Runtime Verification on FPGAs with LTLf Specifications

Tommy Tracy II
University of Virginia
Charlottesville, Virginia 22904
Email: tjt7a@virginia.edu

Lucas M. Tabajara
Rice University
Houston, Texas 77005
Email: lucasmt@rice.edu

Moshe Vardi
Rice University
Houston, Texas 77005
Email: vardi@rice.edu

Kevin Skadron
University of Virginia
Charlottesville, Virginia 22904
Email: skadron@virginia.edu

Abstract*—Runtime verification is a technique that evaluates a system's execution trace at runtime against a formal specification. This approach is particularly useful for safety-critical and autonomous systems to verify system functionality and allow for graceful recovery or intervention in the case of system faults. Specifications are often provided in a high-level form using some type of temporal logic, which can then be compiled into an automaton to be used as a monitor for the system. Existing work has mainly focused on implementing such monitors in software. In recent years there has been extensive research, however, in hardware acceleration of automata applications, which can potentially be extended to runtime monitoring. In this paper, we introduce an open-source framework for translating formulas in Linear Temporal Logic over finite traces (LTL_f) into automata implementations on FPGAs for high-efficiency and high-performance runtime monitoring. By using the spatial dimension of FPGAs, we run many of these automata in parallel, significantly reducing the latency between violation and monitor report and achieving significant throughput. We compare the performance of four different architectures corresponding to the combinations of deterministic or nondeterministic automata with an explicit or symbolic representation, and determine the design parameters that result in efficient hardware utilization and higher clock frequencies. We found that explicit automata tend to use more hardware resources, in particular Lookup Tables (LUTs), than symbolic automata. An exception to this is in the case of Flip-Flop (FF) usage, where symbolic DFAs tend to use more FF resources than explicit NFAs for smaller designs. We also found that explicit NFAs can run at higher clock frequencies, except for very large automata with high edge densities. Symbolic NFAs use fewer Look-Up Table resources and run at higher clock frequencies than symbolic DFAs, whereas symbolic DFAs required fewer Flip-Flop resources, except in the case of very simple small automata with lower edge densities. Finally, we found that explicit automata hardware utilization significantly increases with input signal widths, motivating the use of symbolic automata for wide input signals.

I. Introduction

While other types of formal verification seek to verify a system before it is deployed, the goal of runtime verification is to monitor the execution of a system in real time in order to detect behavior that violates the system's formal specification [1], [2], [3]. This gives the system a chance to mitigate, recover from, or document the error. Runtime verification is particularly valuable for safety-critical and autonomous systems [4], where errors that are not immediately dealt with can have catastrophic consequences. Such systems also often operate in physical environments, which are hard to model accurately and often behave in unexpected and unpredictable ways. Therefore, even if the system has been formally verified beforehand, it is possible that it might still display errors during runtime due to assumption violations.

Most existing work on runtime verification has focused on monitors implemented in software [1], [2]. Motivated by the slowing down of Moore's Law and the end of Dennard Scaling [5], there has been a recent trend to use specialized hardware [6]. Specialized hardware that is designed to perform a particular task can be optimized for that task much more than it would be possible for general-purpose hardware. Furthermore, the application can benefit directly from the natural parallelism that hardware provides. For runtime verification, an on-board hardware implementation translates to more efficient real-time monitoring with lower latency.

Monitors used for runtime verification usually take the form of (deterministic or nondeterministic) finite-state automata. Automata applications have already been a target of hardware acceleration, as exemplified by Micron's Automata Processor [7], [8] and subsequent work targeting FPGAs [9], [10]. Specialized architectures for simulating automata have been employed for a number of data- and string-processing applications, including bioinformatics [11], [12], machine learning [13], [14], and natural language processing (NLP) [15]. As an application that also runs automata over streaming data - in this case traces of the system's execution - the extension to runtime verification is a natural one.

Unlike data- and string-processing applications, automata used for formal verification, including runtime verification, are often generated from formal specifications given as formulas in a temporal logic [16], [17], rather than directly as finite automata. A major difference between automata constructed from temporal-logic formulas and those obtained for other applications is the alphabet construction. Data- and string-processing application usually assume a static and relatively small symbol alphabet. For example, an NLP automaton would likely use ASCII as the symbol alphabet, and bioinformatics applications may only need four symbols corresponding to the four nucleotide bases A, T, C and G.

In the case of automata used for formal verification, the symbol alphabet consists of propositional interpretations of the

https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_10

atomic propositions in the formula. The number of possible such interpretations, and therefore the size of the symbol alphabet, is exponential in the number of propositions, leading to potentially much larger alphabets. The problem of the exponential alphabet is usually solved in formal-methods applications by not explicitly representing the transitions on each symbol, but instead labeling transitions by Boolean expressions, with the understanding that a transition is activated by an interpretation if that interpretation satisfies the expression.

Tools that construct automata from temporal formulas [18], [19] often represent transitions in this way. While being very natural when the automaton is generated from a logic formula, this symbolic representation of the transition relation is not supported by Micron's Automata Processor, for example, which uses eight bits per symbol in the alphabet and a memory column of 256 bits per state to recognize the unique symbols in the alphabet. Sadredini et al. [20], with their Flexamata compiler and subsequent Grapefruit [10] FPGA implementation, which integrates Flexamata into a full-stack automata processing framework, addressed this concern by converting among automata of differing symbol alphabet sizes. They do this by trading off symbol alphabet size with automata size and throughput. Unfortunately, this requires temporal multiplexing, where the system signals would need to be buffered and serialized at the reduced width. This approach could work for lower sampling rates of the system signals, but could also bottleneck the system for some applications.

Our main contribution in this work is an investigation among four possible architectures for implementing automata-based monitors for temporal-logic properties on a field-programmable gate array (FPGA). These four architectures are defined by two axes: deterministic vs. nondeterministic and explicit vs. symbolic. The first axis specifies whether the temporal logic specification is converted into a deterministic finite automaton (DFA) or a nondeterministic finite automaton (NFA). It is difficult to predict a priori which of these representations is more efficient in terms of the number of states. Although NFAs have an exponentially smaller worst-case size, DFAs have an exact minimization algorithm that runs in polynomial time, while for NFA minimization, which is PSPACE-complete [21], we are forced to rely on heuristics. The second axis determines whether the state space of the automaton should be represented explicitly or symbolically. In an explicit representation, each state has its own hardware component, called a State Transition Element (STE), that is activated when the automaton moves into that state. Each STE has its own state memory and logic to match the input to the matching symbol set of that state. In this architecture, hardware parallelism allows nondeterminism to be simulated at no additional cost, as multiple STE can be active at the same time. In a symbolic representation, the current state (or set of states, in the case of an NFA) is represented by a bitvector, which is given along with the current input to a logic circuit that computes the bitvector representing the next state. An advantage of DFAs in the symbolic representation is that the current state can be represented in a logarithmic number of bits, while NFAs require a bit per state. On the other hand, the logarithmic encoding in the DFA might lead to more complex (deeper) combinational logic.

