# Accelerating Legacy String Kernels via Bounded Automata Learning

Kevin Angstadt
University of Michigan
Computer Science and Engineering
Ann Arbor, MI, USA
angstadt@umich.edu

Jean-Baptiste Jeannin
University of Michigan
Aerospace Engineering
Ann Arbor, MI, USA
jeannin@umich.edu

Westley Weimer
University of Michigan
Computer Science and Engineering
Ann Arbor, MI, USA
weimerw@umich.edu

## Abstract

The adoption of hardware accelerators, such as FPGAs, into general-purpose computation pipelines continues to rise, but programming models for these devices lag far behind their CPU counterparts. Legacy programs must often be rewritten at very low levels of abstraction, requiring intimate knowledge of the target accelerator architecture. While techniques such as high-level synthesis can help port some legacy software, many programs perform poorly without manual, architecture-specific optimization.

We propose an approach that combines dynamic and static analyses to learn a model of functional behavior for off-the-shelf legacy code and synthesize a hardware description from this model. We develop a framework that transforms Boolean string kernels into hardware descriptions using techniques from both learning theory and software verification. These include Angluin-style state machine learning algorithms, bounded software model checking with incremental loop unrolling, and string decision procedures. Our prototype implementation can correctly learn functionality for kernels that recognize regular languages and provides a near approximation otherwise. We evaluate our prototype tool on a benchmark suite of real-world, legacy string functions mined from GitHub repositories and demonstrate that we are able to learn fully-equivalent hardware designs in 72% of cases and close approximations in another 11%. Finally, we identify and discuss challenges and opportunities for more general adoption of our proposed framework to a wider class of function types.

***CCS Concepts.*** • **Computer systems organization** → *Reconfigurable computing*; • **Theory of computation** → **Regular languages**; • **Software and its engineering** → **Software design techniques**.

***Keywords.*** automata learning, automata processing, legacy programs

**ACM Reference Format:**
Kevin Angstadt, Jean-Baptiste Jeannin, and Westley Weimer. 2020. Accelerating Legacy String Kernels via Bounded Automata Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378503

## 1 Introduction

The confluence of several factors, including the significant increase in data collection [34], demands for real-time analyses by business leaders [28], and the lessening impact of both Dennard Scaling and Moore's Law [73], have led to the increased use of hardware accelerators. Such devices range from Field-Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) to more esoteric accelerators, such as Google's Tensor Processing Unit or Micron's D480 Automata Processor. These accelerators trade off general computing capability for increased performance on specific workloads; however, successful adoption requires additional architectural knowledge for effective programming and configuration. Support for acceleration of legacy programs that limits the need for architectural knowledge or manual optimization would only aid the adoption of these devices.

While present in industry for prototyping and application-specific deployments for quite some time, reconfigurable architectures, such as FPGAs, are now becoming commonplace in everyday computing. In fact, FPGAs are in use in Microsoft datacenters and are also widely available through Amazon's cloud infrastructure [2, 25, 49].

Adopting hardware accelerators into existing application workflows requires porting code to these new programming models. Unfortunately, porting legacy code remains difficult. The primary programming model for FPGAs remains Hardware Description Languages (HDLs) such as Verilog and

VHDL. HDLs have a level of abstraction akin to assembly-level development on traditional CPU architectures. While these hardware solutions provide high throughputs, programming can be challenging—programs written for these accelerators are tedious to develop and difficult to write correctly.

Higher levels of abstraction for programming FPGAs have been achieved with high-level synthesis (HLS) [61], languages such as OpenCL [79], and frameworks such as Xilinx's SDAccel [95]. While HLS may allow existing code to compile for FPGAs, it still requires low-level knowledge of the underlying architecture to allow for efficient implementation and execution of applications [85, 99]. In fact, optimization techniques for OpenCL programs have been demonstrated to have detrimental impacts on performance when software is ported to a new architecture [10]. Therefore, there is a need for a new programming model that supports porting of legacy code, admits performant execution on hardware accelerators, and does not rely on developer architectural knowledge. We focus our efforts on automata computations, a class of kernels with applications ranging from high-energy physics [93] to network security [71] and bioinformatics [68, 69, 83], for which hardware acceleration is both strongly desired and also capable of providing significant performance gains [8, 33, 35, 36, 39, 80, 88, 91, 96].

In this paper, we present AutomataSynth,[1] a new approach for executing code (including legacy programs and automata computations) on FPGAs and other hardware accelerators. Unlike HLS, which statically analyzes a program to produce a hardware design, AutomataSynth both dynamically observes and statically analyzes program behavior to synthesize a *functionally-equivalent* hardware design. Our approach is based on several key insights. First, state machines provide an abstraction that has successfully accelerated applications across many domains [68–71, 82, 83, 90, 92, 93, 98] and admit efficient implementations in hardware [33, 88, 96], but typically require rewriting applications. Second, there is an entire body of work on query-based learning of state machines (e.g., see Angluin for a classic survey of computational learning theory [7]), but these algorithms commonly rely on unrealistic oracle assumptions. Third, we observe that the combination of software model checking (e.g., [15, 16]) and recent advances in string decision procedures (e.g., [31, 47, 81, 84]) can be used in place of oracles for certain classes of legacy code kernels, such as those that recognize regular languages.

While AutomataSynth is based on a general approach for synthesizing hardware designs from high-level source code, we focus in this paper specifically on synthesizing Boolean string kernels (functions that return true or false given a string). We propose to accelerate these string kernels using automata processing, which requires representing functions as finite automata [33, 88, 96]. We demonstrate how software model checking, using a novel combination of bounded model checking with incremental loop unrolling augmented with string decision procedures, can answer oracle queries required by Angluin-style learning algorithms, resulting in a framework to iteratively infer automata corresponding to legacy kernels.

We evaluate our prototype implementation of AutomataSynth on a benchmark suite of string kernels mined from public repositories on GitHub. Our evaluation demonstrates that our approach is viable for small functions and exposes new opportunities for improving current-generation tools. We identify four key challenges associated with using state-of-the-art methods to compile legacy kernels to FPGAs and suggest paths forward for addressing current limitations.

In summary, this paper makes the following contributions:

- AutomataSynth, a framework for accelerating legacy string kernels by learning equivalent state machines. We extend an Angluin-style learning algorithm to use a combination of iterative bounded software model checking and string decision procedures to answer oracle queries.
- A proof that AutomataSynth terminates and is correct (i.e., relatively complete with respect to the underlying model checker) for kernels that recognize regular languages. The proof leverages the minimality of machines learned by L* and the Pumping Lemma for regular languages.
- An empirical evaluation of AutomataSynth on 18 indicative kernels mined from public GitHub repositories. We learn 13 exactly-equivalent models and 2 near approximations.

## 2 Background and Related Work

AutomataSynth employs a novel combination of tools and techniques from multiple computing disciplines. To the best of our knowledge, this is the first work to combine model learning algorithms, software verification, and architectures for high-performance automata processing. We position our work in the context of related efforts from each area and provide background helpful for understanding our approach.

### 2.1 Finite Automata

In this work, we model the behavior of legacy source code using *deterministic finite automata* (DFAs) to enable efficient acceleration with FPGAs. Formally, a DFA is a five-tuple, $(Q, \Sigma, q_0, \delta, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite alphabet of input symbols, $q_0 \in Q$ is the initial (or starting) state, $\delta : Q \times \Sigma \rightarrow Q$ is a transition function encoding transfer of control between states based on an observed input symbol, and $F \subseteq Q$ is a set of accepting states. A DFA processes input data through the repeated application of the transition function with each subsequent symbol in the

[1] https://github.com/kevinaangstadt/automata-synth

input string. After the application of the transition function, a single state within the DFA becomes active. If an accepting state is active after all input characters have been processed, the DFA *accepts* the input (i.e., the input matches the pattern encoded by the DFA).

## 2.2 Accelerators for Finite Automata

There is substantial research on the acceleration of finite automata computation. Automata processing is considered an extremely challenging task to accelerate and is part of the "thirteenth dwarf" in the Berkeley parallel computation taxonomy [11]. Reconfigurable computing has emerged as a suitable platform for accelerating this form of computation [64, 91]. Automata have enabled the acceleration of a wide variety of applications across many domains, including: natural language processing [98], network security [71], graph analytics [70], high-energy physics [93], bioinformatics [68, 69, 83], pseudo-random number generation and simulation [90], data-mining [92], and ML [82].

