MODELS TESTING MODELS IN CONTINUOUS INTEGRATION OF MODEL-DRIVEN DEVELOPMENT

Vladimir Estivill-Castro, René Hexel and Josh Stover
Griffith University
Nathan Campus, 4111, Australia
email: {v.estivill-castro,r.hexel,joshua.stover2}@griffith.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.

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

1 Introduction

The complete spectrum of modern human activity is now controlled and shaped by underlying software. Hence, software testing plays a fundamental and critical role ensuring sufficient vetting and quality assurance to guarantee that the final product will not cause harm to humans (or reduce quality of life), nor will it cause economic losses [4]. Testing is crucial to ensure software reliability but typically causes 50% or more of total software cost [4]. Improving techniques for system testing is vital not only because it directly flows on to software cost, but also prevents the potential negative socio-economic and human impact that software failures may cause.

As the world becomes more interconnected and the promise of the Internet-of-Things (IoT) to revolutionise all aspects of human productivity and social life [20], it also becomes essential to ensure the reliability of software systems. In 2015, 4.9 billion things are already connected in the current IoT, up 30% from 2014. Gartner estimates 736 million smart devices in the top three industry vertical sectors using IoT now: manufacturing, utilities, and transportation. Their forecast for 2020 predicts that the ranking will change, with utilities being first, manufacturing second, and the government sector placed third, totalling 1.7 billion IoT units installed. Gartner’s vice-president Jim Tully justifies the increase in the utilities sector as the result of the investment in smart meters and in smart street and area lighting for energy saving reasons. Such a vast amount of connected things brings with it radical changes in society, creating new services and usage scenarios and driving new business models. However, many are alert to the dangers of malfunction in smart embedded systems as a result of insufficient software quality testing [29, 27, 28].

The opportunity for improvements in software testing are enormous; however, experts suggest that the software models for IoT protocols as well as for the behaviour of the smart things are likely to be based on some sort of state machine [4]. Logic-labelled finite-state machines (LLFSMs) have been shown to be very effective in carrying out the promise of Model-Driven Software Development (MDSD). Software development is faster as the behaviour of the embedded device is specified on a higher level of abstraction than traditional programming languages. In particular, use-case traces can be naturally mapped to paths through states and transitions. In fact Behaviour Engineering [24], a form of requirements engineering, creates these traces and then integrates them into Behavior Trees, from which finite-state machines describing the behaviour of components can be readily synthesised. The precise semantics of LLFSMs makes them overcome some of the criticisms that MDSD has received [26] while enhancing the advantages. Arrangements of logic-labeled finite-state machines are not only automatically transformed into a working software, they are compiled directly [12]; thus they themselves constitute an executable model [10]. Furthermore, they also enable formal verification [14]. As a consequence changing, improving and maintaining behaviours of embedded systems and robots using LLFSMs is also more cost-effective. The high-level modelling means that the behaviour is easier to understand. In the systems engineering and robotics community, state-machines are ubiquitous. MDSD leads to more uniform quality, the LLFSM compiler executes a general uniform code that has been structured to produce minimum overhead. But most importantly, modelling with MDSD reduces the already mentioned heavy cost of testing software, as the high-level design is usually verifiably correct and comprehensive; thus
the effort of testing focuses more on acceptance testing (that is validating the functionality). Because of the use of visual models of LLFSMs, the resulting behaviours are more transparent and the gap between business analysts, requirement engineers, and developers is reduced.

Other advantages of MDSD with LLFSMs include software being less sensitive to technology changes (from the operating system, and compiler versions and platforms) to the actual hardware (which, for example, can be quite different from one robot to another). However, as the models captured by LLFSMs implement the actual behaviour, testing them is even more crucial. Recently, we proposed that the testing of models could be performed by tester-models themselves [15]; however, one rapidly discovers that the tester models themselves must be configured in a multitude of ways to increase the coverage of use-cases evaluated. In this paper we show how to incorporate Combinatorial interaction testing (CIT) [30] with Test-Driven Development [23] and Continuous Integration (CI) [8] to produce a virtuous cycle of productivity in embedded and robotic systems software.

