An OO and Functional Framework for Versatile Semantics of Logic-Labelled Finite State Machines

Callum M^{c}Coll Vladimir Estivill-Castro René Hexel

School of Information and Communication Technology
Griffith University, Nathan QLD 4111, Australia
callum.mccoll@griffithuni.edu.au
v.estivill-castro@griffith.edu.au
r.hexel@griffith.edu.au

***Abstract*—Logic-Labeled Finite State Machines (`LLFSMs`) offer model-driven software development (MDSD) while enabling correctness at a high level due to their transparent semantics that enables testing as well as formal verification. This combination of the three elements (MDSD, validation, and verification) results in more reliable behaviour of software components, but semantics is constrained to specific scheduling. We offer a framework that allows to obtain significant variations that suit specific domains while maintaining the capability to generate Kripke structures for formal verification or to execute corresponding monitor or testing `LLFSMs` for validation in a test-driven development framework. The framework is Object-Oriented so new software patterns for scheduling can be derived to suit a particular embedded, robotic, or cyber-physical system, while at the same time enabling functional programming constructs.**

Keywords–Logic-labelled finite-state machines; Model-Driven Engineering; Real-Time Systems; Verification; Validation.

I. INTRODUCTION

By following a transparent semantics that includes a synchronous model, Logic-Labelled Finite State Machines (`LLFSMs`) enable the design of software that can achieve high levels of complexity and sophistication while guaranteeing deterministic execution and facilitating formal verification [1].

The semantics specify precisely when variables affected by sensors outside the system are inspected as well as the particular points in the execution of the software where snapshots of the environment variables are taken [2]. However, this constrains the execution to just one specific semantics, and in particular, to one specific frequency and pace, which may not be suitable in another robotic or embedded system. It should be possible to configure rapidly and efficiently the semantics and constructs of `LLFSMs` providing developers the freedom to adapt or tailor the system semantics to particular cases. This paper enables such versatility. We provide the capacity to instantiate new scheduling semantics with incarnations of template methods and classes while still providing the capacity to generate the corresponding Kripke structure for formal verification with standard tools, such as NuSMV.

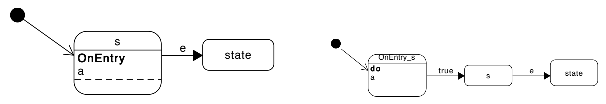
Therefore, this new framework removes the need to adhere to the strict semantics currently implemented in tools such as `clfsm`. Importantly, we maintain the ability to perform formal verification. We illustrate two areas where we create abstractions to the semantics of `LLFSMs` and show how instantiation of these abstractions into concrete derivations maintain the ability to perform formal verification. We introduce `swiftfsm` [3], a framework for `LLFSMs` written in `Swift`, which enables formal verification, but allows developers more freedom to design, adapt and create new `LLFSM` models that are particular to application-specific use cases.

II. LOGIC-LABELLED FINITE STATE MACHINES

Finite state machines are ubiquitous models of system behaviour. Variants of finite-state machines appear in many system modelling languages, most prominently SysML [4] and UML [5], [6]. Despite their widespread use and penetration in model-driven software development, the semantics of SysML [4] and UML [7] are ambiguous [8] and restricted versions are offered to create executable models [9], real-time systems [10] or enable formal verification [11]. Moreover, languages such as SysML and UML have historically adopted the event-driven form of finite-state machines inspired by Harel's STATEMATE. Unfortunately, event-driven systems cannot offer a simple semantics, as it becomes cumbersome to manage event queues and the concurrent arrival of events while handling the current event. The issue is intrinsic to these types of machines, where a system is modelled as being in a finite set S of states, and where transitions 'immediately' fire upon arrival of an event (more complexity usually results as executing a transition can itself fire a series of other events).

Complementary to this, `LLFSMs` model a system as being in a finite set S of states. As before, each state ($s \in S$) represents a possible situation that the system may find itself in. But here it is more explicit that while in that state, the `LLFSM` will execute some actions. The system also moves from state to state by means of transitions. However, in sharp contrast with the event-driven approach, each transition is predicated by a logical expression. States are executable states. A state machine is not waiting for events to happen and reacting to them. It is executing its current state s_c, and at a precise point in the execution, the expressions labelling the associated transitions are evaluated. If one of these expressions evaluates to true, the system moves to the target state of the transition, updating the current state. Each `LLFSM` has a state designated as the initial state ($s_0 \in S$), representing the state at the point when execution commences.

