Learning Concise Models from Long Execution Traces

Natasha Yogananda Jeppu, Thomas Melham, Daniel Kroening
Department of Computer Science
University of Oxford, UK
natasha.yogananda.jeppu@cs.ox.ac.uk

John O'Leary
Intel Corporation
Portland, Oregon
john.w.oleary@intel.com

***Abstract*—Abstract models of system-level behaviour have applications in design exploration, analysis, testing and verification. We describe a new algorithm for automatically extracting useful models, as automata, from execution traces of a HW/SW system driven by software exercising a use-case of interest. Our algorithm leverages modern *program synthesis* techniques to generate predicates on automaton edges, succinctly describing system behaviour. It employs trace segmentation to tackle complexity for long traces. We learn concise models capturing transaction-level, system-wide behaviour—experimentally demonstrating the approach using traces from a variety of sources, including the x86 QEMU virtual platform and the Real-Time Linux kernel.**

***Index Terms*—program synthesis, system modelling**

I. Introduction

In modern system design, hardware and software components are designed by different teams—perhaps even in different companies—and integrated only when first silicon or a stable hardware emulation model is available. Early on, it is hard to see how a given hardware IP will fare under a software workload and how software behaviour is affected by hardware design decisions. Co-design of hardware and software to meet desired specifications is therefore very challenging.

Emulation and virtual platforms provide a way to exercise software components in an environment that simulates the eventual hardware behaviour. They can easily be instrumented to record execution traces for analysis, but the traces generated are large and unstructured. So they are difficult to correlate with a high-level view of the system and its requirements. Concise, human-readable models that express high-level hardware-software interactions can provide users with a better insight into the working of the system. This can, in turn, aid in design exploration, analysis, testing and verification applications.

Several methods have been proposed to reverse-engineer automata that model system behaviour from execution traces [1]–[5], but the labels on transition arcs are limited to Boolean events that are explicit in the traces. Realising abstract and understandable models requires the user to know what abstract conditions are significant in the system evolution, for example 'the FIFO became more than half-full', and to instrument the system to record such conditions. Extensions of these traditional algorithms [6], [7] generate Extended Finite-State Machines (EFSMs) with syntactically-expressed predicates on transitions edges, but require a substantial number of trace samples from simulation of the learned model for predicate inference.

In this paper we present a new algorithm for model learning that employs program synthesis [8] to construct transition predicates that are not explicit in the trace data. Program synthesis has high computational complexity, so our algorithm uses a trace segmentation strategy to make it scalable to long execution traces. In principle, the algorithm should be applicable to trace data obtained from any source: modelling, emulation, simulation, or the system itself. But in this paper, we focus our experiments on systems modelled by virtual platforms or directly in software, and illustrate the motivation for our work in this setting. In Section IV, we show that our algorithm achieves high fidelity to published data-sheet diagrams for high-level system behaviour.

Contribution. The primary contribution of this paper is a new, scalable method for learning finite-state models from execution trace data that produces abstract, concise models. Our algorithm integrates a SAT-based approach with program synthesis techniques. It learns succinct models from traces without the additional information that is typically required by state-merging [1]. The resulting models also feature informative transition-edge predicates that are not explicit in the trace. We make an algorithmic improvement to model learning by making it scale to long traces with a segmentation approach.

II. Formal Model

We suppose that we can collect execution traces of the system we are interested in by observing a finite set of user-defined variables, $X = \{x_1, \ldots, x_k\}$, over some domain D (we simplify presentation by assuming all variables have the same domain.) The set $X' = \{x'_1, \ldots, x'_k\}$ contains corresponding primed variables, also over domain D. A primed variable x'_i represents an update to the unprimed variable x_i at the end of a discrete step. The variables in X could stand for concrete values directly observable in the system or some function of such values, depending on the user's intent. A *valuation* $v : X \to D$ maps the variables in X to values in D. An *observation* at time step t is a valuation of the variables at that time, and is denoted by v_t. A *trace* is a sequence of observations over time; we write a trace σ with n observations as a sequence of valuations $\sigma = v_1, v_2, \ldots, v_n$.

Our aim is to construct an automaton from a trace to represent behaviour captured by the trace. The learned automaton is a Non-Deterministic Finite Automaton (NFA).

Definition 1 (Non-Deterministic Finite Automaton): An NFA $\mathcal{M} = (\mathcal{Q}, q_0, \Sigma, F, \delta)$ is a state machine where $\mathcal{Q}$ is a finite set of states, $q_0 \in \mathcal{Q}$ is the initial state, Σ is a finite alphabet, $F \subseteq \mathcal{Q}$ is the set of accepting states, and $\delta : \mathcal{Q} \times \Sigma \to \mathcal{P}(\mathcal{Q})$ is the transition relation.

