High-Level Executable Models of Reactive Real-Time Systems with Logic-Labelled Finite-State Machines and FPGAs

Vladimir Estivill-Castro
School of ICT, Griffith University
Brisbane, Australia
v.estivill-castro@griffith.edu.au

René Hexel
School of ICT, Griffith University
Brisbane, Australia
r.hexel-castro@griffith.edu.au

Morgan McColl
School of ICT, Griffith University
Brisbane, Australia
morgan.mccoll@griffithuni.edu.au

***Abstract*—Modelling complex, real-time systems at a high level of abstraction is becoming increasingly important with the prevalence of embedded systems. While the Unified Modelling Language (UML) has been at the forefront of the promise of model-driven development, difficulties with their semantics have prevented FPGA implementations of exact UML executable models thus far. In this paper, we propose to adapt logic-labelled finite-state machines with a well-defined, time-triggered semantics suitable for FPGA implementation. We demonstrate the viability of our approach with an implementation of communicating, cellular automata with strict timing requirements.**

***Index Terms*—Logic-labelled finite-state machines, reactive systems, embedded real-time systems, executable models, FPGA**

I. Introduction

With the emergence of digital technology, massive amounts of embedded systems are deployed today in increasingly ubiquitous scenarios [1]. The embedded system industry has created many trends, such as the Internet of Things (IoT) [2]. A 2017 report [3] estimates 22.5 billion IoT devices by 2021, and measured the count as 15.9 billion installed in 2016. While Gartner's tally for 2016 is 3.9 billion, it is clear embedded systems are pervasive and ubiquitous. Correctness becomes crucial as their sophistication and complexity increases, creating a pressing need for rigorous modelling and validation. Importantly, modelling leverages abstraction and enables planning, focusing on fundamental elements rather than less relevant details. Model-driven development (MDD), promises automated translation from models to implementation that is free of human error. Furthermore, modelling communicates designs in human understandable and interpretable forms, allowing modelling artefacts to be adopted and standardised. In the field of embedded systems, the notions of *reactive system* and *real-time system* are of the utmost importance (although we will make a distinction between these two notions). The most prevalent modelling instrument for behaviour is typically some form of finite-state machine [4]. With the penetration and standardisation of the Unified Modelling Language (UML), state-diagrams in UML have received significant interest as executable models [5]. However, the predominant view of UML's state diagrams is *event-driven*[1].

How is one to unify and combine the three notions, reactive system, real-time system, and executable model? What should a suitable high-level modelling tool look like, that enables high parallelism, formal verification, and readily automatic translation into hardware? This paper provides a clear pathway to an alternative to the UML's event-driven state diagrams, using logic-labelled finite-state machines (LLFSMs). We show here that while LLFSMs have, thus far, only been considered for software systems, we can, for the first time, introduce and demonstrate their mapping into embedded hardware systems. In particular, we detail a full case study of their implementation in FPGAs.

We will start off by detailing why the notions of reactive system, real-time system, and event-driven are related but not strictly the same. We then look at why designers and engineers find state machines suitable models for these systems, while pointing out the assumptions and natural omissions that the abstractions, brought by the modelling artefacts, generate. We will discuss previous work, particularly around the causality and timing implications, and how previous attempts to use UML state-diagrams for modelling behaviour of embedded systems have been forced to move away from the original, event-driven nature of UML. Also, we show that attempts to implement UML state-charts fail to be faithful to UML's run-until-completion, informal semantics of UML state-chart execution. With this background, we will introduce our argumentation for adopting LLFSMs. We shall then explore the challenge that motivates this paper; that is the unification of executable models of reactive, real-time systems, offering deterministic semantics, a high level of parallelism, formal verification and automatic translation into a hardware specification language.

[1]Samek [17, Chapter 2] offers a speedy introduction to UML's state charts and is the base of "A Crash Course in UML State Machines" (www.state-machine.com/doc/AN_Crash_Course_in_UML_State_Machines.pdf.)

978-1-7281-1968-7/18/$31.00 ©2018 Crown

II. Modelling Reactive Systems

Reactive-systems are responsive systems without much processing; their counterparts are deliberative systems (which show artificial intelligence capabilities such as reasoning, planning, and learning) [6]. A Graphical User Interface (GUI) in common desktop operating systems or tablets is perhaps a good example of a *reactive system* that is not a *real-time system*: many users have experienced how the "mouse" turns into a spinning sand-clock icon for an indefinite amount of time.

