Automatic Verification of High-Level Executable Models Running on FPGAs

Morgan McColl(✉), Callum McColl, and René Hexel

Griffith University, Brisbane, Australia
morgan.mccoll@griffithuni.edu.au, {c.mccoll,r.hexel}@griffith.edu.au

Abstract. The increasing complexity of Field-Programmable Gate Array (FPGA) applications necessitates high-level design and formal verification. Traditional approaches often fall short, prompting a shift towards Model-Driven Development (MDD) strategies utilising executable models. Executable models simplify the design process by directly translating high-level, human-readable models into executable code, eliminating manual transcoding errors. However, the challenge of verifying these models in an automated manner remains largely unsolved. The contribution of this paper is a model-driven software engineering methodology utilising logic-labelled finite-state machines (LLFSMs) that enable the automated generation of executable FPGA code from high-level, human-readable models as well as associated Kripke structures for the verification (through model-checking) of high-level executable models running on FPGA platforms. We present a method that utilises the semantics of logic-labelled finite state machines on an FPGA to significantly reduce the size of the created Kripke structures compared with existing LLFSM approaches.

Keywords: Automatic Verification · Model-Driven Software Engineering · Logic-Labelled Finite State Machines · FPGAs

1 Introduction

The evolution of technology has brought about increasing reliance on Field-Programmable Gate Arrays (FPGAs) for their versatility, speed, and real-time performance capabilities. These unique properties have made them instrumental in the design and development of real-time systems, telecommunication networks, computer vision, and embedded systems. However, as FPGA applications grow more complex, their design and verification become increasingly challenging. This necessitates a shift from traditional design methods towards Model-Driven Development (MDD) approaches, where high-level, human-readable models are employed to design, verify, and implement these systems.

Model-Driven Development provides an abstraction layer that enables designers to focus on the functionality of the system while abstracting away lower-level hardware details. This high-level approach simplifies the development process

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
É. André and J. Sun (Eds.): ATVA 2023, LNCS 14216, pp. 217–234, 2023.
https://doi.org/10.1007/978-3-031-45332-8_11

and reduces the likelihood of errors that arise from manual translation into code. However, the verification of these high-level models in an automated manner is a challenge that remains to be addressed.

The primary contribution of this paper is an MDD design that allows the automated generation of Kripke structures that allow verification of high-level executable models running on FPGA platforms. We discuss the issues and limitations of existing methods and how to overcome them to handle the complexity of models that utilise FPGA designs while maintaining the advantages of high-level model abstraction.

The rest of this paper is structured as follows. Section 2 explores the background of verifiable MDD utilising high-level, human-readable models. This is followed in Sect. 3 by our detailed design of executable models of logic-labelled finite-state machines running on an FPGA. We then demonstrate how we can create Kripke structures for automated model-checking (Sect. 4). Finally, we summarise our findings in Sect. 5.

2 Background

The Unified Modelling Language (UML) has unified the various disparate notational systems for software design and modelling and has become the de-facto standard for model-driven software engineering (MDSE). There has been remarkable progress in MDSE [1–3] in creating verifiable executable models that define behaviour at a high level. However, the automatic verification of deployable systems remains a challenge. By far, the biggest challenge with common modelling systems are semantic variations. In other words, while the UML has unified notation, formal correctness of resulting execution semantics only holds in some scenarios [4], even in current versions of executable UML (fUML) [5]. Moreover, concurrency and parallelism are severely hampered by the fact that execution paths may diverge due to race conditions or other sensitivities towards instruction execution order. This is the case, even if the same sources and systems are utilised in building and executing the model, as ambiguities remain when it comes to the semantics of these systems [6]. In essence, this violates the promise that executable models allow us to create a fully automated one-to-one mapping between design and implementation so that we can perform formal verification on the real software that executes on the target system. These ambiguities in UML semantics have repeatedly been identified as causing confusion in the industry and leading to inconsistencies and suboptimal results [7,8]. This is not helped by the fact that almost from the beginning, inconsistencies have been created in different UML implementations with different semantics [9] that frustrate the reproducibility of model checking.

It is, therefore, unsurprising that the predominant focus has been on testing approaches, such as test-driven development [10] and continuous integration [11]. However, we argue that, while pragmatic, this is a flawed approach, as testing is, by definition, incomplete. In particular, testing can only prove the existence of defects, not their absence. We will therefore focus on formal methods for

verifying the correctness of executable models. We have already demonstrated earlier [3] that we can achieve consistent and faithful execution semantics of executable models of real-time systems that allow verification in both the time and value domains. Without going into detail here, suffice it to say that while event-driven modelling has been the predominant paradigm that has been productive for traditional systems such as desktop computers or cloud architectures, it has long been recognised that such systems fundamentally follow a best-effort approach that cannot guarantee bounded execution and completion times sufficient to meet hard deadlines [12]. By contrast, time-triggered systems guarantee consistency in both the value and time domains, making formal verification feasible for a much larger class of systems [13].

A key challenge that remains is the creation and verification of graphs used for model-checking. While some systems create a combinatorial state explosion when deriving their Kripke structures, making them too complex to formally verify as-is, we have already shown that we can utilise dependency analysis to isolate modules and verify them independently without jeopardising the verifiability of the composite system as a whole [14]. Moreover, we have shown that we can utilise the same MDSE approach that has traditionally been used on computer systems, but achieve orders of magnitude higher execution speeds when utilising FPGAs [15]. This is essential as embedded architectures often exhibit semantic differences between the original UML design and the compiled software executing on the embedded device [4]. This phenomenon has been especially pronounced when examining UML translations into FPGAs [16–18]. Common translation differences in FPGAs include implicit priorities on transitions, distributed event queues and semantic differences between formal event processing and the limitations of the underlying hardware.

While we have been able to overcome these inconsistencies and are now able to create consistent, high-level executable models that run on FPGAs [15], formal verification has, thus far, eluded us. Before we detail the design that allows us to now create Kripke structures of an executable model on an FPGA, we need to recall the principles of logic-labelled finite-state machines that make these executable models possible.

