Versatile but Precise Semantics for 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 well-defined semantics that enables testing as well as formal verification. While this combination of the three elements (MDSD, validation, and verification) results in more reliable behaviour of software components, semantics is severely constrained in several areas. Here, we offer a framework that allows flexibility in execution semantics to suit specific domains while maintaining rigour and the capability to generate Kripke structures for formal verification or to execute corresponding monitoring or testing LLFSMs for validation in a test-driven development framework. Through the use of modern constructs that extend the object-oriented paradigm, the framework is able to define a set of semantics that enables versatile approaches to LLFSM definition and execution, as well as enabling functional programming constructs. This vastly increases the versatility and usefulness of LLFSMs, making them more adaptable to different domains, without sacrificing the benefits of executable models and the ability to perform formal verification.**

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

I. Introduction

We argue here that it should be possible to rapidly and efficiently configure the semantics and Logic-Labelled Finite State Machines (LLFSMs) constructs to provide developers with the freedom to adapt or tailor the system semantics to their particular scenario. This paper shows that we can enable such versatility. We provide the capacity to instantiate new scheduling semantics with incarnations of template methods and classes, while retaining the capacity to generate corresponding Kripke structures for formal verification. The generated Kripke structures can be formally verified with standard tools [1], such as NuSMV.

By following a limited, precise semantics that is based on a synchronous concurrency model, LLFSM enable the design of software that can achieve high levels of complexity and sophistication while guaranteeing deterministic execution and facilitating formal verification [2]. The LLFSM semantics specifies 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 [3]. However, this constrains the semantics of the executable model to one specific frequency and pace, which limits the expressiveness of the designer in a way that may not be well-suited for a specific robotic or embedded target system.

Therefore, our new framework removes the need to adhere to the strict semantics currently implemented in tools such as clfsm [3]. Importantly, we demonstrate that maintain the ability to perform formal verification. To this end, we illustrate two areas, where we create abstractions to the LLFSM semantics and show how instantiation of these abstractions into concrete derivations maintain the ability to perform formal verification. We extend swiftfsm [4], a framework for LLFSMs written in Swift, enabling formal verification, while allowing developers more freedom to design, adapt and create new LLFSM models that suit particular, 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 [5] and UML [6], [7], [8]. Despite their widespread use and penetration in model-driven software development, the semantics of SysML [5] and UML [9] are ambiguous [10] and restricted versions are offered to create executable models [11], real-time systems [12] or enable formal verification [13], [14]. 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 the possibility of concurrent event arrivals (e.g., from the environment or other components), can create unintended complexities emerging from subsystem interaction. This 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 event handling can itself fire a series of resulting events). The mathematical model of instantaneous (zero-time) transition is rapidly discarded because *"the software actually consumes time when processing those events"* [8, Page 50].

An example of the event-driven approach is the Specification and Description Language (SDL) fomalised by the ITU-T [15]. SDL allows the modelling of reactive systems [1], and with SDL-RT [16], an extension to SDL,

[1]Later we highlight the distinction between event-driven systems, reactive systems, and real-time system

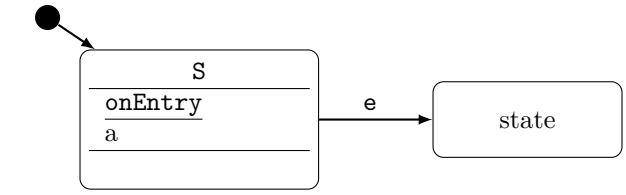
the notation aims at modelling of real-time systems. SDL (and SDL-RT) model a system as a set of asynchronously executing finite state machines called processes. Each process can communicate through the use of signals which constitute communication channels between processes. A sender does not wait for an acknowledgement from the receiver and the receiver places the signal on a queue where it remains until it is able to handle the event.

Each process may be listening for external events, i.e., events that are generated from other processes or the environment. Lamport [17] demonstrated the fundamental composability limitations of event-driven systems. Event-driven systems are designed for the best or the average case, but can result in unbounded delays or message queue sizes in the worst case, resulting in devastating consequences for real-time systems. This is because the execution of the process is completely dependent on the ordering and timing of events that the process has no control over. In extreme situations, such as an event shower, this leads to race conditions resulting in non-deterministic behaviour. To avoid running out of memory, event queues used to store events often are of a fixed size. During an event shower, such a queue can reach capacity, thus resulting in events being missed. This can therefore lead to ambiguous and nuanced behaviour, that is difficult to reproduce, as the process could potentially be in a state which it is waiting for events that have now gone undetected.

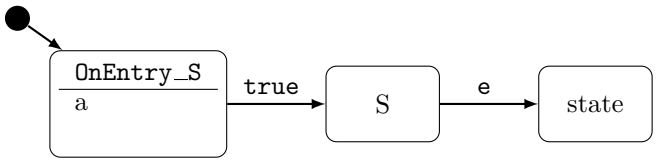
An event shower also has the added problem of making the time it takes to react to an event non-deterministic. The reason for this is that (under the standard *run-till completion* semantics [6], [8]) the system must process all events in the queue before reacting to future events. When a new event happens, the current queue size can hold a few or a large number of disparate events, making it all but impossible to predict how long the new event will be in the queue until it can be dealt with. Most notations, such as UML, SysML, and even SDL, compound this problem by allowing processes to generate new events as part of the transition actions: "A transition performs a sequence of actions. During a transition, the data of an agent is possibly manipulated and signals possibly output (depending on the content of the transition)." [18, p. 53]. That is, the firing of a transition triggers further events "instantaneously" and "simultaneously"; again, violating the ideal mathematical model that *"as close as they might be in time, events are never simultaneous"* [8, Page 50].

