUP0`.

The logic theory for switching from Moving_Down to Moving_Down_Manual is now analogous to the theory in Fig. 12 and can be constructed by replacing `Up` for `Down`; thus it is omitted.

So we move on to the predicate to halt the motor mov-

```
name{SHALLGOUPMANUAL}.
input{passangerLongUp}.  input{driverLongUp}.

MUP0: {} => ~shallGoUpManual.

MUP1: passangerLongUp => shallGoUpManual.  MUP1>MUP0.
MUP2: driverLongUp => shallGoUpManual.     MUP2>MUP0.

output{b shallGoUpManual,"shallGoUpManual"}.
```

Fig. 12. DPL coding for the predicate `shallGoUpManual`.

```
name{SHALLSTOPUPMANUAL}.
input{passangerLongUp}.  input{driverLongDown}.
input{driverShortDown}.  input{driverLongUp}.  input{atTop}.

SUP0: {} => shallStopUpManual.

SUP1: passangerLongUp => ~shallStopUpManual.  SUP1> SUP0.

SUP2:driverLongDown => shallStopUpManual.     SUP2> SUP1.
SUP3:driverShortDown => shallStopUpManual.    SUP3> SUP1.

SUP4:driverLongUp => ~shallStopUpManual.      SUP4> SUP2. SUP4>SUP3.

SUP5: atTop => shallStopUpManual.             SUP5>SUP4.

output{b shallStopUpManual,"shallStopUpManual"}.
```

Fig. 13. DPL coding for the predicate `shallStopUpManual`.

ing up manually; this is `shallStopUpManual`. The DPL theory for this predicate appears in Fig. 13 The development by iterative refinement of this theory follows the same method. The last line is the identification of the predicate `shallStopUpManual` as the output and the first line is the identification of the theory. To make decisions about whether or not the manual raising of the window shall stop the desirable inputs are those listed with an `input` clause. If nothings is known, it is safer to stop (we need active signals from the driver or the passenger to keep going up in manual mode). This is the default rule `SUP0`. Analysing what could reverse this determination, it could be that the passenger is controlling the window manually going up. This is rule `SUP1`.

But, since the driver button has precedence over the passenger, `driverLongDown` or `driverShortDown` becoming TRUE revises the conclusion. However, if the driver is indicating manual control up, we reverse this. This is now rule (`SUP4`. The final stage of the step-by-step development is that, if `atTop` became TRUE, then we do have to stop the motor pushing the window up. This is rule `SUP5` that defeats `SUP4`. The logic for stopping when going down is analogous and will therefore also be omitted here.

This completes the modelling. This model can now be interpreted and simulated. It is a complete software controller for the car-window consisting of the 3 LLFSMs (two sensors and one motor) and the logic theories for the predicates `shallGoUp`, `shallGoDown`, `shallGoUpManual`, `shallGoDownManual`, `shallStopUpManual`, and `shallStopDownManual`. The entire model fits in about one page. We anticipate the learned software engineer would have significant ease in grasping the behaviour from the presentation of the model. In the humble opinion of these authors, the 30-Page *Event-B* specification [18] of the powered window using UML-B state-machines, by comparison, is impenetrable.

If the LLFSMs model is to be compiled, or if it is going to be formally verified by constructing its Kripke structure and performing model-checking with `NuSMV`, the logic theories are converted into Boolean expression using the the `+c` option of the DPL tools (http://www.ict.griffith.edu.au/arock/DPL/DecisivePL.pdf). With this translation, the predicate `shallGoUp` defined by the logic theory in Fig. 9 is the expression in Fig. 14.

This expression is actually quite simple to interpret now. The motor shall go up if any of the buttons (driver or passenger) indicate up, but not when the window has hit the top or an obstacle, nor if the driver is indicating down.

```
shallGoUp≡   ( driverLongUp
               || driverShortUp
               || passangerLongUp
               || passangerShortUp)
          &&!
               ( atTop
               || obstacleFound
               || driverLongDown
               || driverShortDown).
```

Fig. 14. Simple-C expression for the logic theory in Fig. 9.

