Synthesizing State Machines for Data Planes

Xiaoqi Chen
Princeton University
xiaoqic@cs.princeton.edu

Andrew Johnson
Princeton University
aj3189@princeton.edu

Mengying Pan
Princeton University
mengying@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

ABSTRACT

The emergence of programmable switches such as the Intel Tofino has made it possible, in theory, to implement many network monitoring applications directly in the network data plane. In practice, however, such implementations are often more challenging than expected. A key difficulty is that such applications often depend, in part, on recognizing traffic patterns that are easy to specify as a deterministic finite state automaton (a DFA) but hard to implement thanks to stringent hardware constraints: to maximize throughput and avoid race conditions, state machine updates must be completed in a single Tofino pipeline stage, but the limited computational resources make finding an implementation a challenging puzzle. This paper presents a solution to such puzzles—a general framework for synthesizing DFA implementations automatically. A key insight is that such a synthesis system is free to renumber state machine states and implement transitions using any available arithmetic or logical operations over that renumbering, provided the resulting implementation is semantically equivalent to the input specification. To produce such a synthesizer, we model the required state machine semantics and the available single-stage switch operations using SMT constraints. An off-the-shelf SMT solver finds a solution to the constraints, and this solution is then translated to P4 code. We evaluate the effectiveness of our methods by synthesizing state machines for a variety of useful applications, including those that monitor TCP handshakes and video conference streams.

Author list alphabetically ordered.

SOSR '22, Oct 18, 2022, Virtual

ACM ISBN 978-1-4503-9892-3/22/10…$15.00
https://doi.org/10.1145/3563647.3563650

CCS CONCEPTS

• **Networks** → **Data path algorithms**; • **Theory of computation** → Automated reasoning.

KEYWORDS

State Machine, DFA, Programmable Data Plane, P4, SMT

ACM Reference Format:
Xiaoqi Chen, Andrew Johnson, Mengying Pan, and David Walker. 2022. Synthesizing State Machines for Data Planes. In *Symposium on SDN Research (SOSR 22), October 19–20, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3563647.3563650

1 INTRODUCTION

Many network programming tasks must track the state of a connection or protocol across a series of packets that flow through the network. For instance, one may want to monitor TCP connections to detect errors, provide security, bill consumers, collect statistics, or identify attacks. In such a scenario, a system that identifies the canonical 3-way TCP handshake may categorize the connection as "normal," whereas deviations from the protocol will indicate security or performance anomalies that warrant further scrutiny [2]. Table 1 presents several other applications.

While it is often simple to specify expected behavior of these protocols using regular expressions or finite state machines (DFAs), implementing such specifications correctly, at line rate, in modern programmable switches can be surprisingly challenging. For example, the Intel Tofino [6], and other ASICs that implement the PISA architecture [3], are structured as a series of stages. Each stage contains its own local memory, which is inaccessable to other stages, as well as a small set of arithmetic-logic units that may perform simple computations. Hence, in general, to process packets at line rate while maintaining a state machine, the state machine implementation must reside in just a single stage of the pipeline, and every state machine transition must be implemented using one of the small number of possible computations available. In practice, the array of constraints

Name	Description
TCP Handshake	Track the 3-way handshake packets during the setup of a TCP connection.
Video Conference	Track the set of active participants as they join and leave a video call.
Mobile Device	Track the state of mobile devices by observing signaling packets in cellular core networks.
Fingerprinting [10]	Classify the content of encrypted traffic by observing packet size patterns of a session.

Table 1: State machines in example network monitoring applications

facing the programmer is daunting, and it takes substantial creativity, time, and effort to find an implementation.

To solve this problem systematically for the community, we developed a tool to synthesize P4 implementations of state machines automatically. Our key insight is that many state machines may be implemented more efficiently in a constrained PISA architecture when a new numbering of their states is chosen. When a state space is renumbered, the transitions between states may suddenly be implementable using available (and perhaps unexpected) arithmetic or logical operations. For instance, a transition from state 5 to state 7 may suddenly be implementable (perhaps as a left-shift operation), along with all other transitions, if a state renumbering transforms the requirement into a transition from 4 to 8. In essence, we exploit the fact that a particular state machine is isomorphic to, and semantically indistinguishable from, a set of other state machines. Our synthesizer should be free to choose any semantically equivalent implementation that fits the constraints of a device like the Tofino. Moreover, while enumerating possible state numberings is a challenging, tedious, and error-prone process for humans, it can be implemented by encoding the state machine and hardware constraints as an SMT query and deploying an off-the-shelf solver to find a solution to the logical puzzle.