Therefore, an event shower, rather than been avoided by the notation, can, in fact, be fostered by the modelling language and caused by the composition of, individually benign subsystems; thus, placing a huge burden on system designers as they must now anticipate subsystem interactions, including how all possible combinations of events could influence the execution of the system. It quickly becomes impossible to discern how the combinations of all executing process contribute to the execution and timing of a particular sub-system. The solution is to move a way from the event-triggered paradigm entirely.

Complementary to event-driven fsms, `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.



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



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

Figure 1. Equivalence Wagner et al. [19] **Entry Actions** in terms of states without sections and transitions.

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 (those familiar with UML would find this as restricting the labels of transitions to only using guards; and such models have been named *procedural* [8]). States are executable states. A state machine is not waiting for events to happen and reacting to them. Instead, it keeps executing its current state s_c, and at a precise point in the execution within its own sphere of control, 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 [19]:

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** [19] 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 clear 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 such as 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 (as we mentioned, in UML and OMT these are named *guards*) involve variables. The first issue is the scope of the variables and the second issue is that of potential race conditions that could be generated due to these variables being 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, without the software being able to control or influence the timing of these changes. 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 clfsm [3] implementation of 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. This need inspires the framework proposed in the present paper.

III. Protocol Oriented Design

Protocols in Swift are a refinement of an object-oriented construct that enables a developer to define the intent or semantics of a type without necessarily providing a concrete implementation. In this respect, protocols are similar to Java interfaces; a type that implements (or conforms to) a protocol enters into a contract, where the conforming type must implement the constraints imposed by the protocol. The advantage is that any type that conforms to the protocol conforms to the same set of semantics as any other type that also conforms to the same protocol. This enables the use of either type interchangeably. In this sense, protocols are a way to enforce semantic constraints. They enable the modelling of individual pieces of the software without the burden of defining how each piece is implemented. This creates loose coupling between types as a type may depend on a protocol (or rather the semantics which are defined within the protocol) and any type which conforms to the protocol may provide the necessary implementation.

Unlike interfaces, however, Swift takes this idea further with protocol extensions, providing a mechanism of aspect-oriented programming. This enables conforming types to receive default implementations. In other words, a conforming type may receive some or all of its implementation for free, based on the contracts provided by protocols alone. This is not to be confused with standard class inheritance. Standard class inheritance imposes a strict implementation hierarchy and close coupling on inherited types. An implementation within a class may depend on private members or other strict types, an implementation received by a protocol extension simply models how variables and functions in the protocol can be leveraged to provide a default implementation. Importantly, a protocol extension depends only on the contract it and other protocols provide, and only on the public interface provided by these protocols.

However, the true power of a protocol extension is with conditional extensions. A conditional extension enables the developer to define rules on which conforming types receive which default implementation (if any). This way, the protocol extension can also be restrictive so that only types that conform to a specific set of protocols may receive the implementation.

Consider the example in Figure 2, showing the Collection protocol that contains a generic parameter Element. In Swift, a generic parameter on a protocol is known as an *associated type*. This protocol models a collection that contains zero or more elements. Element represents the type of the elements within the collection. Two concrete implementations conform to this protocol: Array and Set. These behave differently: Set only contains unique elements, while Array may contain duplicates. Swift models equality operations in the Equatable protocol. This protocol defines that conforming types must provide == and != functions. A protocol extension can be created on the Collection protocol which provides a default implementation for the == and != functions. These functions compare all the elements within two collections to find if they are equal, assuming that the individual elements within the collections can be compared. This is shown in Extension 1.

Extension 1. *Extend Collection where Element is Equatable:*

```
function == (lhs: Collection, rhs: Collection)
    for le in lhs, re in rhs
        if le != re
            return false
        end
    end
    return true
end

function != (lhs: Collection, rhs: Collection)
    return !(lhs == rhs)
end
```

Examples:

```
nums := [1, 2, 3]
nums == nums // Ok

people := [Person(), Person(), Person()]
people == people // Person is not Equatable
```

Now all types that conform to `Collection` (`Array`, `Set`) are able to use equality operators, assuming that the elements within the collections are able to be compared.

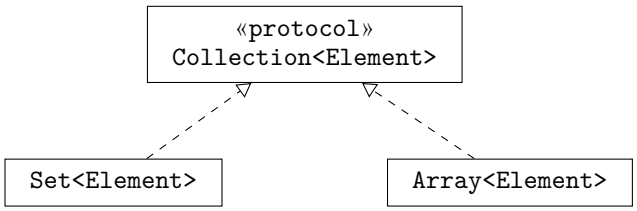


Figure 2. Collections Hierarchy

We use protocols and protocol extensions extensively in `swiftfsm` to define rules on how certain things must be implemented. This way, the developer is constrained to semantics that enable verification and model checking without otherwise restricting the language.

The `swiftfsm` framework minimises the possibility of creating a semantics whose tools for composition result in combinatorial explosion. For example, nesting in UML's statecharts is what enables the modelling of larger systems; however *"the Cartesian product machine is used as the interlingua semantics of statecharts"* [8, Page 63]. On the other hand, we shall not restrict expressivity so that only trivial systems can be modelled. We want to remain Turing complete although all properties of some executable models would not be verifiable (such as the famous Halting problem). The challenge is to enable developers to tailor the semantics to their most effective constructs while retaining small Kripke structures verifiable by standard tools.