A GUI is also an example of an *event-driven system*; namely one typically based on a software architecture built around stimuli-driven callbacks, a subscribe mechanism, and listeners that enact such call-backs. Reacting to stimuli in this way implies uncontrolled concurrency (e.g. using separate threads or event queues). Another example of an event-driven system is the interrupt mechanism for device I/O. Lamport [7] provided fundamental proofs of the limitations of event-driven systems. The counterpart to an event-driven system is a time-triggered system. The counterpart of the interrupt mechanism is polling. *Real-time systems* are required to meet time-deadlines in response to stimuli [8]. Therefore, although closely related, these three types of systems are not the same. The ubiquity of state charts to model these three types of systems has blurred the distinctions; for example, Sommerville [9, p. 544], refers to UML as an illustration that "state models are often used to describe real-time systems". However, Lamport [7] has provided solid and persuasive arguments to why real-time systems may be better served by time-triggered systems and pre-determined schedules, rather than the unbounded delays that may occur in event-driven systems.

A. Modelling Issues with Event-Triggered FSMs

Finite automata have long been used to model dynamic systems and thus computer system behaviour [10]. State diagrams have been prominently incorporated into the most popular modelling language, the Unified Modelling Language, UML [11]. Importantly, UML is restricted to defining a notation rather than a method for analysing and designing systems [12]. Consequently, UML offers advice on how to describe a system, not how to go about building a system [12]. This issue is exemplified by the inconsistency of event queuing, nesting, and ordering semantics for different implementations that use UML's notation [13].

The UML prevalence for modelling systems, and in particular, system behaviour through some form of state charts has perhaps over-emphasised event-driven systems, as UML state charts derive from Harel's State Charts [14] which are event-driven. Ideally, an event takes no time, it is something that happens in an instant and has no duration [10, Page 85]. Consequently, the UML and many authors consider events always as *"pair-wise mutually exclusive"* [15, Page 3]. Moreover, *"since they are mutually exclusive, input events arrive one at a time as inputs to the component under design"* [15, Page 3]. The UML community considers all scenarios as environments where events are never missed and *"assume that events are lost forever after being received and processed by the component under design"* [15, Page 4]. The initial assumption that events arrive in a sufficiently paced sequence propels those authors to assume that each event is handled in a *cycle* [15, Page 12], and essentially equate a cycle to a clock-tick. These assumptions are probably suitable at a macro scale, but are questionable for embedded systems and certainly doubtful for the applications one would use FPGAs for.

We invite the reader to reflect on the UML abstraction that *"events consume no time: they are zero time episodes "* [15, Page 50]. Because of this premise, software modellers *"can safely assume that two or more events are never received simultaneously"* [15, Page 50]. *"As close as they might be in time, events are never simultaneous"* [15, Page 50]. But rather quickly, this simplistic premise and corresponding assumption run into difficulties. First, *"the software actually consumes time when processing those events"* [15, Page 50]. Second, to achieve sophistication and complexity, UML state charts allow composability and nesting into a hierarchy. However, these seemingly innocent composition capacity creates all sorts of havoc with the semantics of state-chart execution. First, transitions with no label *"take place in the next cycle, or clock-tick"* [15, Page 51]. But in a hierarchy of state charts which of the potentially many transitions with no label is to be executed next?

Similarly, some scholars [16] interpret that when transitions are labelled by *guards* alone (no event), then they fire as soon as they become true. This assumption results in all sorts of race conditions as the updating of one variable could flip the value of many transitions. The semantics of such situations requires additional descriptions regarding *preemption* [15, Page 52] and race conditions [15, Page 56]. These issues usually imply the executable models and the code generated is multi-threaded, in uncontrolled concurrency (driven by the handling of events), which complicates formal verification. Perhaps also of concern is that *"the order of execution of these actions depends on the tool"* [15, Page 59]. The initially apparent simple assumptions now reveal a series of challenges: For hierarchical nesting, *"the Cartesian product machine is used as the interlingua semantics of statecharts"* [15, Page 63]. Moreover, *"an event can trigger a transition in all active threads, in some action threads, or in none"* [15, Page 63]. However, some implementors [2] recognise that there are scenarios where the parallel threads can be executed in a prescribed deterministic schedule without impacting on the semantics of the model: *"There is no apparent benefit in performing these blocks concurrently rather than using a pre-defined schedule"* [15, Page 64]. For some particular examples, where there is a logical separation of actions, the controller performs behaviours *"not necessarily in a truly concurrent manner. Instead, it can perform those control decision and actions computations serially"* [15, Page 65]. The most common semantics for UML is the *Run-to-Completion Execution Model* [17, Page 2.2.8]: that is, the

[2]Drusinsky is the implementor of `StateRover`, one of the earliest code generators from UML statecharts.

system queues events, whenever the system is still not finished handling an earlier event. We will not elaborate on the many issues derived from this. Suffice it to say that such queueing implies unbounded delays and the impossibility to warranty response times [7].

Thus, can the UML be used for model-driven development of systems in FPGAs? Can we actually take advantage of the parallelisation offered by FPGAs, when the alternative to UML's unbounded concurrency is sequential scheduling? A key problem when trying to tackle complexity from inception and analysis system requirements to implementations is the possibility of introducing errors at every step of the way. Probably the most important promise of model-driven development is the elimination of human error in the implementation process: ideally, once a formal specification of a system exists in the form of a formal model, no extra translation steps, such as the implementation in a low-level language such as VHDL or Verilog, need to be performed manually. Such automation would eradicate the human error potential to modify system behaviour in an undesired way.