2.1 Logic-Labelled Finite-State Machines

Logic-labelled finite-state machines (LLFSMs) are an approach to modelling that follows the ubiquitous footsteps of modelling with finite-state machines (FSMs). However, in contrast to the event-driven nature of UML statecharts, LLFSMs utilise expressions in a decidable logic to decorate transitions [19]. In other words, they follow the principles of UML state diagrams where only guards, but not events are used, effectively capturing the fact that an event has occurred into a Boolean expression that reflects this. This avoids the impossibility of translating mathematically perfect events, i.e. a semantics of zero duration with perfect order, into a feasible implementation that has to work with event queues and other approximations. With the departure from the notion that reactiveness needs to be driven by events, we can achieve real-time object-oriented modelling [20].

LLFSMs utilise the notion that execution is subdivided into atomic ringlets. Each ringlet execution is an activity that comprises a sequence of actions:

1. A snapshot of external input variables is taken.
2. Where a state ringlet is executed for the first time (e.g. a different state was executed previously), an *OnEntry* action is run.
3. All transitions are evaluated in a pre-defined order of priority.
4. If a transition fires, an *OnExit* action gets run.
5. Otherwise, an *Internal* action gets executed.
6. A snapshot of resulting output variables is written.

We will now detail our design that, for the first time, enables us to run an executable LLFSM model on an FPGA while creating the corresponding Kripke structure on-device to enable formal verification.

3 Design

We begin this section by presenting our design for creating executable models in FPGAs. We then discuss the methods used to reduce the Kripke structure by using the semantics of our models. We have previously used LLFSMs on FPGAs by segregating each ringlet into a periodic process that maps to clock edges [15], as shown in Fig. 1. This mapping creates a synchronous execution between parallel FSMs that allows for the separation of *read* and *write* actions between shared variables. This structure is fundamental to LLFSMs on FPGAs, allowing more complex forms of behaviour by composing and ordering different ringlets. Importantly, this overcomes any race conditions that might otherwise occur between finite-state machines running in parallel and facilitates atomic execution. In other words, the ringlet process is never executed partially but instead presents an indivisible structure that always completes once started [15].

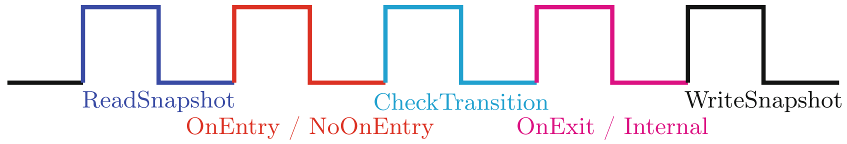

Fig. 1. The Ringlet Execution Cycle [15]

Our FPGA models are executable in the complete sense, as we define state actions and variables in VHDL, a hardware description language used to describe hardware constructs in the FPGA. We perform code generation to translate the graphical depiction of our LLFSM into the executable code that the FPGA enacts. This automatic translation removes semantic differences between design and implementation as the translation software moves the user's code within the machine into the correct sections within a VHDL template. We have constructed

our LLFSM template to enforce the formal semantics of LLFSMs without incurring semantic differences or gaps.

Our main contribution here is that we can now generate corresponding VHDL that performs automatic and optimised Kripke structure construction for our executable models on-device. Generating Kripke structures in this way creates a graph directly derived from the behaviour of the software on the target hardware as it executes. This graph is in a format that can be used to run automated proofs using a model checker such as nuXmv [21]. The Kripke structure reflects precisely what is executed on the FPGA, ensuring there is no semantic gap between the high-level model and the executable. This process is well aligned with formally verifying safety-critical and dependable systems, as formal verification takes place on structures derived directly from the physical hardware.

Since we will be examining the Kripke structure of our LLFSMs, it is important to be aware of some key variables used in the VHDL template to enforce the LLFSM semantics. We will use three variables in our Kripke structures that are generated in the VHDL template, namely *internalState*, *previousRinglet* and *targetState*. The *internalState* variable is used to track which step in the ringlet the LLFSM is currently in (see Sect. 2.1) and reflects the values within the timing diagram of Fig. 1. The *previousRinglet* variable captures the state the machine executed immediately before the current ringlet the machine is executing. This variable allows us to determine whether or not *OnEntry* should be executed in the current ringlet. The final variable, *targetState*, is used to determine if the LLFSM is about to transition to a new state. This variable then contains the state the LLFSM will transition to in the next ringlet.

Another key contribution is the reduction of the Kripke structure size by taking advantage of the LLFSM semantics. We can leverage a ringlet's execution cycle (see Fig. 1) to minimise the Kripke structure size while supporting parallel execution. Let us examine how these Kripke structure optimisations work by examining a simple LLFSM. Consider the LLFSM depicted in Fig. 2 containing the variables within Table 1. This LLFSM starts in an empty initial state that performs no functions and then transitions to state S_0. Within state S_0, the machine sets the value of external variable y to the value of external variable x. The external variables x and y are examples of variables that map to inputs

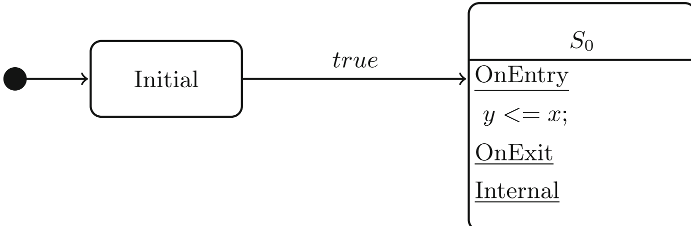

Fig. 2. LLFSM I, depicting an initial state and a state S_0 executing a simple VHDL value assignment.

Table 1. The variables within LLFSM *I*.