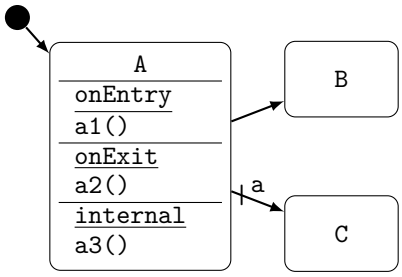


Figure 3. A simple scenario

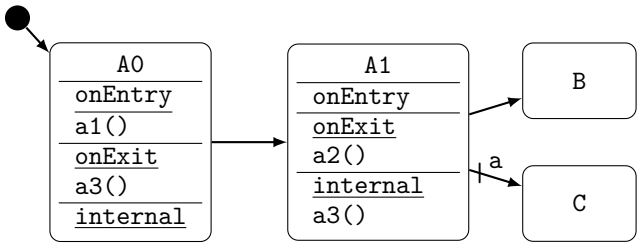


Figure 4. Implementing "atLeastOnce" semantics in `clfsm`

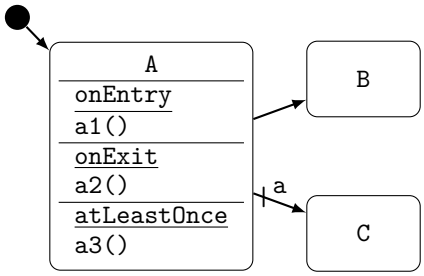


Figure 5. Implementing "atLeastOnce" semantics in `swiftfsm`

IV. Modelling States and Transitions

We are now ready to present our first abstraction: the type used 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. 3). The `clfsm` semantics [2] 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 a, then b. 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, with this restricted semantics, it is not possible to implement an *atLeastOnce* semantics for the *Internal* action without adding another state. If transitions a or b cause a state transition, (in the `clfsm` semantics [2]), then the *Internal* action will never execute. If this functionality is required, a pattern similar to Figure 4 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 that represent such a semantics. The result is shown in Figure 5. As can be seen, the figure demonstrates how a developer may define the new *atLeastOnce* action. While this approach may seem unnecessary to apply in such a simple case, it does show the added expressiveness of the semantics which the developer may choose to employ.

Nothing is stopping the developer from creating further changes to the semantics of how states execute. Ideas such as transition actions (actions which are triggered when specific transitions are taken) become possible. The ability to create more complicated models which could utilise ideas such as state clustering or hierarchies of state machines is also viable now that a developer has the ability to proactively define the state models that they are working with. This allows the developer to tailor the tool-set to the problem they are trying to solve. Most importantly we can achieve all of this, while maintaining a strong type hierarchy and verifiable semantics as shown in Section VII.

The design of custom state types is achieved using a protocol hierarchy. Figure 6 demonstrates how this protocol hierarchy can be used to define the `clfsm` se-

mantics. First, we present the protocol at the top of the hierarchy: `Identifiable`. Instances of types conforming `Identifiable` are able to be distinguished from each other. This is achieved using the name field. Every unique instance is required to have a unique name. The `Identifiable` protocol conforms to the `Equatable` protocol, allowing equality checks by comparing the names of the two instances.

The next protocols are `StateType` and `Transitionable`. These are the protocols that are responsible for representing states and transitions. Because of the wide breadth of state models, `swiftfsm` only assumes that a state has a unique name. therefore, `swiftfsm` defines the `StateType` protocol only containing that name (which is inherited from `Identifiable`). The developer has complete freedom to define any number of phase actions that make up a state. This is done by defining another protocol. For the `clfsm` semantics, this is encapsulated within the `CLFSMActions` protocol. Each action that a state may perform is represented as a member function of the protocol.

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:

$$StateContext \rightarrow Boolean$$

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. We profit and explicitly use referential transparency for the generation of Kripke structures. That is, transitions emanating from a state will be evaluated with a fixed state context variation that has 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 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. Importantly, this means that the evaluation of any transition is referentially transparent, as it is a pure function with explicit inputs and outputs and no side-effects. 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.

Notice that `CLFSMState` does not contain an `addTransition` function. This is provided by a default implementation shown in Extension 2

Extension 2. *Extend `StateType` where `Self` is `Transitionable`:*

```
function addTransition(t: _TransitionType ...)
    transitions := transitions ∪ t
end
```

V. Modelling Logic-Labelled Finite State Machines

The ability to create custom `LLFSM` semantics can be beneficial. Within the confines of the `clfsm` semantics, an `LLFSM` is capable of being suspended, restarted or stopped. The `LLFSM` is also responsible for executing states. While this semantics is expressive, it can be somewhat restrictive. For example, why not make it possible to allow an `LLFSM` to control a given set of `LLFSMs` in a master/slave relationship? This would make it possible to create strict `LLFSM` hierarchies which form a tree structure. This is quite different from the `LLFSM` hierarchies described in the literature [20] that does not enforce any hierarchy and allows any `LLFSM` to control any other `LLFSM`. Our stricter approach here helps ensure clarity and safety.

The way in which `swiftfsm` models `LLFSMs` allows the developer to define custom semantics is again achieved using a protocol hierarchy (Figure 7). The `LLFSM` protocol hierarchy, while more complex, follows the same methods we employed to create the state semantics hierarchy. A base protocol is used and further features are added using separate protocols.

We first present the simplest protocol `StateContainer`. This protocol contains no members, instead it simply defines the `_StateType` associated type. This associated type must be defined to allow conforming types to refer to a single `StateType` type. That is to say, types that are conforming to `StateContainer` are manipulating or interacting with a single type of state in some way.