B. Related Work

Implementation of finite-state machines (FSMs) in FPGAs has a long history [18]. FSMs were typically shown diagrammatically using graph-schemes of algorithms (GSA) and more importantly, transitions were labelled by *logical conditions* [18]. In some cases [19], transitions were labelled directly as an equality test with an input signal. Such transitions essentially were decorated by Boolean expressions, or in the language of UML, guards. That is, models of behaviour for FPGAs have historically avoided the event-driven nature of UML state charts.

The importance of an MDD approach to the synthesis of behaviour intended for embedded systems has been strongly emphasised [20]; particularly stressing that UML state charts are the prevalent diagrammatic tool for behaviour. Wood et al. [20] revised earlier attempts to generate VHDL from UML and found that either translation was incomplete (aimed at simulation and not hardware synthesis, covered a very small subset of UML's state chart notation or were short of following closely an MDD methodology). The MDD approach attempts to directly define a model transformation semantics from the syntactic construct of UML state charts to the VHDL code, but soon ran into issues that imposed limitations; for example the asynchronous nature of UML was replaced by synchronous specification in VHDL [20, Page 1362] (other semantic changes are also listed [20, Page 1364], as well as a table of unsupported features [20, Table 11]). The run-until-completion semantics of UML is avoided by requiring designs where all the consequences of an event must terminate by the next clock click [20]. This essentially eliminates events from labelling transitions (essentially a Boolean expression of the form `event_has_happened` that test an input signal state replaces each instance of an event) and allowing only guards that monitor signals (a system of syntactic priorities is enforced in case several signals become true in nesting machines, but for each model transitions must cover all cases and be mutually exclusive).

Similarly, other authors, who faced the huge challenge to replicate the semantics of UML into VHDL implementation, chose to side-step it, although with different means (using an intermediate language to translate UML to VHDL): Björklund and Lilus use SMDL where "state transitions are represented using the `goto` statement with an `if` statement checking for the presence of events". Other authors [21] place the `if` statement checking for the presence of events inside a loop, but the point is the same, events are eliminated (as before, from an interrupt mechanism to a pooling mechanism). This circumvention of UML's event-driven nature existed before: events were tripped from finite-state machines not represented with UML [22]. In this way, the sophisticated issues of UML are side-stepped when a corresponding transition is evaluated [20], [21]. Similar modifications to event semantics can be found in the translation of other models such as those developed with the Event-B formalism [23]. To the best of our knowledge, the only exception to the treatment of UML behaviour diagrams as MDD tools for VHDL (or similar hardware languages), where transitions are not reduced to Boolean expressions, is the use of UML activity models presented by Balderas-Contreras et *al.* [24]. However, they use activity diagrams for data-flows, and essentially use UML as a GSA. To the best of our knowledge, implementations of UML models that require concurrency and communication have been limited to only a few state charts participating in the system.

C. Defined Semantics: Logic-Labelled Finite-State Machines

Logic-labelled finite-state machines (LLFSMs) are a formalism that provides an alternative to the event nature of UML statecharts [25]. avoid the issues of translating the semantics of mathematically perfect events (of zero duration with perfect order) into implementations (that utilise, e.g., finite-length event queues). LLFSMs provide executable models with a precise, defined semantics [25]. LLFSMs are an alternative to the notion that reactiveness derives from event-driven modelling and that such modelling facilitates real-time characteristics [26, Page 63].

LLFSMs can be considered UML state diagrams with only guards, pooling the occurrence of events into Boolean variables that reflect whether an event has occurred or not. They have been named *procedural* [15]; because their execution can continue without pausing them. Traditionally, to run an arrangement of LLFSMs concurrently, the notion of a ringlet is required (this is basically the amount of work a single LLFSM performs while in its current state before passing the token of execution to the next LLFSM in the arrangement). The semantics of a single ringlet execution is broken up into a number of ordered actions:

1) A snapshot is read from the external variables.
2) If coming from a different state, *OnEntry* is run. The current state is unchanged.
3) The transitions are evaluated.