Each state contains a set of executable actions. These actions are executed at specific times and under certain conditions. For example Wagner *et al.* define four distinct types of actions [12]:



(a) Diagram for **Entry Actions**.

(b) Diagram providing semantics for **Entry Actions**.

Figure 1. Equivalence Wagner et al. [12] **Entry Actions** in terms of states without sections and transitions.

1) **Entry Actions**: Executed when the system first enters a state.
2) **Exit Actions**: Executed when the system leaves the state.
3) **Transition Actions**: Executed when the system is transitioning between states.
4) **Input Actions**: Executed when an input satisfies a particular condition. These actions can be independent of the state.

We use Wagner *et al.* to illustrate the first point of why a general framework is of interest. We suggest that the fundamental execution cycle is the very simple notion of two states between a transition: a source state s_s and target state s_t. The distinction of an Entry Action a is merely semantic sugar for the removal of an extra state. We illustrate this in Figure 1. Wagner *et al.*'s **Entry Actions** [12] are essentially a pre-state to the state s. Figure 1a is the construct that actually has the semantics of Figure 1b. This is important, because if the expression e in Figure 1 is also `true`, it becomes very transparent that the action a will be performed at least once even if execution exits state s immediately (we note that ambiguities of this type were already identified in standards like `SCXML`).

The proper specification of semantics becomes even more important when the actions in a state access a set of variables that affect subsequent actions and transitions. That is, the attached Boolean expressions (usually named *guards*) involve variables. The first issue is the scope of the variables and the second issue is the potential race conditions that could be generated upon such variables if they are shared in some way. Common cases of variables that are shared are the variables where sensors record a status of the environment. Thus, while the software is executing, the value of a sensor variable may change. Similarly, control variables for effectors are shared. The software modelled by `LLFSMs` may set a control variable and the driver of the effector reads such a variable to act. The prototype `clfsm` [2] for `LLFSMs` provides three levels of scope for variables.

1) **External Variables**: Variables external to the system from the perspective of the software, usually corresponding to the sensors and effectors. They may change at any point in time.
2) **FSM Local Variables**: These are variables that are shared between all states within a single `LLFSM`.
3) **State Local Variables**: These are variables that are local to a state.

Naturally, one can specify more variants. For example, why not have variables that are shared between all the `LLFSMs` of a system, but not sensors and effectors? Why not have variables whose scope is even more local than that of a state, e.g., only local to the **OnEntry** section? These examples illustrate the need for a flexible approach to extending the possibilities of `LLFSM` constructs and form the proposed framework of this paper.

III. Protocol Oriented Design

Protocols (akin to interfaces in `Java`) are a common mechanism to establish the contract a module (or set of classes under a main class) is to adhere to in order to participate and implement some functionality. he protocol itself defines the signatures (names and parameters) of the methods (and if appropriate return values with types) in order for objects to cooperate. In some cases the protocol also specifies invariants and exceptions.

Our `swiftfsm` framework uses protocols extensively to stipulate the required functionality. However, typically, the protocols themselves contain no implementation (although it is possible in `Swift` to have a default implementation), thus a type (class) that conforms to a protocol provides its specific implementation for the functionality that the protocol encapsulates. We use protocol-oriented design to model the semantics of a model, and thus, we focus on describing a set of protocols. When a software engineer wants to develop an implementation of the semantics; these shall conform to a specific set of protocols and implement the required functionality. This therefore enables a developer to design how different parts of the system interact and function, without the need to create a global implementation of behaviour or a new implementation to generate Kripke structures for verification. Moreover, the framework allows the developer to create different implementations for specific, convenient modelling of constructs that conform to the same semantics modelled by these protocols.