The first protocol which conforms to `StateContainer` is `FiniteStateMachineType`. This protocol represents an `LLFSM`. However, similar to how the `StateType` protocol made no assumptions about the actions contained within a state, the `FiniteStateMachineType` protocol makes

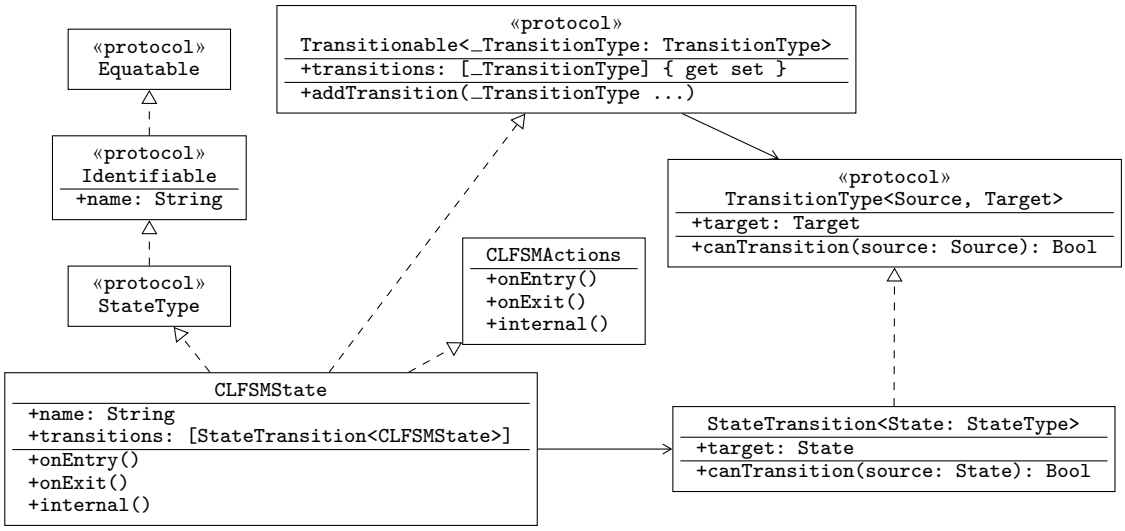


Figure 6. Defining The `clfsm` State Semantics

no assumptions on what the `LLFSM` model may do. The `FiniteStateMachineType` protocol does not even assume that it will execute a state. This is modelled by the `StateExecuter` protocol.

We now introduce a new abstraction over the original concept of an `LLFSM` ringlet [2]. 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.

$$State \rightarrow State.$$

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 that determine how their states are executed. To illustrate the adaptability of this approach, it is also possible to create different ringlets that execute the same set of 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 state currently 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 developer wishes 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.

The `StateExecuter` protocol (in combination with `IncrementalExecuter`) provides a `next` function. This function is responsible for executing the current state. However, if the `LLFSM` contains a ringlet, then the `next` function may delegate the execution of the state to the ringlet. This functionality is encapsulated in the `StateExecuterDelegator` protocol that defines a `StateExecuter`, which contains a `Ringlet`. `FiniteStateMachineType` provides a default implementation for the `next` function when conforming to `StateExecuterDelegator`. This is shown in Extension 3.

A similar pattern is followed for providing the remaining features of the `LLFSM`. In this sense, a feature is modelled as a protocol, and further protocols are created to enable default implementations. A further example is with the `Restartable` protocol. This protocol provides a `restart` function which, when called, should restart the `LLFSM` so that the initial state becomes the current state.

Extension 3. *Extend* `FiniteStateMachineType` *where* `Self` *is* `StateExecuterDelegator` *and* `Self` *is* `PreviousStateContainer` *and* `_StateType` $=$ `RingletType._StateType`*:*

```
function next()
    temp := ringlet.execute(currentState)
    previousState := currentState
    currentState := temp
end
```

However, recall that the `clfsm` semantics state that the `onEntry` action is only executed if the previous state does not equal the current state. If the `LLFSM` was truly to restart, then the previous state should also reset to its initial value. This way, there does not exist a situation where the `LLFSM` does not execute the `onEntry` action. To provide the default implementation