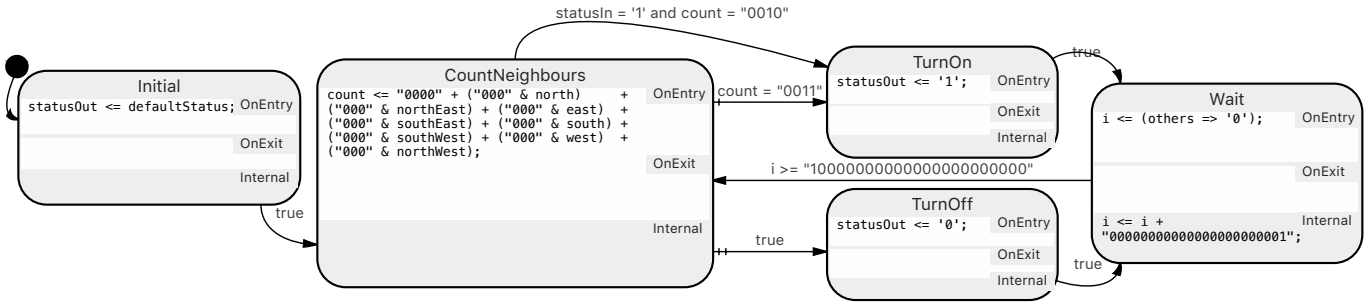

Fig. 1: An LLFSM implementing the behaviour of a cell in Conway's *Game of Life*.

4) If a transition has fired then *OnExit* is run, and the target state of the transition becomes the current state. Otherwise, the *Internal* action is run. The current state is unchanged.
5) The snapshot is written back to the external variables.

External variables are those that have scope beyond a single LLFSM and shared within the arrangement. Otherwise, variables have the scope of the LLFSM they live (machine scope) or the state they live (state scope). The snapshot taken before any code is executed ensures that all transitions are evaluated within the same context, and the sequential scheduling within the arrangement eliminates any race-conditions. However, this sequential scheduling seems to suggest that LLFSMs would not be suitable for massive parallelism when ported to FPGAs. We now proceed to show this is possible.

III. MDD: Compiling LLFSMs into FPGAs

Our method is the following.

1) The user creates a visual behaviour specification (e.g. Fig. 1) in an LLFSM-editor.
2) Our LLFSM-editor uses a machine format consumed by our code generator which generates the corresponding source code.
3) Each LLFSM can be instantiated many times and execute concurrently with other instances.
4) The designer can deploy and test the machines and integrate them into a project.

We will support the semantics of LLFSMs for specifying system behaviour, enabling the designer to capture requirements at a high level. Our design process to facilitate the deployment of LLFSMs into FPGAs (and thus furthering a framework of Model-Driven Development) consists of the following steps.

First, we extended an existing visual LLFSM-editor that was initially built to design machines in C++ [27]. Our extension to the visual editor enables high-level VHDL code to be included in LLFSM states and transitions, and supports saving into a suitable format for the next step. Second, we have created a tool that takes a configuration and accepts a set of machines that get instantiated and joined to generate a single VHDL project ready for compilation on an FPGA. Third, we design strategies to facilitate LLFSM ringlet semantics on the FPGA while enabling a large number of LLFSMs to execute in parallel and to communicate without race conditions. Our framework extends the standard semantics of an arrangement of LLFSMs in software (for example C++) that enabled concurrency but no parallelism, i.e., machines had to be scheduled and executed sequentially. The main contribution of this paper is the description and demonstration of the ringlet semantic implementation into the FPGA.

We now describe how we achieve parallelism while maintaining precise ringlet semantics for each LLFSM. A fundamental element of our approach is a VHDL template. We present this template using a case study that also illustrates the parallel execution of multiple, communicating instances of the same LLFSMs in a synchronised manner. Without loss of generality, our VHDL template applies to any design of LLFSMs.

The case study is Cellular Automata that can be modelled by LLFSMs as each cell is a simple automaton. Importantly, the arrangement of the Cellular Automata demands communication and synchronisation. Each cell's behaviour is defined as a finite-state machine whose next state is not only depending on its current state but is also influenced by the machines around it [28]. Thus, although each cell's behaviour is well modelled by a state machine, such machine must communicate (read the values) of its neighbours and change in synchrony its state to all others. A famous example of cellular automata is Conway's *Game of Life* [29]. Consider a square grid (called the board) with each node (cell) being represented by an instance of a machine that determines whether the cell should be on or off (thus resulting in only two states). The resulting state of the machine for any cell is determined by the eight machines that border its perimeter. The rules determining whether the state in the next cycle should be on or off are:

- If a cell is on, it will remain on if two or three of its neighbours are on.
- If a cell is off, then it can only be turned on if exactly three of its neighbours are on.
- Otherwise, the cell is turned off.

The state of each machine is updated periodically and all state changes happen synchronously. The dynamics of the system is directly determined by the initial state of the board. The board can have one of three outcomes:

1) All machines are (and stay) off.
2) A static structure where some (but not all) cells are permanently on.

3) An oscillatory pattern that repeats indefinitely.

This type of system can be defined as an arrangement of LLFSMs. Fig. 1 shows the model for each LLFSM.