IV. Modelling States and Transitions

We are now ready to present our first abstraction: the type for transitions. To introduce the idea, consider the following scenario where allowing developers to create custom semantics leads to more robust designs. Let's focus on a state A (Fig. 2). The `clfsm` semantics [1] explicitly specifies that the *onEntry* action will execute once and only once for each state, after which the sequence of transitions will be evaluated in the order α, then β. If the associated expression (not shown) evaluates to `true`, the corresponding transition will fire and the state will execute its *onExit* action. If none of the transitions fire, the *Internal* action will be run. In either case, the execution token passes to the next `LLFSM` in the arrangement.

Importantly, this way it is not possible to implement an *atLeastOnce* semantics for the *Internal* action without adding another state. If transitions α or β cause a state transition, (in the `clfsm` semantics [1]), then the *Internal* action will never execute. If this functionality is required, a pattern similar to Figure 3 needs to be implemented. Note that this involves creating two states and copying (duplicating) implementation, obstructing factorisation and creating the danger of introducing failures. Both $a1$ and $a3$ need to be copied into the new state $A0$ in order to implement the *atLeastOnce* semantics. State $A1$ is almost the same as the original state A. This becomes arduous to maintain and modify as the developer must keep the $A0$ actions in sync with the $A1$ actions.

With `swiftfsm`, we overcome this problem by allowing developers to define custom state types. The result is shown in Figure 4. Because of the wide breadth of state models, `swiftfsm` only assumes that a state has a unique name.

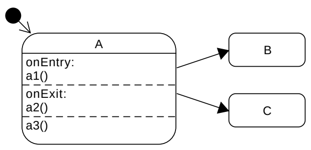


Figure 2. A simple scenario

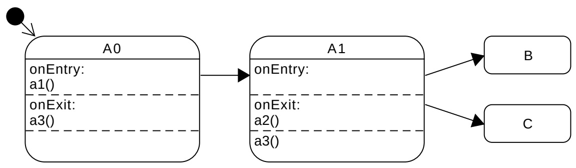


Figure 3. Implementing “atLeastOnce” semantics in `clfsm`

Therefore `swiftfsm` defines a `StateType` protocol only containing that name. The developer has complete freedom to define any number of phase-actions that make up a state.

The `swiftfsm` framework does not even assume that a state can transition. This is a separate requirement, modelled as a separate protocol. The `Transitionable` protocol adds a sequence of transitions to conforming states. All transitions contain

1) a predicate function that, when it evaluates to `true`, represents a situation where the `LLFSM` will transition; and
2) a *target* state the `LLFSM` will transition to.

The type of the transition predicate function is defined as:

$$\mathit{StateContext} \rightarrow \mathit{Boolean}$$

This abstracts a state context type that encapsulates all (and only) the necessary variables that influence the evaluation of the predicate function. In this way, a transition function can access the necessary variables through its source state. This is an important concept when generating the corresponding Kripke structure of an executable model in order to perform formal verification. The generation of the Kripke structure depends on referential transparency, i.e., transitions will be evaluated with any possible combination of state context variations passed in with no further dependencies or side-effects.

This allows for an important optimisation. Typically, an `LLFSM` state corresponds to several Kripke states, because of

- state sections (e.g., **onEntry**, **onExit**, **Internal**, **atLeastOnce**, etc.), and
- the potential semantics of snapshotting external variables between these state sections.

However, our semantics recognises that external variables that are not involved in a transition will not need to create a new transition evaluation context. Therefore, the above transition type is side-effect free and removes the need to consider all possible combinations of external variables outside those appearing in the transaction.

The traditional conceptualisation of the class of transitions is that transitions have a source and a target state. Such a conceptualisation complicates the optimisation we just mentioned, as the transition is in a static relationship with its source

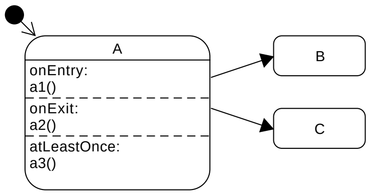


Figure 4. Implementing “atLeastOnce” semantics in `swiftfsm`

state (typically implemented as a reference). Our approach does not need to change the source state of a transition in an `LLFSM` to create the Kripke states for sections. Our framework only updates the possible changes to the external variables of relevance, and submits the State with this new context for evaluation to the transition (which is a pure function). Importantly, this means that the evaluation of any transition is referentially transparent as it is a pure function with explicit inputs and outputs. The Kripke structure generated in this way is guaranteed to obtain the effect of evaluation of the transition without possible side effects influencing the transition as all the variables are in the context attached to the state.