We evaluate each of these four options on a set of randomly-generated formulas in Linear Temporal Logic over Finite Traces (LTL_f) [22], formed by taking random conjunctions of common temporal patterns [23]. LTL_f was chosen because it is a convenient way of specifying events that happen in a finite time, such as the ones that runtime verification seeks to detect, and a lot of machinery exists for translating such formulas into finite automata [24], [25]. The formula set is converted into separate automata, which are then implemented on one FPGA. Each automaton monitors a different property, but the set shares input signals corresponding to shared propositions between the formulas. We scale our benchmarks by varying the number of formulas, the number of different variables across all formulas (number of unique system signals), and the number of conjuncts per formula (formula complexity).

The results of our evaluation provide insight on the different tradeoffs that emerge when considering solutions implemented directly in hardware as opposed to software. We found that symbolic automata tend to use less hardware than explicit automata and that explicit NFAs tend to run at higher clock frequencies, except in the case of very small formulas or very complex formulas. Overall, we find that symbolic NFAs tend to perform best of all of our evaluated architectures across most experiments with the lowest hardware utilization.

Finally, we found that explicit automata hardware utilization significantly increases with the size of the symbol alphabet, motivating the use of symbolic automata for wide input signals, which happens when the formula has a high number of propositions. Our investigation allows us to better understand the considerations that must be taken into account when implementing runtime monitors in hardware, and concludes that, while no particular approach dominates, each one has its own pros and cons that should be considered when deciding how to accelerate runtime verification for a specific application.

II. Background

A. Linear Temporal Logic

Linear Temporal Logic over Finite traces (LTL_f) is a variant of Linear Temporal Logic that is interpreted over finite rather than infinite traces. Its syntax is identical to LTL over infinite traces, and is defined as follows for a set of propositions $\mathcal{P}$:

$$\varphi ::= \top \mid p \in \mathcal{P} \mid (\neg\varphi) \mid (\varphi_1 \vee \varphi_2) \mid (\mathbf{X}\varphi) \mid (\varphi_1 \mathcal{U} \varphi_2)$$

Lichtenstein, Pnueli, and Zuck showed in [26] that every LTL formula is equivalent to a formula of the form $\bigwedge_i^n (GF\phi_i \vee FG\psi_i)$, where ϕ_i and ψ_i contain only past operators. In other words, ϕ_i and ψ_i are finite-trace formulas. Thus, finite-trace monitors are the foundation on which one can build a monitoring framework for LTL [27], which motivates our focus on LTL_f.

$\mathbf{X}$ and $\mathcal{U}$ are the temporal connectives "next" and "until". We can define other temporal connectives such as $\mathbf{F}$

("eventually"), $\mathbf{G}$ ("always") and $\mathcal{W}$ ("weak until") in terms of those. A *propositional interpretation* $\tau \in 2^{\mathcal{P}}$ is a set of propositions representing the propositions that are true at a given time. A *trace* is a finite sequence $\rho \in (2^{\mathcal{P}})^*$ of propositional interpretations $\rho_0, \rho_1, \ldots, \rho_n$, where ρ_i is the set of propositions that are true at time i. We denote that an LTL_f formula φ is satisfied by a trace ρ at time i by $\rho, i \models \varphi$, and shorten $\rho, 0 \models \varphi$ to $\rho \models \varphi$. Refer to [22] for the semantics of LTL_f formulas. The *language* of an LTL_f formula φ, denoted by $\mathcal{L}(\varphi)$, is the set of finite traces that satisfy φ. The *reverse* of a trace $\rho = \rho_0, \ldots, \rho_n$, denoted by ρ^R, is the trace $\rho_n, \ldots, \rho_0$. The *reverse* of the language of a formula φ, denoted by $\mathcal{L}^R(\varphi)$, is the set of traces ρ^R for $\rho \in \mathcal{L}(\varphi)$.

B. Finite State Automata

A Finite State Automaton (FSA) is a mathematical model of the form $A = (S, \Sigma, I, \Delta, F)$, where: S is a finite set of states, Σ is a finite set of symbols called the input alphabet, $I \subseteq S$ is a set of initial states, $\Delta \subseteq S \times \Sigma \times S$ is a transition relation indicating the successor states of a given state when the automaton reads an input symbol in Σ, and $F \subseteq S$ is a set of accepting states. FSA are often represented by a graph of nodes connected by edges. Figure 1.A shows an example FSA, where the left-most state is an initial state, and the right-most is an accepting state. Edges represent transitions, and are labeled by the corresponding transition symbols from the input alphabet. FSAs process input signals by transitioning between states. The computation begins at the initial state and proceeds at every time interval, evaluating an input symbol. If that input symbol matches a transition symbol, a transition is made to the next state, and so forth. If an accepting state is reached, then the automaton has accepted the input; if not, the input is not accepted. The set of finite traces accepted by an FSA A is the *language* of the FSA, and denoted by $\mathcal{L}(A)$.

FSAs can be deterministic or nondeterministic. Deterministic Finite Automata (DFAs) have at most one initial state, and at most one transition from each given state on a given input symbol. Nondeterministic Finite Automata (NFAs), on the other hand, are more general and can have multiple transitions from each state on the same input symbol. As a consequence, when running an NFA over a sequence of inputs, multiple transitions can be taken at once, and multiple states can be active at the same time. Every NFA can be determinized into a DFA that recognizes the same language, but in the worst case the smallest DFA for a given language may be exponentially larger than the smallest NFA. Because of this, DFAs potentially yield a significant increase in memory utilization, while NFAs are memory-bandwidth bounded by potentially many parallel transitions. However, DFAs have an efficient and exact minimization algorithm, while NFAs can practically only be minimized heuristically [28].

Previous work has demonstrated how finite state automata can be used to accelerate a variety of applications that go beyond the usual string matching applications, including bioinformatics [11], [12], machine learning [13], [14], and natural language processing [15]. These works represent FSAs in a *homogeneous* representation, where matching computations are done on the states, rather than the edges. More specifically, a homogeneous automaton is one where all transitions into a state have the same symbol set [29], [30]. We depict such automata by placing the symbol sets on the states rather than on the edges. Figure 1 depicts a non-homogeneous automaton and its equivalent homogeneous automaton. Homogeneity is used in hardware implementations to simplify the mapping of automata to hardware for parallel transition computation on the nodes, as demonstrated in Micron's Automata Processor [7]. This transformation also comes at a significant increase in the number of states in the automata, scaling with the edge density of the non-homogeneous representation.