In the rest of this paper, we explain our DFA synthesis algorithm in more depth and demonstrate that it works well on a range of practical examples. Indeed, our prototype can synthesize implementations of DFAs with dozens of states and transitions. Depending on the size and complexity of the DFA, synthesis sometimes takes just a few seconds and sometimes a few minutes. For pedagogic purposes, the focus of this paper is on the synthesis of a single state machine. However, in practice, it is just as easy to implement an array of state machines, with each state machine in the array implementing the same transition system, but being used to track the state of a separate flow. The number of flows supported by such an array of state machines depends on the hardware and will be the same as the number of flows supported by any other data plane switch program with per-flow state.

Our prototype code is publicly available on GitHub[1].

[1]https://github.com/Princeton-Cabernet/DFA-synthesis

2 RUNNING STATE MACHINES UNDER MEMORY UPDATE CONSTRAINTS

In order to achieve high throughput and low forwarding latency, high-speed programmable switches based on the Protocol Independent Switch Architecture (PISA) [3] use a pipeline architecture in the data plane and impose strict requirements on register memory updates. To avoid any memory access hazards, specific register memory regions are only accessible from a particular pipeline stage. As the switch processes a packet within a pipeline stage, it can run a microprogram called a Register Action to perform a Read-Modify-Write operation on the value stored in register memory.

Each Register Action performs very simple computations. In Figure 2, we show simplified pseudocode illustrating its capability. It first performs a comparison between r, the old value stored in memory, and a constant value or a field from the packet. Depending on the result, it branches and performs an arithmetic or logical operation between r and another expression, saving the new value r' in the same register memory. Here, the expressions are a constant or a variable in the packet's header or metadata. Note that the expression's value is given before we execute the Register Action and cannot depend on the in-memory value r. We can pre-program several different Register Actions (up to 4 for the Tofino switch) for a single register memory, and choose one of them to execute for each packet.

One naive way to implement a DFA in the data plane is to store the state in the register memory, and program the state transition rules as a match-action lookup table. To process a packet, we: (1) read the state from register memory, (2) consult the transition lookup table to obtain the new state, and (3) write the new state to the same register memory. However, we cannot both read the current state and consult the lookup table in the same hardware pipeline stage. If we read the old state in the Register Action and query the lookup table in the *next* pipeline stage, we must recirculate the packet to write the new state, which lowers the pipeline's throughput. Furthermore, recirculation leads to a race condition between the delayed memory write (step 3) and reads made by future packets (step 1), possibly causing errors.

Therefore, our goal is to implement the state transition in a single stage of the pipeline. That means we must finish the state transition in a Register Action without the help of

```
If r cmp sym :
    r' ← r op sym
Else:
    r' ← r op sym
```

sym ∈ { constant, header, metadata }
cmp ∈ { ≥, =, ≤, ≠ }
op ∈ { +,-,|,&,⊕ }

Figure 2: Pseudocode of a simple Register Action.

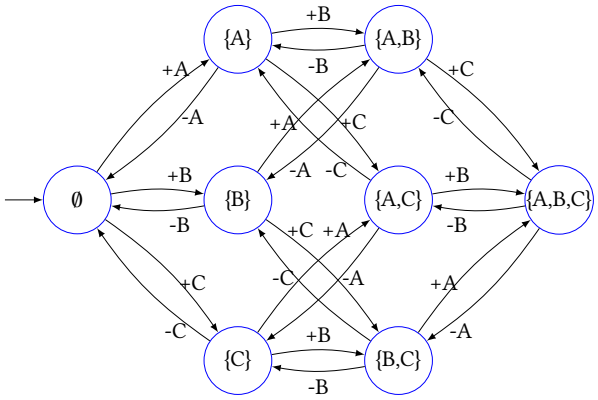


Figure 3: A DFA that tracks the set of participants in a video call, with up to 3 users (self-loops omitted).

State	Number
∅	0
{A}	1
{B}	2
{C}	3
{A,B}	4
{A,C}	5
{B,C}	6
{A,B,C}	7

(a) Naive mapping.

State	Number
∅	0 = 0b000
{A}	4 = 0b100
{B}	2 = 0b010
{C}	1 = 0b001
{A,B}	6 = 0b110
{A,C}	5 = 0b101
{B,C}	3 = 0b011
{A,B,C}	7 = 0b111

(b) A better mapping.