TABLE I: Cellular Automaton Variables

Scope	Name	Type	Mode
External	north	std_logic	*in*
External	northEast	std_logic	*in*
External	east	std_logic	*in*
External	southEast	std_logic	*in*
External	south	std_logic	*in*
External	southWest	std_logic	*in*
External	west	std_logic	*in*
External	northWest	std_logic	*in*
External	statusIn	std_logic	*in*
External	defaultStatus	std_logic	*in*
External	statusOut	std_logic	*out*
Machine	count	unsigned (3 downto 0)	
Machine	i	unsigned (22 downto 0)	

An essential element of the LLFSM semantics and its implementation are the variables (and their communication scope). Table I shows the variables for this LLFSM. Variable `i` is not relevant for the cellular automata, it is used in our Model-Driven LLFSM to achieve busy-waiting, slowing visualisation so humans perceive the state change.

A. VHDL Variables and State Representation

The first issue that our template addresses is the representation of LLFSM states, its variables, and their different scope. We represent the states as `std_logic_vectors` of variable size: the size corresponds with the number of states in the machine. For VHDL, we prefix each state with `STATE_`. Our array of states for an LLFSM starts with *0* and is incremented by *1* for each state; that is, VHDL states are named by numerals. Because each variable has a scope of external, machine, or state, we prefix them accordingly. We place external variables in the entity declaration as a `port` parameter, and we place machine and state variables in the architecture declaration. By default we supply a `port` clock signal, `clk` which is a `std_logic` input. Our code generation includes four additional internal state constants to track the sections (*OnEntry*, *OnExit*, *Internal* & *CheckTransition*). Also, we define two additional internal state constants for the implementation of LLFSM snapshot semantics: (*ReadToSnapshot* and *WriteFromSnapshot*). We use four additional internal signals to track the current state of the machine as well as the actions to be performed in a section: `internalState`, `currentState`, `targetState` and `previousRinglet`. Our tool creates local snapshot variables for each external variable in the design. In order to ensure that the user is referring to only the snapshot variables at any point in time, external variables get prefixed with `EXTERNAL_` in the `port` declaration. The snapshot variables which are represented as architecture signals (and thus have machine scope) do not have this prefix attached.

To illustrate, consider the first ringlet of the Cellular Automata example (Fig. 1). Fig. 2 shows the VHDL code our template generates for the relevant variables and states of this

```
library IEEE;
use IEEE.std_logic_1164.All;
use IEEE.numeric_std.all;

entity CellularAutomaton is
    port (
        clk: in std_logic;
        EXTERNAL_statusOut: out std_logic;
        EXTERNAL_defaultStatus: in std_logic
    ...
    );
end CellularAutomaton;

architecture LLFSM of CellularAutomaton is
    --Internal State Representation Bits
    constant OnEntry: std_logic_vector (2 downto 0) := "000";
    constant CheckTransition: std_logic_vector (2 downto 0) := "001";
    constant OnExit: std_logic_vector (2 downto 0) := "010";
    constant Internal: std_logic_vector (2 downto 0) := "011";
    constant ReadToSnapshot: std_logic_vector (2 downto 0) := "100";
    constant WriteFromSnapshot: std_logic_vector (2 downto 0) := "101";
    signal internalState: std_logic_vector (2 downto 0) :=
        ReadToSnapshot;
    --State Representation Bits
    constant STATE_Initial: std_logic_vector (2 downto 0) := "000";
    constant STATE_CountNeighbours: std_logic_vector(2 downto 0) :=
        "100";
    ...
    signal currentState: std_logic_vector (2 downto 0) :=
        STATE_Initial;
    signal targetState: std_logic_vector (2 downto 0) := currentState;
    signal previousRinglet: std_logic_vector (2 downto 0) :=
        STATE_Initial xor "111";
    --Snapshot of External Variables
    signal statusOut: std_logic;
    signal defaultStatus: std_logic;
    ...
begin
...
end LLFSM;
```

Fig. 2: VHDL variable representation.

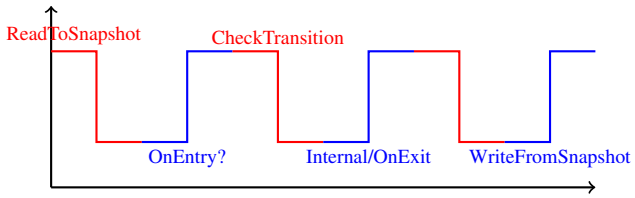

Fig. 3: Ringlet timing diagram.

ringlet. Because the other variables are analogous, we skip the display of their VHDL code.

The next step is to create VHDL code that matches the semantics of LLFSMs. Each action indicated above is performed at a given point in time, based on the rising and falling edge of the clock. The different actions are executed in a process block where the clock is the only signal in the sensitivity list. The implementation we have created can be represented by the timing diagram of the reference clock seen in Fig. 3. Each ringlet takes three clock cycles, starting on the falling edge. The advantage of this constant time execution, is that it facilitates determinism and synchronisation.