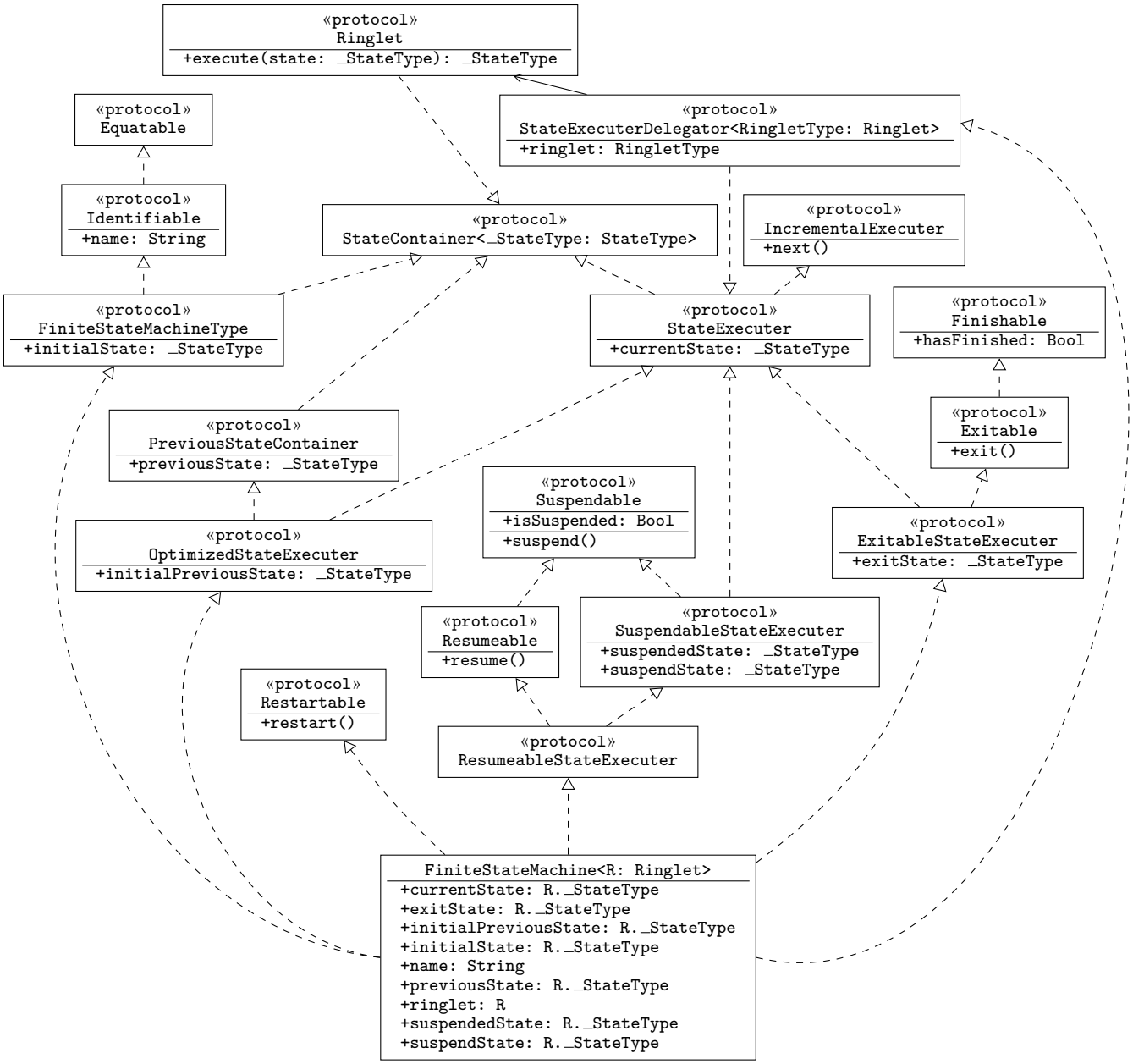


Figure 7. The LLFSM Protocol Hierarchy

for a restart function, the `FiniteStateMachineType` protocol requires that there should also exist a conformance to `OptimizedStateExecuter`. This protocol enables an `initialPreviousState` member which represents the first previous state. With this, a default implementation for the `restart` function is now possible. The `restart` function effectively sets the current state and the previous state to their initial values.

To put everything together, the `FiniteStateMachine` type implements the default model in `swiftfsm`. This implements the same semantics as `clfsm`. Notice, however, that the `FiniteStateMachine` type does not need to provide any function implementations, as the type receives its entire implementation from protocol extensions.

VI. Scheduling

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 appear 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. We detach from such ideal conceptions that "events consume no time: they are zero time episodes " [8, Page 50] and "as close as they might be in time, events are never simultaneous" [8, Page 50] because even in UML it is extremely easy to create an event for which there would be several listeners whose response would be the generation of an event (creating simultaneous events). Or admitting upon further analysis that "the software actually consumes time when processing those events" [8, Page 50]. Thus, we avoid 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" [21, 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" [22, p. 89]. We also do not follow the simplistic approach that assumes any external stimulus (change of external variables) will not happen until all internal changes take place "giving priority to internal actions over external actions" [23].

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.

VII. Formal Verification

If one strictly follows the derivation of Kripke structures from the artefact of sequential program constructs [24], 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 [25], [26]. 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 [25]. For consistency, we configured a version of `swiftfsm` that followed such an approach [4].

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's execution. 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.

To demonstrate the approach for creating a Kripke structure, consider the `Incrementing LLFSM` shown in Figure 8. This `LLFSM` is responsible for incrementing a counter by one. When the value of the counter reaches a non-zero number,the `LLFSM` transitions to the `Exit` accepting state. However, this `LLFSM` provides two boolean external variables, each representing the state of a button in the environment. When both buttons are pushed, the `LLFSM` transitions to the `Exit` state regardless of the value of the counter.

Using the round-robin sequential scheduler requires that a snapshot be taken before and after a state is executed. This is represented in the Kripke structure. An execution of a state results in several nodes. Firstly, taking a snapshot will result in a node being created for each combination of external variables. For the `Incrementing LLFSM`, this will result in a total of 4 nodes. However, the advantage with `swiftfsm` is, as stated previously, treating the execution of a ringlet as an atomic action. Each 4 nodes that represent the reading of a snapshot will each transition to a single node. We therefore classify each node within the Kripke structure as an R (read) node or a W (write) node. R nodes represent the state of the system