Figure 4: Two state-to-number mappings. With (b), state transitions can be done more easily using logical operations.

lookup tables. As an example, consider the "Video Conference" DFA in Table 1 that tracks which of the three users A, B, and C are actively participating in a video call as shown in Figure 3. To begin, we order the states and assign a number to each state (Figure 4a), and try to implement the transitions as a Register Action.

We begin with the input symbol `+A`, which is triggered when new traffic from user A is observed. If the old state is {B}, we need to transition to new state {A,B}, which means updating the old in-memory value from 2 to 4. Old state {C} needs to be updated to {A,C} (numerically, 3⟶5). It appears that one of the branches in the Register Action can implement $r' \leftarrow r + 2$. Meanwhile, state ∅ must transition to {A} (0⟶1), which is not covered by this branch; we have to use the other branch to implement $r' \leftarrow r + 1$. Yet, several states need to be kept unchanged (1⟶1, 4⟶4, 5⟶5, 7⟶7). Each Register Action only has two branches, and we don't have a third branch to implement $r' \leftarrow r$! The transitions for `+A` appear to be too complex to be completed in one Register Action.

The **key idea** of this paper is that rather than implementing the given DFA as-is, we may implement an **isomorphic** one, i.e. we may renumber states in any way that is convenient. Figure 4b shows an alternative number assignment for the same DFA. Under this numbering, the transition `+A` simply sets the first binary bit to 1, and can be implemented as $r' \leftarrow r|\texttt{0b100}$ unconditionally. Similarly, transition `-A` (when user A disconnects) sets the first bit to 0 and can be expressed as $r' \leftarrow r\&\texttt{0b011}$. We further note that transitions `+A`, `+B`, and `+C` can be executed using the same Register Action $r' \leftarrow r|f$ by setting the metadata variable f to `0b100`, `0b010`, or `0b001` respectively ahead of time.

Meanwhile, we also note that we are allowed to **choose** a different Register Action based on which symbol we see, as this information is available once we see the network packet, *before* accessing the register memory. We can program a second Register Action to execute $r' \leftarrow r\&f$. When we need to perform transitions `-A`, `-B`, or `-C` upon seeing a connection termination packet, we set the metadata variable f to `0b011`, `0b101`, or `0b110` respectively, and then choose to run the second Register Action instead of the first one. This way, we have implemented all six transitions correctly.

With the right numbering, we can share the same operation when implementing multiple transitions. By carefully choosing a state-to-number mapping and pre-computing an auxiliary input f, we can actually simplify the computation required for performing state transitions, and the seemingly unwieldy DFA can now run smoothly in the data plane.

However, for more complex DFAs, it is challenging and likely impossible to manually design the ideal state-to-number mapping coupled with the correct values for the metadata variables and which register action to choose. Instead, we seek help from an SMT solver.

3 SYNTHESIZING STATE MACHINES USING AN SMT SOLVER

In this section, we discuss how to use an SMT solver to find a DFA state numbering, such that all its state updates can be executed as simple arithmetic or logical operations in data-plane memory.

3.1 Definition and Notation

A DFA is a five-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

(1) $Q = \{q_0, q_1, \cdots\}$ is a finite set of states; initial state q_0.
(2) $\Sigma = \{\sigma_0, \sigma_1 \cdots\}$ is a finite set of alphabet symbols.
(3) $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.
(4) $F \subseteq Q$ is the set of accepting states.

Our goal is to find a mapping from states to natural numbers $R : Q \rightarrow [2^M - 1] = \{0, 1, 2, 3, \ldots, 2^M - 1\}$. Here M is the number of bits stored per memory word, and is usually chosen between 8, 16, or 32. We require R to map states to