In our setting, all states of the automaton are accepting states, i.e., our automaton rejects by running into a 'dead end'. A symbol a of the alphabet Σ is a function $a : (X \cup X') \to D$, i.e., a pair of observations of the system. Let $\sigma = v_1, \ldots, v_n$ be a trace of the system. The symbol a_i for $i = 1, \ldots, n-1$ is

$$\begin{aligned} a_i(x) &= v_i(x) \\ a_i(x') &= v_{i+1}(x) \end{aligned} \tag{1}$$

The automaton accepts a word $w = a_1, \ldots, a_p$ over Σ for $p < n$ if there exists a sequence of automaton states $q_1, \ldots, q_{p+1}$ such that

- $q_1 = q_0$
- $q_{i+1} \in \delta(q_i, a_i)$ for $i = 1, \ldots, p$.

III. Model Learning with Program Synthesis

Our model-learning algorithm is provided in Algorithm 1. The algorithm fits into the following overall framework:

A. *Tracing infrastructure.* This records traces and performs any required pre-processing.

B. *Transition predicate synthesizer.* This generates transition predicates from trace data using program synthesis.

C. *Model construction algorithm.* This iteratively constructs an automaton from a sequence of predicates obtained from the previous step and checks its compliance with the sequence input.

We describe these components in detail in the sections that follow.

A. Tracing Setup

We use implementations of the system of interest to obtain trace data. For most of our experiments, execution traces are produced simply by instrumenting source code with print statements. This provides flexibility in getting the required information from the simulation runs. Target components of interest are identified and trace statements added at relevant points in the source code based on the end goal for analysis. Traces can also be produced using any other means, for example inbuilt tracing or logging frameworks.

B. Transition Predicate Synthesis

For observations that include non-Boolean variables, we need a way to consolidate the information they represent into expressions that will serve as transition predicates in our automaton. We use a program synthesiser to generate a state transition function $next(x)$ that provides the value of the given variable $x \in X$ in the next state. The method used is an instance of *synthesis from examples* [9]. There are many algorithms that implement this synthesis technique. We discuss choices for the synthesis algorithm in Section VII.

Trace data are used to provide concrete samples for $next(x)$, which in turn serve as constraints in the synthesis tool (line 3). For example, consider a system with a single variable x_1, and let x_1=1, x_1=2, x_1=3, x_1=4 be a trace of that system. The corresponding examples for deriving $next(x_1)$ are $next(1) = 2$, $next(2) = 3$, $next(3) = 4$. From these examples, the synthesis tool might generate $next(x_1) = x_1+1$, which we then use to model the behaviour of the variable x_1 in our NFA. Consider another system with two variables $X = \{x_1, x_2\}$ and suppose the next-state value of x_1 depends on x_2:

$$x_1' = \begin{cases} x_1 + 1 & \text{if } x_2 = 0 \\ x_1 - 1 & \text{if } x_2 = 1 \end{cases} \tag{2}$$

One trace for this system might be this: (x_1=1, x_2=0), (x_1=2, x_2=0), (x_1=3, x_2=1), (x_1=2, x_2=0). The corresponding examples for synthesis of a definition of $next(x_1, x_2)$ are $next(1, 0) = (2, 0)$, $next(2, 0) = (3, 1)$, and $next(3, 1) = (2, 0)$.

The synthesised function $next(x)$ is used to define a predicate '$x' = next(x)$' that relates observations in the current and next states, and will serve as a transition predicate in our learned automaton. To tackle the problem of synthesis complexity for long traces, a sequence of these predicates is generated by feeding segments of the trace one after the other into the predicate synthesis algorithm, using a sliding window (lines 9–13).