Xie et al.'s REAPR framework enables high-throughput automata processing using modern FPGAs [96]. Additionally, application-specific accelerators for various types automata processing have been proposed, such as Micron's AP [33], the Cache Automaton [80], ASPEN [8], and other ASICs [35, 36, 39, 88]. While accelerators for state machine computation are performant, they require that input problems be phrased in an explicit state machine model, which is uncommon in extant software. Indeed, writing an automaton has been demonstrated to be error-prone and difficult [4, 77], thus leaving an abstraction gap and hindering widespread adoption of automata processing accelerators.

Our development of AutomataSynth is complementary to these efforts: we enable the "compilation" of legacy source code for execution on—and acceleration with—automata processing architectures.

## 2.3 State Machine Learning Algorithms

We briefly summarize learning of state machines here, detailing the most relevant instance in Section 3.1. These algorithms are a subset of *model learning* in learning theory and have been the subject of study for several decades [5, 30, 78, 86]. The most common approach is to use *active learning* in which the model is learned by performing experiments (tests) on the software or system to be learned. State machine learning has been applied to the domains of internet banking [1], network protocols [32, 37], legacy systems [54, 72], and describing machine learning classifiers [94].

Most efforts have focused on developing suitable algorithms for learning finite automata [6, 19]. More recent advances simplify the internal data structures of the algorithms, reduce the number of tests necessary to learn a model, or combinations thereof [43, 44, 48, 67]. Learning an equivalent state machine from software remains challenging, and most approaches employ some form of approximation [6, 52].

In this work, we apply this body of model learning research to the problem of adapting legacy source code for efficient execution on hardware accelerators. Our approach attempts to learn a model that is fully equivalent to the original program using software verification techniques, but may also produce approximate results in some situations.

## 2.4 Program Synthesis and Verification

*Program synthesis* is a holistic term for automatically generating software from some input description. Recent efforts have focused on different applications of synthesis, such as sketching [3, 75, 76], programming by example [40], and automated program repair [56, 63]. Many of these approaches employ *counterexample-guided inductive synthesis* (CEGIS) to produce a final solution [76]. CEGIS is an iterative technique that constructs candidate solutions that are tested (typically via formal methods) for equivalence. A *counterexample*, or model of undesirable behavior, is provided if the candidate solution is incorrect, and begins the next iteration of synthesis. We note that CEGIS is largely equivalent to the techniques used in the learning theory community for model learning.

Program verifiers and *software model checkers* prove that a program adheres to a specification or produce counterexamples that violate the specification [15]. These tools typically interleave the control flow graph (CFG) and a specification automaton and explore the resulting graph to determine if any path leads to an error state in the specification.

There has been significant research and engineering effort applied to making these techniques scalable and applicable to real applications [17, 27, 55]. Of particular relevance here are bounded or iterative techniques that address recursive control flow [18, 45], which typically unroll loops a fixed number of times before checking if an error state is reachable in the straight-line portion of the CFG. Most closely related to our work has been the use of bounded model checking to verify string-processing web applications; however, this work often focused on secure information flow rather than constraints over strings [42]. There are also theoretical results on the decidabilty of straight-line programs on strings, which naturally arise in bounded model checking [53].

A related body of research focuses on extracting program behavior from legacy code for acceleration using domain-specific languages (DSLs), an approach referred to as *verified lifting*. Examples, include extracting stencil computations [46, 57], database qeuries [26], and sparse and dense linear algebra calculations [38]. By targeting DSLs, verified lifting can leverage known properties of the given problem domain to aid extraction and acceleration. Our development of AutomataSynth complements these efforts: we focus on learning hardware descriptions of string kernels for acceleration with FPGAs. For generality, we intentionally limit the domain-specific assumptions leveraged by our approach.

We propose a novel combination of insights from counterexample guided synthesis, theorem proving and automata learning to synthesize behaviorally-equivalent automata from legacy source code in Section 3.3.

### 2.5 High-Level Synthesis for FPGAs

High-Level Synthesis (HLS) allows development for FPGAs at a much higher level of abstraction than HDLs [61]. Indeed, HLS has been demonstrated to reduce the time to develop FPGA designs [51]. Most tools support programs written in C-like languages, suggesting that HLS would be amenable for adapting and accelerating legacy code bases. However, the performance of designs constructed using HLS can be unimpressive, requiring significant optimization [85, 99]. HLS tools may also not support all features of the language (e.g., dynamic data structures), meaning that legacy code must be refactored before the approach is applicable.

In this work, we present an alternative to HLS that decouples the existing design and implementation of legacy code from the final design produced for an FPGA. In doing so, we avoid many of the limitations of HLS techniques.

## 3 Learning State Machines from Legacy Code

We propose AutomataSynth, a framework for learning functional behavior models for off-the-shelf, legacy code implementing regular languages and synthesizing hardware descriptions from those models. Our approach extends Angluin's L* algorithm [6] by (1) using bounded software model checking with incremental unrolling to implement one of its assumptions, (2) using software testing to implement another of its assumptions, and (3) transforming learned models into homogeneous DFAs for hardware synthesis.

### 3.1 L* Primer

Dana Angluin's foundational L* algorithm was popularized in 1987 [6]. Because many of our framework decisions (such as how to implement its required queries and counterexamples in a legacy source code context) and results (such as correctness and termination arguments) depend on the steps and invariants of her algorithm, we sketch it here in some detail. We claim no novelty in this subsection and readers familiar with L* can proceed to Section 3.2.

At its core, the L* algorithm relies on a *minimally adequate teacher* (MAT) to answer two kinds of queries about a held-out language, $L$. First, the MAT must answer *membership* queries, yielding a Boolean value indicating if the queried string is a member of $L$. Second, the MAT must answer *conjecture* or *termination* queries.[2] Given a candidate regular language $A$, typically expressed as a finite state machine, the MAT responds with `true` if $A = L$ or responds with a *counterexample* string for which $A$ and $L$ differ. (Note that automata learning is used in applications where $L$ is not a DFA, and thus this query is typically not resolved by standard DFA equivalence checking.)

[2]These are also called equivalence queries, but we avoid this term to prevent confusion with similar uses of the term in software verification.

These queries are used to construct an *observation table* that can be transformed directly into a DFA. This table may be defined as a 3-tuple, $(S, E, T)$, where $S$ is a nonempty, finite, prefix-closed[3] set of strings over $\Sigma$; $E$ is a nonempty, finite, suffix-closed set of strings over $\Sigma$; and $T$ is a function mapping $((S \cup S \cdot \Sigma) \cdot E)$ to $\{\texttt{true}, \texttt{false}\}$. $(S, E, T)$ may be visualized as a two-dimensional array where rows are indexed by a value $s \in S \cdot \Sigma$, columns are indexed by a value $e \in E$, and entries are equal to $T(s \cdot e)$. For ease of notation, Angluin defines $row(s)$ to be a finite function, $f$, mapping values from $E$ to $\{\texttt{true}, \texttt{false}\}$ defined as $f(e) = T(s \cdot e)$. Informally, $row(s)$ denotes the values in a particular row of the observation table.

An observation table must be both *closed* and *consistent* before a DFA may be correctly constructed. A table is *closed* if for every $t \in S \cdot \Sigma$, there exists an $s \in S$ such that $row(t) = row(s)$. A table is *consistent* if, for all $s_1, s_2 \in S$ where $row(s_1) = row(s_2)$, $row(s_1 \cdot a) = row(s_2 \cdot a)$ for all $a \in \Sigma$. These properties ensure that there is a valid transition out of each state in the DFA (closed) and that transitions on any character remain the same regardless of the characters already processed (consistent). Given a closed and consistent observation table, a DFA over the alphabet $\Sigma$ may be constructed as follows:

$$\begin{aligned}
Q &= \{row(s) \mid s \in S\}\,,\\
q_0 &= row(\varepsilon),\\
F &= \{row(s) \mid s \in S \wedge T(s) = \texttt{true}\}\,,\\
\delta(row(s), a) &= row(s \cdot a).
\end{aligned}$$

Each unique row in the observation table becomes a state in the candidate automaton, and outgoing transitions from a state are defined by the row indexed by the current row's prefix concatenated with the transition character.

Pseudocode for the L* algorithm is provided in Algorithm 1. A closed, consistent observation table is constructed using membership queries. Then, the table is transformed into a candidate automaton for a termination query. If the MAT responds with a counterexample, the counterexample and its prefixes are added to the observation table. The process repeats until the MAT responds to a termination query in the affirmative. The final automaton is the learned language.