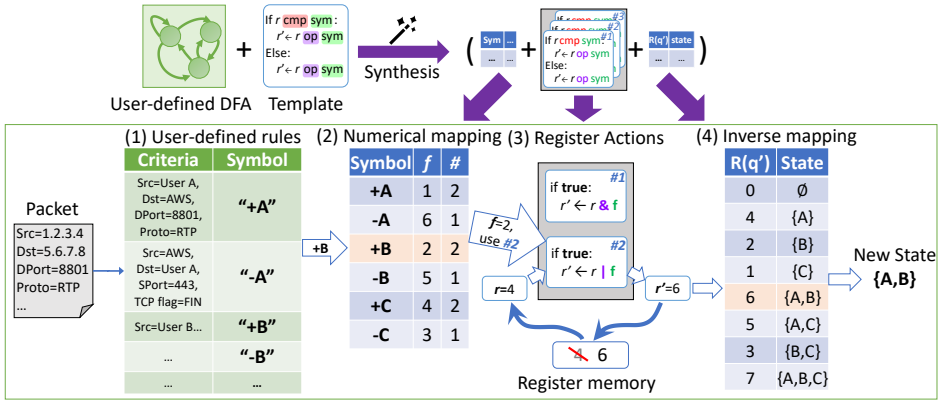


Figure 5: When a packet arrives in the data plane, it maps to a symbol according to user-defined rules. We then choose one Register Action and an input value f according to the synthesis result. After executing the Register Action, we apply the inverse mapping to obtain the new state.

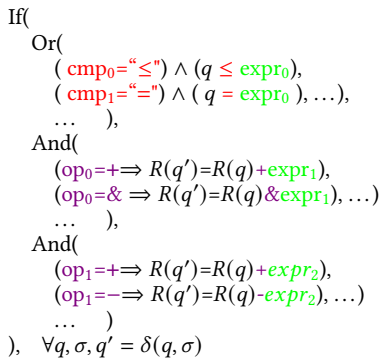


Figure 6: Different implementations of state transitions are formulated as SMT constraints.

unique numbers because we also need to map the numbers in memory back to states using the inverse mapping R^{-1}.

Besides the state mapping R, we also define two more mapping functions from the transition symbols to numbers, $f, g : \Sigma \to [2^M - 1]$, as Register Actions can take at most two variables as input. They are used as operands in the arithmetic operations to update the in-memory value. For example, if we can find f such that $R(q') = R(q) + f(\sigma)$ satisfies all transitions $q' = \delta(q, \sigma)$, we can simply perform $r' \leftarrow r + f$ in the Register Action.

Finally, when multiple Register Actions are used, we also define a function *whichop*: $\Sigma \to \{1, 2, \dots\}$ to designate which Register Action a transition uses for the memory update.

3.2 Hardware Implementation

Figure 2 presents a template for a Register Action, which corresponds to a conditional memory update operation using arithmetic and logical operations supported by the switch. When we run the Register Action, based on a comparison result, we have to reach the desired new in-memory value r' from the old value r via a single operation $r' \leftarrow r$ op sym, where op is an arithmetic or logical operation and sym is either a constant or a value already stored in packet header or metadata. This means we need to satisfy an equation $R(q') = R(q)$ op sym, i.e., transform $R(q)$ into $R(q')$ using a single operation for all transitions.

Fortunately, basic arithmetic and logical operations are expressive enough for us to manipulate the in-memory value to create complex updates. Here we make an important observation: we can apply arbitrarily complex $f(\sigma)$ and $g(\sigma)$ mappings by using lookup tables in pre-processing, since we know σ before accessing the register memory. Also, we can choose any state-to-number mapping R as long as the inverse number-to-state mapping R^{-1} exists.

Figure 5 illustrates the workflow of our hardware data-plane program:

(1) User-defined application logic first maps a packet (or a series of packets) to a particular transition symbol σ. Here we also derive an array index, if the register memory array contains DFAs for multiple flows.
(2) Pre-processing: given σ, we calculate $f(\sigma)$ and $g(\sigma)$, as well as which Register Action to apply.
(3) By executing the chosen Register Action, with $f(\sigma)$ and $g(\sigma)$ as input, we update the old value stored in memory r into the new value r', which is returned. This step needs to guarantee that for any state q where $r = R(q)$, we have $r' = R(q')$ where $q' = \delta(q, \sigma)$ is the new state.
(4) Post-processing: given $R(q')$, we use the inverse mapping to obtain the new state q', and pass it back to the application logic.

By storing different instances of a DFA in a register memory array and using a flow ID to generate the array index, we can have one DFA instance per flow for millions of different flows.

3.3 Synthesizing Mappings

Although it is possible to manually design the numerical mappings given a particular DFA, it is a tedious and difficult process for most DFAs. Furthermore, this process needs to be coupled with finding the correct combinations of comparison and arithmetic or logical operations for each Register Action.