Name	Type	Scope	Mode
x	*std_ logic*	*External*	*input*
y	*std_ logic*	*External*	*output*

and outputs respectively, as associated, for example, with sensors and actuators. This variable assignment only occurs once during the first ringlet of the S_0 state (*OnEntry*). All subsequent ringlets will perform no function as the *Internal* action of state S_0 is empty.

We can now derive a partial Kripke structure for the first ringlet of state S_0 (Fig. 3). This Kripke structure assumes that the variable x has the value of *'0'* (logic low) at the beginning of the ringlet. We have also removed the snapshot semantics to demonstrate the state explosion without our optimisations. This is very close to typical UML translations into FPGAs. The Kripke structure in Fig. 3 shows the Kripke states between the *OnEntry* action and the *CheckTransition* phase of state S_0's first ringlet. Notice that several Kripke states are describing the *CheckTransition* phase of this ringlet due to the different values that are possible for the variable x. This Kripke structure represents the actual code that is executed on an FPGA and must account for each value of x (a *std_ logic* variable in VHDL). There are 9 possible values that a *std_ logic* variable can be in, however, these values will generally resolve to a combination of logic low and logic high bit values when executed on the actual hardware. This resolution is dictated by the synthesiser and FPGA fabric [22], so we cannot ignore this for the general case.

Since x is an external variable representing a sensor reading, this value can change at any point in time and must be accounted for in the Kripke structure in these circumstances. For each *CheckTransition* Kripke state in this figure, we will also have the same edges for the next Kripke state (*Internal* in this case) where the same sensor may change its value. The total number of Kripke states for this ringlet is 171 (for each *OnEntry* state, we have 9 *CheckTransition* states and 9 *Internal* states). There are 9 *OnEntry* states in total, therefore the total Kripke structure for the first ringlet is $9 \cdot 19 = 171$. This combinatorial state space represents an example of the state explosion problem for our LLFSM.

3.1 Reducing the Kripke Structure Size Using Snapshot Semantics

We now demonstrate how the Kripke structure is altered by introducing snapshot semantics. Before the start of each ringlet, a snapshot of all input external variables is taken. This snapshot takes the input (sensor) value at the start of the ringlet and writes it into a local copy that the LLFSM acts upon during the ringlet's execution. The LLFSM acts upon local copies of the output external variables during the ringlet's execution as well. At the end of the ringlet, the output local variables are written back to the external variables of the LLFSM.

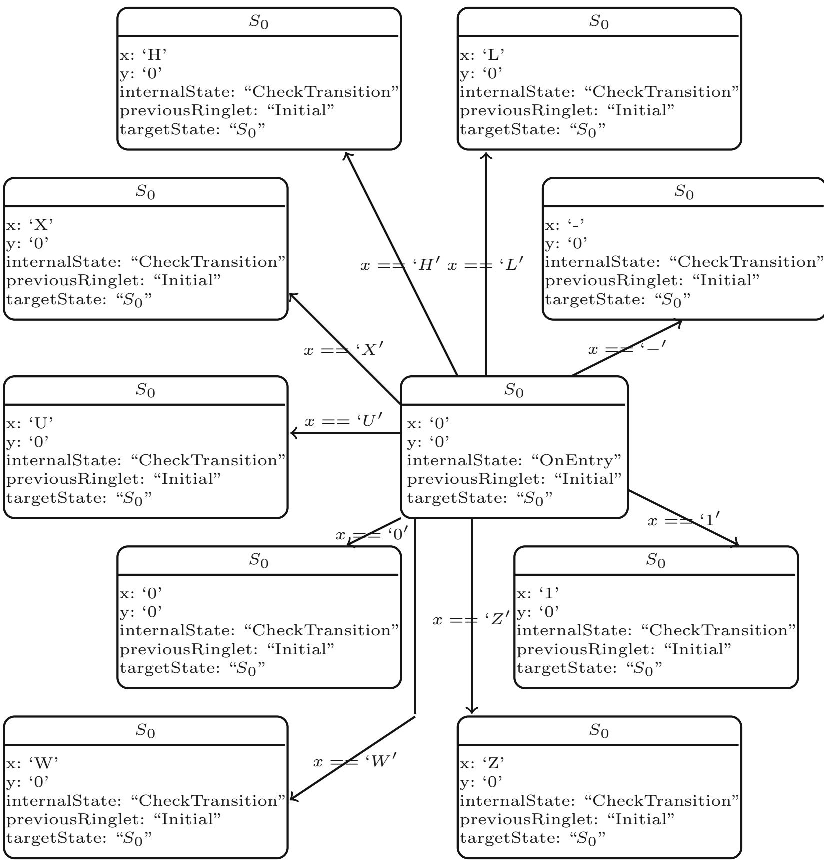

Fig. 3. The Kripke Structure of the First Ringlet in State S_0 without Snapshot Semantics.

These *read* and *write* sections of an LLFSMs execution are depicted as *ReadSnapshot* and *WriteSnapshot* in Fig. 1 respectively.

The advantage of this semantics is that the local snapshot variable representing x in our example is not modified after the *ReadSnapshot* phase of the ringlets execution. The result of this semantics drastically reduces the number of Kripke states in our ringlet. We have updated the Kripke structure for this example in Fig. 4. Please note that x now represents the snapshot variable x while *EXTERNAL_x* represents the corresponding sensor variable x. We again assume that the first sensor reading is $x =$ '0'. For brevity, we have reduced the number of edges in this figure into a set of values. The actual Kripke structure would represent each value in the sets labelling the edges as a separate edge. In the Kripke structure in Fig. 4, we have omitted the external variable *EXTERNAL_x* as the number of states to depict would be too large to show here. The main trend that is observable from this Kripke structure is that the x snapshot

