Efficient Modelling of Embedded Software Systems and Their Formal Verification

Vladimir Estivill-Castro  
School of ICT,  
Griffith University,  
Nathan 4111, Australia

René Hexel  
School of ICT,  
Griffith University,  
Nathan 4111, Australia

David A. Rosenblueth  
IIMAS,  
UNAM  
01000 México D.F, México

Abstract—We propose vectors of finite-state machines whose transitions are labeled by formulas of a common-sense logic as the modelling tool for embedded systems software. We have previously shown that this methodology is very efficient in producing succinct and clear models (e.g., in contrast to plain finite-state machines, Petri nets, or Behavior Trees). We show that we can capture requirements precisely and that we can simulate and validate the models. We can, therefore, directly apply Model-Driven Engineering and deploy the models into software for diverse platforms with full traceability of requirements. Moreover, the sequential semantics of our vector of finite-state machines enables model-checking, formally establishing the correctness of the model. Finally, our approach facilitates systematic Failure Modes and Effects Analysis (FMEA) for diverse target platforms. We demonstrate the effectiveness of our methodology with several examples widely discussed in the software engineering literature and compare this with other approaches, showing that we can prove more properties, and that some claims about verification in such approaches have been exaggerated or are incomplete.

Index Terms—Software testing, verification and validation, formal methods in software engineering, software requirements engineering, model-driven engineering.

I. INTRODUCTION

Today, high-level specifications of behaviours are largely represented by models using mechanisms such as finite-state machines (FSMs) or Behavior Trees. Such modelling fits well the model-driven engineering (MDE) agenda [1] for software development. The direct interpretation (or translation) of the models minimises faults (with respect to traditional software development and deployment that translates requirements into implementations). This also also results in quality software for diverse platforms and offers traceability of requirements as well as rapid refinement and adaption.

However, all this is spurious if the model is not correct in the first place. Hence the critical importance of subjecting models to formal verification. Many systems consist of components, each of which has its behaviour specified by a FSM, making the space of state combinations for the system extremely large. Traditionally, a system state has to be modelled as a combination of all of the components’ individual states, resulting in an extremely large number of combined states (combinatorial state explosion). Therefore, model-checking techniques face an enormous scalability challenge.

Proper system behaviour can be ensured when each FSM behaves properly in just a single schedule that ensures each component operates correctly and timely with respect to the others (this has additional properties such as prevention of starvation). Ensuring that all possible schedules are correct is, in fact, not necessary. As we show here, through deterministic scheduling we can obtain a concise (Kripke) structure to which one can apply state-of-the-art model-checking tools.

Using FSMs where transitions are labeled with common-sense logic statements is remarkably expressive [2] when compared with other paradigms for modelling behaviour, such as Petri Nets, Behavior Trees, and even standard FSMs, e.g., those of executable UML [3] or StateWorks [4]. In particular, for the four case studies (Microwave, Mine Pump, Industrial Press and Insulin Pump), the models constructed with FSMs are significantly more concise, usually fitting within one A4 page, while models with Behavior Trees have far more boxes and arrows, requiring multiple A4 pages. The Behavior Tree approach could then verify only a few properties (the FSMs approach proposed here verifies far more properties for those case studies). The reason is simple: the Behavior-Tree notation intrinsically uses forks [5, Fig. 3, page 1236] whose semantics [6, Fig. 3, page 4] consists of the launching of independent (parallel) threads of execution with complete freedom and rate of progress for each. Thus, the corresponding Kripke structure (whose Kripke-states encompass all possible states of the system and whose Kripke-transitions include all possible paths of execution of the system) is extremely large. Therefore, in earlier model-checking of the same case studies [5], CPU times reported are of the order of days. With the FSMs here, all model-checking was performed in a few seconds using the NuSMV shareware on a desktop computer.