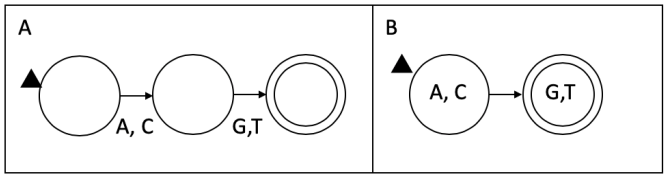

Fig. 1. Non-homogeneous (A) vs. Homogeneous automaton (B).

Every LTL_f formula φ over a set of propositions $\mathcal{P}$ can be converted into a (deterministic or nondeterministic) FSA A_φ with alphabet $2^{\mathcal{P}}$, such that $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$. In the worst case, the smallest NFA for an LTL_f formula may be at most exponential in the size of the formula, while the smallest DFA may be doubly-exponential. The tool MONA [18] implements an algorithm for constructing a minimal DFA from a formula in Monadic Second-Order Logic (MSO). Since every LTL_f formula can be converted into MSO [22], MONA can be used to generate minimal DFAs for LTL_f formulas. As part of our framework, we present a solution to using MONA for generating NFAs as well.

C. Automata Acceleration with FPGAs

Field Programmable Gate Arrays (FPGAs) are used in computing systems to implement reconfigurable hardware. Existing automata engines including REAPR [9], REAPRpp [31], and Grapefruit [10] accelerate a variety of applications with explicit automata on FPGAs. Figure 2 illustrates how *explicit automata* are represented in hardware by these explicit engines, with constituent states of the automata instantiated with separate memory and logic resources. This requires that the spatial resources used by the design grow in the size of the automata, but also allows all automata states to make transitions in parallel, making this approach particularly efficient for processing NFAs, where the number of active states can be variable and for evaluating multiple automata in parallel.

REAPR works by generating Verilog from ANML [7] automata description files, an XML-like homogeneous FSA representation, and generates an architecture that is very similar to Micron's Automata Processor (AP) [7], using the homogeneous automata representation. One limitation REAPR inherited from the AP is the static 8-bit symbol width. AP-like automata processing assumes an input symbol of 8 bits and a

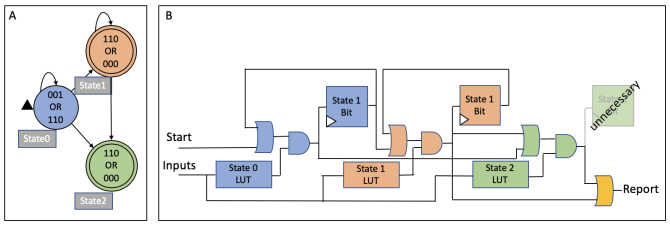

Fig. 2. Architecture of explicit automata.

corresponding 256-bit matching column for representing the full 8-bit symbol alphabet. Although useful when considering pattern matching on ASCII or byte-level data, when implementing runtime verification, this limits the number of system signals that our formulas could process in one cycle to 8.

Rahimi et al. [10] implement the Flexamata [20] compiler in their Grapefruit full-stack automata engine. They overcome the symbol-width limitation by extending ANML and allowing for arbitrary bitwidths by trading off symbol alphabet size with automata size and throughput. In addition to this, Grapefruit also includes heuristic-based NFA minimization, which allows the tool to reduce the size of explicit finite state automata in hardware. We utilize Grapefruit and extend its functionality to also include symbolic automata (see Section III-B.

D. Runtime Monitors

Although there are many types of runtime monitors with different semantics, in this work we define a monitor for an LTL_f formula φ as an FSA (NFA or DFA) that accepts a finite trace iff this trace satisfies φ. As the monitor reads the trace produced by the system, it continuously reports whether the finite trace observed from the beginning of the execution until the current time step satisfies the formula.

Previous work on runtime monitors has focused on automatically generating runtime system monitors on CPUs as well as on FPGAs. Drusinsky [32] introduces a verification tool that generates code from LTL and MTL assertions written into C, C++, Java, Verilog and VHDL code to evaluate runtime systems against the formulas at runtime.

Tabakov et al. [2] introduce a technique for automatically generating SystemC runtime monitors from LTL formulas. They identify four important components that they optimize to minimize runtime overhead: state minimization, alphabet representation, alphabet minimization, and monitor encoding. They then identify the configurations that offer the best monitor performance in terms of runtime overhead. Pike et al. [33] introduce Copilot, a domain-specific language built on top of Haskell for programming runtime monitors for distributed real-time systems. Boule and Zilic [34] use a recursive technique that breaks properties into syntax trees. Each node in the tree is used to create a sub-property automaton which are concatenated with the rest in an automata generation algorithm.

Geist et al. [35] implement runtime observers in their system on processors implemented on their FPGA. Meredith et al. [36] with MOP and Pellizzoni et al. [37] with BusMOP use the Monitoring Oriented Programming (MOP) framework to synthesize hardware monitors for runtime verification. BusMOP generates monitor blocks from temporal logic specifications. These monitor blocks use symbolic DFAs to verify system properties at runtime.

Jaksič et al. [38] translate Signal Temporal Logic (STL) assertions into hardware runtime monitors on FPGAs. They synthesize *temporal testers*, or transducers, that output a signal if a specification has been satisfied. Selyunin et al. [39] translate Signal Temporal Logic (STL) and Timed Regular Expressions (TRE) into hardware monitors on FPGAs. They demonstrate a High-Level Synthesis (HLS) and automata-based approach for temporal tester transducers.

Selyunin et al. [40] apply runtime monitoring for automata systems and use HLS to synthesize monitors for FPGAs. Baumeister et al. [41] compile RTLola, a stream-based specification language used for real-time properties, into VHDL for FPGA deployment. Convent et al. [42] introduce the Temporal Stream-based Specification Language (TeSSLa) used to specify constraints on railway cyber-physical systems. Their approach differs considerably from previous approaches, because they allow for runtime reconfigurability. They do this by creating a set of event processing units that can be combined at runtime to monitor for complex properties.

Schumann et al. [4] introduce R2U2, a monitoring and diagnosis framework for unmanned aerial systems. R2U2 is implemented on an FPGA and monitors streams of data from the GPS and ground control station, flight software status, sensor readings, and actuator outputs. They implement their runtime monitors in logic as presented by Reinbacher et al. [3].

While previous work has implemented runtime monitors on FPGAs, our work differentiates itself in a few ways. First, we take advantage of recent progress made in hardware acceleration of automata by using state-of-the-art approaches from that field. We also focus on LTL_f as a specification language for runtime properties, allowing us to also use recently-developed techniques for converting LTL_f formulas into automata. Finally, as far as we are aware we are the first to perform an experimental comparison between deterministic and non-deterministic as well as symbolic and explicit automata, in order to determine the advantages and disadvantages of each representation in an FPGA implementation.