C. Model Construction

Our model construction algorithm takes as input a sequence of predicates $P = p_1, \ldots, p_k$ of the form just described (line 13). Each predicate is represented by an expression (a syntax tree) over variables in $(X \cup X')$. The automaton $\mathcal{M}$ to be constructed is represented as an array, each element of which encodes a transition. The i-th element in the array is a triple comprising the following symbolic variables: a state variable q_i for the state from which the transition occurs, a variable p_i' for the corresponding transition predicate and a next state variable q_i' for the state the system moves to (line 14). The sequence of predicates is divided into segments using a sliding window w, a parameter that can be tuned, and unique segments are processed further (line 15). These predicate segments are later used to encode transition sequences in the automaton. The parameter w determines the input size, and consequently the algorithm runtime. Choosing $w = 1$ will not capture any sequential behaviour but only ensures that all trace events appear in the automaton. For model learning, we would like to choose a value for w that results in a small input size but is not trivial ($w = 1$). For our experiments we performed multiple runs of the algorithm, randomly selecting a different value for w between $2 \leq w \leq |P|$ for each run, and obtained the same automaton in all scenarios. The strategy we adopt for our experiments is to fix window length $w = 3$, which is small to ensure quick results, and yet captures interesting trace patterns. The result of segmentation is the *set* of all unique subsequences of P of length w. Segmentation significantly improves runtime, especially

Algorithm 1 Model Learning Algorithm

```
1: procedure GeneratePredicate(σ' : trace of length w)
2:     σ' = v_i, v_{i+1}, ..., v_{i+w-1}
3:     next(v_k) ← v_{k+1}, for k = i, i+1, ..., (i+w-2)
4:     Synthesize next(x)
5:     return '(x' = next(x))'
6: end procedure

7: procedure GenerateModel(σ : trace of length n)
8:     w ← sliding window size
9:     Divide σ into {σ_1, σ_2, ..., σ_k}, k = (n+1-w)
10:    for each σ_i do
11:        p_i ← GeneratePredicate(σ_i)
12:    end for
13:    P ← p_1, p_2, ..., p_k
14:    Target automaton M is represented as
         M = (q_1, p'_1, q'_1), (q_2, p'_2, q'_2), ..., (q_m, p'_m, q'_m)
15:    Divide P into {P_1, P_2, ..., P_{k+1-w}} where
         P_i = p_i, p_{i+1}, ..., p_{i+w-1}
16:    N ← 1                                  ▷ Number of automaton states
17:    Constraint c_0 = ∃i, j ∈ {1, ..., m} ∋ (q_i = q_j ∧ p'_i = p'_j ∧ q'_i ≠ q'_j)
18:    Set of blocking constraints C ← {c_0}
19:    Generate the following C program:
20:        assume 1 ≤ q_i, q'_i ≤ N, for i = 1, 2, ..., m
21:        j ← 0
22:        for each P_i do
23:            for y = i to i + w - 1 do
24:                p'_j ← p_y
25:                assume q_{j+1} = q'_j
26:                j ← (j + 1)
27:            end for
28:        end for
29:        assert ⋁_{c∈C} c
30:    Run CBMC with the above program as input
31:    if assertion holds then                ▷ No candidate automaton found
32:        N ← (N + 1)
33:        go to 19
34:    else                                   ▷ Candidate automaton M found
35:        M ← learned candidate automaton from counterexample
36:        l ← length of transition sequence for subsequence check
37:        S_l ← set of all transition sequences of length l in M
38:        P_l ← set of all subsequences of P of length l
39:        if S_l ⊈ P_l then                  ▷ Subsequence check failed
40:            Encode sequences (S_l - P_l) as blocking constraints and add to C
41:            go to 19
42:        else                               ▷ Subsequence check successful
43:            return M
44:        end if
45:    end if
46: end procedure
```

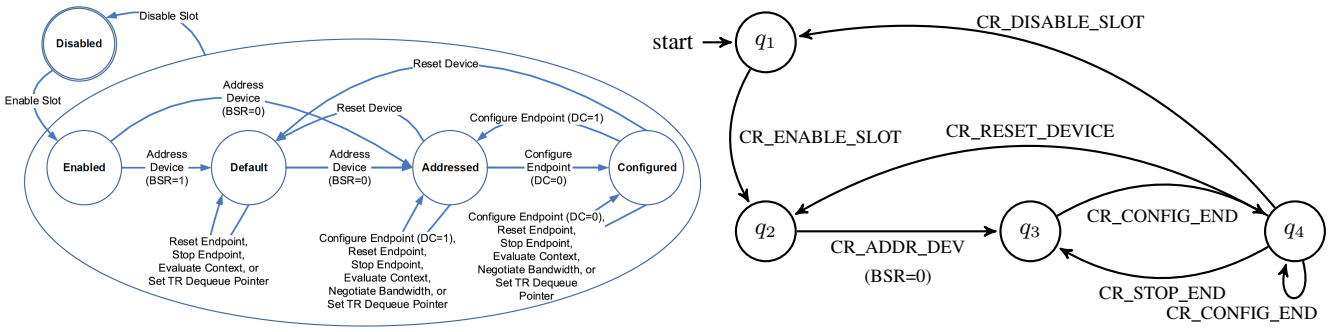


Fig. 1: USB Slot state machine provided in (a) Intel datasheet [10] and (b) model learnt by our framework.

for long traces, by leveraging the presence of repeating patterns in the trace. A detailed discussion of the need for segmentation and its effect on scalability is given in Section V.