Instead, we can use the help from an SMT solver, which is designed to search for solutions given various constraints between variables. We transform the pseudocode template

shown in Figure 2 into variable constraints shown in Figure 6 and task the SMT solver with finding the right variable assignments, including the choice of operands as well as the numerical mappings.

The variables in our SMT query are:

(1) A BitVector variable representing $R(q)$ for each state q;
(2) For each letter σ, two BitVector variables representing the values of $f(\sigma)$ and $g(\sigma)$, as well as boolean indicator variables representing *whichop* (σ);
(3) For each Register Action, $\lceil \log_2 N \rceil$ boolean variables representing which of the N choices it is making for arguments and operators to use.

The constraints include:

(1) Constraints specifying that each state q is mapped to a unique number $R(q)$;
(2) For each transition $q \times \sigma \rightarrow q' \in \delta$, given the choice of Register Action and $f(\sigma), g(\sigma)$, the Register Action's expression evaluates to $R(q')$ correctly.

If a satisfying solution is found, it will include the state-to-number mapping R, as well as the f, g and *whichop* functions. We use them to populate the lookup table in the preprocessing step, and also initialize the values stored in memory to the number corresponding to the initial state, $R(q_0)$. For convenience, we can also add a constraint requiring $R(q_0)$=0 so we don't need to initialize a zeroed-out register memory. The Register Actions themselves can be set up by filling in the template using the boolean indicator variables, which chooses the comparison and arithmetic or logical operators. Finally, we can set up the post-processing lookup table using the inverse of state-to-number mapping R^{-1}.

4 OPTIMIZING FOR EXPRESSIVENESS AND SYNTHESIS TIME

The basic synthesis technique described in Section 3 is sufficient for executing simple DFAs in the data plane. However, sometimes the solver struggles to find a model for more complex DFAs, either taking too long to find a solution (timeout) or reporting it is impossible to find a mapping (unsatisfiable). We describe how we use multiple code templates to balance between expressiveness and solving time.

The Tofino programmable switch supports a rather complex conditional update logic inside a Register Action. Although using the most expressive template provides the most possibility for synthesizing complex DFAs, such expressiveness is often unnecessary and slows down the synthesis significantly.

Instead, we prepared several different Register Action templates with varying complexities, as shown in Figure 7. Sometimes the expressiveness of complex templates are necessary as the simpler ones will cause the SMT solver fail to find any solution (output "unsatisfiable"); other times a simpler template is sufficient and will help the solver find a solution much faster. We also note that we have the option to use one or multiple Register Actions, and restricting the solver to use only one or two fewer Register Actions also reduces synthesis time (at the cost of expressiveness).

Note that the two most complex templates (Figure 7d and 7e) use a hardware feature available on Tofino called "paired mode" that stores a pair of two integers (r_0, r_1) in one register memory address. For these templates, we define the state-to-number mapping using two functions $R_0, R_1 : Q \rightarrow \{0,1\}^M$ in the synthesis, and only require $R_0(q)$ to be unique across different states (and use it for the inverse lookup). Although these Register Action templates are tailored for the Tofino switch, we could synthesize for different programmable hardware by removing unsupported templates and adding new ones matching the hardware's capability. In Section 5 we show empirical evaluation results comparing the solving time and expressiveness when using different templates, as well as varying the number of arithmetic and logical operators to choose from.

In practice, given an input DFA with unknown complexity, we run multiple solver instances in parallel using different template configurations until any one instance returns a satisfying assignment.

5 EVALUATION

We run evaluation experiments to investigate how to achieve the best performance by choosing the right level of expressiveness during synthesis. In this section, we present the solving time for DFAs using different templates, as well as insights we learned from the experiments and a prototype for the Tofino programmable switch. Our framework successfully synthesized data-plane representations of all example DFAs presented in Table 1 in under 6 minutes.

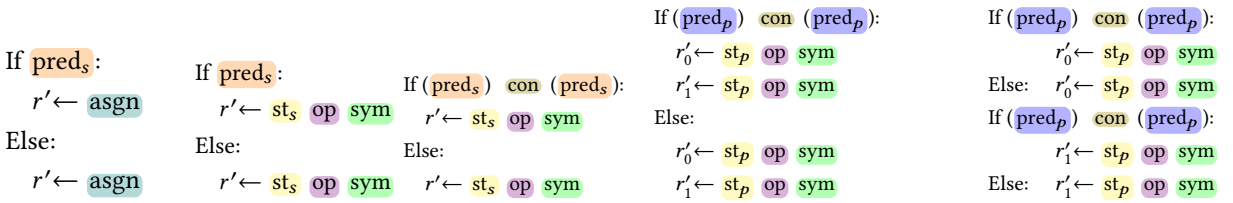
5.1 Experiment Setup

We use z3 [4, 14] v4.8.17 as our SMT solver. Our python driver program parses and validates the DFA using a custom-defined JSON format, and generates the SMT constraints for solving; it has approximately 500 lines of code. Separately, we have a data-plane Register Action simulator script that validates the solver's output.