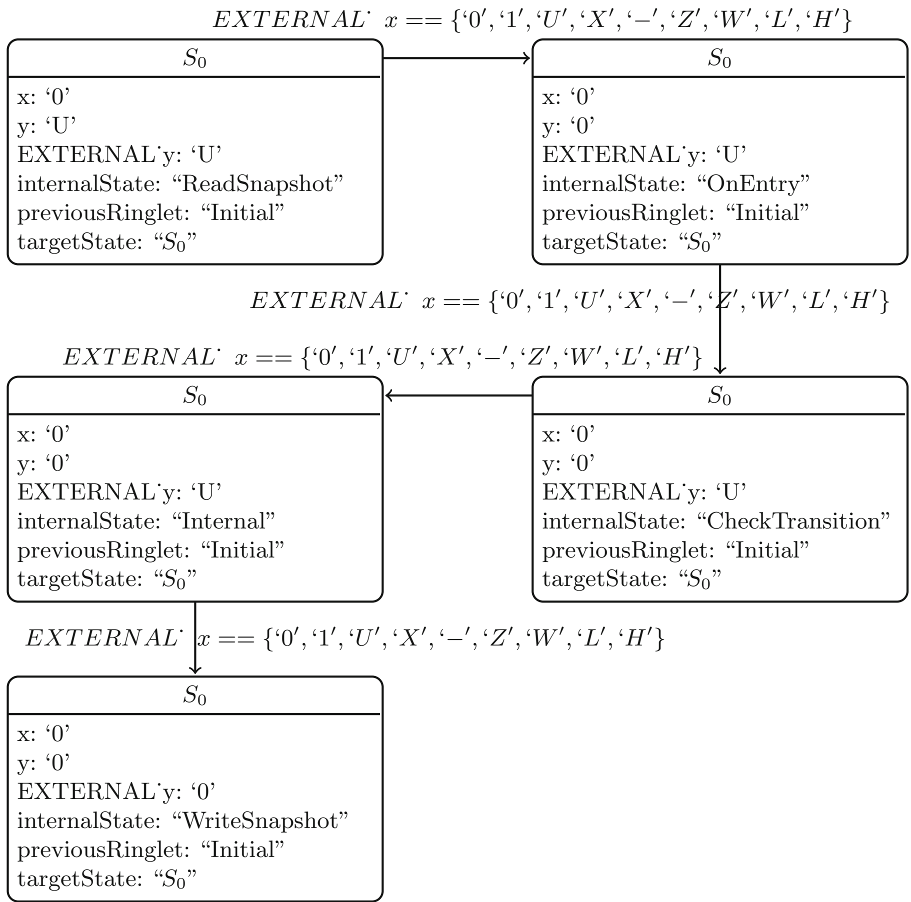

Fig. 4. The Kripke Structure of the First Ringlet in State S_0 with Snapshot Semantics.

variable no longer needs to be mutated during every phase of the ringlet. To illustrate this, assume that *EXTERNAL_x* represents a sensor measuring some property in the environment. Even though the corresponding sensor values may continuously change, we are only interested in the value that is present when we take a snapshot. Therefore, the sensor reading has no further influence over the LLFSM until the next ringlet's *ReadSnapshot* phase. We can also observe that the *EXTERNAL_y* variable is not assigned the value of snapshot variable y until the *WriteSnapshot* Kripke state. We can use these properties to further reduce the complexity and size of the Kripke structure.

Consider the new Kripke structure in Fig. 5. In this Kripke structure, we have two Kripke states designated $S_{0_{Read}}$ and $S_{0_{Write}}$. These states represent the state of the LLFSM immediately before *ReadSnapshot* and immediately after *WriteSnapshot* respectively. Due to the snapshot semantics, our external vari-

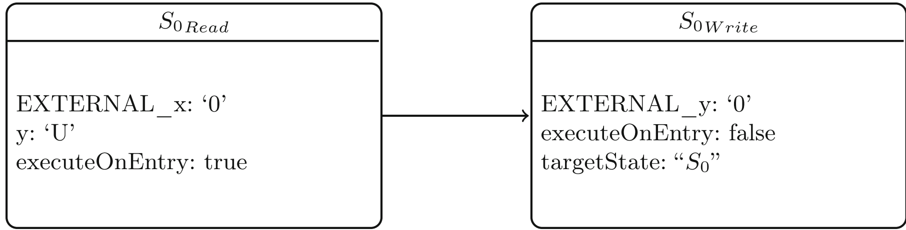

Fig. 5. The Optimized First Ringlet of State S_0 when $EXTERNAL_x$ = '0'.

ables (which map to sensors and actuators) only affect (or are affected by) the execution of our machine during these points in time. Therefore, since these values do not impact the machine's runtime during the ringlet execution, we can treat the ringlet as a black box removing the intermediate states from the Kripke structure entirely.

We can also remove unnecessary variables from the Kripke structure that are used to track and execute the LLFSM semantics. For example, we can remove *previousRinglet* since the only influence this variable has is to decide whether to execute *OnEntry* or not. Since this condition is a boolean, we can replace the *previousRinglet* variable with a simple *executeOnEntry* variable. This optimisation removes the possibility of state explosions from additional state transitions in the LLFSM. We also remove *internalState* since this variable is used to track the ringlets execution. Since we are treating the ringlet as a black box, we do not need to include this variable in the Kripke structure.

In S_{0Read}, we can also remove *targetState* since this variable has no influence over how the ringlet is executed and the x snapshot since it is the same as $EXTERNAL_x$. In S_{0Write}, we can remove the y snapshot since it is equal to $EXTERNAL_y$ and the $EXTERNAL_x$ variable and x snapshot since their values will be modified in the next *Read* state. The x variables in the $Write$ state do not influence the next *Read* state.

The new Kripke structure contains 2 states per value of $EXTERNAL_x$ (*Read* and $Write$). Since there are 9 values of x in this ringlet, we have a total of $9 \cdot 2 = 18$ Kripke states for this ringlet. This number presents an 89% reduction in the size of the Kripke structure for this ringlet.

There is one further optimisation that we have included in the LLFSM semantics. During the *ReadSnapshot* phase of the ringlets execution, we read all input external variables into local copies. We have modified this semantics slightly to only read the input variables that a state is using in one of its ringlets. This further optimisation removes additional external variables from a *Read* Kripke state if those variables were not used in that Kripke states ringlet. This approach allows us to use many different variables in different states without creating a combinatorial state explosion.