### 3.2 AutomataSynth Problem Description

In this subsection, we formalize the problem of learning a state machine from a legacy string kernel.

AutomataSynth operates on a function that takes one string argument and returns a Boolean value:

```
kernel : string -> bool
```

[3]A set is prefix-closed if $\forall s \in S$, every prefix of $s$ is also a member of $S$. Suffix-closure is defined similarly.

**Algorithm 1: Angluin's L* Learner [6]**

```
Data: MAT for held-out language, $L$
Result: A DFA, $M$, representing the held-out language, $L$
Initialize $S$ and $E$ to $\{\varepsilon\}$;
Ask membership queries for $\varepsilon$ and each $a \in \Sigma$;
Construct initial observation table $(S, E, T)$;
repeat
    while $(S, E, T)$ is not closed or not consistent do
        if $(S, E, T)$ is not consistent then
            Find $s_1, s_2 \in S, a \in \Sigma, e \in E$ such that
              $row(s_1) = row(s_2) \wedge T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$;
            Add $a \cdot e$ to $E$;
            Extend $T$ to include the new suffix with membership
              queries;
        end
        if $(S, E, T)$ is not closed then
            Find $s_1 \in S, a \in \Sigma$ such that $row(s_1 \cdot a) \neq row(s)$ for
              all $s \in \Sigma$;
            Add $s_1 \cdot a$ to $S$;
            Extend $T$ to include the new prefix with membership
              queries;
        end
    end
    Construct DFA, $M$ from $(S, E, T)$;
    Make termination query with $M$;
    if MAT responds with counterexample $t$ then
        Add $t$ and all prefixes to $S$;
        Extend $T$ to include the new prefixes using membership
          queries;
    end
until the MAT responds with true to the termination check on $M$;
return DFA $M$
```

We assume that the source code for this function is provided and that the function halts and returns a value on all inputs (i.e., `kernel` is an algorithm). If `kernel` recognizes a regular language, AutomataSynth returns a state machine, $M$, with equivalent behavior to `kernel`. That is, for all $s \in \Sigma^*$, $M(s) = \texttt{kernel}(s)$. For runs which exceed a resource budget or expose incompleteness in the underlying theorem prover (including functions that are non-regular), our prototype implementation alerts and provides *approximate* equivalence, where $M(s) = \texttt{kernel}(s)$ when the length of $s$ is less than an arbitrary fixed length (see Section 4).

In Section 4.4, we present a formal proof that our framework produces an equivalent DFA for input kernels that recognize regular languages. Our empirical evaluation in Section 6 demonstrates that real-world legacy string kernels either recognize regular languages, or our tool can produce an approximation of the original function. We discuss the challenges associated with supporting a broader class of functions in Section 7.1.

## 3.3 Using Source Code as a MAT

We propose extending Angluin's L* algorithm to learn a DFA representation of a legacy string kernel. To succeed, we must construct a MAT that can answer membership and termination queries about an input string kernel. While the L* algorithm provides a framework for query-based DFA learning, the original work does not define any one mechanism for implementing the teacher. Our proposed MAT implementation leverages the source code of the target function.

***Membership Queries.*** We observe that a membership query for a string, $s$, may be implemented by executing the legacy kernel on $s$. The result returned by the function is the answer to the query. For languages akin to C employing integers, we interpret Boolean values in the standard way (i.e., 0 is `false` and all other values are `true`). While direct and intuitive in theory, we note that there are several challenges in practice. Following the C standard, many runtime systems assume that strings are null-terminated (i.e., a null character must only occur as the final character in a string). In practice, however, we find that legacy string kernels will sometimes allow null characters in other positions. This often occurs when the length of the input string is known *a priori*. While seemingly innocuous, this deviation from the standard limits the runtime mechanisms by which the legacy kernel may be invoked. We found that compiling the kernel to a shared object and then invoking the function dynamically provided the best stability in our experiments.

***Termination Queries.*** At the heart of our problem formulation is the challenge that a legacy string kernel does not admit a direct means for answering termination queries. A recent survey of model learning indicates that testing for equivalence queries [86]; however, our initial efforts found testing alone to be unsuitable for termination queries in this domain. Our insight is that verification strategies from software model checking may be used to test for equivalence between the kernel and a candidate automaton. Traditionally in verification, equivalence would be proven using bisimulation or interleaving of the automaton and the source kernel. However, this formulation presupposes that the "state transitions" are directly encoded in the source code and can be aligned with the state transitions in the candidate automaton. We do not make this assumption in our problem definition in Section 3.2, and we prefer an approach that does not require manual annotation. Indeed, we do not even assume that the states of the equivalent automaton are visited "in order" during the execution of the legacy kernel.

We propose to verify an alternate reachability property that places additional constraints on the input string. In particular, we observe that a counterexample $t \in \Sigma^+$ is in either $L(\texttt{kernel})$ or $L(M)$ but not in both, and thus will always satisfy the constraint $t \in L(\texttt{kernel}) \oplus L(M)$, where $\oplus$ is the symmetric difference operator. Therefore, we ask the software verifier to prove that there is no execution of `kernel` on $t$ where `kernel` returns `true` and $t \notin L(M)$ *or* `kernel` returns `false` and $t \in L(M)$. To test this reachability property, we use a novel combination of bounded model checking with incremental loop unrolling augmented with a string constraint solver. We discuss the implementation of this verification task in depth in Section 4.

Software verifiers are *relatively complete* (e.g., [12]), meaning that there are certain programs that cannot be fully verified due to limitations in the underlying SMT solvers. Verifiers often return an answer in three-valued logic: `true` in our application means that the kernel and candidate automaton were proved equivalent, allowing for termination; `false` in our application means that the property was not satisfied, and there is a counterexample to provide to the L* algorithm; and `unknown` in our application means that the verifier was unable to prove equivalence, but also does not provide a counterexample. In the case of an `unknown` answer from the verifier, we halt our algorithm and warn the user that the resulting automaton is *approximate*; there may be inputs for which the automaton returns an incorrect answer.

## 3.4 Synthesizing Hardware Descriptions from Automata

Once a state machine has been learned using the L* algorithm with our custom MAT, the kernel is now amenable to acceleration. There has been a significant effort to accelerate automata using FPGAs [96] and other custom ASICs (e.g., GPUs [91, 97] and Micron's AP [33]). We convert the learned automaton to a hardware description and synthesize the design for loading onto an FPGA. Other execution platforms are possible [9], but we focus on FPGAs in this work due to their widespread deployment.

## 3.5 System Architecture

Figure 1 depicts the high-level system architecture of our framework. The L* learner (shown to the left) queries a MAT (shown to the right) consisting of the legacy source code, software model checker, SMT solver, and string decision procedure. The legacy string kernel is used by the MAT to answer membership queries. Termination queries are transformed by a mapper into a software verification problem that searches for string that distinguish the language of a candidate automaton from the target language implemented in the kernel. The output of the Learner is a DFA that encodes the same computation as the Kernel. We use this DFA to synthesize a hardware design for execution on an FPGA.

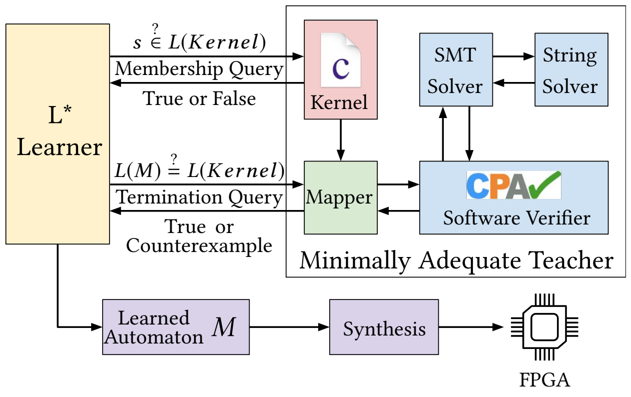

**Figure 1.** AutomataSynth System Architecture. The Minimally Adequate Teacher uses the legacy kernel to answer membership queries. The kernel is combined with a candidate automaton in the mapper to produce a software verification problem. Using bounded software model checking combined with string decision procedures, we search for a counterexample that distinguishes the target language from the language of the candidate automaton. Finally, we synthesize the learned automaton for execution on an FPGA.