For the logic theory of the predicate `shallGoDown`, we obtain the following logic expression that can be compiled for `C++` (ot `java`) as well.

```
shallGoDown≡ !atBottom
             &&( obstacleFound
               ||( !driverLongUp && !driverShortUp
                 &&( driverLongDown
                     ||driverShortDown
                     ||passangerLongDown
                     ||passangerShortDown
                     )
                 )
               ).
```

Again, once our process is performed, the logic expression above seems obvious. The window shall go down when its not at the bottom, and if an obstacle has been found; or when the driver or the passenger indicate to go down; the last case provided that the driver is not signaling up. The expression for `shallGoUpManual` is also obtained by running the DPL tools with the `+c` option with the corresponding input from Fig. 12. And this case is simple, probably one could figure it out without DPL.

```
shallGoUpManual ≡
    driverLongUp || passangerLongUp.
```

Thus, we move to the compilation of the logic theory for `shallStopUpManual` that was presented in Fig. 13.

```
shallStopUpManual≡ atTop||(
                              !(driverLongUp
                          &&( driverLongDown
                              ||driverShortDown
                              !passangerLongUp
                              )
                          ).
```

A. *Model-checking illustration*

To illustrate some of the model-checking we can perform, we now present some simple examples of properties that the model of the car-window satisfies. For example, the LLFSM that handles the input signals `passangerSensorUpPressed` and `passangerSensorDownPressed` for the passenger (see Fig. 8) should be such that it never forwards contradictory signals to the LLFSM in Fig. 10. That is, of the output signals it produces (see Fig. 16) non two simultaneously can be TRUE. The CTL [19], [20] formula (using `NuSMV` syntax) in Fig. 15 says that, in all states of the Kripke structure it is globally true (thus in all future states) that if the signal `passangerShortUp` is TRUE, then the other 3 signals must be FALSE (except if this is the very first execution of the initial

```
SPEC
AG (    (passangerShortUp = 1 ->
                ((passangerShortDown = 0
                & passangerLongDown = 0)
                & passangerLongUp = 0))
        | pc = M0S0R0)
```

Fig. 15. CTL formula that verifies that only `passangerShortUp` is TRUE once some computation has happened.

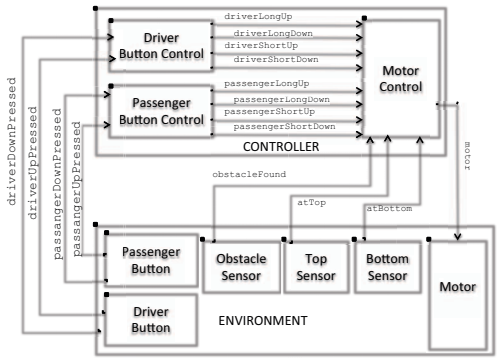


Fig. 16. Communication channels of the car-window controller and its environment; the closed world model.

state). While the reader may find it surprising that we make a statement about the program counter (the variable `pc` in the Kripke structure), it is simply because we make absolutely no assumption about the state of any of the variables at the start of computation. If the term `pc = M0S0R0` is removed, `NuSMV` indicates the CTL formula is false and exhibits as evidence a trace with an initial Kripke-state where no computation has happened at all (where indeed two or more of the output signals of Fig. 8 could be TRUE). However, with the slightest progress of the computation, the initialization of the IDLE takes place, and from there on there will never be a future state where contradictory signals can be produced. Thus, this shows the strength of our model-checking that it makes no assumption about the state of variables when computation starts; and thus, ensures that the model properly initializes them.

Clearly, analogous CTL expressions are verified for the other four signals out of the button for the passenger.

The reader may notice that it would be possible to have just one channel of communication between the LLFSM from the button controller to the motor controller. This would then be a whiteboard variable called `passangerButtonSetting` that now takes one of possible 5 values (as opposed to the current 4 Boolean variables; recall that all these Boolean variables set to FALSE means the button is idle). This is, in fact, what is done with the channel out of the motor-controller into the motor, where a variable `motor` takes the values `Up`, `Down` or OFF. LLFSMs allows for both approaches. Although at the level of the model, this may be a matter perhaps of modeling style, later on we would indicate that using a single variable with 5 possible values is better as it reduces the size of the generated Kripke structure.