More importantly, Behavior Trees have no transparent semantics regarding the interaction of the environment with the execution of the system software. Thus, the Behavior-Tree approach makes assumptions about the sensitivity of sensors, and the speed of the software with respect to consecutive sensors readings. Such prioritisation, that assumes internal events in the system happen faster than external events of the system, essentially means that the model-checking is unsound (or in the words of those authors dangerous [5, page 1249]). The approach here is free of these assumptions. We do not require initialisation guarantees on when the software starts (the software properly initialises values of its variables). Thus, the system can start in any state, and we prove the corresponding
properties correct, that is, the software is correct on all possible starting states (for example, our models are verified as correct even in surprisingly contradicting system states; in the case of the Microwave, for example, the software may start its execution with the door open, the sensor for the door indicating the door closed, the engine not running, but the variable for the motor set to on). Thus, the model-checking discussed here constitutes proper correctness verification by formal methods. Moreover, the approach here enables simulation and direct implementation. Such simulation also enables validation of the models. The Behavior-Tree approach has not reached this level of maturity, and the models published with Behavior-Trees for these case studies incur is serious faults (the Behavior Tree model [7, Fig. 7] for the Microwave model fails to ring the bell after finishing cooking [2], the Mine Pump has a bad model for the button of the supervisor [8], [9], the Industrial Press exhibits a continuous running behaviour [8], [9] if the operator keeps the button pressed, and we will see here that in the case of the Insulin Pump a continuous ring occurs rather than three warning beeps).

Previously, the efficient and practical model-checking techniques discussed here were applicable to only a single sequential FSM [10]. This paper discusses the extension to a collection of FSMs. The strong modelling capabilities of collections of FSMs (whose transitions are labelled with queries to an expert system with non-monotonic logic formulas) have been confirmed using case studies widely used in the literature of software engineering and model checking. The succinct model that can be naturally achieved with sequential FSMs for the Industrial Press fits as one figure [8, Fig. 6]. For the Mine Pump a similar situation occurs [8, Fig. 4]. We can also perform full virtual simulations of the model and execute the automatically derived code on actual robots (a video for the model of Industrial Press in two platforms appears in youtu.be/b10pMdh14PM while a video for the Mine Pump appears in youtu.be/y4mufP0jA8U). This has been achieved due to the clear semantics of our model and the implementation of these semantics in an interpreter.

This accomplishes a key promise of model-driven engineering. We extend the implementation that builds Kripke structures for a single sequential FSM to one that derives rules for a collection of sequential FSMs that operate following a deterministic sequence. The same interpreter that implements and runs these rules also constructs the corresponding Kripke structure. As a result, we demonstrate formal verification of correctness properties using standard tools such as NuSMV. Moreover, we contrast that the size of Kripke structures we obtain are smaller and more properties are verified than with previous descriptions of such case studies.

We now give an overview of the modelling tool and its implementation.

II. THE MODELLING TOOL

A. Sequential FSMs labeled by common sense logic

A FSM consists of a set S of states, and a transition function \( T : S \times E \rightarrow S \). There is a distinguished state \( s_0 \in S \), named the initial state. Typically, the set \( E \) is a set of events, or a set of input symbols, but in our case this is a set of Boolean expressions. Moreover, \( T \) is usually a partial function, that is, there are pairs \( (s_i, e) \) for which \( T \) is not defined; so, \( T \) is usually called the transition table.

The standard general description of the semantics for \( T(s_i, e) = s_j \) is that when the machine is in state \( s_i \in S \) and the expression \( e \) evaluates to true, the machine will move to state \( s_j \). However, this requires that \( e \) and \( e_s \) never be simultaneously true. That is, for any \( t \neq s \), if \( T(s_i, e_t) \) and \( T(s_i, e_s) \) are defined, then \( T(s_i, e_t) = T(s_i, e_s) \) every time \( e_t \land e_s \) is true.

We simplify the burden for the behaviour designer 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 \( (e_s, \forall s < t) \) in the sequence \( T(s_i, \cdot) \) evaluates to true. Note that while this is simply syntactic sugar, it does make the task of the behaviour designer markedly simpler.

States in a FSM model a space in time where an action takes place. However, there are three sections where actions are grouped. An OnEntry section is executed upon arrival to the state, while actions in the OnExit section are executed as the machine departs. These actions in these two sections are executed only once. The third section is a section for internal actions which are executed only if none of the transitions fire. When the internal actions are completed, the machine 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. Therefore, execution of a ringlet consists of one of the following possibilities.

1) Execution of the OnEntry section (because this is the first ringlet on arrival to this state), evaluation of expressions labelling transitions, and if one evaluates to true, execution of the OnExit section.

2) Execution of the OnEntry section (because this is the first ringlet on arrival to this state), evaluation of expressions labelling transitions, and if none evaluate to true, execution of the internal section.

3) On a later ringlet, evaluation of expressions that label transitions, and if one evaluates to true, execution of the OnExit section.