III. Implementing LTL_f Monitors in Hardware

We present an open-source software pipeline[43] for converting LTL_f formulas into automata-based runtime monitors implemented on a cloud-deployed FPGA. We explore four possible automata representations placed along two axes: deterministic/non-deterministic and explicit/symbolic. Each representation is described later in this section.

For generating the automata from the temporal formulas, we employ an approach centered on the tool MONA [18], which can construct finite automata from formulas in Monadic Second-order Logic (MSO), a logic strictly more expressive than LTL_f. We chose this tool based on its performance and versatility. Other possible options for converting LTL_f to

automata would be the tools SPOT [19] and LISA [44]. Previous comparisons, however, have shown MONA to perform better than SPOT [24], while LISA only has support for DFAs. Because MONA is based on MSO, we can use a technique based on reversing the formula to construct NFAs as well, as described in Section III-A.

Although there are several existing tools for deploying automata on FPGAs, most focus on memory-based DFA solutions. We found the FPGA automata processing framework Grapefruit [10] to be the best solution that provides both DFA and NFA functionality as well as a full end-to-end solution. Grapefruit also demonstrated higher performance over previous work such as REAPR. Grapefruit generates an explicit Hardware Description Language (HDL) module from a description of a homogeneous automaton. We extend Grapefruit to also generate HDL modules that represent logic transitions for symbolic non-homogeneous DFAs and NFAs.

A. Generating Finite Automata from LTL_f Formulas

The first half of our pipeline takes in an LTL_f formula φ and constructs an abstract non-homogeneous representation of a finite automaton that recognizes the language of φ. As previously mentioned, we explore two different constructions, one which produces a deterministic and another which produces a non-deterministic automaton. We start by describing the deterministic construction:

1) Translate the LTL_f formula φ into a formula $fol(\varphi)$ in First-Order Logic (FOL) over finite traces. This is possible since FOL has the same expressive power as LTL_f. A translation algorithm can be found in [22].
2) Use the tool MONA to convert $fol(\varphi)$ into a DFA A_φ that recognizes the same language. This is possible because MONA accepts inputs in MSO, which is a superset of FOL.

It is important to point out that the DFA constructed by MONA is minimal, meaning that it is the smallest DFA that recognizes the language.

Because the construction algorithm implemented in MONA heavily relies on the fact that DFAs can be efficiently minimized, the automaton output by the tool is always deterministic. Yet, it is known that there are languages for which the smallest DFA is exponentially larger than the smallest NFA [45]. Therefore, if our construction algorithm can exploit non-determinism, we may obtain an exponentially smaller automaton. Furthermore, recall that non-determinism allows us to take advantage of the natural parallelism among multiple active states in each automaton, as well as parallelism across multiple automata, and leverages the high degree of parallelism afforded by FPGAs. In order to use MONA to generate an NFA instead, we make use of a technique introduced in [46]:

1) Convert the LTL_f formula φ into a $PastLTL$ formula φ^R such that $\mathcal{L}(\varphi^R) = \mathcal{L}^R(\varphi)$, i.e., φ^R is satisfied by exactly those traces that are the reverse of a trace that satisfies φ. To do this, it is enough to replace all future temporal operators in φ with past temporal operators. See [25] for details.
2) Translate φ^R into a FOL formula $fol(\varphi^R)$ describing the same language. See [25] for a translation algorithm.
3) Use MONA to construct a DFA A^R_φ for $fol(\varphi^R)$. Note that this DFA accepts the reverse language of φ.
4) Reverse A^R_φ by turning initial states into accepting states and vice versa, and swapping the source and destination states of each transition. The result is an NFA A_φ that accepts the reverse language of A^R_φ, and therefore the same language of φ [46].

The minimal DFA for the reverse language of an LTL_f formula is guaranteed to be at most exponential in the size of the formula (see [22] on converting an LTL_f formula to a linear-sized alternating automaton, and [47] on obtaining an exponential-sized DFA for the reverse language of an alternating automaton). In contrast, the DFA for the formula itself can be doubly-exponential. Therefore, the NFA generated by this approach has the potential to be exponentially smaller than the DFA that would be constructed by simply using MONA directly.

B. Implementing Monitors in FPGA

Having obtained an automaton from the LTL_f formula, we explore two ways to implement them on an FPGA: *explicitly* or *symbolically*. In either case, each input signal of the circuit corresponds to a proposition in the formula, and multiple LTL_f formulas can be processed in parallel, up to the capacity of the FPGA.

The *explicit implementation* follows a similar architecture to REAPR and Grapefruit as presented in Section II-C. In this architecture, each state of the automaton is represented by a separate hardware module called a State Transition Element (STE). The STE consists of an activation bit and logic corresponding to the transition condition of this state (the explicit implementation is based on homogeneous automata, so the transition condition is associated with the state, not the edge). The activation bit for a state is set to 1 if any of its predecessors were active in the previous step and the current input satisfies the state's transition condition. Note that if the automaton is an NFA, multiple STEs can be active at the same time. The STE for an accepting state also generates a report bit. Given an automaton (DFA or NFA) A_φ generated by MONA, we perform the following operations to implement A_φ explicitly:

1) Convert A_φ from a non-homogeneous representation given in the output format of MONA into a homogeneous automaton in the ANML format.
2) Use Grapefruit to heuristically minimize the automaton (and remove unreachable states) and generate HDL.
3) Synthesize and target FPGA.

It is important to note that the conversion algorithm to homogeneous automaton may turn a non-homogeneous DFA into a homogeneous NFA, and may come at an increase in automata size. Therefore, when we refer to an "explicit DFA" implementation, we only mean that the automaton was initially constructed and minimized as a DFA, but the

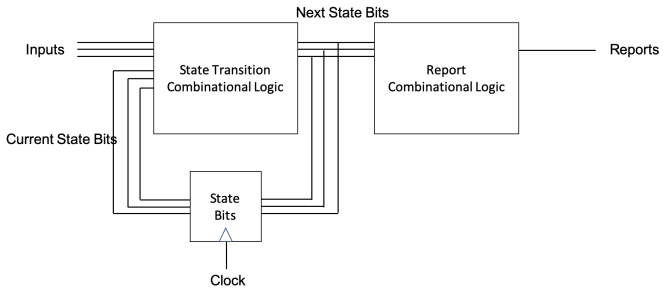

Fig. 3. Architecture of symbolic automata.

homogeneous automaton implemented in the FPGA may be non-deterministic. As a result, the main difference between the explicit DFA and explicit NFA approaches is that they produce automata with a different number of states and transition logic.