V. SCHEDULING

Here, we introduce a new abstraction over the original concept of an `LLFSM` ringlet [1]. A ringlet defines how the sections within a state are executed, and more specifically, how and in what order each action is executed. We propose to view ringlets as pure functions that take a state and return the next state to execute. Therefore we have them as objects of the following type.

$$\mathit{State} \rightarrow \mathit{State}.$$

If a new state is returned, then the `LLFSM` has transitioned. By modelling a ringlet in this fashion, we enable developers to create custom ringlets which determine how their states are executed. As an illustration of the adaptability of this approach, it is also possible to create different ringlets that execute the same states in different ways. Importantly, the execution of the state becomes orthogonal to the definition of the state.

However, in practice it is common that a ringlet may require to modify state information. To this end, the `swiftfsm` framework provides the `Ringlet` protocol which defines an `execute` function. If we look at previous semantics for `LLFSMs`, and in particular to the semantics offered by the `clfsm` compiler, we can see that the ringlet only executes the *onEntry* section when the previously executed state does not equal the current state being executed (in particular, if a state has a transition to itself, this is a legal construct, but if the transaction executes, in `clfsm` this does not re-run the *onEntry* section). If a developers wished to extend the semantics that all arriving transitions (including self-transitions) cause the *onEntry* section to execute, our framework here allows the creation of a `CLFSMRinglet` that contains a `previousState` member variable that the `execute` function refers to and manages when executing the current state. That is we are using the `Method` pattern, and the developer supplies the method that defines the specific ringlet to sequence sections of a state.

Because `LLFSM` are not event-driven, they are scheduled using a round robin scheduler. We provide such scheduling as the default in the framework `swiftfsm`. Therefore, a single ringlet, for the current state of each `LLFSM`, is executed in a sequential fashion. This creates concurrent execution in a predictable manner reducing state explosion for formal verification. The sequential execution avoids thread management and avoids complexities associated with parallel execution, (there are essentially no critical sections or mutual exclusion challenges). Because of the sequential scheduling, we have a deterministic execution of an arrangement of the `LLFSMs`, thus when the Kripke structure is created for the entire arrangement, we have a smaller Kripke model (a smaller NuSMV input file) that with unconstrained concurrency of event-driven systems. By preventing side-effects (as shown in the previous Section), we further reduce the size of the Kripke structure enhancing the feasibility of performing model checking.

Furthermore, `swiftfsm` uses a stricter snapshot semantics when executing the ringlets. A snapshot is taken of the external variables before the ringlet is executed. The state then uses the snapshot when executing actions and evaluating transitions (recall our execution context). Only once the ringlet has finished executing, any modifications made made visible externally (e.g., to the environment). This defines the granularity at which the system is reactive to changes observable by sensors in the environment and does not need to make a dangerous assumption of well-behaved environments and that the software always runs faster than any external part of the system. Compare this with many formal verification approaches that only work with ideal event-driven systems, that do not exist in practice. For example, approaches where extended finite-state machines handling of external variables is simply assumed to be irrelevant. "During a macrostep, the values of the inputs do not change and no new external events may arrive; in other words, the system is assumed to be infinitely faster than the environment" [13, p. 172]. Alternatively, the environment is assumed to be well-behaved, so that it sends the input the software requires at the right time, forming "a closed model corresponding to the complete mathematical simulation of the pair formed by the software controller and the environment" [14, p. 89]. Finally, a simplistic approach where any external stimulus (change of external variables) will not happen until all internal changes take place "giving priority to internal actions over external actions" [15].

We argue that the specification of when a snapshot is taken defines the level of atomicity of the sections within the state run by the ringlet with respect to the external variables. This becomes particularly important when performing formal verification.

VI. Formal Verification

If one strictly follows the derivation of Kripke structures from the artefact of sequential program constructs [16], the corresponding Kripke states would not only be the boundaries of sections of `LLFSM` states, but every assignment and operation in those sections correspond to extended `FSMs`, containing programming language statements (e.g., in `Swift`). The sequential execution of `LLFSMs` and its default snapshot semantics enables more succinct Kripke structures, where the delicate point is the handling of the external variables [17], [18]. Nevertheless, as we mentioned, such a default semantics requires recording all of the variables influencing the execution before and after every state section in order to generate the Kripke structure [17]. For consistency, we configured a version of `swiftfsm` that followed such an approach [3].