4) On a later ringlet, evaluation of expressions that label transitions, and if none evaluate to true, execution of the internal section.

An operational semantics of the execution of a single machine has been already provided [10].

B. Individual, sequential FSMs as components

FSMs lead to well-defined languages and run-time semantics [10] for control systems of autonomous robots [11], [12]. They are based on frameworks [12], [13] that support all stages of development and reconfiguration at run-time. These frameworks constitute two basic tools: (1) a development environment (where models can be drawn, syntactically verified, and communication patterns analysed) and (2) a run-
time environment (where models are simulated or executed, debugged and traced, and execution is logged). Moreover, the run-time scheduling of several models of behaviour (each represented by a FSM) can be made deterministic [12]. This leads to verification of many aspects of the communications or structure of the models before execution [12], [13]. The important aspect of deterministic scheduling is the possibility of performing model-checking [12]. It has been suggested that sequential pre-computed scheduling would cope with the combinatorial state explosion and complexity problems of mixed synchronous/asynchronous systems that include data flow and task-control derived from using several FSM models [12], but actual integration with model-checking technology was left for further work [12].

We now describe how a sequence of FSMs with transitions labeled by Boolean expressions are interpreted in a single thread. By providing this semantics, the construction of a Kripke structure effectively becomes the construction of a Kripke structure for a sequential program. The idea is simple and effective. A sequence of FSMs is executed in a circular round-robin fashion in the order of the sequence. Each machine runs a ringlet [10] of its current state. That is, if the current state is different from the previous state (typically, when a state is first entered), then the OnEntry part of the current state is run. Thereafter, a snapshot is taken of all the external and shared variables (see below), and the machine progresses with the evaluation of expressions in the sequence of transitions out of the current state. If one of these transitions fires\(^1\) the OnExit part is executed, followed by passing control to the next FSM in the sequence (with the first machine considered to follow the last one, completing the circle). If none of the transitions fire, then the internal actions of the current state are executed and control passes to the next machine in the sequence (again considered circular).

**Suspending and resuming state machines:** Each state machine in the collective has the ability to suspend execution and effectively become dormant. This is modelled through a separate state called the suspend state. The suspend state is a state that typically does not have any explicit transitions in and out, and therefore does not have to be modelled explicitly by the user (instead, the state machine will create such an empty suspend state when it first gets suspended).

Before the above sequence is run for a state machine, the machine checks whether a transition into the suspend state has been requested. In that case, a transition from the current state (for the sake of suspension called the resume state) to the suspend state will fire. Then the normal sequence of steps gets executed for the suspend state, including OnEntry, checking for transitions firing, and possibly running of internal actions. For a typical suspend state, these are all empty, effectively rendering the machine dormant.

If resumption of a state machine is requested (again, this is checked before any of the above steps is executed), a transition from the suspend state to the resume state will fire, moving the machine back into the state it was in before suspension. Alternatively, a restart can be requested, which will fire a transition from the suspend state to the initial state.

**Submachines:** The above mechanism of suspension and resumption can be triggered from within an activity from any state machine. This effectively allows for the introduction of a hierarchy of controlling and controlled state machines, where a controlling machine, a supermachine, suspends and resumes/restarts a controlled machine, a submachine.

A well-formed tree of such state machines is used to implement a subsumption architecture. This allows multiple components of a complex system to be modelled in a hierarchical fashion, where supermachines can operate at a higher level of abstraction while submachines implement the behavioural details of the system. Note that this differs from traditional component modelling that treats each component as an independent, concurrent entity (and consequently introduces significant complexity associated with nondeterministic temporal behaviour and task synchronisation). By contrast, in our system, the subsumption architecture is modelled (and implemented) through the same mechanism of sequential execution of states in a collective of machines, enforcing determinism throughout.

### C. Generation of a Kripke structure for concurrent machines

The Kripke structure that corresponds to a collective of concurrent state machines can conceptually be simple to describe. Any valuation of the variables of all the FSMs, including its program counters is a Kripke state. If the computation can possibly move from one Kripke state to another we have a link in the Kripke structure.

