Modeling, Validation and Continuous Integration of Software Behaviours for Embedded Systems

Vladimir Estivill-Castro  
School of ICT  
Griffith University  
Nathan Campus, 4111, Australia  
Email: v.estivill-castro@griffith.edu.au

René Hexel  
School of ICT  
Griffith University  
Nathan Campus, 4111, Australia  
Email: r.hexel@griffith.edu.au

Josh Stover  
School of ICT  
Griffith University  
Nathan Campus, 4111, Australia  
Email: joshua.stover2@griffithuni.edu.au

Abstract—We propose to test software models with software models. Model-Driven Software Development proposes that software is to be constructed by developing high-level models that directly execute or generate most of the code. On the other hand, Test-Driven development proposes to produce tests that validate the functionality of the code. This paper brings both together by using Logic-Labeled Finite-State Machines to deploy executable models of embedded systems and also to configure the corresponding tests. The advantage is a much more efficient validation of the models, with more robust and durable representations, that ensure effective and efficient quality assurance throughout the development process, saving the costly exercise of formal model-checking until the system is complete enough to meet all requirements.

Keywords—Software Models, Real-Time Systems, Test-Driven Development, Model-Driven Development, Validation and Model-Checking, Finite-State Machines.

I. INTRODUCTION

Gartner estimates the value of the Internet of Things (IoT) to be $300 billion by 2020. IDC’s prediction is that the current worldwide market for the IoT will grow from $1.9 trillion in 2013 to $7.1 trillion in 2020. This is just one more illustration of the pervasive and ubiquitous presence of interconnected digital controllers and embedded systems surrounding all aspects of human life. The processors in these systems are becoming more powerful and the demand for behaviour sophistication is resulting in more complex software in all sorts of systems. Kopetz [1] has highlighted the impact on industry, life-style, energy production and consumption, and naturally the massive implications for safety and security that worldwide networking of smart systems and smart objects is bringing. Even medical devices and health systems are now included part of the smart planet [1] despite there have been recent articles highlighting potential risks [3], [4]. Sadly, more and more system failures and software faults are being reported, many of which affect safety-critical systems. Recent examples that have gained significant media attention include the security risks in some Range Rover models that enable unauthorised individuals to gain control of the vehicle remotely. Earlier, reported faults in the “ConnectedDrive” technology of over 2 million BMW, Mini and Rolls Royce cars made it possible for unauthorised persons to unlock their doors.

The difficulty to scale up model-checking technologies to the complex behaviours that are now expected in smart objects and devices makes it imperative to find mechanisms to validate and establish their safety. To quote Kopetz [1]:

“The pervasive deployment of smart objects that collect data and control the physical environment from distance poses a severe challenge to the security and safety of the world and the privacy of our lives.”

Despite their increasingly complex structure, the systems integrated into the IoT, by their very nature, are real-time systems. Several scholars have alerted to the serious challenges posed by the delicate timing issues [2].

“A new economy built on the massive growth of endpoints on the Internet will require precise and verifiable timing in ways that current systems do not support.”

This becomes even more critical due to the increasing use of such systems in a safety-critical context. On May 1st, 2015, for example, the FAA issued an airworthiness directive for Boeing’s 787 because a software bug shuts down the plane’s electricity generators every 248 days:

“all four GCUs will go into failsafe mode at the same time, resulting in a loss of all AC electrical power regardless of flight phase”.