4 Automatic Kripke Structure Generation

To evaluate our design, we have implemented several tools to perform Kripke structure generation for LLFSMs on FPGAs. The process of creating LLFSMs for FPGAs and performing a formal verification is as follows.

1. Design the LLFSM in an LLFSM editor.
2. Perform code generation using the built-in compilation options in the editor. The generated code will contain the VHDL code for the LLFSM and the Kripke structure generator.
3. Synthesis, place and route.
4. Generate the bitstream and load onto the FPGA.
5. Use external interfaces such as Ethernet, or UART to retrieve the generated Kripke structure.

Our Kripke structure generator takes advantage of the parallel architecture of the FPGA to generate the correct Kripke structure. To parallelise our generation, we have decomposed the LLFSM into ringlets that need to be executed and evaluated. The generator begins in the initial state of the LLFSM and executes the first ringlet to determine the next reachable state to execute. The generator then follows a specific procedure until it has executed all reachable states. The procedure of the generator is composed of the following steps.

1. Choose the next state to execute. This will include the state to execute, the value of its output external variables, the machine variables, the state variables, and whether the LLFSM needs to execute *OnEntry*.
2. Choose all possible combinations of input external variables that this state uses and execute them on the LLFSM in parallel for each combination. In the example shown in Fig. 2, this would be 1 LLFSM for the *Initial* state and 9 LLFSMs for the S_0 state.
3. Execute the LLFSMs starting in *ReadSnapshot* and wait until they reach *WriteSnapshot*.
4. Stop the LLFSMs and save the $Read$ and $Write$ states, removing all duplicates.
5. Determine the next set of states to execute by observing the $Write$ states and the previous states that were executed.
6. If there are no more states to execute, stop generating, otherwise go to Step 1.

Following this procedure, the LLFSM dictates the states to explore in the Kripke structure generation. Each new Kripke state that is *discovered* is recorded by the generator and used to explore the state-space of the LLFSM further. If the Kripke state is *new*, then the generator will place it onto a queue of states to explore further (i.e. subsequent *Read* states resulting from the new *Write* state). Following this process, the generator will continue to discover new Kripke states until it has explored the entire reachable state space of the LLFSM. Throughout the Kripke structure generation, the generator tracks previously explored Kripke

states to remove duplicate pathways. If a *Read* state has been explored previously, then the Kripke structure generator will not execute that ringlet again. The generator finishes when there are no more Kripke states left to explore.

The accuracy and completeness of the Kripke structure generation is guaranteed since the execution of the LLFSM drives the entire process. The Kripke structure generator is simply *observing* and *recording* the Kripke states as the LLFSM encounters them. When the generator *observes* all reachable Kripke states (i.e. there are no more *Read* states queued), the generation is *finished* and the Kripke structure is complete. Moreover, the structure is completely representative of the behaviour of the LLFSM as the structure is directly generated from the LLFSM execution.

The Kripke structure generator is generated for each machine from the states and variables contained within the LLFSM. The entire Kripke Structure generator exists within a small number of VHDL source files, including the generated code for the LLFSMs. These files may be included in existing HDL projects to facilitate interoperability between other components within the project. No additional tooling is required to facilitate the new VHDL source files in these projects.

4.1 Furnace Relay Case Study

We now demonstrate our approach with an example taken from the literature. Consider the UML state machine shown in Fig. 6. This FSM is taken directly from McUmber and Cheng's paper [17]. McUmber and Cheng's FSM is converted to VHDL using their code generator. Due to the embedded hardware in FPGAs, this translation is not exact and incurs semantic differences between model definition and execution. The authors use a process block to represent each state within the *FurnaceRelay* FSM. This process block does not contain any signals in its sensitivity list, nor does it incorporate any clock sources. This code is not synthesisable as signals within this FSM are latched with flip-flops during the rising edge of a clock signal. The authors of this paper have ignored this and tried to create a purely event-triggered semantics consistent with UML by setting event signals high within extremely small durations (1 fs in this example) before exiting. This behaviour tries to emulate the instantaneous nature of

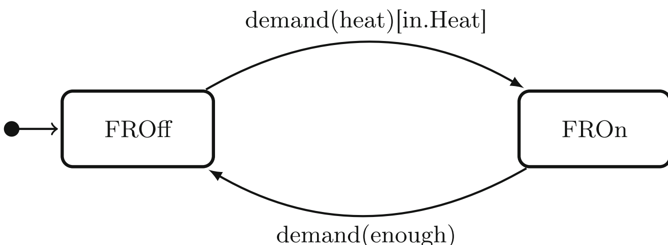

Fig. 6. The FurnaceRelay UML State Machine

formal events but incurs a small, but finite amount of time as a result of executing on real hardware. Physics dictates that instantaneous events are not possible within digital circuits, as it takes time to latch and propagate signals within the FPGAs fabric. The delays are considerable (when compared to the 1 fs specification) using the limited clock frequencies of FPGAs. More importantly, they can create an inconsistent behaviour due to the partial ordering of events [23,24] creating divergence points in the UML semantics.

4.2 Utilising LLFSM Semantics

Even though the authors reference event-driven UML semantics [17], the actual semantics of their FSM is more akin to that of an LLFSM, but hiding the crucial fact that a latched logic value is what is being evaluated. An LLFSM polls its variables and assigns new values at specific points in time [19]. This process translates well into FPGAs as they latch signals based on clock edges [15]. We may thus translate LLFSM models into FPGAs without incurring semantic differences [15]. Therefore, to formally verify the FSM in Fig. 6, it is preferable to perform a mapping into an LLFSM first.