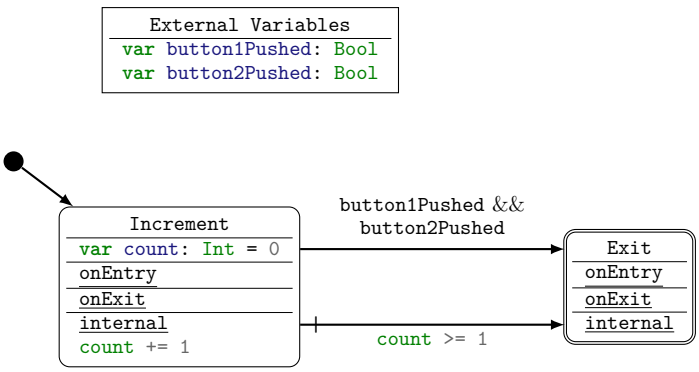


Figure 8. The `Incrementing` LLFSM

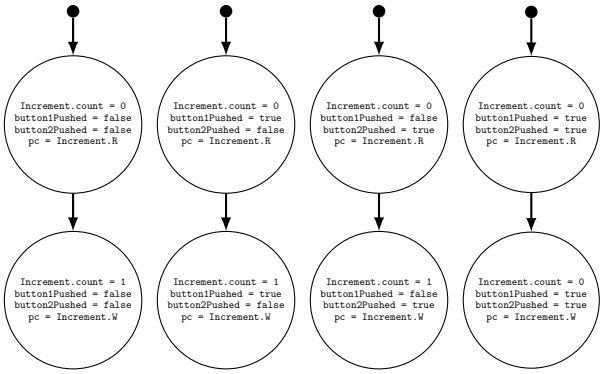


Figure 9. Executing the initial ringlet of the `Incrementing` LLFSM

before a ringlet has been executed, whereas the W nodes represent the state of the system after the ringlet has finished executing. When executing using the round-robin scheduler, a single ringlet execution will result in multiple R nodes each leading to a single W node. This is shown in Figure 9.

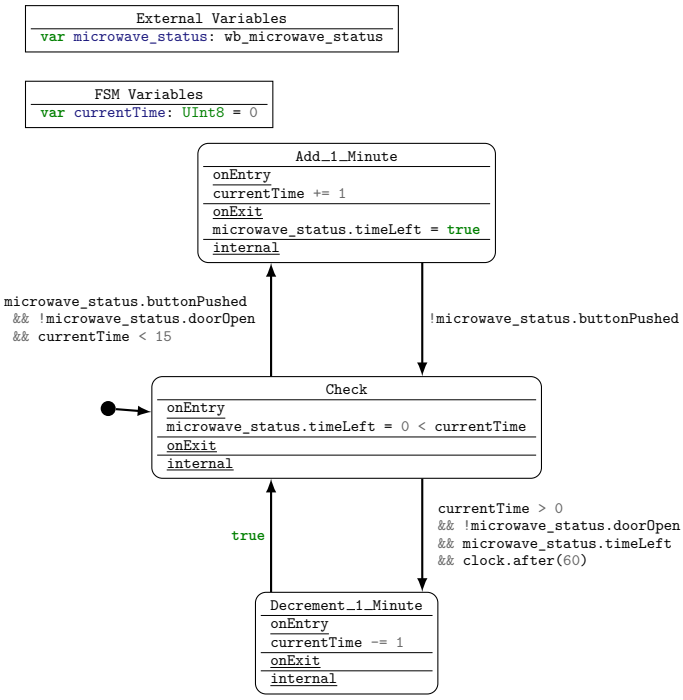
VIII. Microwave Case Study

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 [27]. This model has been extensively studied in formal verification [24, 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 [28, p. 2]. Software models for microwave behaviour are widely discussed [29], [30], [31], [32], [33], [34]). Figure 10 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_1_Minute`). 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 [34, p. 27, Table 1] or in Shlaer and Mellor [30, 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.

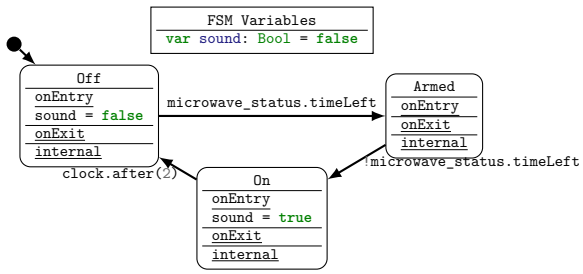
The new model would globally replace `timeLeft` by `0<currentTime`. All declarations of `extern timeLeft` disappear from all `LLFSMs`. Thus, the timer machine changes to Figure 11. This change results in slightly different behaviour. With the executable model of Figure 10, when the button is pressed for the first time and not released, nothing would happen. Now, 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 10, 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 requirements (Figure 12 illustrates the concurrent execution of the arrangement of `LLFSMs` and documents their state changes for a use-case). 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 for each execution of one schedule over the arrangement of `LLFSMs`. Each `LLFSM` would therefore share the same snapshot and any modifications made to the snapshot will only be saved once all `LLFSMs` have executed their current state.

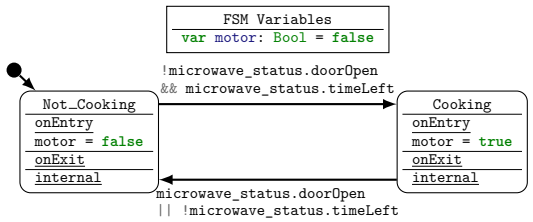
This has a drastic impact to the number of Kripke States that are generated for the Kripke Structure. Consider all possible combinations of a snapshot of the external variables. The microwave uses three Boolean variables, therefore this results in $2^3 = 8$ possible combinations. There are normally four 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



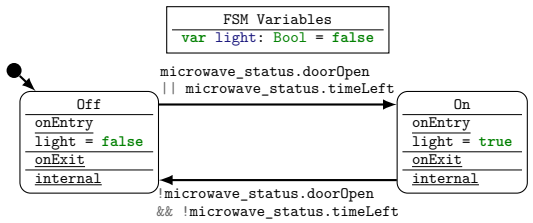
(a) A 3-state machine for the timer.