Three key approaches by the software engineering community stand out when responding to the challenges of producing more versatile software that synthesises the behaviour of such embedded systems, in particular, Model-Driven Software Development (MDSD), Test-Driven Development TDD and Continuous Integration (CI). In this paper, we incorporate these three ideas by modelling and testing models with logic-labeled finite-state machines (LLFSM). Use of arrangements of LLFSMs is a MDSD approach, which has proven to be very effective [5], [6] as it scales from embedded systems to robotic systems [7]. Moreover, this not only produces compiled/executeable software directly from high-level models (as opposed to, e.g., Event-B [8]), but also facilitates formal verification with model checking [9]. However, model-checking is still a laborious exer-
cise, not only because of the complex strategies required to minimise combinatorial explosion [10], but also because formulating the statements to formally verify human-language requirements is non-trivial. For example, the NuSMV model-checker supports CTL and LTL as the mechanisms to express the properties for verification. It is complex to accurately express (often ambiguous) requirements written in natural language as these formalisms. Thus, the danger is that what is verified is an incorrectly formulated property that does not properly reflect the original requirement [12]. Although logic-labeled finite-state machines deliver deterministic sequential models that minimise the state-explosion of model-checking [10], to facilitate composability and scalability of complex systems underpinning the IoT, it is important that validation mechanisms can be performed early and incrementally on the executable system.

Thus, in addition to formal verification, it is natural to detect faults in software systems through simulation and validation, as part of CI, avoiding the costly exercise of formal model-checking upfront. Testing is a cost effective approach to early and continuous software quality assurance. However, we point out that successful testing alone is not definite verification as tests themselves may have faults and typically validate a smaller set of situations or traces. This is particularly important in the context of safety-critical real-time systems where code coverage of tests constitutes only one dimension. The other (often overlooked) dimensions are value coverage and timing coverage at the component and system level. Moreover, traditional unit testing frameworks do not integrate well with finite-state machines, necessitating different approaches for testing and validation [13].

We propose a method for the validation of models that are systematically constructed from high-level requirements. To this end, we utilise LLFSMs in our design. Perhaps more interesting is that the tests themselves are now constructed using logic-labeled finite-state machines. The environment, the sensors, and the actuators that surround the system being validated and defined by an arrangement of LLFSMs are also simulated by LLFSMs. Moreover tester-LLFSMs are used to reproducibly construct the scenarios under scrutiny in a test. This testing is also incorporated into a continuous integration server; thus, any updates or extensions to the behaviour are continuously validated automatically.

We demonstrate our approach with two case studies widely used within the software engineering and real-time systems literature. First, the microwave oven as software models of its behaviour not only appear in the literature of software modelling [14], [15], [16] but also appear in discussions on behaviour engineering [17]. This allows us to show validation and construction of tests for continuous integration in the value domain, similar to what is achievable using CTL and LTL formalisms. However, for systems that demand verification on statements regarding response time (commonly real-time systems), model-checking demands more complicated formalisms [18] that very quickly become too complex to model, modify, or maintain, in stark contrast to the promise of simplicity of high-level executable models.

To demonstrate testing and continuous integration within the time domain our second safety-critical case study is a pacemaker; also previously studied in the field of formal verification of real-time systems [19], [20].

II. FIRST CASE STUDY

A. Accelerating validation using GUIs

We first demonstrate the applicability of LLFSMs for TDD, MDSD and CI with the example of the One Minute Microwave. Because testing embedded devices demands deployment of the software onto the device in order to use the actual hardware, and a human operator to go through the test procedure (see Figure 1), the idea of MDSD with a graphical user interface (GUI) to emulate the device has been a subject of recent interest [21]; not only for automated testing and validation, but also user testing (Figure 2). The approach follows the model-view-controller (MVC) paradigm. The behaviour is described by LLFSMs, which constitutes the controller component of the MVC pattern. The status of the system is stored on a blackboard and constitutes the model in MVC. The view is the GUI simulating the hardware UI. The advantage of this scenario is that the design of the device itself can be evaluated in software, and the software (the arrangement of LLFSMs) does not change when it is ported to the actual embedded system (again, the software defining the behaviour and developed using MDSD is the same in Figure 1 and in Figure 2). Nevertheless, in
these figures, the box labeled MODEL is a software model; and in particular, an arrangement of finite-state machines that describes and defines the behaviour of the system. In the case study here, the GUI has been built with Qt Creator and communication uses control/status messages. We forward the status of buttons, or any widget in the GUI that produces input, to the arrangements of LLFSM and post control messages to modify the status of any widget in the GUI we require to update.