To perform an accurate formal verification, we first convert the UML FSM into an LLFSM. We have attempted to match the UML FSM with variables of the same size and type. However, some of the code is missing from the publication and conservative assumptions were made to create a synthesisable LLFSM. Specifically, the *demand* variable needed to support at least 4 different values, so we have chosen a 2-bit *std_logic_vector* instead of the enumeration in the original publication. We have also replaced the *instate* signal that was indicating the state of the relay with a *std_logic* signal called *relayOn*. The *instate* signal definition was not shown in the original publication. The resulting LLFSM is depicted in Fig. 7 with the variables in Table 2. The resulting VHDL code for this LLFSM and its Kripke structure generator is available on *GitHub* [25] and consists of 729 *FurnaceRelay* LLFSMs executing in parallel. The Kripke structure size of this LLFSM is 4543 Kripke states and it took 740 ns to generate the entire graph on the FPGA at 125 MHz. The Kripke structure size of the corresponding UML FSM is estimated to be greater than 4,782,969 Kripke states. The LLFSM-equivalent Kripke structure is thus reduced by more than 99.9%. To highlight the Kripke structure generation, we have provided screenshots of simulation results for some of these ringlets.

Table 2. The variables within the *FurnaceRelay* LLFSM.

Name	Type	Scope	Mode
heat	*std_logic*	*External*	*input*
demand	*std_logic_vector(1 downto 0)*	*External*	*input*
relayOn	*std_logic*	*External*	*output*

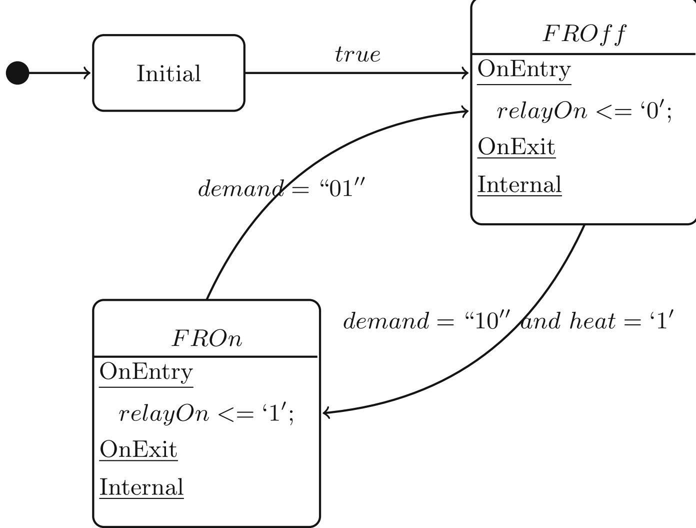

Fig. 7. The FurnaceRelay LLFSM.

We begin in the *Initial* state by executing the initial ringlet of the LLFSM (Figs. 8 and 9).

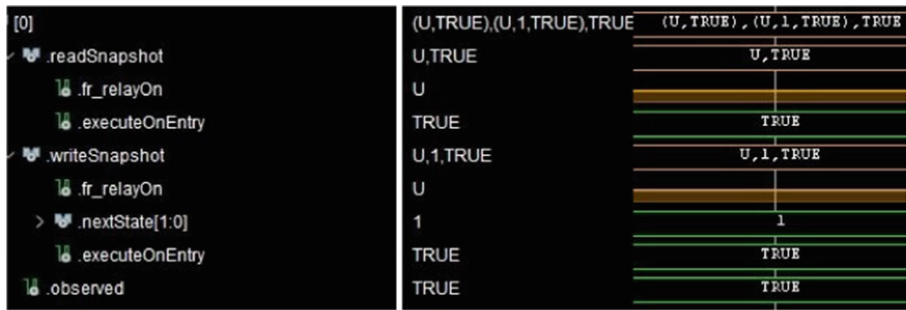

Fig. 8. The *Initial* Ringlet Waveforms.

This ringlet transitions the LLFSM into the *FROff* state. Once in that state, the LLFSM will either transition to *FROn* if *demand* is "10" and *heat* is '1' (Figs. 10 and 11) or otherwise remain in *FROff* (Figs. 12 and 13). Please note that the figures for the *FROff* ringlets represent the ringlet where the previous state was *FROn*. This is why *relayOn* is logic high during the *Read* phase of this Kripke structure. When the LLFSM is in *FROn*, it will either transition back to *FROff* when *demand* is "01" (Figs. 14 and 15) or otherwise remain in *FROn* (Figs. 16 and 17).

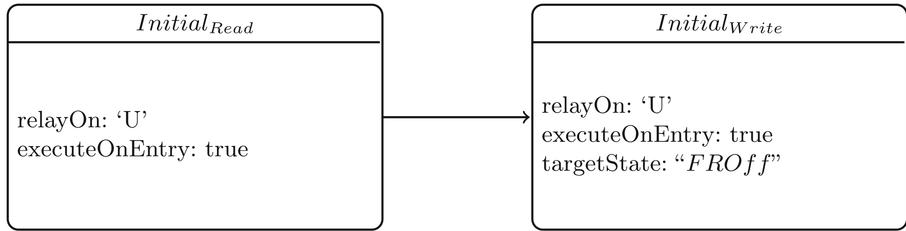

Fig. 9. The *Initial* Ringlet.

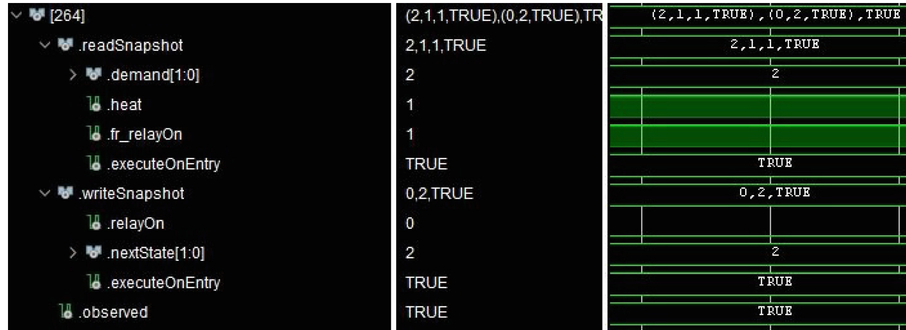

Fig. 10. Waveforms of an *FROff* Ringlet that Transitions to *FROn*.