# 4 Implementation and Correctness

In this section we lay out formal properties of our implementation, first demonstrating that iterative, bounded software model checking conforms to the required properties for MAT termination queries. We then prove correctness and termination for AutomataSynth. Because AutomataSynth operates on arbitrary source code and employs theorem proving techniques, correctness and termination are *relative*.

## 4.1 Bounded Model Checking

There are several SMT-based model checking algorithms that have been employed to verify properties of software. We use bounded model checking, an algorithm best-suited for the queries currently supported by string constraint solvers. In particular, string solvers do not currently support most interpolation queries (e.g., [29]), which are used heavily by counterexample-guided [27] algorithms such as SLAM and BLAST [13, 15]. Developing interpolation algorithms that support string constraints is beyond the scope of this work.

*Bounded model checking* enumerates all program paths up to a certain bound that reach a target error state (e.g., an error label in the source code) [18]. For each path, the algorithm generates a set of constraints over the program's variables, and the disjunction of these constraints is passed to an SMT solver to determine if the constraints for at least one path are satisfiable. If so, the set of variable assignments represents a configuration of the program that would result in an error condition at runtime. In this approach, loops are unrolled a fixed number of times (the "bound"). We follow the standard practice of incremental unrolling (cf. [20]), which iteratively applies bounded model checking for increasing unrolling depths. For programs with unbounded loops this strategy results in a semi-algorithm; however, we demonstrate in Section 4.4 that there exists a finite unrolling that fully verifies a kernel deciding a regular language for our property.

## 4.2 Reasoning about Strings

As described in Section 3.3, AutomataSynth must verify that there are no strings in the symmetric difference of the legacy kernel and a candidate automaton. We propose encoding the language of the candidate automaton as a regular expression constraint on the input string parameter of the kernel. We then solve the encoded problem using a bounded model checker that reasons about strings. A suitable string decision procedure must support (at minimum):

- Unbounded string length,
- Regular expression-based constraints over strings,
- Access to individual characters of strings,
- Comparison of individual characters and strings,
- Reasoning about the length of strings, and
- The ability generate strings that satisfy a set of constraints.

Additional features supported by string decision procedures can be helpful for representing standard library string functions. Recent decision procedures, such as Z3str3 [14] or S3 [84], support these required properties.

To combine bounded model checking with string decision procedures, we extend the CPAChecker extensible program analysis framework [16] to generate and solve string constraints. We modify CPAChecker's predicate analysis algorithm to generate "String" sort constraints for string-like types in C programs (e.g., `char` and `char*` types [62]). We produce a character extraction constraint for each occurrence of indexing of (and dereferencing) string variables. Additionally, we add support for functions such as `strlen`.

The C language specification does not directly support regular expressions. To embed these constraints in a program's source code, we also add an additional function for checking if a string variable conforms to a regular expression.

## 4.3 Verification for Termination Queries

Listing 1 demonstrates our formulation of termination queries using bounded model checking with incremental loop unrolling and string decision procedures. We construct a wrapper around the source code for the kernel that adds additional assertions to the path constraints used by the software verifier. When the kernel returns `true`, we add the constraint that the input string cannot be represented by the regular expression representing the candidate automaton. We add a similar constraint for the `false` case as well. Finally, we ask the verifier to prove that the error label (line 23) is unreachable (note that `assume` constraints influence reachability).

## 4.4 Correctness

In this subsection, we conclude our formal development of termination queries based on the combination of bounded model checking with incremental loop unrolling and string decision procedures. We demonstrate that this approach satisfies the Angluin constraints for MAT termination queries

```
1  // KERNEL is the legacy kernel function
2  // REGEX is the language of the candidate automaton
3  int termination_query(char* input) {
4
5    // call the kernel and record result
6    int ret = KERNEL(input);
7
8    if (ret) {
9      // if kernel accepts, candidate DFA must reject
10     __VERIFIER_assume(
11       !__VERIFIER_inregex(input, REGEX)
12     );
13     goto ERROR;
14   } else {
15     // if kernel rejects, candidate DFA must accept
16     __VERIFIER_assume(
17       __VERIFIER_inregex(input, REGEX)
18     );
19     goto ERROR;
20   }
21
22   // Error state to prove unreachable
23   ERROR:
24   return ret;
25 }
```

**Listing 1.** Formulating termination queries as a software verification problem. We embed regular expression constraints to force the legacy kernel and candidate automaton to disagree on the input string. If the return statement is unreachable, the two representations are equivalent. Otherwise, there is a string counterexample that can be used to continue the L* algorithm.

(see Section 3.1). In particular, we prove that this algorithm always halts with a counterexample or proof of equivalence between the legacy string kernel and a candidate automaton (assuming the underlying decision procedure is correct). While incremental unrolling is a semi-algorithm in general, we demonstrate that a finite unrolling is sufficient to prove equivalence *for pure programs that decide a regular language* (i.e., programs that both recognize a regular language and halt on all inputs as described in Section 3.2). We assume that the program is pure to avoid nondeterministic behavior and side-effects (e.g., nondeterministic behavior resulting from I/O) and that the program decides a regular language to leverage formal results from automata theory. While these assumptions restrict the class of programs to which our formal result applies, we note that AutomataSynth can handle more complex functions, but may produce approximate results. Ultimately, our goal is to prove the following theorem:

**Theorem 4.1.** *Let $K$ be a pure program that decides a regular language $L(K)$. There exists a finite unrolling $K'$ of $K$ such that if $M$ is the candidate DFA learned by $L^*$ from $K'$, then $K \equiv M$.*

We write ($\equiv$) to denote equality of accepted languages, i.e., $K \equiv M$ if and only if $L(K) = L(M)$. We will prove this theorem using a sequence of lemmas as well as theoretical results from the L* algorithm. First, we demonstrate that there exists a finite unrolling of a program $K$ that recognizes all strings in $L(K)$ shorter than a given length. The intuition is that the finite unrolling $K'$ corresponds to the use of *bounded* model checking.

**Lemma 4.2.** *Let $p \in \mathbb{N}$ and $K$ be a program that recognizes a subset of $\Sigma^*$. There exists an $n \in \mathbb{N}$ such that the $n$-finitely-unrolled program $K'$ obtained from $K$ (with all loops replaced with the finite unrolling of the first $n$ iterations and all subsequent iterations removed) satisfies $\forall s \in \Sigma^*, |s| < p \implies K'(s) = K(s)$.*

*Proof.* Given $p \in \mathbb{N}$, we construct the set of strings $S = \{s \mid s \in \Sigma^* \wedge |s| < p\}$, on which $K'$ must agree with $K$. We let $n$ be the maximum number of iterations performed by $K(s)$ for all $s \in S$. Because $S$ is a finite set, its maximum value is guaranteed to exist and be finite. We construct $K'$ by unrolling $n$ times the program $K$. By construction, the property $\forall s \in \Sigma^*, |s| < p \implies K'(s) = K(s)$ holds. □

We also reason using the standard Pumping Lemma for regular languages. For reference, we recall the Lemma here without proof as defined by Sipser [74, Theorem 1.70].

**Lemma 4.3** (Pumping Lemma for Regular Languages)**.** *If $A$ is a regular language, then there is a number $p$ such that for all $s \in A$, if $|s| > p$ then $s$ may be divided into three pieces, $s = xyz$, satisfying the following conditions: for each $i \geqslant 0, xy^iz \in A$, $|y| > 0$, and $|xy| \leqslant p$.*

The smallest such $p$ is called the *pumping length*. We call out as a Lemma the association between pumping lengths and minimality [74, Proof of 1.70]:

**Lemma 4.4.** *The (smallest) pumping length of a regular language $L$ is equal to the number of states in the* minimal *DFA that recognizes $L$.*

Additionally, our proof makes use of two theorems about the output of the L* algorithm. We paraphrase these results here [6]. See Section 3.1 for L* definitions, such as $(S, E, T)$.

**Theorem 4.5** (L* [6], Theorem 1)**.** *If $(S, E, T)$ is a closed, consistent $L^*$ observation table, then the DFA $M$ constructed from $(S, E, T)$ is consistent with the finite function $T$. Any other DFA consistent with $T$ but not equivalent to $M$ must have* more *states.*

We will use the following corollary of this result.