Our first key contribution is the integration of MDSD with TDD. The second addition is the incorporation of CI into such a development process. Finally, we top this up by introducing CIT into this process. We illustrate this with one case study from the automotive industry. There is significant interest for time-domain model-checking of a steering box control [19, 25] and a gear box control [22]. In achieving this case study, we show how LLFSMs are capable of emulating timed-automata [3], which in itself is another major contribution, since this enables the tools utilising LLFSMs to also perform formal verification in the time domain (and not only in the value domain).

2 Preliminaries

The three main conceptual software engineering practices that have led to a leap in software productivity and quality are Model-Driven Software Development (MDSD), Test-Driven Development and Continuous Integration. Also, Logic-Labeled Finite-State Machines as the alternative modelling approach to event-driven finite-state machines (ala UML), that jointly with PULL technology and Control/Status messages provide significant advantages in software construction for embedded and robotic systems [11].

Model-Driven Software Development (MDSD) is a variant of round-trip engineering where changes to a model result in an immediate update of the code generated from that model. Here, consistency requires bidirectionality. However, the models are usually at a higher level of abstraction and thus enable human comprehension and validation against requirements. MDSD offers to overcome the inability of third-generation languages to express concepts effectively for complex/multiple platforms. It has been shown that MDSD leads to faster results due to its higher level of abstraction as models are automatically translated into executable code. Models enable changes in direct proximity to requirements as well as traceability of functionality, which leads to increased quality (fewer faults). We show here how MDSD is less error-prone when supported with testing, validation, or simulation tools.

The software development process now known as Test-driven development (TDD) [23] relies on the repetition of a very short development cycle: first the developer writes an automated test case that defines a desired behaviour or new functionality (and in this sense the interface to the new module). Since initially there is no implementation, typically all tests fail. Progressively, developers produce the minimum amount of code required to pass a test, and finally re-factor the new code to acceptable standards. The automated execution of all earlier tests ensures the code still works for all of those. New tests are produced when more sophisticated behavioural aspects require validation. TDD produces simpler designs and builds reliability.

The practice of Continuous Integration (CI) [8] involves the use of a shared repository for the source code and very intensive (usually several times a day) automated integration and validation of code from different developers. Every significant change triggers re-compilation of all modules, and more importantly, all tests (derived from the TDD approach) are executed. The practice includes the running of each developer’s test locally before actually committing/integrating them to the main line. The contribution of this paper is to incorporate MDSD into a CI framework. We will demonstrate MDSD using Jenkins (jenkins-ci.org) as part of software configuration management.

3 Emulating timed-automata

Timed-automata (and its variants) are today the most used formalisms for formal verification of real-time systems [3]. They are an extension of classical finite-state automata with real-valued variables called clocks. Many tools implement timed-automata and associated model-checkers, but perhaps the most prominent is UPAAAL [21]; which includes a simulator as a more inexpensive generator of one trace or visualiser of the trace highlighted by the exhaustive search of the checker as an exemplar of a fault. The problem is that model-checking is still a laborious exercise, not only
because of the complex strategies required to minimise combinatorial explosion [9], but also because formulating the statements to formally verify human-language requirements is non-trivial. For example, the NuSMV [5] 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 in these formalisms. Thus, the danger is that what is verified is an incorrectly formulated property that does not properly reflect the original requirement [13]. In the case of real-time systems, timed-variants of LTL or CTL [22] or more sophisticated formalisms [7] are needed. Moreover, a collection of timed-automata allows complete non-deterministic concurrency of the collection; while in an arrangement of LLFSMs the schedule is deterministic, reducing the combinatorial explosion.