(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 10. Complete model of one-minute microwave.

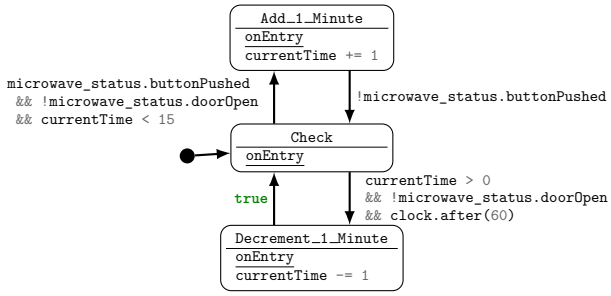


Figure 11. Simplified timer with **onEntry** sections only.

snapshots per schedule cycle, a reduction by three orders of magnitude.

To demonstrate the effect that limiting the snapshots has on the Kripke structure size, we provide a git repository which contains the Kripke structures in several formats for the `Incrementing LLFSM` and the one-minute microwave `LLFSMs`. When following the original semantics (those where a snapshot of the external variables is taken and saved before and after each ringlet execution) the Kripke structure for the microwave contains 369 260 nodes. By using a single snapshot semantics where a snapshot is taken per schedule cycle, the Kripke structure for the microwave contains 60 674 nodes. A massive reduction. The Kripke structures can be located at https://github.com/mipalgu/VersatileKripkeStructures.

IX. Conclusion

In software engineering, there is a prevalence for modelling using UML state charts (which is a derivation of Harel's State Charts [35]) and which are event-driven. Moreover, Sommerville [27], states that "state models are often used to describe real-time systems" [27, p. 544], citing UML. We note that Sommerville also uses a microwave to illustrate how `FSMs` model the behaviour of systems [27, p. 136]. Because of these associations among systems that respond to stimuli, it is important to clarify the terminology regarding what constitutes an event-driven system, a reactive system and more importantly, a real-time system. Reactive-systems are responsive systems without much processing, as opposed to deliberative systems (which reason, plan, learn) [36].

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). Event-driven programming is also illustrative of this mechanism that follows the inversion of control (IoC) design principle; call-backs are custom code only

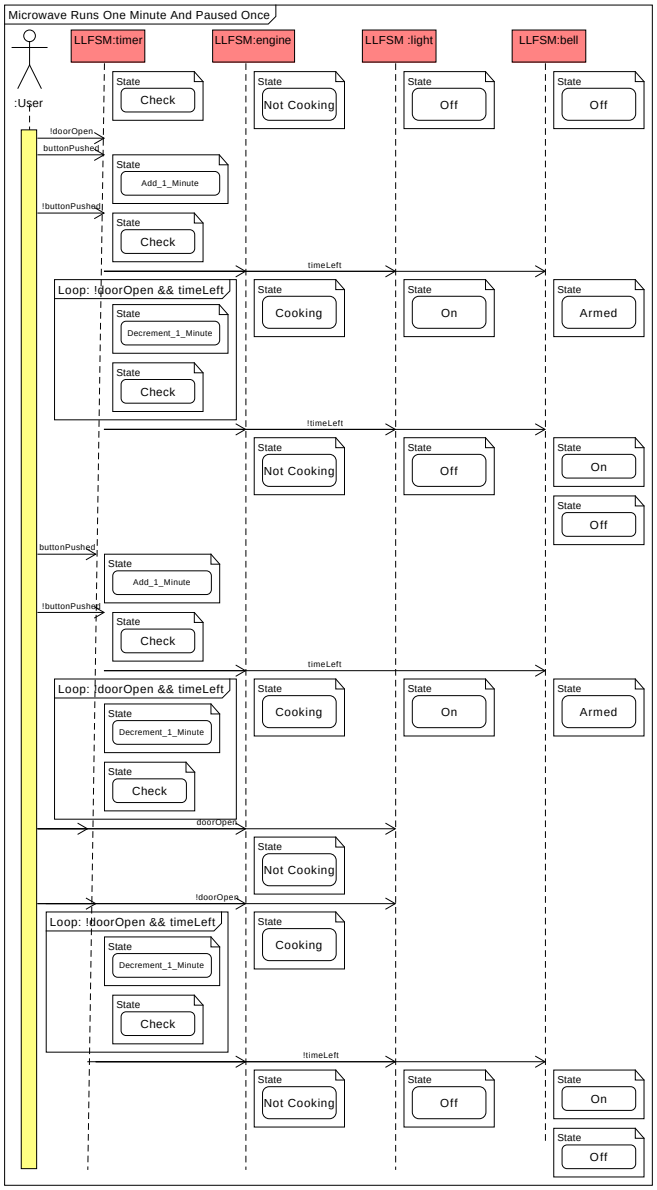


Figure 12. States transitions in a sequence diagram for the use case of running the microwave for one minute and a pause in another minute.

concerned with the handling of events, whereas the event loop and dispatch of events/messages is handled by the framework or the runtime environment. This application of the "Hollywood Principle: Don't call us, we'll call you", while very productive in several contexts (we already mention GUI applications), has serious limitations for real-time applications. We insist that fundamental (mathematically supported assertions) have long being established [17] regarding the limitations of event-driven systems. The counterpart to event-driven systems are time-triggered systems.

Real-time systems are required to meet time-deadlines in response to stimuli [37]. Therefore, although closely related, these terms are not the same, and in this paper, we argue (supported by the work of Lamport [17]) that there are many solid reasons why real-time systems may be better served by time-triggered systems and predetermined schedules, rather than the unbounded delays that may occur in event-driven systems.