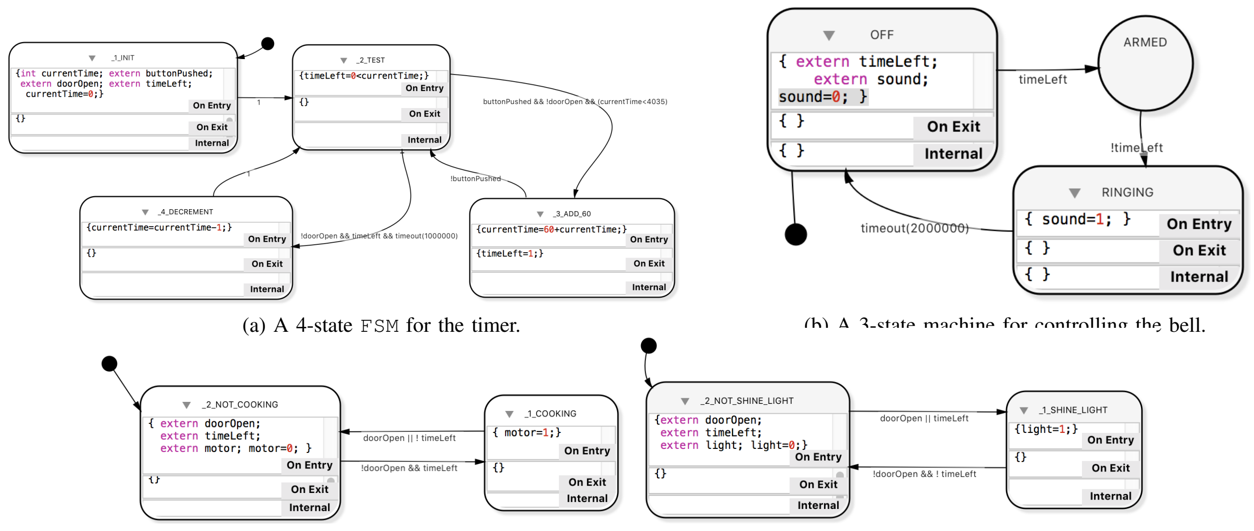
These earlier approaches relied on the ringlet itself to record variables, influencing the execution of a state. However, a more succinct approach can be used and a further optimisation can be made. Since the `swiftfsm` framework not only uses a sequential scheduling similar to `clfsm`, but a ringlet's execution is atomic with respect to the external variables, ringlet execution can now be treated as a black box.

Consequently, a snapshot should only be taken of the variables before and after the entire ringlet for a state is executed. This variation also prevents statements being executed that make modification to variables that are not reflected in the final context for the next Kripke state. For example, a state may make changes to an external variable during an **onEntry** section that is cancelled by a further modification in the **onExit** section. Since no effect of this will occur during the state's execution, as we now identify a Kripke state *before* and *after* an entire ringlet execution, interim changes are not reflected in the resulting Kripke structure.

Importantly, we argue that this is a benefit, not a problem! In `swiftfsm`, the statements within sections of the state operate within a context derived from a snapshot of the external variables, which gets taken precisely when the state is scheduled. There is absolutely no way that any modification could (nor should) affect the environment until the snapshot is saved. External variables are updated precisely once when the ringlet has finished executing. Similarly, since `swiftfsm` uses sequential scheduling, there is no way for the modification of non-external variables to have side-effects and influence the execution of other machines, because the semantics is equivalent to a single thread. The only important record for the construction of the Kripke states (to be part of the Kripke structure or verification) is the context (of the variables) before and after each ringlet is executed.