**Corollary 4.5.1.** *Let $p$ be the pumping length of the target language, $L$, and $M$ be a DFA constructed from a closed, consistent $L^*$ observation table. The pumping length of $L(M)$ does not exceed $p$.*

Finally, we make use of the L* algorithm termination result. The property we use in our proof has been emphasized.

**Theorem 4.6** (L* [6], Theorem 6)**.** *Given any MAT presenting a regular language $L$,* ***$L^*$ eventually terminates and outputs a DFA isomorphic to the minimum DFA accepting $L$****. Additionally, if $n$ is the number of states in the minimum DFA recognizing $L$ and $m$ is an upper bound on the length of any counterexample provided by the MAT, then the total running time of $L^*$ is bounded by a polynomial in $m$ and $n$.*

With these properties in hand, we are now ready to prove our original theorem.

*Proof (Theorem 4.1).* Given a pure program $K$, which decides a regular language, and a candidate DFA $M$ constructed from a closed, consistent L* observation table $(S, E, T)$, let $p$ be the pumping length of $L(K)$. By Lemma 4.4, the minimal DFA that recognizes $L(K)$ has $p$ states. By Lemma 4.2, there exists a finite unrolling $K'$ of program $K$ such that $\forall s \in \Sigma^*.|s| < p \implies K'(s) = K(s)$. We will show that verifying $K' \equiv M$ is sufficient to verify $K \equiv M$ using bounded model checking.

Verifying the property $\nexists t \in \Sigma^*$ such that $t \in L(K') \oplus L(M)$ (the symmetric difference, i.e., $t \in L(K') \cup L(M)$ and $t \notin L(K') \cap L(M)$) with incremental bounded model checking (recall $K'$ is a finite unrolling) can result in two outcomes:

*Case 1*: $\exists t \in \Sigma^*$ such that $|t| < p \wedge K'(t) \neq M(t)$.
*Case 2*: $\forall t \in \Sigma^*$ such that $|t| < p$, $K'(t) = M(t)$ holds.

In the first case, we return $t$ as a counterexample, concluding $K \not\equiv M$. In the second case, we conclude that $K' \equiv M$ and any counterexample must be at least as long as $p$; however, no such counterexample exists. The proof proceeds by contradiction.

Suppose, for the sake of contradiction, that $\exists t' \in \Sigma^*$ such that $|t'| \geqslant p$ and $K(t') \neq M(t')$. Let $n$ be the number of states in the candidate DFA $M$. We now relate $n$ to the number of states in the minimal DFA recognizing $L(K)$. By Lemma 4.4 and Corollary 4.5.1, $n \leqslant p$ because the pumping length of $L(M)$ is at most $p$ and the number of states in $M$ is equal to the pumping length of $L(M)$. Additionally, because the finite unrolling $K' \equiv M$, $n \geqslant p$ by Theorem 4.5. Therefore, the number of states in $M$ is bounded above and below by the pumping length of our target language, implying that $n = p$. Using our assumption about $t'$, we note that $K$ is consistent with $T$ but not equivalent to $M$, and thus by another application of Theorem 4.5 we conclude that the DFA recognizing $L(K)$ must have more than $n = p$ states. This contradicts the fact, from Lemma 4.4, that the minimal DFA recognizing $L(K)$ has exactly $p$ states. Therefore, no such $t'$ exists.

Because L* produces a minimal DFA (Theorem 4.6), and $M$ was produced from a closed, consistent observation table, we can conclude that $M$ must be a DFA isomorphic with the minimal DFA accepting the language $L(K)$. Thus, $K \equiv M$.

This means that, using bounded model checking on the program $K'$ (recall that $K'$ is a finite unrolling and thus admits *bounded* model checking), we either find a counterexample or can conclude equivalence of $K$ and $M$. Therefore, $K' \equiv M \implies K \equiv M$. □

From this result, we can establish the following corollary, which allows us to conclude that our approach may be used in a MAT to answer termination queries.

**Corollary 4.6.1.** *For a given program K, there exists a finite number of iterations of incremental unrolling needed for our approach to respond to a termination query with either a counterexample or a proof of equivalence.*

***Implications.*** In our formulation, termination queries return an answer if the bounded software model checking with incremental loop unrolling and the string decision procedures terminates. Our result is therefore *relative* to the completeness of the model checker and underlying SMT theories (see Ball et al. for a discussion of relative completeness in software model checking [12]). For pure kernels that decide a regular language, we proved that there is a finite bound on the incremental unrolling that will determine equivalence of the kernel and a candidate automaton. In practice, we make use of a timeout on the verification process to ensure timely termination at the expense of correctness in some cases. This design decision results in an approximate solution in cases where either the finite unrolling bound has not yet been reached or the legacy kernel recognizes a non-regular program. The approximate solution is correct for strings of length up to a particular bound but may disagree on larger strings. Our empirical evaluation in Section 6 demonstrates that AutomataSynth successfully learns an equivalent state machine for thirteen of eighteen real-world string kernels mined from legacy source code.

## 5 Experimental Methodology

In this section, we describe our process for selecting real-world, legacy string kernels benchmarks as well as our experimental setup for the evaluation described in Section 6.

### 5.1 Benchmark Selection

In our evaluation, we focus on measuring the extent to which AutomataSynth learns models for real-world string functions using varied library methods. We construct our benchmark suite by mining legacy string kernels from open-source software projects on GitHub using the following protocol. First, we filter all projects for those with C source code and ordered the resulting repositories by number of stars (i.e., popular repositories first). Next, we use the Cil [62] framework to iteratively parse each source file and extract all functions with an appropriate type signature (see Section 3.2). We filter these functions to exclude those that referenced functions or data outside the compilation unit. We allow the use of common library function (e.g., `strlen`, `strcmp`, etc.). In total, we considered 26 repositories and mined 973 separate string kernel functions using this protocol.

After filtering for duplicates and a manual analysis to identify functions that return Boolean values (we note that while C has the `_Bool` data type, many functions still use integers of varying widths), we collected 18 meaningfully-distinct real-world benchmarks. Table 1 provides an overview of these string kernels. We use the function name to refer to each benchmark and also indicate the source project for each. Lines of code (LOC) provides a count of the total number of non-comment lines in the post-processed version of the benchmark. Finally, we also indicate whether the kernel is supported by our prototype system. Our prototype implementation supports all but three of these legacy string kernels. The unsupported kernels use computation that is difficult to capture with present string decision procedures.

The kernels in our benchmark suite interact with strings in various manners. Some kernels, such as `is_numeric_index`, `skipline`, and `cf_is_comment`, loop over all characters in the string checking various constraints. Several also make heavy use of `strcmp` to check for the presence of specific strings (e.g., `checkerrormsg`, `is_mcounted_section_name`, and `start_line`). We also found examples of kernels (e.g., `git_offset_1st_component` and `AMF_DecodeBoolean`) that perform single character comparisons. While a developer will likely not be interested in accelerating a single character comparison, these kernels remain indicative of real-world code and allow us to demonstrate a proof-of-concept for synthesizing designs for accelerators such as FPGAs. An evaluation of benchmarks more typical of kernels accelerated by FPGAs (e.g., long-running kernels with hundreds or thousands of states) is left for future work.

**Table 1.** Benchmark Suite of Real-World, Legacy String Kernels

| Function | Project | LOC | Support |
|---|---|---|---|
| `git_offset_1st_component` | *Git*: Revision control system | 6 | ✓ |
| `is_encoding_utf8` | | 38 | ✘[*] |
| `checkerrormsg` | *jq*: Command-line JSON processor | 4 | ✓ |
| `checkfail` | | 14 | ✓ |
| `skipline` | | 17 | ✓ |
| `end_line` | *Linux*: OS kernel | 11 | ✓ |
| `start_line` | | 11 | ✓ |
| `is_mcounted_section_name` | | 54 | ✓ |
| `is_numeric_index` | *MASSCAN*: IP port scanner | 17 | ✓ |
| `is_comment` | | 11 | ✓ |
| `AMF_DecodeBoolean` | *OBS Studio*: Live streaming and recording software | 2 | ✓ |
| `cf_is_comment` | | 28 | ✓ |
| `cf_is_splice` | | 22 | ✓ |
| `is_reserved_name` | | 39 | ✓ |
| `has_start_code` | | 18 | ✓ |
| `number_is_valid` | *openpilot*: Open-source driving agent | 72 | ✘[†] |
| `utf8_validate` | | 72 | ✘[‡] |
| `stbtt__isfont` | | 24 | ✓ |

[*]Requires `strcasecmp` support [†]Requires `strtod` support
[‡]Performs math on characters

### 5.2 Experimental Setup

Our AutomataSynth implementation produces MNRL, an open-source state machine representation language intended for large-scale automata processing applications [9]. We transform the learned DFA to be *homogeneous*, a property that admits a simplified transition rule while maintaining expressive power and that is amenable to hardware acceleration [10, 23, 33, 91]. We use Brzozowski's algorithm [21] for converting candidate DFAs to regular expressions as part of the software verification step (see Section 4).