We now show how time-automata are converted to arrangements of logic-labeled finite-states machines, and thus become an executable model, amenable to testing. We start by illustrating how the fundamental syntactic time-automata constructs in UPPAAL [21] are converted into an arrangement of LLFSMs. First, a model in UPPAAL is a collection or network of components, that corresponds in LLFSMs to an arrangement of finite-state machines. Each component is a timed-automata that maps to one LLFSM in the arrangement. The components supported by UPPAAL [21] are timed-automata extended with integer variables in addition to clock variables. In UPPAAL, the integer variables and the clocks are global. The equivalent in LLFSMs is that clocks and integer variables will be messages in the PULL-based shared-memory middleware named gusimplewhiteboard [12].

Second, the nodes in UPPAAL correspond to states in LLFSMs. Edges in the automata correspond to transitions in LLFSMs. The delicate aspect is the labels that decorate the edges. In UPPAAL, the labels consist of 3 optional parts: (guard, synchronisation action, and sequence of clock resets and assignments). This aspect is not translated directly. Also, UPPAAL’s timed-automata have optional invariants on nodes (which are a restricted family of Boolean expressions only involving the clocks); empty invariants are equivalent to true.

Conceptually, each timed-automaton in the network progresses at its own pace, bounded by guards or invariants involving clocks or synchronisation mechanisms. The synchronisation action of an edge can be one of four types:

1. empty: it does not exist,
2. sending: a message is sent through a channel $a$ as denoted by $a!$.
3. receiving: a call that blocks until a message is received from channel $a$. This is denoted by $a?$.
4. urgent: this transition must have an empty guard.

Channels are Boolean flags. Figure 1 shows how a sending-edge in a timed automata translates to a LLFSM structure. In the semantics of LLFSMs, the OnExit section will not be executed unless a transition fires. If $!\text{invariant}_1$, then this terminates with EXIT_FAIL. The only transition that leads away from EXIT_FAIL is $\text{invariant}_1 \&\& \text{guard}$. Thus, the signaling on the channel happens immediately after the Boolean guard evaluates to true. If $\text{invariant}_2$ holds, this leads to the TARGET_STATE. If the guard and $\text{invariant}_1$ are true, but $\text{invariant}_2$ is false, then we terminate in
Figure 5. The conversion of several out edges in a timed-automaton to LLFSMs.

EXIT_FAIL. The DUMMY ensures the semantics of timed-automata: The assignments may affect invariant_2, thus the Boolean expression invariant_2 must be evaluated after the assignments from the OnExit section. Figure 2 presents the corresponding translation for an edge that has a receiving synchronisation channel. Again, the LLFSM terminates if !invariant_1 in EXIT_FAIL. If invariant_1 holds, when guard becomes true, the LLFSMs starts looping around and inspects the channel until it becomes true. Once the Boolean flag in the channel is true, it transitions to the TARGET_STATE if the invariant_2 holds, otherwise it terminates in EXIT_FAIL.

If there is no synchronisation, that is the entry for a channel is empty, then the translation of such an edge is even simpler; see Figure 3. Designating a channel as urgent makes operations atomic (recall that in the network of timed-automata each component could progress independently and concurrently). By designating a channel as urgent, there cannot be a guard, and the automaton must transition from the SOURCE_STATE to the next state with the channel activated, and the assignments performed without any other timed-automata carrying out interleaved operations. This is precisely the LLFSM semantics illustrated in Figure 4. At least one transition will evaluate to true. The SOURCE_STATE cannot be executed for more than one ringlet (which is the atomic unit), and the operations regarding the channel and the assignments will be executed immediately after evaluating the transition. This cannot be preempted by any other LLFSM in the arrangement due to its sequential schedule (semantically equivalent to execution in a single thread). We note that in all these transformations, the TARGET_STATE is unreachable if the invariant_2 is false. That is, these transformations ensure that, everywhere in the resulting arrangement of LLFSMs, not even the OnEntry section of a state is executed unless its invariant is true.