B. Semantics Implementation

First in the execution of this machine is the *ReadToSnapshot* action of the Initial state. This code is the same for every state in the machine and is shown in Fig. 5.

```
case internalState is
    ...
    when ReadToSnapshot =>
        north <= EXTERNAL_north;
        east <= EXTERNAL_east;
        south <= EXTERNAL_south;
        west <= EXTERNAL_west;
        statusIn <= EXTERNAL_statusIn;
        northEast <= EXTERNAL_northEast;
        southEast <= EXTERNAL_southEast;
        southWest <= EXTERNAL_southWest;
        northWest <= EXTERNAL_northWest;
        defaultStatus <= EXTERNAL_defaultStatus;
        if ( previousRinglet = currentState ) then
            internalState <= CheckTransition;
        else
            internalState <= OnEntry;
        end if;
    ...
end case;
```

Fig. 5: ReadToSnapshot VHDL code.

```
case currentState is
    ...
    when STATE_Initial =>
        case internalState is
            when OnEntry =>
                statusOut <= defaultStatus;
                internalState <= CheckTransition;
            when Internal =>
                internalState <= WriteFromSnapshot;
            when OnExit =>
                internalState <= WriteFromSnapshot;
            ...
        end case;
    ...
end case;
```

Fig. 6: VHDL Code for Rising-Edge actions.

We read the external variables into the snapshot copies that the ringlet uses. The `previousRinglet` signal then determines whether *OnEntry* should be executed next, on the rising edge of the clock. The `currentState` signal is used in a case statement to determine which FSM state is executing. Each state contains another case statement on `internalState` to discern the different actions that the state is executing. For the actions *OnEntry*, *Internal* and *OnExit*, the designer's corresponding code is placed in the actions section, followed by an update to the `internalState` signal. The code for these actions of the Initial state is shown in Fig. 6. The next action is the *CheckTransition* action on the falling edge of the clock, performing an if-statement on the transitions in their order of priority. When evaluating to true, the `internalState` signal is set to *OnExit* and the `targetState` signal is set to the transition's target state. If no transition evaluates to true, the `internalState` signal is set to *Internal* in the else clause. The *CheckTransition* code for the Initial state is shown in Fig. 7.

```
case internalState is
    when CheckTransition =>
        case currentState is
            ...
            when STATE_Initial =>
                if ( true ) then
                    targetState <= STATE_CountNeighbours;
                    internalState <= OnExit;
                else
                    internalState <= Internal;
                end if;
            ...
        end case;
    ...
end case;
```

Fig. 7: VHDL code for CheckTransition.

The last actions are either the *Internal* or *OnExit*, executed on the next rising edge of the clock, followed by *WriteFromSnapshot* on the next rising edge, writing all of the snapshots back to their corresponding external variables, then setting `internalState` to *ReadToSnapshot*, the `previousRinglet` to the current state, and finally setting `currentState` to `targetState` (Fig. 8).

```
case currentState is
    when STATE_Initial =>
        case internalState is
            ...
            when WriteFromSnapshot =>
                EXTERNAL_statusOut <= statusOut;
                internalState <= ReadToSnapshot;
                previousRinglet <= currentState;
                currentState <= targetState;
            ...
        end case;
    ...
end case;
```

Fig. 8: VHDL Code for WriteFromSnapshot.

IV. Evaluation

The Cellular Automata were evaluated on a Xilinx Spartan 6 LX 25 FPGA on a *miniSpartan6+* evaluation board. A test bench was created with a clock frequency of 100MHz and the waveforms for the first ringlet in the *Initial* state is shown in Fig. 4. The values of the different internal signals were assigned, based on the state-encoding constants shown in Fig. 2. The first ringlet sets the external `statusOut` variable to the current value of the `defaultStatus` external variable in the *OnEntry* section (Fig. 1). The generated VHDL program commences with the following, initial values:

- `internalState` is set to *100*, the *ReadToSnapshot* section.
- `currentState` is set to *000*, the Initial state.
- `targetState` is also set to *000*.

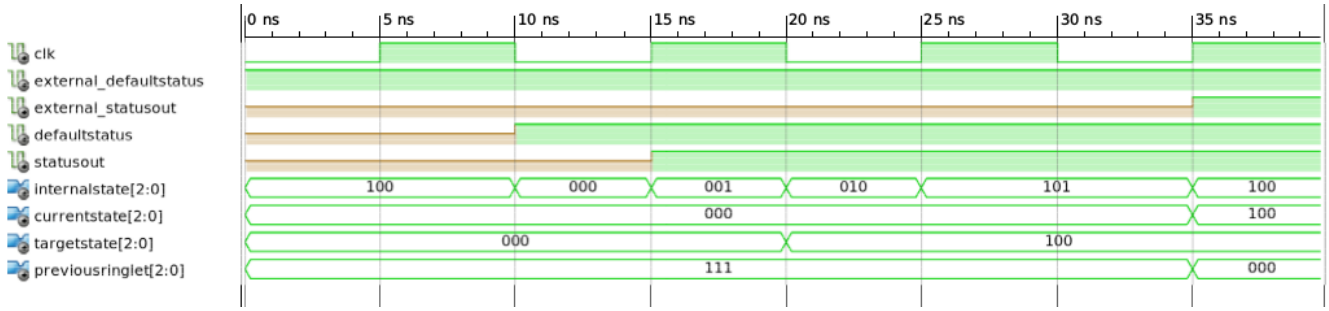

Fig. 4: Simulation Results of the First Ringlet

- `previousRinglet` is set to *111* to be different from Initial state, ensuring that *OnEntry* is run in the first ringlet.
- `EXTERNAL_defaultStatus` is set to *1*.