The *symbolic implementation* instead encodes the current state by a bitvector stored in an internal memory, and uses a single logic circuit to compute the next state as a function of the current state and inputs. For NFAs, the bitvector includes one bit for each automaton state, representing whether that state is active or not. For DFAs, since only one state is active at a given time, this representation would be inefficient. Instead, each state is given a binary encoding in a logarithmic number of bits. Since symbolic automata do not require separate components for each state, their hardware utilization is expected to be less than explicit automata. In order to capture transitions into accepting (or reporting) states, we use a separate piece of combinational logic to determine if the next state is accepting and generate a report signal that is set to 1 if the state is accepting and 0 otherwise. Figure 3 shows how we represent symbolic automata. The steps for implementing symbolic automata in an FPGA are the following:

1) Remove unreachable states if they exist (since MONA generates minimal DFAs, this may only happen in NFA construction),
2) Convert automaton representation given by MONA into truth tables, one for each state bit and one for the reporting bit. Each table maps the value of the current state and the current input to the new value of the state or reporting bit.
3) Use modified Grapefruit to convert truth tables into synthesizable HDL.
4) Synthesize and target FPGA.

We extend Grapefruit to generate symbolic DFAs and NFAs. We do this in two steps. In the first step, we generate an intermediate Truth Table representation (IR) from the MONA output. We generate a separate truth table for each of the state bits. Recall that the number of state bits for NFAs is linear, while for DFAs it is logarithmic. We found that when comparing DFAs to NFAs, the DFAs have fewer truth tables, but these truth tables required deeper logic circuits. Finally, we generate a separate truth table for the bits reporting the accepting states. This shallow truth table checks the state bits of the DFAs and NFAs and generates an output report signal if any of the accepting states are active.

We then generate Verilog lookup table modules from these IRs, to be synthesized into logical circuits in the FPGA. For state transitions, we use sequential logic with case statements. Each bit in the automaton state maps to its own module with a case statement mapping inputs and current state bits to next state bits. For the report truth tables, we use combinational case statements to map current state bits to a report signal. We use Grapefruit's hardware generator to connect all of the truth table modules to the shared input signals including a system clock, reset, and input symbol, as well as a unique output report signal for each accept state in the automata.

IV. Experimental Setup

A. Generating LTLf Formulas

In order to evaluate the effectiveness of our pipeline and different approaches for implementing runtime monitors in hardware, we generate a diverse set of LTL_f formulas of differing complexities. The FPGA synthesis and optimization tools optimize circuitry, including removing redundant hardware, and therefore it is not sufficient to duplicate the same formulas to evaluate scalability. To that end, we generate multiple different formulas by taking random combinations of the 18 LTL_f patterns from [23]. Each pattern is a simple formula with one or two variables. We take conjunctions of multiple small patterns, merging like variables among them, in order to generate more complex formulas. This process is repeated several times to create multiple formulas, which all draw randomly from a pool of common variables. That way, different formulas may have shared variables. All formulas are then converted to automata and implemented on the same FPGA. The four possible combinations described in the previous section (deterministic/explicit, non-deterministic/explicit, deterministic/symbolic, non-deterministic/symbolic) give us four quadrants for our experimental evaluation. We evaluate the performance of the formulas we generate for each of these four quadrants and compare among them.

In more detail, the LTL_f formulas used in our evaluation are generated in the following way:

1) Draw n random formulas from the pool of patterns.
2) For each variable of each pattern, draw an associated pattern variable from a pool of k shared input variables. Different variables in the same pattern are mapped to different shared input variables from the pool, but variables from different patterns can be mapped to the same input variable.
3) Take the conjunction of all n formulas, forming a more complex LTL_f formula.
4) Repeat this process m times with the same pool of shared input variables, producing m complex formulas with shared variables between them.

Each complex formula is then separately converted to an automaton and implemented on the FPGA, according to each of the four quadrants described previously. To evaluate how the architecture defined by each of the quadrants scales as the number and complexity of each rule increases, we vary the three parameters n, k and m above:

- **Number of formulas (n):** Number of separate LTL_f formulas implemented on the FPGA. We vary this parameter from 10-10,000.
- **Number of variables (k):** Size of the total pool of variables to be drawn from by the LTL_f formulas. We select 10 and 100.
- **Formula size (m):** Number of conjuncts per formula. We select 1, 3, and 5.

We experimentally determine the range of values for each of the three parameters. In the case of the number of formulas, we used AutomataZoo [48] as a reference with number of states up to approximately one million. We also determined that explicit automata hardware utilization scales rapidly with the number of variables, which maps to the number of input signals; for this reason we ran experiments for 10 and 100 variables. Finally, we tried to keep automata to a few thousand states, and therefore set a formula size cap to 5. We repeat each of these experiments three times, each time generating a new set of formulas, and we report the average of the three runs.

B. Hardware Setup

We target Amazon's cloud-deployed FPGAs to standardize on a publicly-available platform. Amazon provides Xilinx Ultrascale+ FPGAs in their F1 EC2 instances. In order to synthesize and place-and-route our HDL into a bitstream to configure the FPGA, we used Amazon's FPGA developer Amazon Machine Image (AMI), which provides us the FPGA software tools. For our experiments, we used Amazon FPGA Developer AMI version 1.6.0, which includes Vivado 2018.3. We deployed this AMI on Amazon EC2 c4.8xlarge instances.

V. Experimental Results

A. Comparisons Among Automata Implementations

We report the average results of the three runs in Figures 4, 5 and 6. Although transition density has low variance across these three runs, the variance in automaton sizes increases with the size of the formula. We leave a more detailed analysis of the distribution of automaton sizes to future work and focus here on a general analysis based on the average results.

Figures 4 and 5 show the number of Flip-Flops (FFs) and Look-Up Tables (LUTs) utilized by the FPGA for our randomly-generated formulas, composed of the conjunction of multiple patterns over random variables drawn from 10 binary system signals. FFs are used in the explicit implementations to store the bits indicating whether a state is active, and in the symbolic implementations to store the bitvector encoding the current state. LUTs correspond to logic gates and are used to implement transition and reporting logic. The Xilinx Virtex UltraScale+ VU9P has a total 1,181,768 LUTs and 2,363,536 Flip-Flops. Our results show that explicit automata, both DFAs and NFAs, tend to use more LUT hardware resources than symbolic automata. Our explicit NFAs tend to use fewer Flip-Flop and LUT resources than their DFA counterparts.