The generation of a Kripke structure for one sequential machine poses some issues of concurrency with the effect of the environment on the external variables. These are the variables the FSM reads, but the environment updates, such as, for example, the variable that stores whether a door is open or not in a microwave oven. External variables require some attention because we need to specify the level of temporal granularity at the border between the system and the environment. It should not be possible for the environment to set the variable and then re-set the same external variable so fast that the system does not perceive this change. By taking a snapshot of external variables at pre-specified points in time, as described below, we effectively create sampling points that specify a deterministic boundary of interaction with the environment. Once this boundary is specified, the Kripke structure for one machine can be completed [10].

With this, we now have a sequence of FSMs. Our modelling tool enables three types of variables. The first two types can only be read or updated by FSMs: variables that sit locally within one FSM, and variables that are shared by the FSMs in the sequence. The third type are external variables whose value can be updated or modified by the environment (anything outside the sequence of machines) but can be shared by the
FSMs. To handle the case of external variables as well as shared variables, the semantics of the execution of a FSM is that the machine reads the values of all the variables it uses only once per ringlet and just before the commencement of evaluating the first of the expressions of the transitions out of a state (if there is an OnEntry part, after that part has completed). That is, a machine takes a snapshot of externally visible (and, more importantly: modifiable) variables only once per execution of a ringlet. Thus, there is no Kripke transition out of a Kripke state that represents execution. In other words, evaluation of expressions never changes the value of any variable and the next expression will be evaluated in the same context (valuation).

However, when control leaves a FSM and passes to the next in sequence, the next machine may find external variables modified since the single thread of control read variables last. Some of these variables were modified by the previous execution of a ringlet, while others may have been modified by the environment. Therefore, we have several machines running concurrently, but they do so in a predefined, deterministic schedule that can be mapped to a sequential program. This sequential program is modelled by a Kripke structure, as described in the model-checking literature [14, Chapter 2]. We have implemented this translation in a highly efficient, cross-platform state machine interpreter. Furthermore, we have expanded \texttt{qfsm (qfsm.sf.net)} to enable the construction of the graphical models that systematically correspond to such a Kripke structure. We can now use model-checking tools to formally verify properties for the models. To this end, we have complemented the software that interprets the models (using the above semantics) to also produce NuSMV input files that represent the corresponding Kripke structure.

### III. Ambulatory Infusion Pump

In this paper we discuss a rather more sophisticated case study where software controls a pump to regulate the amount of volume injected to a patient. The controller (AIP), keeps the pump either running or halted. There are several sensors. The \texttt{airDetectorSensor} detects air in the line, the \texttt{VolDetector} measures volume, indicating less than or equal to 0 ml., between 1 ml. and 5 ml., and greater than 5 ml. Other inputs come from the start and stop button (the \texttt{ssButton}), and the \texttt{occlusionSensor} that detects a blockage in the line. Also, the \texttt{batteriesSensor} detects whether the batteries are low, in good order, or disconnected. Actuators of the system are the \texttt{messageScreen} that displays messages to the user, the \texttt{timerScreen} that display the time left, the \texttt{timerBeeper} that makes a sound when the timer goes off, and the \texttt{beeper} that makes a sound if batteries are low.

There are the three situations that are analogous and force that the pumping is stopped and the remaining volume release must be recalculated. (1) If the device has no volume (to inject). (2) If there is blockage on the line. (3) If there is air in the line.

If the batteries are low, the message \texttt{BATTERIES_LOW} must be displayed and the \texttt{beeper} should beep three times. The AIP starts only when the batteries have \texttt{normalCharge}. Removing the batteries turns the AIP off, and also the pump off. If the \texttt{ssButton} is held pressed when the pump is halted and there is no volume, the message \texttt{EMPTY} must be displayed on the screen. Analogously, if the \texttt{ssButton} is held pressed when the pump is halted and the line is blocked, then the message \texttt{LINE_BLOCKED} must be displayed on the screen. If the \texttt{ssButton} is held pressed when the pump is halted and there is air in the line, then the message \texttt{AIR_IN_LINE} must be displayed on the screen.

When commencing the injection of the corresponding dosage, if the batteries have normal charge, there is no air in the line, the flow is not blocked, there is normal volume and the \texttt{ssButton} is held pressed, then a timer sound is made, the display shows dots, the timer is set to 2 min. and the pump starts pumping. The display shows \texttt{PUMP_RUNNING}.