VII. Case Study

We present a case study where we simplify the model of a microwave oven, a ubiquitous example in the software engineering literature of behaviour modelling through states and transitions [19]. This model has been extensively studied in formal verification [20, p. 39], as the safety feature of *disable cooking when the door is open* is analogous to the requirement that a radiation machine should have a halt-sensor [21, p. 2]. Software models for microwave behaviour are widely discussed [22], [23], [24], [25], [26], [27]). Figure 5 shows the standard executable model with `LLFSMs`. While this model is transparent and formal verification establishes requirements, the full machinery of Kripke states for each of the three state-sections is not required (note that all **Internal** sections are empty and the only **onExit** section that is used is in the timer `LLFSM`, in state 3 to `ADD_60`. Moreover, the model would also be simplified if the `timeLeft` variable were to be removed by making it equivalent to the condition `0<currentTime`. With respect to the requirements specified in Myers and Dromey [27, p. 27, Table 1] or in Shlaer and Mellor [23, p. 36] the behaviour of such a simplification is irrelevant. But, for model checking, removing the Boolean variable `timeLeft` alone would half the number of Kripke states (and the corresponding size of the NuSMV file where formal verification is conducted is thus halved). By removing the state sections, the number of Kripke states would be halved again. Thus, it would be advantageous to derive `LLFSMs`, where states have no **onExit** nor **Internal** actions.


_1_INIT
{int currentTime; extern buttonPushed; extern doorOpen; extern timeLeft; currentTime=0;}
On Entry
On Exit
Internal
_2_TEST
{timeLeft=0<currentTime;}
buttonPushed && !doorOpen && (currentTime<4035)
!buttonPushed
_4_DECREMENT
{currentTime=currentTime-1;}
!doorOpen && timeLeft && timeout(1000000)
_3_ADD_60
{currentTime=60+currentTime;}
{timeLeft=1;}
OFF
{ extern timeLeft; extern sound; sound=0; }
ARMED
timeLeft
!timeLeft
RINGING
{ sound=1; }
timeout(2000000)
_2_NOT_COOKING
{ extern doorOpen; extern timeLeft; extern motor; motor=0; }
_1_COOKING
{ motor=1;}
doorOpen || ! timeLeft
!doorOpen && timeLeft
_2_NOT_SHINE_LIGHT
{extern doorOpen; extern timeLeft; extern light; light=0;}
_1_SHINE_LIGHT
{light=1;}
doorOpen || timeLeft
!doorOpen && ! timeLeft

(a) A 4-state FSM for the timer.

(b) A 3-state machine for controlling the bell.

(c) A 2-state machine for controlling the cooking engine.

(d) A 2-state machine for the light.

Figure 5. Complete model of one-minute microwave.

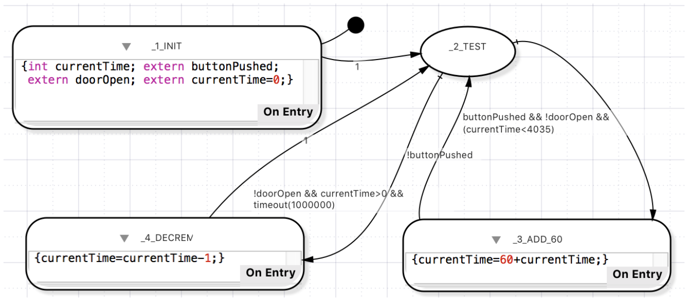

_1_INIT
{int currentTime; extern buttonPushed; extern doorOpen; extern currentTime=0;}
On Entry
_2_TEST
1
buttonPushed && !doorOpen && (currentTime<4035)
!buttonPushed
!doorOpen && currentTime>0 && timeout(1000000)
_4_DECREM
{currentTime=currentTime-1;}
_3_ADD_60
{currentTime=60+currentTime;}

Figure 6. Simplified timer with **onEntry** sections only.

The new model would globally replace `timeLeft` by `0<currentTime`. All declarations of `extern timeLeft` disappear from all LLFSMs. Thus, the timer machine changes to Figure 6. We point out the slight change of behaviour. With the executable model of Figure 5, when the button is pressed for the first time and not released, nothing would happen. With the changes suggested, when the button is pressed for the first time and not released, if the door is closed, cooking will commence and the light will go on. As long as the button is pressed and not released such cooking with the light on will continue and the timer will not be decremented. This behaviour does exist in a slightly similar form in Figure 5, but only happens from the second time onwards. That is, the user must press the button; upon releasing the button, cooking starts and the light turns on. If the user presses and holds the button now that cooking has started, it also blocks timing counting down. Again, we do not consider this subtle difference in behaviour relevant as it is never identified in the requirement. However, the variation simplifies the Kripke structure radically for more efficient formal verification of the requirements. With our framework, the designers can easily alternate between the two executable models, and conduct model checking on both.