For termination queries, we add string constraint handling to CPAChecker 1.8 [16]. We also extend the JavaSMT framework [47] to support the draft SMT-LIB strings theory interface [81]. We use Microsoft's Z3 version 4.8.6 SMT solver [31] with the *Seq* string solver [84] for all queries. All evaluations use an Ubuntu 16.04 Linux server with a 3.0 GHz Intel Xeon E5-2623-v3 with four physical cores and 16 GB of RAM and a maximum time budget of 24 hours.

## 6 Evaluation

In this section, we evaluate AutomataSynth on fifteen real-world, legacy string kernels mined from open-source projects. We first evaluate the correctness of the state machines generated by AutomataSynth and report runtime and query counts. Second, we evaluated the suitability of the generated automata for hardware acceleration.

### 6.1 State Machine Learning

Table 2 presents results from our empirical evaluation of AutomataSynth on a benchmark suite of fifteen legacy string kernels. We do not report results for the three benchmarks that are not supported. We report the number of membership and termination queries executed for each kernel as well as the number of states in the learned automaton and the total runtime in seconds. The final column indicates if AutomataSynth correctly learned the kernel's functionality. A check mark means that our tool learned a fully-equivalent automaton. We also indicate approximate results in which the maximum time limit was exceeded.

On average, it took seven hours to learn an automaton from the legacy string kernel, with more than half of the benchmarks terminating in fewer than five minutes. AutomataSynth correctly learned thirteen of the fifteen benchmarks. The remaining two benchmarks yield approximate solutions, with many of these approximations being extremely similar to the target kernel functionality. In our evaluation this approximation was always the result of timeouts rather than the relative completeness of the SMT solver used for termination queries. There were no instances in our benchmark set for which the SMT solver returned an unknown result due to a limitation in the string decision procedures.

We determined that there were two primary causes for AutomataSynth reaching the timeout without learning a fully-equivalent state machine. First, Brzozowski's algorithm for constructing a regular expressions can produce large expressions that require simplification to remove redundant and superfluous clauses. This was most relevant to kernels that compared string suffixes with a string constant. We believe this performance limitation is an artifact of design choices in our prototype, which could be solved with more careful construction of regular expressions. Second, some SMT queries were significantly less performant than others. We discuss this challenge in more detail in Section 7.

The relative utility of the membership and termination queries varies between the benchmarks. For example, the function `git_offset_1st_component` checks a string to see if the first character is a forward slash (`/`). Using membership queries, AutomataSynth learned that the first character of the string must be a slash and that any number of characters may follow. The termination query provided a single counterexample of a longer string that was initially misclassified. For this kernel, the membership queries provided much of the "learning". This is in contrast to the `stbtt__isfont` kernel, which ultimately compares an input string against four hard-coded strings. In this case, the membership queries only provided minimal information. Instead, the termination queries discovered the string constants in the kernel's source code and provided much of the learned information. In general, membership queries tended to provide more information when each character in the input string was considered separately while termination queries helped to discover string constants used for comparison by the kernels.

AutomataSynth successfully learned automata for fifteen of the eighteen legacy kernels mined from open-source projects. Of these, thirteen were exactly equivalent and two were near approximations.

### 6.2 Hardware Acceleration

In this work, we claim no novelty for accelerating automata using hardware accelerators, such as FPGAs. Instead, we leverage existing work in the area of high-performance automata processing. On FPGAs, Xie et al.'s REAPR framework supports high-throughput processing of data with finite automata on FPGAs [96]. For spatially-reconfigurable architectures akin to FPGAs, the dominant factor affecting performance is the number of hardware resources used by a design. For ANMLZoo benchmarks, which contain tens of thousands of states [91], REAPR successfully synthesized designs running in the range of 200–700 MHz. Because the automata learned by AutomataSynth are significantly smaller, we expect that similar throughputs could be achieved.

The finite automata learned by AutomataSynth fall withing the design constraints of FPGA-based automata accelerators, allowing for high-throughput execution.

**Table 2.** Experimental Results

| Benchmark | Membership Queries | Termination Queries | Number of States | Total Runtime (s) | Correct |
|---|---|---|---|---|---|
| `git_offset_1st_component` | 4,090 | 2 | 2 | 7 | ✓ |
| `checkerrormsg` | 32,664 | 2 | 15 | 86,195 | ✓* |
| `checkfail` | 189,013 | 3 | 35 | 86,308 | ✓* |
| `skipline` | 7,663 | 3 | 3 | 294 | ✓ |
| `end_line` | 510,623 | 4 | 44 | 29,531 | ✓ |
| `start_line` | 206,613 | 2 | 46 | 4,813 | Approx. |
| `is_mcounted_section_name` | 672,041 | 7 | 57 | 86,399 | Approx. |
| `is_numeric_index` | 10,727 | 3 | 4 | 297 | ✓ |
| `is_comment` | 4,090 | 2 | 2 | 14 | ✓ |
| `AMF_DecodeBoolean` | 2,557 | 2 | 2 | 4 | ✓ |
| `cf_is_comment` | 4,599 | 2 | 4 | 300 | ✓ |
| `cf_is_splice` | 1,913 | 2 | 4 | 3 | ✓ |
| `is_reserved_name` | 350,705 | 8 | 42 | 85,469 | ✓ |
| `has_start_code` | 10,213 | 2 | 7 | 5 | ✓ |
| `stbtt__isfont` | 79,598 | 5 | 19 | 13 | ✓ |

*AutomataSynth warned of a potential approximate solution due to timeout, but manual analysis confirmed correctness

## 7 Discussion

At a high level, AutomataSynth learns the behavior of a Boolean string kernel through a combination of dynamic and static analyses and emits a functionally-equivalent state machine that is amenable to acceleration with FPGAs. We believe that approaches such as AutomataSynth are very promising and could offer solutions to limitations inherent to current HLS techniques. HLS relies heavily on the structure of C-like source code to produce a hardware description, which were designed for performance on—and as an abstraction of—von Neumann architectures. As such, HLS is unlikely to produce performant FPGA designs from legacy code that was heavily optimized for CPUs [85, 99]. This places a heavy burden on developers tasked with porting code and represents a significant barrier to adoption. Our approach decouples the implementation choices of the legacy program from the emitted hardware design. This allows us to produce a design using a model of computation—state machines—that is performant on FPGAs [91, 96].

This paper represents an initial effort to understand the benefits and limitations of using state machine learning algorithms to compile code for FPGAs. A significant research effort remains for approaches akin to AutomataSynth to be mature enough for industry adoption. In the remainder of this section we identify four key research challenges whose solutions would lead to significant advances in learning-based synthesis for FPGAs. Additionally, we describe candidate future directions to tackle each of these.

### 7.1 Learning More Expressive Models

We present an approach for accelerating regular language Boolean string kernels with FPGAs. Our prototype soundly transforms such kernels to functionally-equivalent hardware descriptions; Boolean functions with inputs that may be transformed into a serial data stream are also applicable. However, legacy code contains many other types of functions, and these remain an open challenge. Supporting a new function type presents a two-fold challenge: (1) identifying suitable computational models for acceleration and (2) designing or adapting an algorithm suitable for learning these models. Finite automata, as formally defined, produce a single bit of output for each string processed and are limited to recognizing Regular Languages. Additional models, such as Mealy and Moore machines, support transforming an input value into an output value, while others, such as pushdown automata, support more expressive classes of languages.

Several efforts are underway to extend learning algorithms to more expressive computational models [19, 24, 59]. It may also be possible to leverage insights from the architecture community and recent efforts to accelerate automata processing, in which designs often support tagging output *report* signals with additional metadata [33, 89]. Further, existing DFA learning algorithms may admit learning functions that output an enumerated—or even a multi-bit—value.

An additional challenge is that determining program equivalence is, in the limit, undecidable. For example, Angluin notes that termination queries are not generally decidable for context-free languages [6]. However, existing software verifiers suffer from this same challenge and provide relative completeness [12]. Further, this challenge may be addressed in some cases through careful use of approximation.