There is nothing wrong with using a loosely defined semantics in visual and textual notations (in particular the event-driven state charts of UML) when communicating the main ideas of software designs to stakeholders [38]. However, advocates of model-driven software development (MDSD) argue that software developers shall mostly work with models and that UML is the programming language [39]. However, in this scenario the UML has shortcomings. For example, the relevance of MDSD to the synthesis of behaviour intended for embedded systems has been strongly emphasised [40]; particularly stressing that UML state charts are the prevalent diagrammatic tool for behaviour. But, for instance, modelling behaviour for FPGAs has consistently avoided the event-driven approach of UML state charts: Wood et al. [40] reviewed earlier attempts to generate VHDL from UML and found that either 1) translation was incomplete (aimed at simulation and not hardware synthesis), 2) covered a tiny subset of UML's state chart notation or 3) were far from following closely an MDSD methodology. Wood et al. [40] attempt to directly define a model transformation semantics (from the syntactic construct of UML state charts to the VHDL code) also ran into issues; in particular, the asynchronous nature of UML was replaced by synchronous specification in VHDL [40, Page 1362] (other semantic changes are also listed [40, Page 1364], as well as a table of unsupported features [40, Table 11]). Wood et al. [40] removed the run-until-completion semantics of UML requiring designs where all the consequences of an event must terminate by the next clock click [40]. They eliminated events from labelling transitions, and now a Boolean expression of the form `event_has_happened` (that pool the state of an input signal) replaces each instance of an event. Thus, their UML models allow only guards that monitor signals (they enforce a system of syntactic priorities in case several signals become true in nesting state charts, but for each model, transitions must cover all cases and be mutually exclusive). Thus, we have another scenario where `LLFSMs` are being used with precise but particular semantics.

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. 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 redefining while retaining precision of the semantics of executable models [6], [7], Our framework allows 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.

References

[1] C. McColl, V. Estivill-Castro, and R. Hexel, "An oo and functional framework for versatile semantics of logic-labelled finite state machines," in The Twelfth International Conference on Software Engineering Advances, ICSEA 17, J. Lavazza, R. Oberhauser, R. Koci, and S. Clyde, Eds. IARIA, October 8th-12th 2017, pp. 238–243.

[2] 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.

[3] V. Estivill-Castro, R. Hexel, and C. Lusty, "High performance relaying of C++11 objects across processes and logic-labeled finite-state machines," in Simulation, Modeling, and Programming for Autonomous Robots: 4th International Conference, SIMPAR 2014. Springer International Publishing, October 20th-23rd 2014, pp. 182–194.

[4] C. McColl, "swiftfsm - A Finite State Machines Scheduler," Honours Thesis, Griffith University, 170 Kessels Rd, Nathan QLD, 4111, Australia, 10 2016.

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

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

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

[8] D. Drusinsky, Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking. Newnes, 2006.

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

[10] 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 Publishing, May 19th-22nd 2002, pp. 697–701.

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

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

[13] 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.

[14] B. Selic and S. Grard, Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[15] Recommendation ITU-T Z.100: Specification and Description Language – Overview of SDL-2010, International Telecommunication Union, April 2016. [Online]. Available: https://www.itu.int/rec/T-REC-Z.100-201604-I/en

[16] Specification and description language - real time. [Online]. Available: http://www.sdl-rt.org/

[17] 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.

[18] Recommendation ITU-T Z.101: Specification and Description Language – Basic SDL–2010, International Telecommunication Union, April 2016. [Online]. Available: https://www.itu.int/rec/T-REC-Z.101-201604-I/en

[19] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, Modeling Software with Finite State Machines: A Practical Approach. 6000 Broken Sound Parkway NW, Boca Raton, FL 33487-2742: CRC Press Taylor & Francis Group, 2006.

[20] V. Estivill-Castro and R. Hexel, "Verifiable parameterised behaviour models for robotic and embedded systems," in International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, vol. 1. SCITEPRESS Science and Technology Publications, January 22nd-24th 2018, pp. 364–371.

[21] 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.

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

[23] 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.

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

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

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

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

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

[29] 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.

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

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

[32] L. Wen and R. G. Dromey, "From requirements change to design change: A formal path," in 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004). Beijing, China: IEEE Computer Society, 28-30 September 2004, pp. 104–113.

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

[34] T. Myers and R. G. Dromey, "From requirements to embedded software - formalising the key steps," in 20th Australian Software Engineering Conference (ASWEC). Gold Cost, Australia: IEEE Computer Society, 14-17 April 2009, pp. 23–33.

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

[36] R. C. Arkin, Behavior-Based Robotics. Cambridge, Mass.: MIT Press, 1998.

[37] H. Kopetz, Real-Time Systems - Design Principles for Distributed Embedded Applications, 2nd ed., ser. Real-Time Systems Series. Berlin: Springer, 2011.

[38] W. J. Dzidek, E. Arisholm, and L. C. Briand, "A realistic empirical evaluation of the costs and benefits of UML in software maintenance," IEEE Trans. Softw. Eng., vol. 34, no. 3, May 2008, pp. 407–432.

[39] B. Selic, "The pragmatics of model-driven development," IEEE Software, vol. 20, no. 5, Sept 2003, pp. 19–25.

[40] S. K. Wood, D. H. Akehurst, O. Uzenkov, W. G. J. Howells, and K. D. McDonald-Maier, "A model-driven development approach to mapping UML state diagrams to synthesizable VHDL," IEEE Transactions on Computers, vol. 57, no. 10, Oct 2008, pp. 1357–1371.