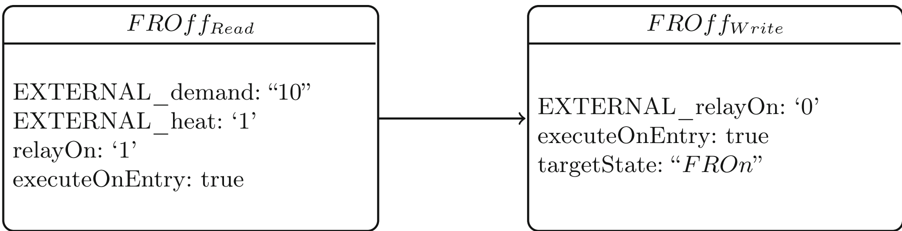

Fig. 11. An *FROff* Ringlet that Transitions to *FROn*.

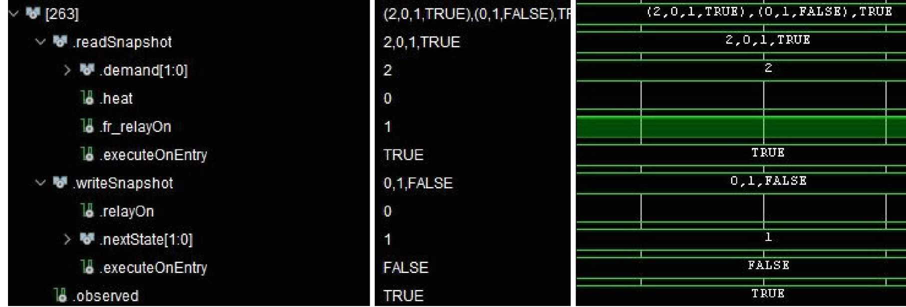

Fig. 12. Waveforms of an *FROff* Ringlet that doesn't Transition.

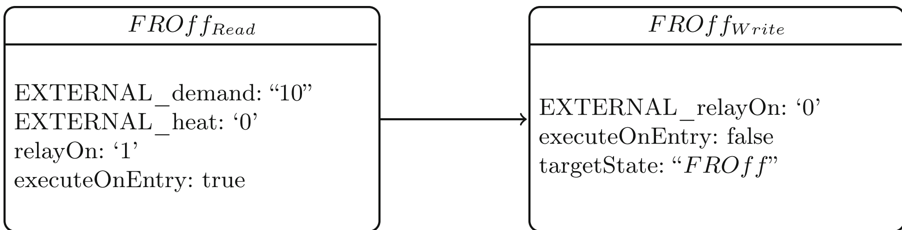

Fig. 13. An *FROff* Ringlet that doesn’t Transition.

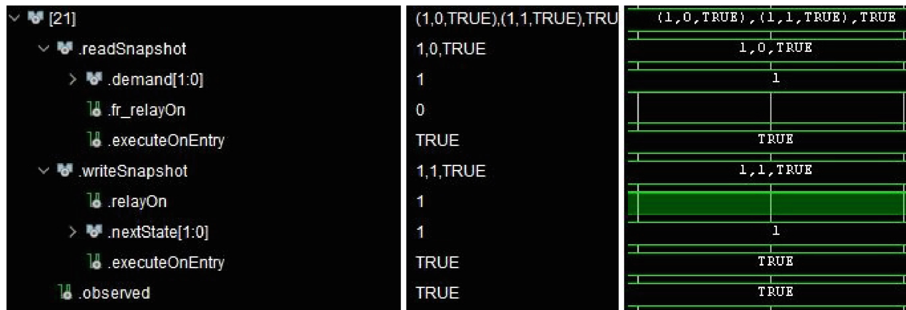

Fig. 14. Waveforms of an *FROn* Ringlet that Transitions to *FROff*.

The previous description provided an overview of the behaviour of our LLFSM by examining the ringlets executed in the Kripke structure. Covering these cases, the Kripke structure generator executes these ringlets until all reachable ringlets have been executed. Once this has been achieved, the generator stops and the Kripke structure is complete.

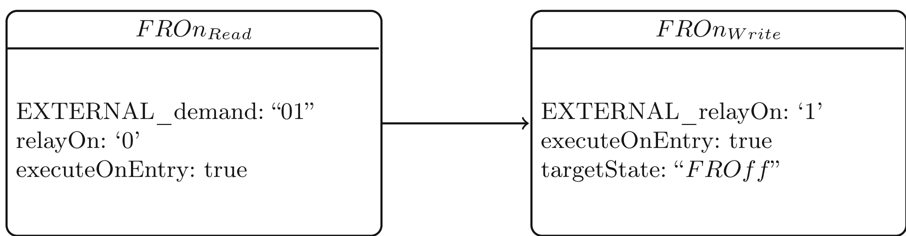

Fig. 15. An *FROn* Ringlet that Transitions to *FROff*.

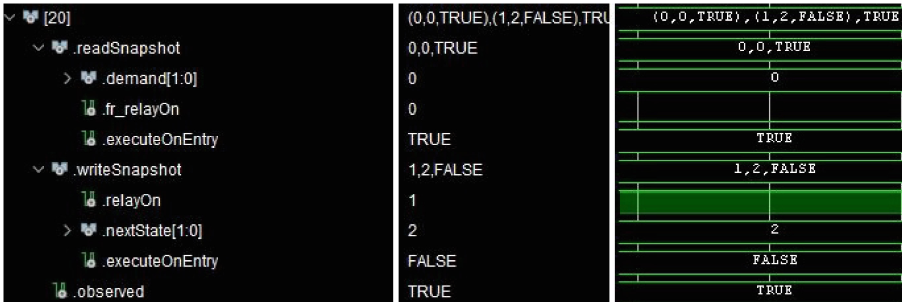

Fig. 16. Waveforms of an *FROn* Ringlet that doesn't Transition.