We determined that transforming the MONA-generated automata to homogeneous automata came at a significant cost in terms of number of states. For our 10-variable, explicit automata, we saw an increase in number of states from 2x in the case of 1-pattern automata to 10x for 5-pattern automata. This increase in states is due to the increase in edge density as automata become more complex. We found the homogeneous state increase to be be a flat multiplier as we scaled our number of formulas. When comparing to the majority of AutomataZoo benchmarks, which have edge/node densities below 2, our conjunctive LTL formula automata had average edge/node densities of 1.36 edges/node for 1 pattern, 4.53 edges/node for 3 patterns, and 8.89 edges/node for 5 patterns, with explicit DFAs and NFAs having approximately the same edge densities. We repeated this analysis with formulas composed of disjunctions instead of conjunctions and found edge/node densities of 1.36 edges/node for 1 pattern, 5.45 edges/node for 3 patterns, and 12.67 edges/node for 5 patterns.

In the case of symbolic automata, we found that symbolic NFAs tend to use more Flip-Flop resources but fewer LUT resources than deterministic implementations. This is due to symbolic NFAs being represented with a lookup-table module per bit in a linear bit-vector ($\mathcal{O}(n)$) representation of the automata, while the DFA implementation represented each bit in a logarithmic bit-vector ($\mathcal{O}(\log n)$) representation. While the implementation does parallelize the bit logic, the DFA logic depth tended to be deeper than NFA logic, resulting in higher clock frequency support for symbolic NFAs. Finally, Vivado was unable to place-and-route 10,000 automata of formula size 5 for any automata type. Each of these automata of formula size five were composed of 100s of states, and we ran out of resources for many of them.

Figure 6 shows the maximum clock frequencies at which the generated hardware monitors can process input signals. We implement our automata in out-of-context mode, which means that our solutions do not include input or output (I/O) circuitry. We removed I/O complications from our analysis as those decisions are application dependent, and can vary significantly in complexity as shown by I/O work by Bo et al [49] and in Grapefruit [10]. These results show that for a larger number of automata (100-10000), the explicit automata maintain a higher clock frequency than their symbolic counterparts. In the case of very small formulas or for very complex formulas, the explicit automata get larger faster and the symbolic implementations can be run at higher frequencies.

Our results are summarized in Figure 7. We find that if hardware utilization is a primary concern, symbolic automata tend to use less hardware than explicit automata. If minimizing Flip-Flip usage, symbolic DFAs are the best option, except in the case of smaller formulas. We see this behavior, because our NFA implementation uses a logic circuit per state in the automaton, while our DFA representation only needs a number of circuits that is the log of the number of states. This larger number of state bits results in a higher FF usage. If minimizing LUT usage, symbolic NFAs are the best option. Symbolic NFAs have more logic circuits, but each of these logic circuits are shallower than the DFA circuits, resulting in a reduced LUT usage. For our experiments, we found that the difference between architectures can result in up to a 5x increase in

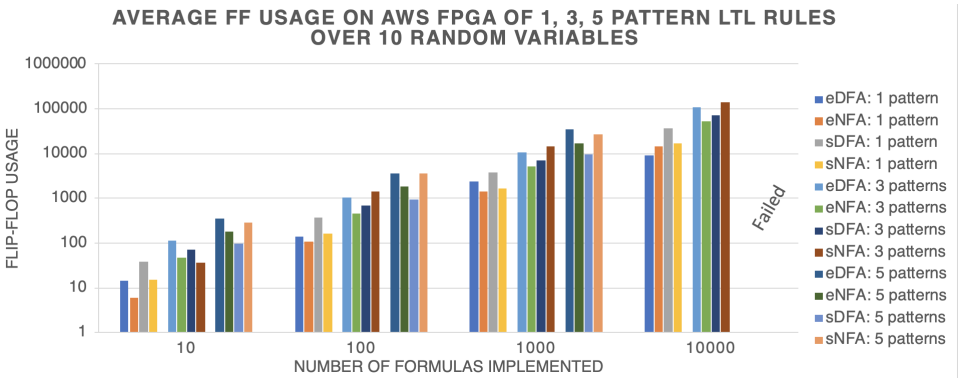

Fig. 4. Flip-Flop usage for for each automata type.

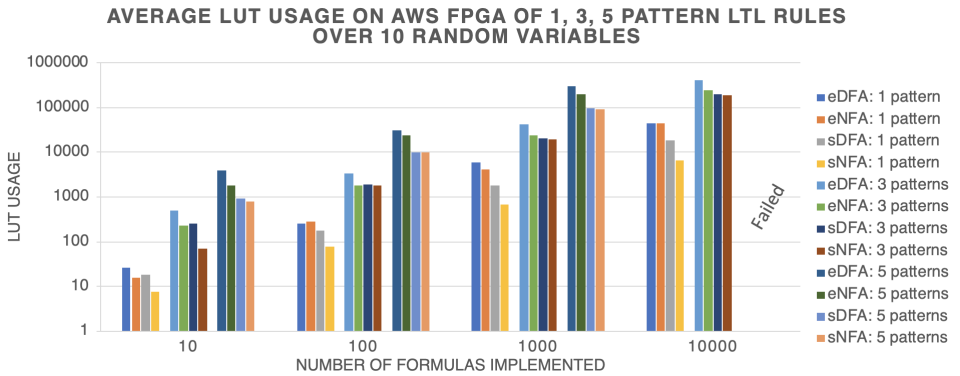

Fig. 5. LUT usage for for each automata type.

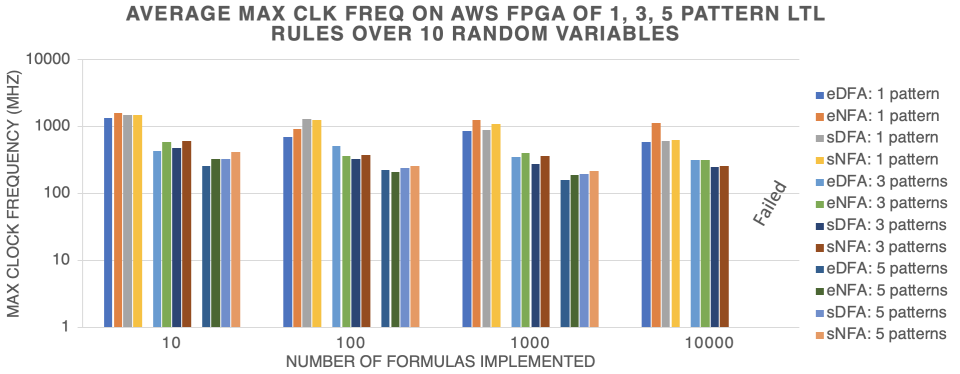

Fig. 6. Maximum clock frequency for each automata type.

	Deterministic FSA	Non-Deterministic FSA
Explicit		• Lower LUT and FF than eDFA • Higher Clk than eDFA, sDFA, sNFA
Symbolic	• Lower FF than sNFA, eNFA for larger automata • Matching LUT with sNFA for larger automata	• Lower LUT than sDFA, eDFA, eNFA • Lower FF than sDFA for smaller automata • Higher Clk than sDFAs

Fig. 7. Comparing explicit vs. symbolic and deterministic vs. non-determinstic automata implemented in hardware.