All our experiments run on a server with two AMD 48-core CPUs (3.1Ghz) and 256GB memory, running Ubuntu 20.04. We disabled Z3's multi-threading and run all experiments in single-thread mode, with timeout set to one hour.

We use several DFAs representative of practical network measurement workload to benchmark our technique. First, we use the examples in Table 1: tracking a TCP handshake,

Key: cmp $\in \{\leq, =, \geq, \neq\}$, asgn $\in \{r, f, g, \text{const}\}$, op $\in \{+, -, \&, |, \bigoplus\}$, sym $\in \{f, g, \text{const}\}$, con $\in \{\&, |, \text{left}, \text{right}\}$, $st_s \in \{r, \text{const}\}$, $st_p \in \{r_0, r_1, \text{const}\}$, $pred_s := (st_s + sym + \text{const}\ cmp\ 0)$, $pred_p := (st_p + sym + \text{const}\ cmp\ 0)$.



(a) Assignment **(b) Conditional** **(c) Two Conditions** **(d) Paired** **(e) Paired+TwoBranch**

Figure 7: Each bubble indicates a distinct choice that the solver makes among the options indicated for that color. Different Register Action pseudocode templates provide a trade-off between expressiveness and synthesis speed.

Input DFA		Assignment	Conditional	TwoCond	Paired	Paired + TwoBranch
Mobile Device		**0.50s**	1.58s	2.20s	2.61s	4.52s
Video Conf.		unsat (0.83s)	13.61s	**5.93s**	155.8s	715.9s
TCP Handshake		unsat (0.71s)	224.9s	**2.16s**	8.83s	49.85s
Simple		**0.12s**	0.17s	0.24s	0.65s	0.71s
Parallel-2		**1.36s**	1.72s	4.37s	7.61s	7.63s
Parallel-3		**13.46s**	30.05s	37.27s	92.37s	106.1s
Parallel-4		**126.8s**	351.9s	377.3s	990.1s	931.4s
Sequential-2		**0.96s**	2.02s	3.48s	6.85s	9.62s
Sequential-3		timeout	timeout	timeout	**913.5s**	timeout
Sequential-4		timeout	timeout	timeout	timeout	timeout
Fingerprinting (String Matching)	1x8	unsat (0.64s)	270s / 61%	161s / 93%	**126s / 100%**	227s / 100%
	1x10	unsat (0.82s)	530s / 33%	377s / 86%	**332s / 100%**	833s / 96%
	1x12	unsat (1.12s)	660s / 16%	553s / 74%	**786s / 95%**	1587s / 65%
	1x16	unsat (1.67s)	110s / 2%	1295s / 29%	**1704s / 44%**	2252s / 16%
	2x4	unsat (0.94s)	350s / 46%	289s / 90%	**244s / 100%**	534s / 100%
	2x6	unsat (1.43s)	1100s / 8%	873s / 41%	**1575s / 81%**	1875s / 41%
	2x8	unsat (2.12s)	858s / 1%	1374s / 17%	**1505s / 24%**	1916s / 3%
	3x4	unsat (1.86s)	1250s / 12%	993s / 42%	**1390s / 74%**	1829s / 52%
	4x4	unsat (2.86s)	all timeout	2345s / 4%	**2197s / 17%**	1915s / 11%

Table 8: Synthesis time for DFAs with different Register Action templates of varying expressiveness.

video conference participants, and mobile device state. These three DFAs are all moderately complex, with 7, 8, and 5 states respectively (and all have 6 symbols).

For video fingerprinting, we need to match quantized video segment sizes against a fingerprint string. To obtain strings for testing, We randomly sampled video segment size patterns from a video fingerprinting dataset provided by Reed and Kranch [10] and quantized the sizes into 16 levels. The "1x12" experiment matches the input to an individual video's fingerprint with length 12. We also attempt to match one input against multiple fingerprint strings simultaneously; for example, "2x8" means simultaneously matching against 2 fingerprint strings, both with length 8. The DFA needs to report which, if any, fingerprint matches with the input sequence. For each experiment, we generate 100 DFAs using different fingerprint strings.