To construct the model, we search systematically for an N-state NFA whose behaviours include all the unique segments previously identified and has at most one transition from any state labelled with any given predicate. The latter is encoded as a blocking constraint (line 17) and added to a set of constraints that we maintain to restrict the automaton throughout the construction process (lines 17–18). For a given N, we hypothesize no such automaton exists and use a model checker to check the hypothesis. This is done by first encoding the hypothesis as a C program (lines 19–29). For the check we use the C Bounded Model Checker (CBMC) [11]. We fix the number of automaton states by restricting the state variables of $\mathcal{M}$ to take values between 1 and N (line 20). Lines 21–28 ensure that the automaton always includes the corresponding transition predicates in the sequence they appear in the segments of P. We assert that at least one of the blocking constraints is true (line 29). The program, along with this assertion, is then fed to CBMC (line 30). If the assertion holds (line 31), it indicates that for all assignment of values in the range 1 to N to state variables q_i and q_i' of $\mathcal{M}$, at least one of the blocking constraints on the automaton hold. This implies that there is no N-state automaton that meets our specifications; in this case we increment N and repeat the search (lines 31–33). We begin model construction with $N = 1$ and increase the number of states by 1 if such an automaton cannot be learned. This ensures that we learn the smallest automaton that contains all subsequences of P of length w.

A counterexample to the assertion is an assignment of values to state variables of $\mathcal{M}$ that encodes an N-state automaton that contains all subsequences of P of length w and also, does not satisfy any of the blocking constraints (line 34). Once CBMC has constructed a candidate model, we check its compliance with P by looking for *invalid* transition sequences in $\mathcal{M}$ (lines 35–39). A transition sequence is said to be invalid if it is not a subsequence of P. We check if all transition sequences in the model of a given length l are subsequences of P (line 39). The parameter l can be tuned to change the degree of generalisation. However, a higher value for l implies tighter constraints on the model moving towards a more exact representation. It is known that learning exact automata from trace data is NP-complete [12]. Hence, we have used $l = 2$ to ensure that it is not too complex for the model checker to solve but at the same time does not over-generalise to fit the trace. We encode invalid sequences as additional blocking constraints on the automaton and repeat the search (lines 39–41). This refinement loop gleans further information from sequence P. The algorithm returns the N-state automaton $\mathcal{M}$ if such a candidate automaton is found and the compliance check is successful (lines 42–43).

IV. Benchmarks

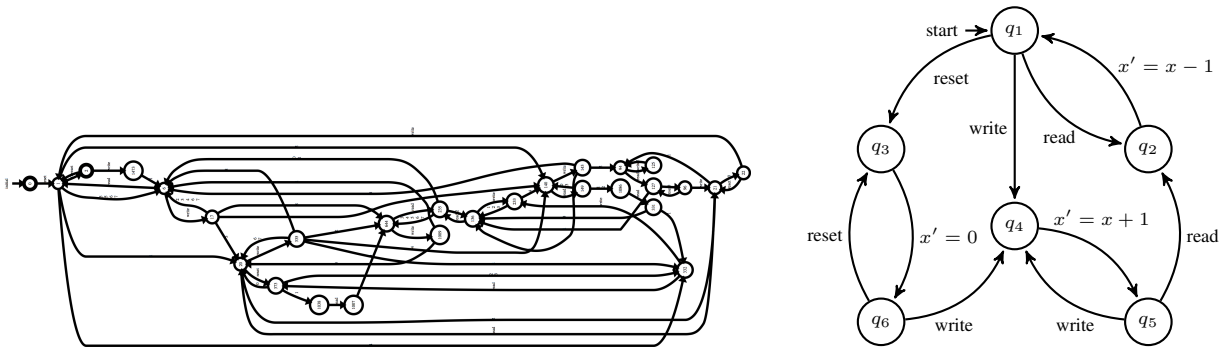
We demonstrate our model learning approach from execution traces for six examples. We compare our approach against the traditional state-merge algorithm and provide experimental evidence to demonstrate scalability of our algorithm. Four of the six benchmarks use the QEMU virtual platform to emulate an x86 system, including the hardware components. The virtual platform runs a full CentOS Linux distribution and is given an application to exercise system behaviours of interest. The other two benchmarks are artificial and enable us to benchmark particular aspects of our method.