### 7.2 Expressive Power and Performance of String Solvers

Our empirical evaluation of AutomataSynth demonstrated some limitations of present string decision procedures. Certain string operations (e.g., case-insensitive lexicographic comparisons and casting between characters and numbers to perform arithmetic operations) occur in real-world software but are difficult to represent as constraints in String theories. Additionally, SMT queries generated by bounded model checking algorithms can result in long-running computation.

These challenges are not new: the formal methods community has been laboring to improve string decision procedures for over a decade. Early efforts often focused on the problem domain of identifying cross-site scripting and SQL code injection vulnerabilities (e.g., [41]) and introduced new constraint types. These efforts often reasoned about fixed-sized string variables (e.g., [50]). Subsequent efforts, such as Z3str3, also focus on improving the performance of these decision procedures and have extended support to unbounded strings [14].

AutomataSynth is one of the first efforts to combine bounded software model checking with string decision procedures. This combination presents a novel and compelling use case for string solvers that requires new constraint types and optimizations. We make our tool and all of the SMT queries automatically generated by our process available[4] to the community to encourage renewed interest in—and efforts to—improve the performance of string solvers.

### 7.3 Scaling Termination Queries

We found, in practice, that termination queries consumed an average of 66% of the total runtime of AutomataSynth. As candidate state machines increase in size, we expect the scalability of termination queries to dominate. This challenge presents an opportunity for innovation. We proposed an approach based on the novel combination of bounded software model checking and string decision procedures; however, alternate formulations of termination queries may provide better performance while maintaining correctness.

Many applications of model learning focus on the use of testing to provide answers to termination queries [86]. We have observed that the application of automated testing presents several challenges, such as producing a suitable quantity and diversity of inputs to identify counterexamples. Test input generators, such as Klee [22], may only support bounded length strings (rather than unbounded).

The application of other software verification techniques may also provide performance gains. Counterexample-guided abstraction refinement verifiers can abstract much of a program's state to gain performance, but require support for interpolation queries. These are not currently support by string solvers, but present an additional area for research.

### 7.4 Characterizing and Taming Approximation

Because scalability and decidability of termination queries are challenges, approximation may play an important role in improving the performance of learning-based approaches to synthesizing FPGA designs. Indeed, there is already significant interest in the architecture and software communities for producing approximate programs [58, 60, 65, 66].

Approximation has been a key parameter in model learning algorithms from the start [6, 86]. Results from learning theory often analyze approximation using Valiant's *probably approximately correct* (PAC) framework, which bounds the probability of the error being less than a fixed threshold for an approximately-learned model [87]. Such results can predict the number of queries necessary to bound the error but *do not characterize the locations or significance of the remaining error.* Anomalous results for frequently-used inputs have a very different impact than anomalous results for seldom-used inputs. Given the design of Angluin-style algorithms, it may be possible to determine which inputs result in approximate solutions. For example, pre-populating the observation table with rows pertaining to known inputs (i.e., those taken from the test suite) ensures that the learned state machine produces the correct output for those relevant values.

## 8 Conclusions

We present AutomataSynth, a framework for accelerating legacy regular language Boolean string kernel functions using FPGAs. Our approach uses a novel combination of state machine learning algorithms, software verification algorithms, string decision procedures, and high-performance automata processing architectures to learn the behavior of a program and construct a behaviorally-equivalent FPGA hardware description. We demonstrate a proof-of-concept of this approach using a benchmark suite of eighteen string kernels mined from open-source projects on GitHub. AutomataSynth successfully constructs equivalent (or near equivalent) FPGA designs for more than 80% of these benchmarks. We believe this approach shows promise for overcoming some of the limitations of current HLS techniques.

## Acknowledgments

This work is funded in part by: NSF grants CCF-1629450, CCF-1763674, CCF-1908633; AFRL Contract No. FA8750-19-1-0501; and the Jefferson Scholars Foundation. We also wish to thank Kevin Skadron and Lu Feng for their suggestions, encouragement, and expertise in the early stages of this project as well as the anonymous reviewers for their helpful comments and feedback.

## References

[1] F. Aarts, J. De Ruiter, and E. Poll. 2013. Formal Models of Bank Cards for Free. In *Sixth International Conference on Software Testing, Verification and Validation Workshops*. 461–468.

[2] Gustavo Alonso. 2018. FPGAs in Data Centers. *Queue* 16, 2, Article 60 (April 2018), 6 pages.

[3] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013.

[4]See the AutomataSynth repository at https://github.com/kevinaangstadt/automata-synth

Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*. 1–8.

[4] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R. Larus. 2003. Debugging Temporal Specifications with Concept Analysis. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA). 182–195.

[5] Dana Angluin. 1981. A note on the number of queries needed to identify regular languages. *Information and Control* 51, 1 (1981), 76 – 87.

[6] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (Nov. 1987), 87–106.

[7] Dana Angluin. 1992. Computational Learning Theory: Survey and Selected Bibliography. In *Symposium on Theory of Computing*. 351–369.

[8] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. 2018. ASPEN: A Scalable in-SRAM Architecture for Pushdown Automata. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, Piscataway, NJ, USA, 921–932.

[9] K. Angstadt, J. Wadden, V. Dang, T. Xie, D. Kramp, W. Weimer, M. Stan, and K. Skadron. 2018. MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 84–87.

[10] K. Angstadt, J. Wadden, W. Weimer, and K. Skadron. 2019. Portable Programming with RAPID. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (April 2019), 939–952.

[11] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. 2006. *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report UCB/EECS-2006-183. EECS Department, University of California, Berkeley.

[12] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. 2002. Relative Completeness of Abstraction Refinement for Software Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. 158–172.

[13] Thomas Ball and Sriram K. Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software* (Toronto, Ontario, Canada) *(SPIN '01)*. Springer-Verlag, Berlin, Heidelberg, 103–122.

[14] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *Formal Methods in Computer Aided Design (FMCAD 2017)*. 55–59.

[15] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2007. The software model checker Blast. *International Journal on Software Tools for Technology Transfer* 9, 5 (01 Oct 2007), 505–525.

[16] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–190.

[17] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. 2010. Predicate Abstraction with Adjustable-block Encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Lugano, Switzerland) *(FMCAD '10)*. FMCAD Inc, Austin, TX, 189–198.

[18] Armin Biere. 2009. Bounded Model Checking. In *Handbook of Satisfiability*. 457–481.

[19] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker. 2009. Angluin-Style Learning of NFA. In *International Joint Conference on Artificial Intelligence*.

[20] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). 70–87.

[21] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494.

[22] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224.

[23] Pascal Caron and Djelloul Ziadi. 2000. Characterization of Glushkov automata. *Theoretical Computer Science* 233, 1 (2000), 75–90.

[24] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active Learning for Extended Finite State Machines. *Form. Asp. Comput.* 28, 2 (April 2016), 233–263.

[25] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 7, 13 pages.

[26] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 3–14.

[27] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *J. ACM* 50, 5 (Sept. 2003), 752–794.

[28] Computer Sciences Corporation. 2012. Big Data Universe Beginning to Explode. http://www.csc.com/insights/flxwd/78931-big_data_universe_beginning_to_explode.

[29] William Craig. 1957. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic* 22, 3 (1957), 269–285.

[30] Colin de la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA.

[31] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[32] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, Berkeley, CA, USA, 193–206.

[33] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. 2014. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 12 (2014), 3088–3098.

[34] DNV GL. 2016. Are you able to leverage big data to boost your productivity and value creation? https://www.dnvgl.com/assurance/viewpoint/viewpoint-surveys/big-data.html.

[35] Yuanwei Fang, Tung T Hoang, Michela Becchi, and Andrew A Chien. 2015. Fast support for unstructured data processing: the unified automata processor. In *Proceedings of the ACM International Symposium on Microarchitecture (Micro '15)*. 533–545.

[36] Yuanwei Fang, Chen Zou, Aaron J Elmore, and Andrew A Chien. 2017. UDP: a programmable accelerator for extract-transform-load workloads and more. In *International Symposium on Microarchitecture*. ACM, 55–68.

[37] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining Model Learning and Model Checking to Analyze TCP Implementations. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham,

454–471.

[38] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. 2018. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 139–153.

[39] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 44.