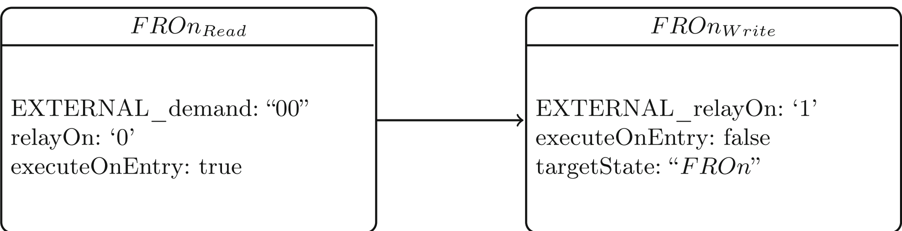

Fig. 17. An *FROn* Ringlet that doesn't Transition.

5 Conclusion

In this paper, we have demonstrated the feasibility of automatic verification of high-level executable models running on FPGAs. We have shown how we can model system behaviour using logic-labelled finite-state machines. Importantly, we can utilise the parallelism that FPGAs provide without jeopardising the atomicity of executing ringlets, implementing snapshot behaviour utilising deterministic, clock-synchronised execution of states across the FPGA fabric.

We have further demonstrated how we can create Kripke structures in an automated fashion that can then be utilised using standard model-checking tools. Moreover, we can optimise these Kripke structures to significantly reduce the state-space needed for system verification. Finally, we can create these Kripke structures directly on the FPGA, ensuring not only consistency in semantics, but also dramatically increasing the speed with which we can create these Kripke structures in hardware when compared to software running on a microprocessor.

References

1. Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Softw. Syst. Model. **19**(1), 5–13 (2020). https://doi.org/10.1007/s10270-019-00773-6
2. Bucchiarone, A., et al.: What is the future of modeling? IEEE Softw. **38**(02), 119–127 (2021)

3. McColl, C., Estivill-Castro, V., McColl, M., Hexel, R.: Verifiable executable models for decomposable real-time systems. In: MODELSWARD, pp. 182–193 (2022)
4. Besnard, V., Brun, M., Jouault, F., Teodorov, C., Dhaussy, P.: Unified LTL verification and embedded execution of UML models. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, pp. 112–122. Association for Computing Machinery, New York (2018)
5. Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., Gérard, S.: Executable modeling with fUML and Alf in Papyrus: tooling and experiments. In: Mayerhofer, T., Langer, P., Seidewitz, E., Gray, J. (eds.) Proceedings of the 1st International Workshop on Executable Modeling Co-Located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Volume 1560 of CEUR Workshop Proceedings, pp. 3–8. CEUR-WS.org (2015)
6. Pham, V.C., Radermacher, A., Gérard, S., Li, S.: Complete code generation from UML state machine. In: Ferreira Pires, L., Hammoudi, S., Selic, B. (eds.) Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, 19–21 February 2017, pp. 208–219. SciTePress (2017)
7. Iqbal, M., Ali, S., Yue, T., Briand, L.: Applying UML/MARTE on industrial projects: challenges, experiences, and guidelines. Softw. Syst. Model. **14**(4), 1367–1385 (2015). https://doi.org/10.1007/s10270-014-0405-5
8. Petre, M.: UML in practice. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 722–731 (2013)
9. Beeck, M.: A comparison of Statecharts variants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994. LNCS, vol. 863, pp. 128–148. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58468-4_163
10. Mäkinen, S., Münch, J.: Effects of test-driven development: a comparative analysis of empirical studies. In: Winkler, D., Biffl, S., Bergsmann, J. (eds.) SWQD 2014. LNBIP, vol. 166, pp. 155–169. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-03602-1_10
11. Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pp. 426–437. Association for Computing Machinery, New York (2016)
12. Lamport, L.: Using time instead of timeout for fault-tolerant distributed systems. ACM Trans. Program. Lang. Syst. **6**, 254–280 (1984)
13. Furrer, F.J.: Future-Proof Software-Systems: A Sustainable Evolution Strategy. Springer, Berlin (2019). https://doi.org/10.1007/978-3-658-19938-8
14. Estivill-Castro, V., Hexel, R.: Module isolation for efficient model checking and its application to FMEA in model-driven engineering. In: Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 218–225 (2013)
15. Estivill-Castro, V., Hexel, R., McColl, M.: High-level executable models of reactive real-time systems with logic-labelled finite-state machines and FPGAs. In: 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–8 (2018)
16. Wood, S.K., Akehurst, D.H., Uzenkov, O., Howells, W.G.J., McDonald-Maier, K.D.: A model-driven development approach to mapping UML state diagrams to synthesizable VHDL. IEEE Trans. Comput. **57**(10), 1357–1371 (2008)

17. McUmber, W.E., Cheng, B.H.C.: UML-based analysis of embedded systems using a mapping to VHDL. In: Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering, pp. 56–63 (1999)
18. Labiak, G., Borowik, G.: Statechart-based controllers synthesis in FPGA structures with embedded array blocks. Int. J. Electron. Telecommun. **56**(1), 13–24 (2010)
19. Estivill-Castro, V., Hexel, R.: Arrangements of finite-state machines - semantics, simulation, and model checking. In: Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, MODELSWARD, vol. 1, pp. 182–189. INSTICC, SciTePress (2013)
20. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley, Hoboken (1994)
21. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
22. Vivado Design Suite User Guide: Synthesis (UG901)
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system, pp. 179–196. Association for Computing Machinery, New York (2019)
24. Kopetz, H.: Sparse time versus dense time in distributed real-time systems. In: 1992 Proceedings of the 12th International Conference on Distributed Computing Systems, pp. 460–467 (1992)
25. McColl, M.: FurnaceRelay LLFSM, version 1.0.0 (2023). https://github.com/Morgan2010/FurnaceRelay