USB xHCI Slot State Machine The Extensible Host Controller Interface (xHCI) specifies a controller's register-level operations for USB 2.0 and above. In this example we look specifically at slot-level operations that take place when we access a virtual USB storage device as implemented in QEMU x86 platform emulation. The framework learns the automaton given in Fig. 1b resembling the Intel datasheet diagram in Fig. 1a.

It is worth noting that the model learning algorithm is able to generate an accurate representation of system behaviours that are exercised under a given application load. Some transitions in Fig. 1a do not appear in the learned model because either QEMU does not implement those scenarios or that the application load does not drive the system into those scenarios. The models generated thus provide valuable coverage information.

QEMU USB Interface Emulation We use the same setup as above but record all interface events that take place when a virtual USB storage device is attached to the virtual platform. The resulting trace records the series of ring fetch and ring write operations between the command ring and event ring of the xHCI protocol. Our algorithm learned a concise 7 state automaton, while the smallest model generated by state-merge had 91 states (Table II).

QEMU Serial I/O Port We used QEMU's x86 emulation of a serial I/O port and recorded changes in queue length. The trace contains (numerical) queue length data along with (Boolean) read, reset and write events on the queue. Our algorithm was able to generate expressions for transition predicates as given in Fig. 2b. We were unable to take the queue to its full capacity due to very quick read-writes and frequent resets.



(a) Model generated by State Merge

(b) Model learnt by our framework

Fig. 2: QEMU Serial I/O Port

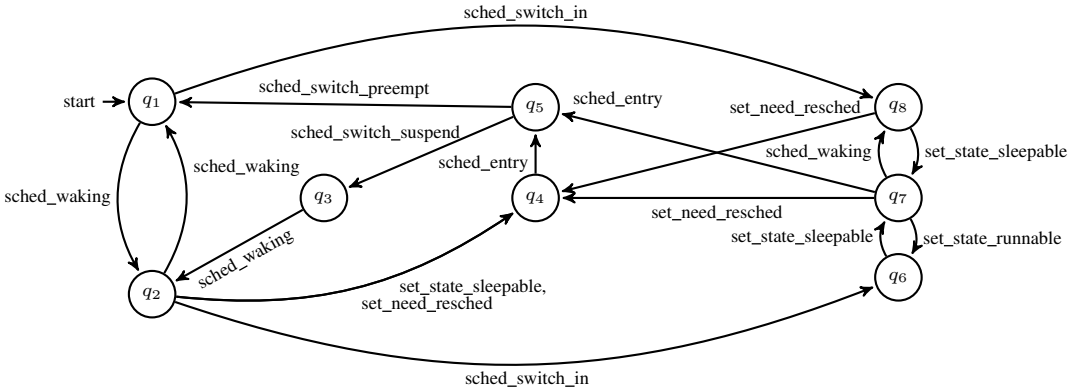


Fig. 3: Model of RT-Linux Kernel Thread Scheduling learnt by our framework

Counter We trace a simple program that counts from 1 up to a threshold T, which we set to 128; after it reaches the threshold it counts back down to 1. This is repeated N times. We observe the value of the counter. The synthesis component of our algorithm automatically generates the expected transition predicates $(x' = x+1)$, $(x \geq 128)$, $(x \leq 1)$ and $(x' = x - 1)$ using just the values from the trace.

This benchmark is particularly interesting as it has constants, like the value of T, in the predicates. The ability to automatically produce these constants depends on the synthesis tool used and the approach to synthesis. We discuss this in detail in Section VII.

Integrator Control applications frequently track an integral of an input signal. We implement an anti-windup integrator where the computed output op is saturated at predefined thresholds, 5 and -5. The input ip is restricted to take values $\{1, 0, -1\}$. The trace contains valuations of (ip, op) pairs at discrete time steps.

The algorithm generates complex transition predicates $(op' = op + ip)$, $(op' = op)$ and $(op = 5 \wedge ip = 1) \vee (op = -5 \wedge ip = -1)$. The transitions on $(op' = op + ip)$ encode integrator behaviour outside saturation. In the learned model, transitions on $(op = 5 \wedge ip = 1) \vee (op = -5 \wedge ip = -1)$ are always followed by transitions on $(op' = op)$; hence, accurately capturing behaviour at saturation.