There is another construct for designating some operations as atomic within UPPAAL’s timed-automata: committed locations. As with urgent channels, committed locations are states that do not preempt the current thread of execution; that is no other component progresses and the current component continues execution. The current scheduler for LLFSMs does not explicitly support this. However, it is very simple to suspend all machines in the arrangement (other than the current one) from the target state of a transition that is a committed location, rather than passing the token of execution to the next LLFSM in the arrangement. This way, the token remains with the current machine. Also, if several edges have the same source but different targets (which in timed-automata usually means that one guard is the logical negation of the alternative), in LLFSMs an intermediate state is required to carry out the potentially different synchronisation and clock assignment. Figure 5 shows the transformation.

4 Combinatorial interaction testing

One of the challenges of creating executable models of LLFSM arrangements (or networks of timed-automata) is that these are eventually materialised in a sequence. While in some cases a particular order is absolutely essential for correctness, it many situations it would be preferable to design the components in such a way that the particular ordering is immaterial for correctness. In that way, the system is significantly more robust to the accidental invocation of the behaviour with some order reversed. An example of such a scenario is the four LLFSMs that constitute the control of the microwave [9]. The microwave is a widely studied machine. Also, if several edges have the same source but different targets (which in timed-automata usually means that one guard is the logical negation of the alternative), in LLFSMs an intermediate state is required to carry out the potentially different synchronisation and clock assignment. Figure 5 shows the transformation.

The challenge here for formal verification and model-checking is that each new order of the n components of the arrangement requires a new formulas in LTL or CTL even in just the value domain. This is again an example where creating tester LLFSMs becomes much more useful. It is more productive to have an LLFSM that generates all the n! evaluation orders. There is no point in pursuing the delicate and costly formal verification exercise if the system already fails tests from the TDD framework. Thus, in the past, we have produced a tester LLFSM that generates all orders and runs all tests for all configurations of the system under test (SUT) [15]. Despite this automation, validating the SUT repeatedly n! times represents exponential growth. For the case of the microwave, there will be 4!=24 iterations of the testing of the corresponding properties. Here, we incorporate Combinatorial Interaction Testing (CIT) in a clever way. CIT [30] acknowledges that fully testing all configurations of a large system cannot be done.
Figure 6. Implementation of the Lucas-Walecki Hamiltonian decomposition algorithm as a tester LLFSMs.
Thus, this approach samples the huge configuration space, testing representative instances of a system’s behaviour. The usual challenge is to produce what are called covering arrays [16]. We acknowledge that the optimality of many t-way covering arrays actually remains unknown (in the homogeneous case where one aims at minimising the size of the array that enumerates all configuration of t valuation out of k variables each with potentially v different values).

However, our proposal here is that an alternative type of Combinatorial Interaction Testing is required, and that we can find a suitable scalable solution. Rather that aiming at full coverage of the n! orderings of the components of a network of timed-automata (or arrangement of LLFSMs), we propose to find a linear number of orders such that any pairs of time-automata appear together (one listed besides the other) in at least one of the proposed orders. Moreover, by running this coverage in the opposite direction we ensure that any pair of LLFSMs is tested adjacently and each takes a turn at executing in sequence before the other.

The combinatorial form of these objects originally received the name of Round-dance Neighbour Designs [2] and it seems that these have been researched by mathematicians since 1897. In the language of combinatorics, this corresponds to the decomposition of the complete graph $K_n$ into the smallest number of Hamiltonian paths [17]. In the language of combinatorial testing, this decomposition covers all edges of $K_n$. The generation of the smallest set of permutations so all $n$ pairs has important connections to visualisation of parallel coordinates [17, 18]; however, despite the strong connections between combinatorics and the discipline of CI, to the best of our knowledge this seems to be the first application of this property in combinatorial software testing. Moreover, the tester algorithm itself (generating the Hamiltonian decomposition of the complete graph) can be presented as a LLFSM. Figure 6 presents the implementation of what is known as the Lucas-Walecki Hamiltonian decomposition algorithm [17] as part of our Combinatorial Interaction Testing of arrangements of LLFSMs. The algorithm requires $O(n)$ time to generate the n orderings used for testing in a TDD setting under the jenkins environment.