Other safety properties we can test using `NuSMV` and the Kripke structure derived from the LLFSM model are those indicating that the driver's control take precedence over the passenger control. For example, checking that *there is never a state where the motor is going down as the passenger is signaling down with a long press (for manual control) and at the same time the driver is signaling a short up* takes the following CTL from in `NuSMV`.

```
SPEC
!  EX ( passangerLongDown =1
       & driverShortUp = 1
       & motor = Down )
```

Naturally, we can also check that the passenger cannot be driving the window down and the motor actually going down if the driver is signaling up with a long press. While these are two cases where the manual driving down of the passenger is overwritten by the driver, the same can be done for the passenger's control in auto mode. And therefore, for the four cases when the passenger's desire is to move the window up but is overwritten by the driver's command to do down.

For claims regarding the forceful transition into another state, we use LTL [21], [20]. For example, if we have that the window is going up under the manual control of the driver and and obstacle is found; then the variable motor will change to `Down` after a few Kripke states where the obstacle remains present (or the other conditions change). Fig. 17 reflects the structure of the corresponding LTL formula.

Perhaps one limitation of the approach here is that model-checker software, like `NuSMV`, is subject to combinatorial explosion on the number of external variables. That is Kripke structures can grow rapidly. For this, signaling that variables like `motor` can only take 3 values (`Up`, `Down`, and OFF) is important. Similarly, the presentation we have made here where we have four Boolean variables for the output of each button switch, results in $2^4 = 16$ assignments for them that represent Kripke states. If alternatively, as already suggested, we had used one output channel (with five values) from the passenger's button control to the motor control, then the Kripke states multiply by 5 (and not by 16).

IV. Conclusion

Here, we have shown that LLFSMs have the same power as *Event-B* for modelling both the environment and its controlling computer system and software. We how shown that LLFSMs support development in stages (or levels) and iterative refinement, ensuring correctness of properties for a model at the given level and identifying requirements that ought to be fulfilled in the next level (because the requirement is actually missing, or it was unclear, or was no the focus yet of the current level). But LLFSMs not only identify the capacities of the current stage by formal verification, they can also be identified by validation through MDD style simulation. Thus, LLFSMs support correctness by construction.

Moreover, we have shown that for well-established case studies, LLFSMs provide a more succinct model than *Event-B* while maintaining the ability for formal verification. Furthermore, LLFSMs can represent closed model systems that allow further safety analysis such as FMEA. Its representation

```
LTLSPEC
G ( ( obstacleFound=1 & atTop=0 & driverLongUp=1 & motor=Up ) ->
       X  ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
              X  ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
                     X  ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
                            X  ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down |
                                   X  ( obstacleFound=0 | atTop=1 | driverLongUp=0 | motor=Down
                                   )
                            )
                     )
              )
       )
) )
```

Fig. 17. Structure of the LTL formula that encodes that and obstacle found will cause the motor to switch direction to going down.

of the environment is more powerful than similar approaches as it does not require the environment to play fair or follow deterministic assumptions in order to simulate, interpret, and execute models or formally verify their properties.

ACKNOWLEDGMENT

The authors would like to thank all members of the MiPal RoboCup Team.

REFERENCES

[1] P. Mohagheghi, V. Dehlen, and T. Neple, "Definitions and approaches to model quality in model-based software development - a review of literature," *Information & Software Technology*, vol. 51, no. 12, pp. 1646–1669, 2009.

[2] B. Dobing and J. Parsons, "How UML is used," *Commun. ACM*, vol. 49, no. 5, pp. 109–113, 2006.

[3] J. Erickson and K. Siau, "Unified modeling language: The teen years and growing pains," in *Human Interface and the Management of Information. Information and Interaction Design - 15th International Conference*, ser. Lecture Notes in Computer Science, S. Yamamoto, Ed., vol. 8016. Springer, July 21-26 2013, pp. 295–304.

[4] G. Reggio, M. Leotta, F. Ricca, and D. Clerissi, "What are the used UML diagrams? a preliminary survey: Technical report," Universitá di Genova, Italy, Dipartimento interscuola di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Tech. Rep., 1998, to appear in Proceedings of 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (EESSMod 2013 co-located with MODELS 2013).