Real Time Linux Kernel We generated an automaton describing the behaviour of threads in the Linux PREEMPT_RT kernel on a single core system. This is motivated by work in [13], [14] where hand-drawn models of the kernel are used as monitors for runtime kernel verification. For this example, we used the built-in Linux tracing infrastructure *ftrace* to trace scheduler-related calls made by the thread under analysis, as described in [14]. We used the Linux PREEMPT_RT kernel version 5.0.7-rt5 on a single core QEMU emulated x86 machine for our experiments.

Initial attempts at modelling thread behaviour with our algorithm, using the *pi_stress* tests from the *rt-tests* suite as system load, revealed that some states in the hand-drawn model provided in [14] are not covered by the given load. On running an additional kernel module to cover these corner cases, we obtain the automaton in Fig 3. This experiment provides evidence in support of potentially using the models learned by our algorithm for functional test coverage analysis.

V. The Benefit of Trace Segmentation

To learn models from execution traces we require efficient and scalable mechanisms for mining useful information from large amounts of trace data. More often than not, execution traces of a system contain recurring patterns, which we exploit to speed up model learning. To evaluate the benefit of our segmentation technique, we give the results of a runtime comparison of the algorithm for segmented and non segmented trace inputs for all six examples (Table I). We observe that the segmentation enables our algorithm to scale: Fig. 4 is a plot of the runtime against trace length for exponentially increasing trace lengths for the integrator example. Segmentation breaks down a large problem into multiple small instances that have manageable runtime. We leverage the presence of trace patterns to significantly reduce execution time as it is sufficient to process repeating segments once.

VI. Comparison with State Merge Algorithms

State merge algorithms are the established approach to model generation from traces. Traces are first converted into a Prefix Tree

Example	N	Trace Length	Full Trace (s)	Segmented Trace (s)
USB Slot	4	39	14.1	**9**
USB Attach	7	259	2249.5	**915.4**
Counter	4	447	249.1	**95.9**
Serial I/O Port	6	2076	23590.5	**60.2**
Linux Kernel	8	20165	>16 hours	**516.3**
Integrator	3	32768	>16 hours	**3495.6**

TABLE I: Runtime comparison for segmented and non-segmented trace input. For a fair comparison, we begin learning with number of states equal to N.

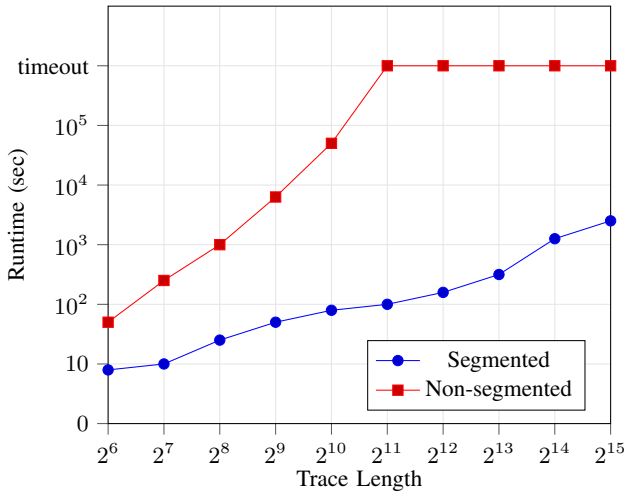


Fig. 4: Graph plot (log–log plot) comparing runtime for segmented and non-segmented trace input for the integrator example.

Acceptor (PTA). Model inference techniques are used to identify pairs of equivalent states to be merged in the hypothesis model. One of the most popular and accurate inference techniques is Evidence-Driven State Merging (EDSM) [2]. The MINT (Model Inference Technique) tool [5], [15], implements a variant of EDSM using data classifiers to classify trace events based on next event. States for which the classifier predicts the same next event are merged. It also provides support for the traditional kTails approach to state merge [1].

A runtime comparison of MINT against our approach reveals that the state-merge algorithm is significantly faster (Table II) but generates large automata that are difficult to comprehend, as shown in Fig. 2a. By contrast, our framework learns models that are much more succinct (Fig. 2b) and capture system behaviour accurately. The MINT tool was unable to produce models from long traces of length >20,000 for the Linux kernel and integrator examples whereas our approach successfully produced concise automata in both cases.

VII. Discussion of Program Synthesis Engines

Virtually all tools that perform program synthesis implement a form of Counter Example Guided Inductive Synthesis (CEGIS) [8]. The program that is generated is usually required to conform to a grammar, which is given as part of the problem description. Tools that require this grammar implement Syntax-Guided Synthesis (SyGuS) [16]. We experimented with two program synthesis tools for generating transition predicates for the automaton: CVC4 version 1.6 [17], [18], which by default employs SyGuS, and fastsynth, which is based on the work done in [19].