5 Illustrative case study

To demonstrate TDD with LLFSMs we use the same case study as previously presented with UPPAAL [22, Section 4]. The main component is the Gear Box Controller which receives commands from a second component: the Interface. The Interface can request a new gear trough the channel ReqNewGear after it has set the global variables FromGear and ToGear. The Gear Box Controller issues commands to other environment components: Gear Box, Clutch and Engine. All of these components are presented as timed automata [22, Figure 4 to 8]. In Figure 7 we represent the LLFSM for the Interface [22, Figure 7] and in Figure 8 the LLFSM corresponding to the Clutch [22, Figure 5]. These figures demonstrate the translation defined in Section 3.

Not only can we test the models for each components with tester LLFSMs but we can also use tester LLFSMs to test the interactions between these models. Moreover, the models for Gear Box, Clutch and Engine are idealized models of the environment for the software represented by the Gear Box Controller. We can evaluate rapidly the impact if the environment does not play by the rules. That is, we can perform fault injection analysis by introducing faults on Gear Box, Clutch or Engine and observing which tests fail. For example, Figure 9 shows an LLFSM to test the model for the Clutch. This tester presents 5 alternating requests for the Clutch be open and then to be closed, checking that after issuing a request using the channel OpenClutch, the effect is that the channel ClutchIsOpen is activated. Similarly for the channel CloseClutch. However, the test also checks that the channel ClutchIsOpen is not activated with the channel ClutchIsClosed. The tester may seem laborious, but it is a result of the simple model for the Clutch: the Clutch has no protection for situations where the commands OpenClutch and CloseClutch are both asserted simultaneously.

The advantages of the LLFSM modeling include the immediate translation form the nondeterministic timed-automata system modeled by UPPAAL to the corresponding deterministic system which facilitates tracking the faults or the causes of any failures in the execution go test from TDD. Using LLFSMs ensures that the error-prone process of coding timed-automata into implementations is eliminated. We insists that LLFSMs are executable models.

The advantage using a LLFSM to test a Use-Case is that the Use-Case itself is a linear story of an actor that provides inputs to the system and receives expected outputs. The role of the actor in TDD here is taken by a tester LLFSM that supplies the inputs and reviews the outputs against the corresponding expected results. If the results are different from those expected, the test fails; otherwise the test is a success. In fact, the current Interface can be regarded as a Use-Case where the operator request to shift gears from gear 1 to gear 2.

More importantly, in this case study, all components of the software are meant to be independent, and the Gear Box Control should both fail just because of differences in timing for the initial states of hose element sin the environment. Thus, in this case study, using the Combinatorial
Interaction Testing (CIT) also pays off. The sequential execution of the arrangement of the resulting LLFSMs should not fail because of the ordering of the components in the arrangement. In particular, all components have default initial states and they shall not move forward unless they receive the corresponding signals by others in their corresponding input channels; but this is best verified by running all components under the orders that guarantee all components will be either before and after but adjacent to every other component. That is, we used the Hamiltonian path decomposition testing proposed in Section 4.

6 Conclusions

In this paper we have used the powerful LLFSMs software modeling and construction tool as part of a discipline of software validation via testing. LLFSMs currently compile standard C++-11 code in the OnEntry, OnExit and Internal sections as well as any Boolean C++ expression in their transitions. They are not event driven, but they are Turing complete; thus, they can be an implementation to any other program. Modeling tools for timed-automata (for example UPPAAL) restrict the expressions and assignments that can be used to construct invariants or to handle signals. It would be impossible to construct tester timed-automata that validate other timed-automata.

Because the costly exercise of formal verification, even tools like UPPAAL recommend some simulation or testing of models before model-checking is attempted. We have shown here that the MDSD advantages of LLFSMs for robotic and embedded systems can be combined with TDD with CI, and moreover we have proposed an alternative bringing the graph theory decomposition to Eulerian tours into Hamiltonian paths to construct combinatorial testing for the arrangements of LLFSM. The systematic translation presented here from timed-automata to LLFSMs enables the simulation and testing under TDD and CI of models developed as timed-automata. This expands the running and validation of these models before the costly exercise of performing model-checking on them.

References