LUT and FF usage. If max clock frequency (throughput) is the primary concern, explicit NFAs maintain higher clock frequencies than symbolic automata for a larger number of automata. In the case of very small formulas or very complex formulas, symbolic implementations tend to run at higher frequencies likely due to the clock delay imposed by edges between explicit nodes. Across all of our experiments, we find that symbolic NFAs tend to perform best of all of our evaluated options, and that the difference between architectures can result in up to a 63% reduction in throughput. Similar results were obtained when we repeated the experiments replacing the conjunctions with disjunctions, and the same general conclusions apply. The most significant difference was that, likely due to the steeper scaling of edge/node density, disjunctions failed for 10,000 automata even with only 10 shared variables (k) and a formula size (m) of 3.

B. The Importance of NFA Minimization

Grapefruit includes a series of heuristic minimization techniques that allow us to significantly decrease the number

of states in our explicit NFAs. FPGA optimization tools are also applied by Vivado during the synthesis and place-and-route phases. We wanted to determine the effect of higher-level automata optimization on hardware utilization and performance, and synthesized and place-and-routed 3-pattern automata with and without Grapefruit optimizations. We observed that our generated explicit DFAs have fewer states than our generated explicit NFAs across most of our formulas, even post-minimization. We find that during the Cross Boundary and Area Optimization steps of synthesis that the NFA states were merged much more than the DFAs, resulting in a net result of less hardware utilization than DFAs. Although we did use Grapefruit's minimization functionality, the resulting automata are not necessarily minimal. We found that Grapefruit heuristics reduced our state count by between 4.5% and 11.0%, with LUT reductions from 4.3% to 8.2% as we scaled the number of automata from 10 to 10,000.

C. Wide Input Signals

One limitation of our explicit automata implementations is the required distribution of input signals to all of the states that make up the automata. FPGA optimization tools only route signals required by the transition logic for each state, but as formula complexity increases, more signals need to be routed to each state, resulting in significant hardware utilization.

We repeated our experiment with 100 input signals and found that even with simple single-pattern formulas, we were able to synthesize 10,000 symbolic automata formula, but only 1000 explicit automata. When moving complexity up to 3 patterns, we could still support over 10,000 symbolic automata, but fewer than 100 explicit automata. With 5 pattern complexity, we could only support 1000 symbolic automata, and could not synthesize even 10 explicit automata.

Wide input signals require serialization on the input, and handling report identification requires serialization on the output. Our analysis does not investigate I/O because it is implementation dependent. When monitors are monitoring an implementation on the FPGA, there may not be a need to transfer signals off the chip. Also, in the case of output signals, there are many approaches to handling monitoring solutions. If the application and monitoring resolution implementation is on chip, there may not be a need to remove report information off the chip. If this information does need to leave the chip, it might be sufficient to send off a single bit of information, as opposed to the entire report bit vector, as demonstrated with other FPGA-based automata implementations.

VI. Conclusions and Future Work

In this work, we introduce a framework for generating FPGA-deployed runtime monitors based on LTL_f formulas with four different architectures: explicit DFA, explicit NFA, symbolic DFA, symbolic NFA. We use our framework to determine performance tradeoffs among these. Our results show that there is no single best hardware representation for automata-based runtime verification, and that there are trade-offs between hardware utilization and the maximum clock frequency for automata transitions. Across all of our experiments, we find that symbolic automata tend to use less hardware (FFs and LUTs) than explicit automata, that explicit automata tend to run at higher frequencies (except in the case of very small or very complex formulas), and that symbolic NFAs tend to perform best of all of our evaluated architectures across the widest range of scenarios. Our experiments also showed us that differences between architectures can result in up to a 5x increase in LUT and FF usage, as well as result in up to a 63% reduction in runtime verification throughput.

We extended Grapefruit to also generate symbolic hardware automata. Although Grapefruit includes many other features including targeting Block RAM, full-stack support with I/O, and support for variable symbol width and striding, we did not use these functions in our experiments. Application-side research could further investigate concerns related to I/O and moving signal data to the automata as well as handling reporting data communication.

We targeted Amazon's F1, cloud-deployed FPGAs to standardize on one FPGA platform. Application-side research could utilize our work for integrating runtime monitors into high performance cloud-deployed applications, including machine learning and bioinformatics workloads. Our framework generates HDL that can target smaller and lower power FPGAs for other applications, including embedded systems. Because our explicit automata use the standard ANML format, automata engines built for other architectures can also be used.

We chose to keep the width of input signals constant across our experiments to determine the performance of our solution when all input signals are processed simultaneously. With flexibility in timing, or with slower sampling, future work could utilize Grapefruit's variable symbol-width functionality to handle many more input signals, albeit at a slower rate, thus making it possible to handle formulas with a much larger number of propositional variables. This would also significantly reduce hardware utilization.

In the future, it would also be interesting to compare with existing frameworks for implementing monitors in FPGA, such as [34], [39]. Since these works use different specification languages (e.g. PSL for [34] and STL and TRE for [39]), this would require establishing a unified set of benchmarks for these different formalisms and separating in the experimental evaluation the impact of the differences between specification languages from the performance of the FPGA framework.

During our analysis, we found that the average automata edges/node density scaled differently for conjunctions vs. disjunctions of patterns. We found that edge density for disjunctions tended to scale with a steeper slope than conjunctions. Future work could explore this relationship between compositions of LTL formulas and automata parameters.

Acknowledgments

Work funded by the NSF XPS and CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, NSF grants IIS-1527668, CCF-1704883, IIS-1830549, and an award from the Maryland Procurement Office.

REFERENCES

[1] K. Havelund, "Runtime verification of C programs," in *Testing of Software and Communicating Systems, 20th IFIP TC 6/WG 6.1 International Conference, TestCom 2008, 8th International Workshop, FATES 2008, Tokyo, Japan, June 10-13, 2008, Proceedings*, ser. Lecture Notes in Computer Science, K. Suzuki, T. Higashino, A. Ulrich, and T. Hasegawa, Eds., vol. 5047. Springer, 2008, pp. 7–22.

[2] D. Tabakov, K. Y. Rozier, and M. Y. Vardi, "Optimized temporal monitors for systemc," *Formal Methods in System Design*, vol. 41, no. 3, pp. 236–268, 2012.

[3] T. Reinbacher, K. Y. Rozier, and J. Schumann, "Temporal-logic based runtime observer pairs for system health management of real-time systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 357–372.