Creating GUIs for validating the enables performing usability testing and users to participate in refinement of the requirements. Moreover, the users can visualise the behaviour because of the diagrammatic display of states and transitions provided by the model. A video\(^1\) demonstrates this with a very simple GUI whose purpose is just to provide the motors and the effectors for the microwave. In fact, engaging user groups with the GUI results in requirements that deviate from the original ones as follows:

“When cooking has been paused by opening the door, most microwaves do not restart immediately when the door is closed, but further confirmation is required by pressing a start button.”

However, the purpose of this paper is to take this much further to automatic testing and CI (see Figure 3). The usability testing approach still demands the involvement of a human operator, and thus cannot support TDD. Moreover, any testing by humans has non-positivist elements, making them a lot less suitable for system testing, particularly when it comes to verifiable predicates and functionality. Such systematic testing is best performed by automatic means that re-execute all tests when updates are incorporated into the software code base. This is the main contribution of CI.

B. The applicability of LLFSMs for TDD, MDSD and CI

In the case of TDD and CI, as with the GUI approach there is no hardware. The sensors are activated by the tests depositing status messages in the middleware. The actuators are observed by the control messages that the executable software model produces. However, here the tests themselves are LLFSMs, executed as part of the project-building infrastructure. The executable software model of the microwave consists of an arrangement of four LLFSMs that execute in a single thread, as described earlier [10].

We now illustrate how models can test models. In this initial example, let’s examine an executable software model (Figure 4) that constitutes a test of the requirement that when

“the microwave is not cooking, opening the door turns the light on while closing the door closes the light of.” [10]

We emphasise that the software being tested does not change, neither the test but as explained before, we can use the middleware to translate I/O pin signals for GUI-testing or for a simulated microwave with an NXT, as per the video mentioned before. Figure 4 shows the LLFSM that manifests this requirement as a test. This test consists of a SET UP phase (indicated by the Initial state and SET_DOOR_CLOSED state). In the Initial state the TIME_LEFT pin is set low (the oven is not cooking). The SET_DOOR_CLOSED state is part of the SET UP so the pin indicates the door is closed. Recall that there is no actual hardware, the LLFSM in Figure 4 is acting as an environment for the tested software and has set a context where the door is closed and no time has been accumulated. This should turn the light off, and thus, if the light is still on after 50ms, the tester LLFSM has detected a fault in the control software of the microwave (transitioning to FAIL_LIGHT, which exits with a failure code). If the light is not on, then !digitalRead(PIN_LIGHT) is true and the tester LLFSM moves to the state TEST_LIGHT_ON_DOOR_OPEN. In this state a signal is placed on the pin that the door is open. As before, if the light does not come on after 50ms, the tester arrives at the terminating FAIL_LIGHT state. If the light does come on, the tester transitions to the state TEST_LIGHT_OFF_DOOR_CLOSED. In this state, the pin is set to indicate a closed door. For the third time, the light is tested, and if the light does not go off in 50ms, the state FAIL_LIGHT is reached. Only if all required behaviours happen, this tester LLFSM arrives at the state

---

\(^1\)www.youtube.com/watch?v=vtjevr4ZXms
The exit values of executing this tester LLFSM are used by the Jenkins infrastructure that evaluates the software compiling the system for several operating systems and cross compilers (cross-compiling for the Arduino as well as clang under MacOS and Linux). Makefiles and the Jenkins environment ensure that build and test targets in the project get re-run on any update to a LLFSM, guaranteeing that required-of behaviour is automatically re-tested against all the tests (see Jenkins documentation for this). Our contribution here is that executable models are testing executable models.

In the literature of model-checking [22, Page 39] this case study is discussed because the safety feature of disabling radiation when the door is open is an analogous to the requirement on the Therac-25 radiation machine, where a failure to properly enforce it caused harm to patients. Thus, to continue the illustration, Figure 5 shows an executable model as a LLFSM that tests this requirement. Note that transitions now have strict Boolean expressions regarding the status of the motor. For example, the transition for the state TEST_MOTOR_ON to the state TEST_MOTOR_OFF is labeled with the expression !digitalRead(PIN_MOTOR), which evaluates to true if the I/O pin for the motor is off and is false otherwise. The CI infrastructure performs these two tests systematically and repeatedly every time there is an update to any component.