[40] Sumit Gulwani. 2016. Programming by Examples: Applications, Algorithms, and Ambiguity Resolution. In *Proceedings of the 8th International Joint Conference on Automated Reasoning*. Springer-Verlag, Berlin, Heidelberg, 9–14.

[41] Pieter Hooimeijer and Westley Weimer. 2009. A decision procedure for subset constraints over regular languages. In *Programming Language Design and Implementation (PLDI)*. 188–198.

[42] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Verifying web applications using bounded model checking. In *International Conference on Dependable Systems and Networks*. IEEE, 199–208.

[43] M Isberner. 2015. *Foundations of active automata learning: an alorithmic perspective.* Ph.D. Dissertation. Technical University of Dortmund.

[44] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *Runtime Verification*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Springer International Publishing, Cham, 307–322.

[45] Ranjit Jhala and Rupak Majumdar. 2009. Software Model Checking. *Comput. Surveys* 41, 4, Article 21 (Oct. 2009), 54 pages.

[46] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 711–726.

[47] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. 2016. JavaSMT: A Unified Interface for SMT Solvers in Java. In *Verified Software. Theories, Tools, and Experiments*, Sandrine Blazy and Marsha Chechik (Eds.). Springer International Publishing, Cham, 139–148.

[48] Michael J. Kearns and Umesh V. Vazirani. 1994. *An Introduction to Computational Learning Theory.* MIT Press, Cambridge, MA, USA.

[49] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with Amorphos. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 107–127.

[50] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 105–116.

[51] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen. 2019. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 898–911.

[52] D. Lee and M. Yannakakis. 1996. Principles and methods of testing finite state machines-a survey. *Proc. IEEE* 84, 8 (Aug 1996), 1090–1123.

[53] Anthony W Lin and Pablo Barceló. 2016. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 123–136.

[54] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. 2004. Efficient Test-based Model Generation for Legacy Reactive Systems. In *Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International (HLDVT '04)*. IEEE Computer Society, Washington, DC, USA, 95–100.

[55] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification* (Seattle, WA) *(CAV '06)*. Springer-Verlag, Berlin, Heidelberg, 123–136.

[56] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 691–701.

[57] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. 2015. Helium: Lifting High-Performance Stencil Kernels from Stripped X86 Binaries to Halide DSL Code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 391–402.

[58] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4, Article 62 (March 2016), 33 pages.

[59] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michal Szynwelski. 2017. Learning nominal automata. In *Principles of Programming Languages (POPL)*. 613–625.

[60] Thierry Moreau, Joshua San Miguel, Mark Wyse, James Bornholt, Armin Alaghi, Luis Ceze, Natalie D. Enright Jerger, and Adrian Sampson. 2018. A Taxonomy of General Purpose Approximate Computing Techniques. *Embedded Systems Letters* 10, 1 (2018), 2–5.

[61] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (Oct 2016), 1591–1604.

[62] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228.

[63] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE, Piscataway, NJ, USA, 772–781.

[64] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. 2017. Demystifying Automata Processing: GPUs, FPGAs, or Micron's AP?. In *Proceedings of the International Conference on Supercomputing* (Chicago, Illinois) *(ICS '17)*. ACM, New York, NY, USA, Article 1, 11 pages.

[65] Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: language support for safe and modular approximate programming. In *Foundations of Software Engineering (ESEC/FSE)*. 745–757.

[66] Martin C. Rinard. 2003. Acceptability-oriented computing. In *Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA)*. 221–239.

[67] R.L. Rivest and R.E. Schapire. 1993. Inference of Finite Automata Using Homing Sequences. *Information and Computation* 103, 2 (1993), 299 – 347.

[68] Indranil Roy. 2015. *Algorithmic Techniques for the Micron Automata Processor.* Ph.D. Dissertation. Georgia Institute of Technology.

[69] Indranil Roy and Srinivas Aluru. 2014. Finding Motifs in Biological Sequences Using the Micron Automata Processor. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium.* 415–424.

[70] I. Roy, N. Jammula, and S. Aluru. 2016. Algorithmic Techniques for Solving Graph Problems on the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*. 283–292.

[71] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru. 2016. High Performance Pattern Matching Using the Automata Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*. 1123–1132.

[72] Mathijs Schuts, Jozef Hooman, and Frits Vaandrager. 2016. Refactoring of Legacy Software Using Model Learning and Equivalence Checking: An Industrial Experience Report. In *Proceedings of the 12th International Conference on Integrated Formal Methods - Volume 9681* (Reykjavik, Iceland) *(IFM 2016)*. Springer-Verlag, Berlin, Heidelberg, 311–325.

[73] J. M. Shalf and R. Leland. 2015. Computing beyond Moore's Law. *IEEE Computer* 48, 12 (Dec 2015), 14–23.

[74] Michael Sipser. 2006. *Introduction to the Theory of Computation* (2nd ed.). Thomson Course Technology.

[75] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 136–148.

[76] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. ACM, New York, NY, USA, 404–415.

[77] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A Type System for Regular Expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs* (Beijing, China) *(FTfJP '12)*. 20–26.

[78] Bernhard Steffen, Falk Howar, and Maik Merten. 2011. Introduction to Active Automata Learning from a Practical Perspective. In *Formal Methods for Eternal Networked Software Systems: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, Marco Bernardo and Valérie Issarny (Eds.). Springer Berlin Heidelberg, Bertinoro, Italy, 256–296.

[79] J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* 12, 3 (May 2010), 66–73.

[80] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. 2017. Cache Automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50)*. ACM, New York, NY, USA, 259–272.

[81] Cesare Tinelli, Clark Barrett, and Pascal Fontaine. 2019. *SMT-LIB 2.6 Strings Theory: Draft 2.1.* Technical Report. Department of Computer Science, The University of Iowa.

[82] Tommy Tracy II, Yao Fu, Indranil Roy, Eric Jonas, and Paul Glendenning. 2016. Towards Machine Learning on the Automata Processor. In *Proceedings of ISC High Performance Computing*. 200–218.

[83] Tommy Tracy II, Mircea Stan, Nathan Brunelle, Jack Wadden, Ke Wang, Kevin Skadron, and Gabe Robins. 2015. Nondeterministic Finite Automata in Hardware—the Case of the Levenshtein Automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[84] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA, 1232–1243.

[85] L. Di Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio. 2017. The Role of CAD Frameworks in Heterogeneous FPGA-Based Cloud Systems. In *2017 IEEE International Conference on Computer Design (ICCD)*. 423–426.

[86] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95.

[87] L. G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (Nov. 1984), 1134–1142.

[88] Jan van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *International Symposium on Microarchitecture*. 461–472.

[89] Jack Wadden, Kevin Angstadt, and Kevin Skadron. 2018. Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 749–761.

[90] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron. 2016. Generating efficient and high-quality pseudo-random behavior on Automata Processors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 622–629.

[91] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron. 2016. ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures. In *International Symposium on Workload Characterization (IISWC '16)*. 1–12.

[92] Ke Wang, Elaheh Sadredini, and Kevin Skadron. 2016. Sequential Pattern Mining with the Micron Automata Processor. In *Proceedings of the ACM International Conference on Computing Frontiers* (Como, Italy) *(CF '16)*. ACM, New York, NY, USA, 135–144.

[93] Michael H.L.S. Wang, Gustavo Cancelo, Christopher Green, Deyuan Guo, Ke Wang, and Ted Zmuda. 2016. Using the Automata Processor for fast pattern recognition in high energy physics experiments — A proof of concept. *Nuclear Instruments and Methods in Physics Research* (2016).

[94] Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, StockholmsmÃďssan, Stockholm Sweden, 5247–5256.

[95] Loring Wirbel. 2014. *Xilinx SDAccel: A Unified Development Environment for Tomorrow's Data Center.* Technical Report. The Linley Group.

[96] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan. 2017. REAPR: Reconfigurable engine for automata processing. In *27th International Conference on Field Programmable Logic and Applications (FPL '17)*. 1–8.

[97] Xiaodong Yu and Michela Becchi. 2013. GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '13)*. ACM, New York, NY, USA, Article 18, 10 pages.

[98] Keira Zhou, Jeffrey J. Fox, Ke Wang, Donald E. Brown, and Kevin Skadron. 2015. Brill tagging on the Micron Automata Processor. In *Proceedings of the 9th IEEE International Conference on Semantic Computing*. 236–239.

[99] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah). Article 35, 12 pages.