[5] J. Erickson and K. Siau, "Can UML be simplified? practitioner use of UML in separate domains," in *12th Workshop on Exploring Modeling Methods for Systems Analysis and Design (EMMSAD'07), held in conjunctiun with the 19th Conf. on Advanced Information Systems (CAiSE'07)*, vol. 365. Trondheim, Norway: CEUR, June 2007, pp. 87–96.

[6] V. Estivill-Castro and R. Hexel, "Arrangements of finite-state machines semantics, simulation, and model checking," in *International Conference on Model-Driven Engineering and Software Development MODELSWARD*, S. Hammoudi, L. Ferreira Pires, J. Filipe, and R. César das Neves, Eds. Barceloan, Spain: SCITEPRESS Science and Technology Publications, 19-21 February 2013, pp. 182–189.

[7] V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth, "Efficient modelling of embedded software systems and their formal verification," in *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, K. R. Leung and P. Muenchaisri, Eds. Hong Kong: IEEE Computer Soceity, Conference Publishing Services, 4th - 5th December 2012, pp. 428–433.

[8] J.-R. Abrial, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[9] V. Estivill-Castro and R. Hexel, "Module isolation for efficient model checking and its application to FMEA in model-driven engineering," in *ENASE 8th International Conference on Evaluation of Novel Approaches to Software Engineering*. Angers Loire Valley, France: INSTCC, July 4th-6th 2013, pp. 218–225.

[10] V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth, "Efficient model checking and FMEA analysis with deterministic scheduling of transition-labeled finite-state machines," in *2012 3rd World Congress on Software Engineering (WCSE 2012)*, P. Wang, Ed., Wuhan, China, 6th-8th November 2012, pp. 65–72.

[11] V. Estivill-Castro and D. A. Rosenblueth, "Model-checking of transition-labeled finite-state machines," in *2011 International Conference on Advanced Software Engineering & Its Applications (ASEA)*, ser. Communications in Computers and Information Science, T.-H. Kim, H. Adeli *et al.*, Eds., vol. 257. Jeju Island, Korea: Springer Verlag, December, 2011, pp. 61–73.

[12] V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth, "Failure mode and effects analysis (FMEA) and model-checking of software for embedded systems by sequential scheduling of vectors of logic-labelled finite-state machines," in *System Safety, The 7th Int. IET System Safety Conf., incorporating the Cyber Security Conference 2012*, Edinburgh, UK, 15th - 18th October 2012, cD-ROM Proceedings. Paper 3.a.1.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *Int. J. on Software Tools for Technology Transfer*, vol. 2, p. 2000, 2000.

[14] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley Publishing Co., 1979.

[15] D. Billington and A. Rock, "Propositional plausible logic: Introduction and implementation," *Studia Logica*, vol. 67, pp. 243–269, 2001.

[16] L. Grunske, K. Winter, N. Yatapanage, S. Zafar, and P. A. Lindsay, "Experience with fault injection experiments for FMEA," *Software, Practice and Experience*, vol. 41, no. 11, pp. 1233–1258, 2011.

[17] M. Satpathy, C. Snook, S. Arora, S. Ramesh, and M. Butler, "Systematic development of control designs via formal refinement," in *International Conference on Model-Driven Engineering and Software Development MODELSWARD*, S. Hammoudi, L. Ferreira Pires, J. Filipe, and R. César das Neves, Eds. SCITEPRESS Science and Technology Publications, 19-21 February 2013.

[18] C. Snook, "Power window case study models - UML-B/Event-B/Simulink," University of Southampton, Faculty of Physical Sciences and Engineering, Electronics and Computer Science, application/zip, 2012.

[19] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. Workshop on Logics of Programs*, ser. Lecture Notes in Computer Science, vol. 131, IBM Watson Research Center, 1981, pp. 52–71.

[20] M. Huth and M. Ryan, *Logic in Computer Science*, 2nd ed. UK: Cambridge University Press, 2004.

[21] A. Pnueli, "A temporal logic of programs," *Theoretical Computer Science*, vol. 13, pp. 45–60, 1981.