C. Contrast with formal model-checking

We do not advocate the exclusion of model-checking, but we take this opportunity here to contrast already this with the generation of tests proposed here. First, the construction of tester LLFSMs is rather intuitive and corresponds very closely with the use cases and the requirements. The essentially linear story of the use case appears reflected in the tester LLFSM for the corresponding use case. We emphasise that requirements and use cases are usually presented as a linear time-trace; so much so that behaviour engineering with behaviour trees [17] builds the specification one trace at a time. In this example the story line of for the cooking being halted when the door opens is as follows.

1) First we need to have the door closed and put time on the microwave with the button.
2) Second, we open the door and the motor stops.
3) Third, closing the door resumes the cooking.

Such a story line produces a pattern in the tester LLFSM. For a story line where the environment produces a sequence of inputs \( \langle I_1, I_2, \ldots, I_n \rangle \) and that results in state changes in the system observable by responses \( \langle R_1, R_2, \ldots, R_n \rangle \), we have a simple structure for the tester LLFSM. For each input \( I_t \), we have a state in the tester that posts the input for \( I_t \) to the middleware. If the observable response equals \( R_t \), the system progresses to the next input, otherwise it reports the test failed. If all the responses match the expected input, the test succeeds. This pattern in shown in Figure 6. We emphasise

---

Figure 5. Tester LLFSM to verify the software model complies with the requirement that opening the door pauses the cooking.

Figure 6. Pattern of a LLFSMs tester.
that unless the requirements change, the tester LLFSMs do not change. And if a requirement changes, changing the corresponding tester is straightforward. In contrast, model-checking with tools like NuSMV demand a particular syntax for the LTL or CTL formulas that is very fragile. Such formulas are complicated, usually demanding deep nesting and their semantics is complicated to follow. It is actually quite challenging to be convinced that these formulas actually represent the requirements. It is quite common to succeed with formulas that do not actually represent a desired system property. Any change in the model being tested requires all LTL and CTL formulas to be adjusted.

Thus, producing the corresponding verification is significantly more time-consuming. Our point here is that they should only be performed once all the tests have passed. Obviously the tests do not establish that in all executions that follow from a certain valid configuration, regardless of time, the configuration is reached, the execution would be correct, but early detection by the models testing models proposed here prevent costly model-checking that potentially only informs about the same fault.

For example, in this seemingly simple example of the microwave oven, we have mentioned that the software consists of 4 LLFSMs [10] (one controls the motor, a second one the bell, a third one drives the light, and a fourth one listens to the button to set the time and count down the seconds). While these authors have conducted full model-checking of the properties defined in the requirements and a check regarding overflow of the timer, this was performed for one arrangement of the 4 LLFSMs. If all arrangements were to be verified, the 4!=24 possible integrations of these LLFSMs into the sequential execution would need to be verified, implying the model-checker NuSMV would need to be run against 24 Kripke structures. This is a laborious and not an autonomous exercise. It is important to emphasise that the 4 LLFSMs are almost independent modules, and part of the quality assurance is that in fact all 4!=24 configurations of the arrangement are correct. That is, the integration cannot accidentally produce a fault just because of the ordering of the modules. Thus, we have produced an LLFSM that generates all permutations of an arrangement of LLFSMs and tests them with the CI server. In this scenario, no tester LLFSMs (Figs. 4 and 5) needs any adaption or changes. By contrast, the formulas for model-checking under NuSMV do require adaption for each of the possible 24 orders (as they are specific to the operators for the next Kripke state).

Thus, this case study demonstrates the benefits of models testing models under continuous integration for quality assurance of embedded systems.