To further challenge our synthesis framework, we also create artificially complex DFAs using composition. Inspired by Kinetic [8], we compose multiple copies of a simple 3-state, 3-symbol DFA into a larger DFA:

- In parallel composition, the larger DFA takes in a tuple of multiple symbols for each transition step, and perform transitions for each underlying DFA individually using one of the symbols. The 2nd, 3rd, and 4th-order composition has 9, 27, and 81 states respectively, with the same number of symbols as the number of states.
- In sequential composition, the larger DFA takes in the same symbol tuple at a time but only perform the transition for one underlying DFA; it starts at the first underlying DFA and only moves on to the next one once the earlier one is at its accepting state. The 2nd, 3rd, and 4th-order composition has 9, 27, and 81 transitions, and 6, 9, and 12 states respectively.

5.2 Register Action Templates

Table 8 shows the time it takes to synthesize mappings using different Register Action templates, which represent different levels of expressiveness. All synthesis runs use 4 Register Actions and bit width M=8. The string matching results shown in Table 8 have two metrics: the fraction of sampled DFAs that are successfully synthesized, as well as the average time spent for non-timeout synthesis runs.

As we can see, using the Two Conditions template is sufficient for synthesizing most DFAs, as it strikes a good balance between performance and expressiveness. However, string matching DFAs are more challenging and require the more expressive Paired template for a high success rate. Meanwhile, the most expressive Paired+TwoBranch template often timed out because of its complexity.

5.3 Choices of Operators

We also analyze the best set of arithmetic and logical operators to consider during synthesis. Figure 9 shows the "slowdown" of synthesis time when considering 3, 4, 6, and 13 different arithmetic and logical operators as part of the Register Action template, compared to successful synthesis runs using only addition operation. 3 uses subtraction

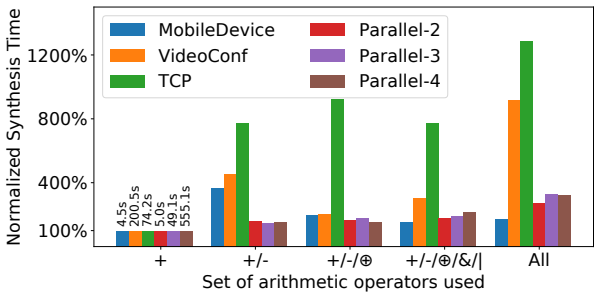


Figure 9: Synthesis time increases as we allow more arithmetic and logical operators to be used.

Resource	ALU	TCAM	Instr. words	Hash Units
Utilization	2.1%	0.5%	3.1%	1.4%

Table 10: Hardware resource utilization for Tofino.

in two directions besides addition; 4 adds XOR, and 6 adds AND/OR. Finally, 13 means we consider all variants of the bitwise logical operators (NAND, NOR, etc) that use bitwise NOT before the operands. As we can see from Figure 9, the synthesis time grows slowly with the number of arithmetic and logical operators considered. However, there is also some non-monotonicity, possibly because the extra expressiveness brought by the additional operators allows the solver to find a simpler solution faster. Also, the solver often fails to find a solution for other more complex DFAs using only arithmetic operators (addition and subtraction). It appears that using 4-6 operators (addition and subtractions, alongside AND/OR/XOR) strikes a good balance between expressiveness and performance.

We also note that the SMT solver uses more memory when the synthesis uses more arithmetic and logical operators. In our experiment, each synthesis run costs 2-4GB of memory when using 4-6 operators; however, this grows to 40GB when all 13 operators are used. This further signifies the need to avoid using an unnecessarily large set of operators.

5.4 Running on Switch Hardware

We built a code generator that transforms an SMT model into P4 code templates. Using the template, the programmer completes the program by filling in the mapping from packet header fields to DFA symbols and by programming the actions to be taken in each DFA state. Table 10 presents the resources used when compiling the P4 program for the Mobile Device example. Note that the resource utilization is nearly constant regardless of what DFA is being implemented, as the complexity of the DFA is embedded in the Register Action micro-programs and the numerical mapping. Meanwhile, the SRAM utilization depends only on the number of DFA instances allocated. Allocating 262,144 DFA instances with M=8-bit uses 5.7% of total SRAM available.