[4] J. Schumann, P. Moosbrugger, and K. Y. Rozier, "R2u2: monitoring and diagnosis of security threats for unmanned aerial systems," in *Runtime Verification*. Springer, 2015, pp. 233–249.

[5] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, 2018.

[6] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3282307

[7] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[8] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. T. II, J. Wadden, M. R. Stan, and K. Skadron, "An overview of micron's automata processor," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*, 2016, pp. 14:1–14:3.

[9] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "Reapr: Reconfigurable engine for automata processing," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–8.

[10] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An open-source, full-stack, and customizable automata processing on fpgas."

[11] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "Searching for potential grna off-target sites for crispr/cas9 using automata processing across different platforms," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 737–748.

[12] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the micron automata processor," *IEEE/ACM transactions on computational biology and bioinformatics*, vol. 13, no. 1, pp. 99–111, 2015.

[13] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning, "Towards machine learning on the automata processor," in *International Conference on High Performance Computing*. Springer, 2016, pp. 200–218.

[14] M. Putic, A. Varshneya, and M. R. Stan, "Hierarchical temporal memory on the automata processor," *IEEE Micro*, vol. 37, no. 1, pp. 52–59, 2017.

[15] E. Sadredini, D. Guo, C. Bo, R. Rahimi, K. Skadron, and H. Wang, "A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 665–674.

[16] A. Bauer, M. Leucker, and C. Schallhart, "Comparing ltl semantics for runtime verification," *J. Log. Comput.*, vol. 20, no. 3, pp. 651–674, 2010.

[17] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–64, 2011.

[18] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm, "Mona: Monadic second-order logic in practice," in *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Proceedings*, 1995, pp. 89–110.

[19] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0 — A Framework for LTL and ω-automata Manipulation," in *ATVA*, 2016.

[20] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Flexamata: A universal and efficient adaption of applications to spatial automata processing accelerators," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 219–234.

[21] H. Gruber and M. Holzer, "Computational complexity of nfa minimization for finite and unary languages." *LATA*, vol. 8, pp. 261–272, 2007.

[22] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013, pp. 854–860.

[23] G. De Giacomo, R. De Masellis, and M. Montali, "Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014, pp. 1027–1033.

[24] S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi, "Symbolic ltlf synthesis," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, pp. 1362–1369.

[25] S. Zhu, G. Pu, and M. Y. Vardi, "First-order vs. second-order encodings for ltlf-to-automata translation," in *Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019, Kitakyushu, Japan, April 13-16, 2019, Proceedings*, 2019, pp. 684–705.

[26] O. Lichtenstein, A. Pnueli, and L. Zuck, "The glory of the past," in *Logics of Programs*, ser. Lecture Notes in Computer Science, vol. 193. Springer, 1985, pp. 196–218.

[27] M. d'Amorim and G. Rosu, "Efficient monitoring of omega-languages," in *Proc, 17th Int'l Conf. on Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 3576. Springer, 2005, pp. 364–378.

[28] H. Björklund and W. Martens, "The Tractability Frontier for NFA Minimization," in *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, 2008, pp. 27–38.

[29] V. M. Glushkov, "The Abstract Theory of Automata," *Russian Math. Surveys*, vol. 16, pp. 1–53, 1961.

[30] R. McNaughton and H. Yamada, "Regular expressions and state graphs for automata," *IRE Trans. Electronic Computers*, vol. 9, no. 1, pp. 39–47, 1960.

[31] T. Tracy II, J. Wadden, T. Xie, K. Skadron, and M. Stan, "Accelerating design convergence of automata processing designs with a tiled hierarchy," in *FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers*. VDE, 2019, pp. 1–8.

[32] D. Drusinsky, "The temporal rover and the atg rover," in *International SPIN Workshop on Model Checking of Software*. Springer, 2000, pp. 323–330.

[33] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, "Copilot: monitoring embedded systems," *Innovations in Systems and Software Engineering*, vol. 9, no. 4, pp. 235–255, 2013.

[34] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of psl properties," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 1, pp. 1–21, 2008.

[35] J. Geist, K. Y. Rozier, and J. Schumann, "Runtime observer pairs and bayesian network reasoners on-board fpgas: flight-certifiable system health management for embedded systems," in *International Conference on Runtime Verification*. Springer, 2014, pp. 215–230.

[36] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the mop runtime verification framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[37] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware runtime monitoring for dependable cots-based real-time embedded systems," in *2008 Real-Time Systems Symposium*. IEEE, 2008, pp. 481–491.

[38] S. Jaksvic, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničković, "From signal temporal logic to fpga monitors," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2015, pp. 218–227.

[39] K. Selyunin, S. Jaksic, T. Nguyen, C. Reidl, U. Hafner, E. Bartocci, D. Nickovic, and R. Grosu, "Runtime monitoring with recovery of the sent communication protocol," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 336–355.

[40] K. Selyunin, T. Nguyen, E. Bartocci, and R. Grosu, "Applying runtime monitoring for automotive electronic development," in *International Conference on Runtime Verification*. Springer, 2016, pp. 462–469.

[41] J. Baumeister, B. Finkbeiner, M. Schwenger, and H. Torfah, "Fpga stream-monitoring of real-time properties," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–24, 2019.
[42] L. Convent, S. Hungerecker, T. Scheffel, M. Schmitz, D. Thoma, and A. Weiss, "Hardware-based runtime verification with embedded tracing units and stream processing," in *International Conference on Runtime Verification*. Springer, 2018, pp. 43–63.
[43] T. Tracy II and L. Tabajara, "Ltlfautomata," https://github.com/tjt7a/LTLfAutomata, 2020.
[44] S. Bansal, Y. Li, L. M. Tabajara, and M. Y. Vardi, "Hybrid compositional reasoning for reactive synthesis from finite-horizon specifications," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 9766–9774.
[45] M. Sipser, *Introduction to the theory of computation*. PWS Publishing Company, 1997.
[46] S. Zhu, L. M. Tabajara, G. Pu, and M. Y. Vardi, "On the Power of Automata Minimization in Temporal Synthesis," *CoRR*, 2020, [Online].
[47] A. Chandra, D. Kozen, and L. Stockmeyer, "Alternation," *J. ACM*, vol. 28, no. 1, pp. 114–133, 1981.
[48] J. Wadden, T. Tracy, E. Sadredini, L. Wu, C. Bo, J. Du, Y. Wei, J. Udall, M. Wallace, M. Stan *et al.*, "Automatazoo: A modern automata processing benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 13–24.
[49] C. Bo, V. Dang, T. Xie, J. Wadden, M. Stan, and K. Skadron, "Automata processing in reconfigurable architectures: In-the-cloud deployment, cross-platform evaluation, and fast symbol-only reconfiguration," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 2, pp. 1–25, 2019.