III. SECOND CASE STUDY

We will now illustrate our approach with the MDSD and TDD construction and testing of software for a

<table>
<thead>
<tr>
<th>Req.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R 1</td>
<td>Monitor the input for a sense signal</td>
</tr>
<tr>
<td>R 2</td>
<td>If no input has been sensed for R1, then output a pace</td>
</tr>
<tr>
<td>R 3</td>
<td>If an input is sensed, inhibit pacing for VRP</td>
</tr>
<tr>
<td>R 4</td>
<td>If a pace is sent, do not monitor or send another pace for VRP</td>
</tr>
<tr>
<td>R 5</td>
<td>If the last event was natural, then turn hysteresis mode on</td>
</tr>
<tr>
<td>R 6</td>
<td>If the last event was paced, then turn hysteresis mode on</td>
</tr>
<tr>
<td>R 7</td>
<td>If hysteresis mode is on, R1 ← R11</td>
</tr>
<tr>
<td>R 8</td>
<td>If hysteresis mode is off, R1 ← LRI</td>
</tr>
<tr>
<td>R 9</td>
<td>No pacing must occur during URI after the last event</td>
</tr>
<tr>
<td>R 10</td>
<td>Pacing must occur if LRI has passed since last event</td>
</tr>
</tbody>
</table>

TABLE I

Pacemaker REQUIREMENTS.
Figure 10. The third LLFSM tester validates that the Pacemaker in Figure 7 does not interfere with a natural beat of two different frequencies.

Figure 7. The LLFSM for the Pacemaker in VVI mode.

Figure 8. The first LLFSM tester validating the Pacemaker in Figure 7 responds before the deadline if the beats stop.

Figure 9. The second LLFSM tester validating the Pacemaker in Figure 7 does not anticipate a beat.

heart-beat of the patient. The tester machine emulates the situation where beats are produced at two different rates. If the tester detects a pulse from the Pacemaker when, in fact, all that happened was a change of frequency in the beat generated by the tester, then we have a fault. The LLFSM that implements the test appears in Figure 10. It performs a more elaborate beat generation pattern than the loop in Figure 9. For the first 5 beats it loops fast, but the next 5 beats are slower. Generating a logic formula to formally verify this type of property is extremely complicated and challenging because of the numerical values of the time variables play a role in the property.

IV. DISCUSSION AND CONCLUSIONS

Perhaps the most relevant work to the approach presented here is Model-Based Testing (MBT) [24], a term much more relevant to automated testing in the sense of automatic generation of tests, influenced by the model. That is, the
model is the source of generated code and generated testing-code (the latter is referred as the test). This is illustrated by many techniques and tools in embedded systems software, in particular for medical devices [25] and automotive systems [26]. In those cases, tools and artefacts such as SysLog and SysML are used to construct the models and to generate typically C code for the embedded device, while frameworks for hardware-In-the-Loop (HIL) are used to run the testing code. Our approach here is radically different in that the actual models are executable, guaranteeing that it is the actual models that are tested.

We have shown that tester models can test software models of behaviour because both are represented in executable arrangements of logic-labeled finite-state machines. This combines the benefits of Model-Driven Software Development (MDSD) with those of Test-Driven Development (TDD) and the productive context of Continuous Integration.

The two case studies here illustrate the efficiency of the method as it avoids the complexities and delicacies of formal verification via model-checking. In the first case study we have shown that integration of 4 LLFSMs results in a large number of combinations that makes the application of model-checking tools complex, tightly-coupled and fragile. A different integration pattern results in a different Kripke structure and thus new formulas must be generated (a complex, non-automatable exercise).

The second case study shows we can use the arithmetic and system calls in the tester machines to validate deadlines and properties in the time domain. Again, a large test coverage can be achieved here under controlled conditions, even where these conditions are too complex for formal verification in the time domain.

Our tester models are generic and robust. They can be derived from the linear traces of use-cases, mirroring them closely. This traceability from requirements facilitates validation completeness in an efficient and effective manner.

REFERENCES