6 RELATED WORK

A number of data plane technologies use regular expressions and/or automata to analyze traffic. For instance, Jepsen et al. [7] will identify whether the contents of a *single* packet match a regular expression. BOLT [12] is similar, though it attempts to match a number of different patterns in parallel. Rather than analyzing the contents of a single packet, our tool is designed to analyze a series of packets and to recognize whether that series of packets adheres to some protocol, expressed as a state machine.

DBVal [9] allows programmers to specify and monitor the sequence of tables and actions that is applied to a *single* packet as it passes through the programmable switch. It uses the Ball-Larus encoding [1] to minimize the number of bits required to identify a particular control-flow path through the switch program. Again, our goal, and the underlying algorithms, are different as we are attempting to identify patterns in a series of packets, rather than one, and to so efficiently, we must squeeze our implementation into a single stage. The Bell-Larus encoding trick is not useful here.

NetQRE [13] uses regular expressions to define queries over a sequence of network packets, which can be translated into finite state machines. NetQRE's implementation runs these queries on a CPU—the key contribution of the current is to demonstrate how such queries could be implemented on a programmable switch such as the Intel Tofino.

Chipmunk [5] uses syntax-guided program synthesis to automatically generate user-specified transaction logic written in the high-level Domino [11] language for the data plane. A key difference is that Chipmunk is solving a more general code generation problem and hence is more constrained in its solution. Our system exploits the fact that we are synthesizing state machines, which are equivalent up to consistent renumbering of states.

7 CONCLUSION

We present a tool that transforms state machine specifications into efficient implementations for programmable switches such as the Intel Tofino. The key idea is to exploit the fact that a given state machine has many semantically equivalent implementations: By changing the state numbering and implementing transitions using the available arithmetic and logic operations, one may implement state machines of surprising complexity in a single stage of a programmable switch. If the state machine semantics and the constraints of the underlying hardware are specified using logical formula, an off-the-shelf SMT solver such as Z3 is able to find numberings and transition implementations automatically. We show that synthesis takes 3-6 minutes on a range of useful examples.

ACKNOWLEDGEMENTS.

This work was supported in part by NSF Grant 1837030. Any opinions, findings, and conclusions expressed herein are those of the authors and do not necessarily reflect those of the NSF. We thank the anonymous reviewers of SOSR'22 for their helpful comment and feedback. We would also like to thank our shepherd Fernando Ramos, as well as Ratul Mahajan and Ryan Beckett for discussions about related research.

REFERENCES

[1] Thomas Ball and James R. Larus. 1996. Efficient path profiling. *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29* (1996), 46–57.

[2] Kevin Bock, George Hughey, Louis-Henri Merino, Tania Arya, Daniel Liscinsky, Regina Pogosian, and Dave Levin. 2020. Come as you are: Helping unmodified clients bypass censorship with server-side evasion. In *ACM SIGCOMM 2020*. 586–598.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013*. 99–110.

[4] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

[5] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. Switch code generation using program synthesis. In *ACM SIGCOMM 2020*. 44–61.

[6] Intel. 2022. Intel Tofino Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[7] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast string searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR 19)*. 21–28.

[8] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 59–72.

[9] K Shiv Kumar, Ranjitha K, P S Prashanth, Mina Tahmasbi Arashloo, Venkanna U., and Praveen Tammana. 2021. DBVal: Validating P4 Data Plane Runtime Behavior. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)* (Virtual Event, USA) *(SOSR '21)*. Association for Computing Machinery, New York, NY, USA, 122–134. https://doi.org/10.1145/3482898.3483352

[10] Andrew Reed and Michael Kranch. 2017. Identifying https-protected netflix videos in real-time. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 361–368.

[11] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM 2016*. 15–28.

[12] Shicheng Wang, Menghao Zhang, Guanyu Li, Chang Liu, Zhiliang Wang, Ying Liu, and Mingwei Xu. 2022. BOLT: Scalable and Cost-Efficient Multistring Pattern Matching With Programmable Switches. *IEEE/ACM Transactions on Networking* (2022), 1–16. https://doi.org/10.1109/TNET.2022.3202523

[13] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative network monitoring with NetQRE. In *ACM SIGCOMM 2017*. 99–112.

[14] z3. 2022. z3 solver. https://github.com/Z3Prover/z3/.