The SyGuS-based approach requires a grammar for the program. The key effort is to determine the constants that are required; they

Example	Trace Length	Runtime (s)		Number of States	
		State Merge	Model Learning	State Merge	Model Learning
USB Slot	39	**8.7**	14.5	6	**4**
USB Attach	259	**35.1**	3615.1	91	**7**
Counter	447	**12.1**	98.6	377	**4**
Serial I/O Port	2076	**28.6**	137.4	28	**6**
Linux Kernel	20165	≈ 5 h	**4173.6**	no model	**8**
Integrator	32768	≈ 5 h	**3497.2**	no model	**3**

TABLE II: Runtime analysis of state-merge vs. model learning.

have to be adjusted manually for every model. Fastsynth also implements CEGIS but does not rely on a user-specified grammar to restrict the search space. Fastsynth ignores any grammar given as part of the problem and produces the smallest function that satisfies the constraints. Any constants that may be required are generated automatically.

CVC4 also implements an alternative algorithm that does not require syntax guidance; however, that produces trivial solutions. For example, given the trace sequence 1, 2, 4, 8, CVC4 generates *(ite (= x 4) 8 (ite (not (= x 2)) 2 4))* whereas fastsynth produces the expression $x + x$, which is a better fit for our problem. The type of transition predicates synthesised depends on the ability of the program synthesis tool and the approach to synthesis. A suitable synthesis tool can be chosen based on the target models we wish to obtain and the application domain.

VIII. Related Work

Manually creating abstract system models is time-consuming and error-prone, and this has prompted numerous research efforts aimed at automated model learning. The most common approach to automatically reverse engineer models from execution traces is state merging [1]. The process involves converting traces into prefix tree acceptors (PTA), and then applying model inference techniques to determine which states are to be merged. In the traditional kTails approach two states in the PTA are merged if they are *k-equivalent*. The parameter k is used to change the degree to which the model generalises. A variant of the algorithm [5] additionally uses data classifiers to determine state equivalence.

Conventional automata learning approaches are partial because they fail to model how system variables change during execution. An extension of the state merge algorithm [7] generates "computational" state machines. Data update functions over transitions are automatically generated using genetic programming. This method requires additional trace data, over and above the trace data used to generate the initial model, for transition predicate inference. The GK-Tails [6] algorithm integrates Daikon [20] with the kTails approach to derive transition predicates for Extended Finite State Machines (EFSM) that represent software behaviour. The type of expressions generated is however restricted. kTails based algorithms use instances of only positive behaviour and hence run the risk of over-generalising [12].

A popular model inference algorithm, Evidence-Driven State Merge (EDSM) [2], overcomes the problem of over-generalisation by using both positive and negative instances of behaviour to determine equivalence of states to be merged based on statistical evidence. In an extension of this work [4], finite automata inference is mapped to a graph-colouring problem based on the red-blue EDSM framework [2]. Models are generated by converting the problem into SAT and using state-of-the-art SAT solvers to get an optimal solution.

To avoid over-generalisation in the absence of labelled data, the EDSM algorithm was improved to incorporate inherent temporal behaviour in the models [3], [21]. Models are checked against LTL

properties to validate state merges as they are encountered. In an attempt to model and verify software systems [22], state machines describing software behaviour are generated by checking a hypothetical, manually drawn model against the code. The user specifies a set of states and state invariants which are translated into relevant pre and post conditions in the code. State merge based algorithms do not focus on producing the most succinct models but rather produce a good enough approximation that conforms to the trace [23].

SAT-based approaches to model generation have thus gained popularity due to their ability to produce exact state machines [23], [24]. Similarly, several algorithms have been developed that use SAT together with state merge to generate automata from positive and negative traces [4], [25], [26]. In general, these methods work by first representing the problem using Boolean variables and generating a Boolean formula that constrains them. A SAT checker then generates a hypothesis model, which is verified using LTL properties of the system. The SAT-based approach has been put to practice to construct plant models as Moore machines using behavioural instances, with LTL properties used as constraints [25], [26].

A classic automata learning technique, Angluin's L* algorithm [27], employs a series of equivalence and membership queries to an oracle, the results of which are used to construct the automaton. When the trace does not explicitly contain transition predicates, L* fails to learn behaviour seen in the data. The absence of an oracle often restricts the use of this algorithm for abstracting large systems.