A volume is considered normal if it is greater than 5 ml. If the volume is between 1 ml. and 5 ml., then the display shows the message \texttt{VOLUME_LOW} even if the pump is running. If the dosage was interrupted because of an incident and then the situations is fixed, the recalculating will reset the timer once the \texttt{ssButton} is held press again. Every time the pump starts, the \texttt{timeBeeper} sounds for 3 seconds. If the \texttt{ssButton} is released during those 3 seconds, the pump does not actually start and the display shows the message \texttt{PUMPING_ABORTED}.

Only a few properties have previously been formally verified for this case study using model-checking techniques [5].

Property-1 “If there is blockage in the line, the pump must be stopped.”

Property-2 “If there is air in the line, the pump must be stopped.”

Property-3 “As soon as the pump operation is interrupted, the drug volume must be re-calculated.”

We can not only formally verify these properties, but many others that are part of the requirements. For space reasons, we have not included the coding of the earlier properties in NuSMV, but Fig. 1 shows the coding (using LTL) of the three properties above. The coding illustrates other aspects of our interpreter and Kripke structure generator. For example, \texttt{ESS} is a prefix for external variables that can change value at any time, from any Kripke state to the next. So, the first proposition says that if there is air in the line, it either drops in the next four Kripke states or the motor is shut-down (the \texttt{motor} variable is shared by the vector of FSMs). The last three NuSMV propositions in Fig. 1 all correspond to Property 3. In this case, we have statements about the program counter (\texttt{pc}) in the Kripke structure. The \texttt{pc} encodes with \texttt{M} the machine identifier, with \texttt{S} the state within that FSM, and with \texttt{R} the stage in the ringlet. Thus, \texttt{pc != MS0R0} means the \texttt{pc} in the Kripke structure is not in the OnEntry stage of the initial state of the first machine (in this case the Master Control). If we remove this, the proposition is false, because the software may start while the motor is running and there is air in the pump, and it would not get to the \texttt{RECALCULATING}
state, but it will shut down the motor (which is what is supposed to happen, since this is not an interruption).

On inspection of the Behavior Tree associated with such analysis [5], we discovered that the three beeps ring repeatedly while the battery is low, which in a sense, makes it not three beeps (as per the requirements), but a continuous stream of beeps. In our models, we place a 10-second silence between the next set of three beeps, if the battery charge remains low. Perhaps more serious is the emergent behaviour (in the Behavior Tree model) that results from keeping the button pressed. In this case the pump injects volumes continuously without ever stopping, dosage after dosage (as long as there is a snapshot of the environment is taken, we do not make these types of assumptions. We establish very clearly the point in the ringlet of a finite-state machine that a snapshot of the environment is taken, we do not make any assumptions about the speed with which any sensor is updated. In our models, sensors can also be updated much faster than the software may be able to execute (e.g. when delayed in a multi-tasking operating system).

IV. Conclusion

To our knowledge, we can perform model-checking for the four case studies discussed beyond anything reported in the literature. Our belief is based on the fact that earlier models have issues, inaccuracies, or emergent unusual behaviour. Moreover, we can simulate the models as well as other components of the system and perform Failure Modes and Effect Analysis [15]. See [url]youtube.be/FpVUSrvL10c[/url] for a video of the simulator operating concurrently on all the FSMs of the model of the Industrial Press for the corrected version over the Behavior Tree version. In the video sensor data is supplied through the monitoring tool. The host computer acts as a robot through our whiteboard architecture and produces speech also from the FSMs. This simulation capability again shows that our approach compares favourably with Behavior Trees.

There is another aspect where our approach compares favourably. With the Behavior Tree approach, several threads and channels of communication occur within the Behavior Tree, resulting in computations too complex to perform any sort of model checking without a large number of simplifying assumptions. None of our verifications required more than a few seconds, although in some cases, the generated Kripke structures result in files of several Mb.

Moreover, previous research [5] has assumed that updates of internal variables always takes precedence over external events (arguing that the software runs much faster than the possibility of a user pressing and releasing a button, but there is also an admission that this is risky [5] and their model checking is not sound). We do not make these types of assumptions. We establish very clearly the point in the ringlet of a finite-state machine that a snapshot of the environment is taken, we do not make any assumptions about the speed with which any sensor is updated. In our models, sensors can also be updated much faster than the software may be able to execute (e.g. when delayed in a multi-tasking operating system).

**References**


Fig. 2: Complete model of the ambulatory pump that formally satisfies the properties and includes corrections over the Design Behavior Tree alternative.