As illustrated in Fig. 3, the ringlet starts on the falling edge of the clock, where the *ReadToSnapshot* copies the external input signals into their corresponding snapshot signals. E.g., `defaultStatus` changes from being uninitialised to a logic high. Next, `internalState` is changed to *000*, signifying the *OnEntry* section. On the rising edge of the clock, the corresponding code shown in Initial of Fig. 1 is executed, setting the `statusOut` snapshot variable to the value of the `defaultStatus` snapshot, i.e. The `statusOut` signal gets initialised to a value of *1*. `internalState` is then incremented to *001* (*CheckTransition*). On the subsequent falling edge, *CheckTransition* evaluates all transitions and sets the `targetState` accordingly, i.e. the always-true transition from Initial to CountNeighbours fires. `targetState` is, accordingly, set to the value of the `CountNeighbours` constant, i.e. *100*. Then `internalState` is set to *010* corresponding to the *OnExit* section, which does not contain any user code, so only `internalState` gets updated to *101*, *WriteFromSnapshot*, on the rising edge of the clock. No action occurs on the subsequent falling edge of the clock. On the next rising edge, the snapshot variables `defaultStatus`, and `statusOut` get copied to their externally visible counterparts, `internalState` gets reset to *ReadToSnapshot* and the `currentState` is set to `targetState` (CountNeighbours) concluding the transition from the Initial state to CountNeighbours.

As illustrated in the evaluation of the above execution steps (that we have repeated for the remaining states), the VHDL code generated from the high level, visual model matches the semantics of LLFSMs. We now present a proof of concept, where we have replicated the machines to simulate the *Game of Life* on a grid, to demonstrate the ability of our FPGA approach to go beyond the capabilities of software-implemented LLFSMs and create a massively parallel array of communicating FSMs without race conditions.

We organized 15×15 machines to form a 15 by 15 grid with an outside perimeter that is always off. We tested again our synthesised VHDL for this array of LLFSMs on the same FPGA on our *miniSpartan6+* evaluation board. We connected the output of each machine to drive an HDMI signal and display it on a TV so that the results could be observed. The clock frequency for this example was 50MHz with a period of 20ns. Each ringlet takes three clock cycles, starting on the falling edge, so each ringlet takes 60ns. We could construct well-known shapes by setting the initial values of the cells in accordances with the examples shown in the literature surrounding the game of life [29].

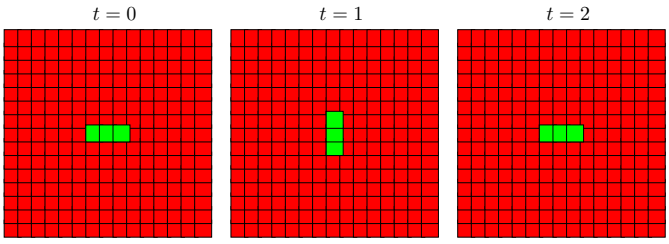

Fig. 9: The alternation known as *Blinker*.

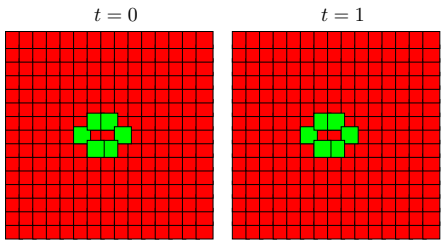

Fig. 10: The alternation known as *Beehive*.

Fig. 9, Fig. 10, and Fig. 11, show some of the shapes that were generated; where green and red represent on and off respectively. One of the requirements for the game of life is that the machines update their states at the same time, to ensure synchronisation and determinism. This demonstrates the advantage of our snapshot semantics that enforces this semantics and prevents any race conditions, as external variables are only ever written to at specific points in time, and any attempts by two machines to concurrently modify the same output would result in a compile-time error, as variables cannot be written to from multiple drivers. Moreover, the *ReadToSnapshot* and *WriteFromSnapshot* phases occur at different clock cycles, ensuring that reading and writing operations never coincide.