A further optimisation can be made when considering how `swiftfsm` currently handles the snapshots of external variables. Recall that a snapshot is taken before the ringlet executes, and then saved back to the environment once the ringlet has finished executing. By changing these semantics to a per-schedule cycle, as opposed to a per-ringlet cycle, we can further minimise the number of Kripke States that are generated. Taking the microwave as an example, instead of taking a snapshot of the external variables before executing each state, we instead take a single snapshot of the environment before executing the ringlet for the current state within each LLFSM. Each LLFSM would therefore share the same snapshot and any modifications made to the snapshot will only be saved once each LLFSM has executed its current state.

This has a drastic impact to the number of Kripke States that are generated for the Kripke Structure. Consider all possible combinations of a snapshot of the external variables. The microwave uses three Boolean variables, therefore this results in $2^3 = 8$ possible combinations. There are normally four snapshots taken per schedule cycle as there are four LLFSMs executing and a snapshot is taken when a ringlet in each LLFSM is executed. Therefore, there are $2^{3^4} = 4096$ possible combinations of snapshots per schedule cycle. When taking a single snapshot at the start of the schedule cycle, the result is $2^{3^1} = 8$ possible combinations of snapshots. Removing the `timeLeft` variable further reduces this to

$2^{2^1} = 4$ combinations of snapshots per schedule cycle, a reduction by three orders of magnitude.

VIII. CONCLUSION

In this paper, we have introduced a flexible semantic model for logic-labelled finite-state machines. Compared to traditional event-driven state machines and LLFSMs, our approach allows a more direct mapping of UML semantics [5], [6], allowing high-level, executable models, which are less error-prone and eliminate duplication. Moreover, we have shown these semantics can be modelled in a referentially transparent way that creates simpler Kripke structures, allowing formal verification of our executable models, that is orders of magnitudes faster for the same model than previous approaches.

In software engineering, there is a prevalence for modelling using UML state charts (which is a derivation of Harel's State Charts [28]) and which are event-driven. Moreover, Sommerville [29], states that "state models are often used to describe real-time systems" [29, p. 544], citing UML. We note that Sommerville also uses a microwave to illustrate how FSMs model the behaviour of systems [29, p. 136]. Because of these associations among systems that respond to stimuli, we thank the reviewers for suggesting to clarify the terminology regarding what constitutes an event-driven system, a reactive system and more importantly, a real-time system.

We refer to an event-driven system as one typically based on a software architecture built around stimuli-driven call-backs, a subscribe mechanism and listeners that enact such call-backs (very much as GUIs are composed for desktops today). Reacting to stimuli in this way implies uncontrolled concurrency (e.g. using separate threads or event queues). The counterpart to event-driven systems are time-triggered systems. Lamport [30] provided fundamental proofs of the limitations of event-driven systems. Reactive-systems are responsive systems without much processing, as opposed to deliberative systems (which reason, plan, learn). Real-time systems are required to meet time-deadlines in response to stimuli. Therefore, although closely related, these terms are not the same, and in this paper, we argue (supported by the work of Lamport [30]) that there are many solid reasons why real-time systems may be better served by time-triggered systems and pre-determined schedules, rather than the unbounded delays that may occur in event-driven systems.

The work presented in this paper illustrates how LLFSMs can be used as executable models. Moreover, we argue that their deterministic execution and verifiability is more suitable for real-time systems than systems where threads proliferate.

REFERENCES

[1] V. Estivill-Castro and R. Hexel, "Arrangements of finite-state machines - semantics, simulation, and model checking," in MODELSWARD, S. Hammoudi, L. F. Pires, J. Filipe, and R. C. das Neves, Eds. SciTePress, 2013, pp. 182–189.

[2] V. Estivill-Castro, R. Hexel, and C. Lusty, "High performance relaying of c++11 objects across processes and logic-labeled finite-state machines." Springer Int. 2014, pp. 182–194.