IX. Conclusion and Prospects

In this paper we have outlined a novel scalable program synthesis based approach to learn models from long execution traces. The models produced are concise and accurately represent the system's behaviour. Our approach can handle traces that contain more than just Boolean events by synthesising expressions for system variable state predicates. We have compared our approach with state-merge algorithms for a range of benchmarks and evaluated the scalability of our algorithm. Our abstract models have several potential applications: they can summarise which aspects of system behaviour have been covered by a suite of tests, they provide starting points for model-based test generation, perhaps to close coverage holes, and they could be used as candidate inductive invariants [28], which, in turn, could be used to prove properties of the system.

Going forward, we wish to look at these applications and address the question of how to efficiently exercise the system to produce relevant traces. We are particularly interested in its utility in invariant synthesis for property verification using the models as candidate invariants in the inductive invariant refinement loop.

Although we demonstrate our approach on systems given as virtual platforms, we wish to explore its value in other domains as well.

X. Acknowledgements

This research was funded by the Semiconductor Research Corporation, Task 2707.001, and Balliol College, Jason Hu scholarship. We thank Daniel Bristot for his help with the RT Linux Kernel.

References

[1] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. Comput.*, vol. 21, no. 6, pp. 592–597, Jun. 1972.

[2] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm," in *Grammatical Inference*. Springer, 1998, pp. 1–12.

[3] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, Sep. 2008, pp. 248–257.

[4] M. J. H. Heule and S. Verwer, "Software model synthesis using satisfiability solvers," *Empirical Software Engineering*, vol. 18, no. 4, pp. 825–856, Aug 2013.

[5] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions," *Empirical Software Engineering*, vol. 21, no. 3, pp. 811–853, Jun 2016.

[6] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *International Conference on Software Engineering (ICSE)*, 2008, pp. 501–510.

[7] N. Walkinshaw and M. Hall, "Inferring computational state machine models from program executions," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 122–132.

[8] S. Gulwani, A. Polozov, and R. Singh, "Program synthesis," in *Foundations and Trends in Programming Languages*, vol. 4. NOW, August 2017, pp. 1–119.

[9] S. Gulwani, "Synthesis from examples," in *3rd Workshop on Advances in Model-Based Software Engineering (WAMBSE)*, 2012.

[10] Intel, *eXtensible Host Controller Interface for Universal Serial Bus (xHCI)*, 2017 November. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf

[11] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 2988. Springer, 2004, pp. 168–176.

[12] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, pp. 302–320, 1978.

[13] D. de Oliveira, T. Cucinotta, and R. S. de Oliveira, "Efficient formal verification for the Linux kernel," in *Software Engineering and Formal Methods*. Springer, 2019, pp. 315–332.

[14] D. Bristot de Oliveira and S. Sant'anna, "Modeling the behavior of threads in the PREEMPT RT Linux kernel using automata," in *Proceedings of the Embedded Operating System Workshop (EWiLi)*, 11 2018.

[15] N. Walkinshaw, *MINT framework Github repository*, 2018. [Online]. Available: https://github.com/neilwalkinshaw/mintframework

[16] 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, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 1–8.

[17] *CVC4*. [Online]. Available: http://cvc4.cs.stanford.edu/web/

[18] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification (CAV)*, vol. 6806. Springer, 2011, pp. 171–177.

[19] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Counterexample guided inductive synthesis modulo theories," in *Computer Aided Verification*. Springer, 2018, pp. 270–288.

[20] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *International Conference on Software Engineering (ICSE)*. ACM, 1999, pp. 213–224.

[21] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin, "Reverse engineering state machines by interactive grammar inference," in *Proceedings of the 14th Working Conference on Reverse Engineering*, ser. WCRE '07. IEEE Computer Society, 2007, pp. 209–218.

[22] W. Said, J. Quante, and R. Koschke, "Reflexion models for state machine extraction and verification," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 149–159.

[23] V. Ulyantsev, I. Buzhinsky, and A. Shalyto, "Exact finite-state machine identification from scenarios and temporal properties," *CoRR*, vol. abs/1601.06945, 2016.

[24] V. Ulyantsev and F. Tsarev, "Extended finite-state machine induction using SAT-solver," in *International Conference on Machine Learning and Applications and Workshops*, 2011, pp. 346–349.

[25] I. Buzhinsky and V. Vyatkin, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1521–1530, Aug 2017.

[26] I. Buzhinsky and V. Vyatkin, "Modular plant model synthesis from behavior traces and temporal properties," *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–7, 2017.

[27] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[28] E. M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*, 2nd ed. MIT Press, 2018.