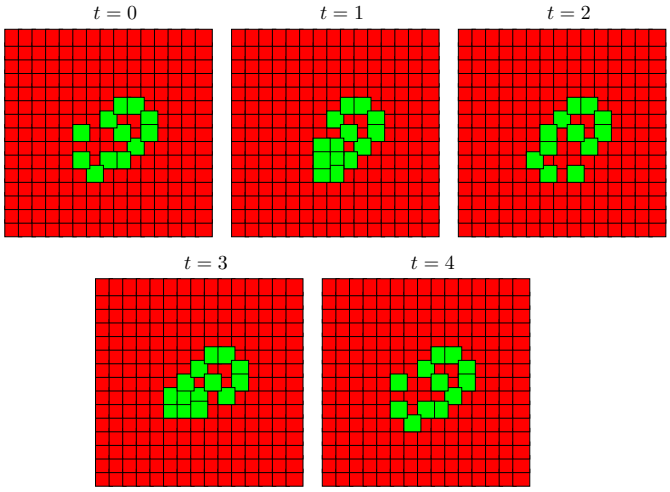

Fig. 11: The alternation known as *Mold*.

V. Conclusion

In this paper we demonstrated that massively parallel, communicating LLFSMs with well-defined semantics can be implemented on an FPGA. Not only does our approach enforce determinism in the value domain, but due to deterministic timing for each state, we can create executable models of complex real-time behaviour with perfect knowledge of timing requirements at design time. Moreover, time-triggered, deterministic behaviour allows communication between multiple machines without race conditions or the requirement for and the complexity of explicit synchronisation constructs. We have demonstrated this with a case study of massively parallel, communicating, logic-labelled finite-state machines, implementing and visualising cellular automata.

References

[1] K. Elk. *Embedded Software Development for the Internet Of Things: The Basics, the Technologies and Best Practices*. CreateSpace Independent Publishing Platform, 2016.

[2] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[3] *INTERNET OF THINGS 2017 REPORT*. BI Intelligence, 2017.

[4] R. Bryce and R. Kuhn. Software testing [guest editors' introduction]. *Computer*, 47(2):21–22, Feb 2014.

[5] S. J. Mellor. Executable and translatable UML. *Embedded Systems Programming*, 16(2):25–30, 02 2003.

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

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

[8] H. Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, Berlin, second edition, 2011.

[9] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.

[10] J. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modelling and Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

[11] The Object Management Group. *Information technology - Object Management Group Unified Modeling Language (OMG UML), Infrastructure*. ISO/IEC 19505-1:2012(E). ISO, April 2012.

[12] S. J. Mellor. Demystifying UML. *Embedded Systems Design*, 19(3):22–33, Mar 2006.

[13] W. Damm, B. Josko, Votintseva A., and A. Pnueli. A formal semantics for a UML kernel language. Deliverable IST/33522/WP 1.1/D1.1.2-Part1, OMEGA Correct Development of Real-Time Embedded Systems, 2003.

[14] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

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

[16] W. Schamai, P. Fritzson, and C.J.J. Paredis. Translation of UML state machines to Modelica: Handling semantic issues. *Simulation*, 89(4):498–512, April 2013.

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

[18] A. Barkalov, L. Titarenko, M. Kolopienczyk, K. Mielcarek, and G. Bazydlo. Chapter 2: Finite state machines and field-programmable gate arrays. In *Logic synthesis for FPGA-based finite state machines*, Studies in systems decision and control, pages 7–34. Springer, 2016.

[19] A. Tiwari and K. A. Tomko. Saving power by mapping finite-state machines into embedded memory blocks in FPGAs. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 916–921 Vol.2, Feb 2004.

[20] 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*, 57(10):1357–1371, Oct 2008.

[21] W. E. McUmber and B. H. C. Cheng. Uml-based analysis of embedded systems using a mapping to vhdl. In *Proceedings 4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 56–63, 1999.

[22] A. T. Abdel-Hamid, M. Zaki, and S. Tahar. A tool converting finite state machine to vhdl. In *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)*, volume 4, pages 1907–1910, May 2004.

[23] S. Ostroumov and L. Tsiopoulos. VHDL code generation from formal Event-B models. In *2011 14th Euromicro Conference on Digital System Design*, pages 127–134, Aug 2011.

[24] T. Balderas-Contreras, R. Cumplido, and G. Rodríguez. Synthesizing VHDL from activity models in UML 2. *Int. J. Circuit Theory Appl.*, 42(5):542–550, May 2014.

[25] V. Estivill-Castro and R. Hexel. Arrangements of finite-state machines semantics, simulation, and model checking. In *International Conference on Model-Driven Engineering and Software Development*, pages 182–189, 2013.

[26] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons Inc, New York, 1994.

[27] V. Estivill-Castro, R. Hexel, and C. Lusty. High performance relaying of C++11 objects across processes and logic-labeled finite-state machines. In D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, editors, *Simulation, Modeling, and Programming for Autonomous Robots - 4th International Conference, SIMPAR 2014*, volume 8810 of *Lecture Notes in Computer Science*, pages 182–194. Springer, October 20th-23rd 2014.

[28] Stephen Wolfram. Cellular automata as models of complexity. *Nature*, 311:419 EP –, Oct 1984.

[29] M. Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.