[3] C. M^{c}Coll, "swiftfsm - A Finite State Machines Scheduler," Honours Thesis, Griffith University, Nathan QLD, 4111, Australia, 2016.

[4] S. Friedenthal, A. Moore, and R. Steiner, A Practical Guide to SysML: The systems Modeling Language. San Mateo, CA: Morgan Kaufmann, 2009.

[5] M. Samek, Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems. Newton, MA, USA: Newnes, 2008.

[6] D. Pilone and N. Pitman, UML 2.0 in a Nutshell. O'Reilly Media, 2005.

[7] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd ed. Boston, MA, USA: Addison-Wesley Longman, 2003.

[8] R. Rumpe, "Executable modeling with UML – a vision or a nightmare? –," in Issues and Trends of Information Technology Management in Contemporary Associations Volume 1, M. Khosrowpour, Ed. Idea Group, 2002, pp. 697–701.

[9] S. J. Mellor and M. Balcer, Executable UML: A foundation for model-driven architecture. Reading, MA: Addison-Wesley, 2002.

[10] B. P. Douglass, Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition). Redwood City, CA, USA: Addison Wesley Longman, 2004.

[11] A. Krupp, O. Lundkvist, T. Schattkowsky, and C. Snook, "The adaptive cruise controller case study — visualisation, validation, and temporal verification," in UML-B Specification for Proven Embedded Systems Design, J. Mermet, Ed. Springer US, 2004, pp. 199–210.

[12] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, Modeling Software with Finite State Machines: A Practical Approach. CRC Press, Boca Raton, FL 2006.

[13] W. Chan, R. J. Anderson, P. Beame, D. Notkin, D. H. Jones, and W. E. Warner, "Optimizing symbolic model checking for statecharts," IEEE Trans. Softw. Eng., vol. 27, no. 2, Feb. 2001, pp. 170–190.

[14] J.-R. Abrial, Modeling in Event-B - System and Software Engineering. Cambridge Uni., 2010.

[15] 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, 2011, pp. 1233–1258.

[16] E. M. Clarke, O. Grumberg, and D. Peled, Model checking. MIT Press, 2001.

[17] V. Estivill-Castro and D. A. Rosenblueth, Model Checking of Transition-Labeled Finite-State Machines. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 61–73.

[18] V. Estivill-Castro, R. Hexel, and D. A. Rosenblueth, "Efficient modelling of embedded software systems and their formal verification," 19th Asia-Pacific Software Engineering Conf., vol. 1, 2012, pp. 428–433.

[19] I. Sommerville, Software engineering (9th ed.). Boston, MA, USA: Addison-Wesley Longman, 2010.

[20] E. M. Clarke, O. Grumberg, and D. Peled, Model checking. MIT Press, 2001.

[21] C. Baier and J.-P. Katoen, Principles of model checking. MIT Press, 2008.

[22] S. J. Mellor, "Embedded systems in UML," OMG White paper, 2007, www.omg.org/news/whitepapers/ label: "We can generate Systems Today" Retrieved: April 2017.

[23] S. Shlaer and S. J. Mellor, Object lifecycles : modeling the world in states. Englewood Cliffs, N.J.: Yourdon Press, 1992.

[24] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, Modeling Software with Finite State Machines: A Practical Approach. NY: CRC Press, 2006.

[25] L. Wen and R. G. Dromey, "From requirements change to design change: A formal path," 2nd Int. Conf. Software Engineering and Formal Methods (SEFM 2004). Beijing, China: IEEE Computer Soc., 2004, pp. 104–113.

[26] R. G. Dromey and D. Powell, "Early requirements defect detection," TickIT Journal, vol. 4Q05, 2005, pp. 3–13.

[27] T. Myers and R. G. Dromey, "From requirements to embedded software - formalising the key steps," 20th Australian Software Engineering Conf. Gold Cost, Australia: IEEE Computer Soc., 2009, pp. 23–33.

[28] D. Harel and M. Politi, Modeling Reactive Systems with Statecharts: The Statemate Approach. New York, NY, USA: McGraw-Hill, 1998.

[29] I. Sommerville, Software Engineering, 9th ed. USA: Addison-Wesley, 2010.

[30] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," ACM Transactions on Programming Languages and Systems, vol. 6, 1984, pp. 254–280.