Proceedings Paper:
Walkinshaw, N. and Hall, M. orcid.org/0000-0002-9408-2996 (2017) Inferring Computational State Machine Models from Program Executions. In: 32nd IEEE International Conference On Software Maintenance And Evolution (ICSME). IEEE , pp. 123-133.

https://doi.org/10.1109/ICSME.2016.74

Inferring Computational State Machine Models from Program Executions

Neil Walkinshaw
Department of Computer Science
The University of Leicester, UK
Email: n.walkinshaw@le.ac.uk

Mathew Hall
Department of Computer Science
The University of Sheffield, UK
Email: mathew.hall@sheffield.ac.uk

***Abstract*—The challenge of inferring state machines from log data or execution traces is well-established, and has led to the development of several powerful techniques. Current approaches tend to focus on the inference of conventional finite state machines or, in few cases, state machines with guards. However, these machines are ultimately only partial, because they fail to model how any underlying variables are computed during the course of an execution; they are not *computational*. In this paper we introduce a technique based upon Genetic Programming to infer these data transformation functions, which in turn render inferred automata fully computational. Instead of merely determining whether or not a sequence is possible, they can be simulated, and be used to compute the variable values throughout the course of an execution. We demonstrate the approach by using a Cross-Validation study to reverse-engineer complete (computational) EFSMs from traces of established implementations.**

I. Introduction

Reverse-engineered models that accurately capture the behaviour of a software system are useful for a broad range of software maintenance, validation, and verification tasks. State machine inference, which is the subject of this paper, has been recently used to expose security vulnerabilities in Android apps via UI analysis [1], [2], to provide API usage models of black-box software components [3], to infer requirements from user scenarios [4], and to devise new tests for network protocols [5].

State machine inference is a well-established activity, – numerous approaches have been proposed since Bierman and Feldmann's seminal *k*-tails paper [6]. Such approaches have traditionally been concerned with the inference of conventional Finite State Machines from traces [7], [8], [9], [10], [11], [12], [13]. Recently, however, several attempts have been made to infer *Extended* Finite State Machines (EFSMs [14]). EFSMs capture the possible sequencing of events that can occur in a system (e.g. inputs or method calls) , along with the corresponding conditions on and changes to the underlying data state[1]. Lorenzoli *et al.* [17] proposed the first approach – GK-Tails – which links the traditional *k*-tails algorithm [6] with Daikon [18], which infers guards on transitions. This has since been expanded upon by several authors [19], [20], [21], [3].

Current attempts to infer EFSMs suffer from one important, overarching limitation: The inferred models are only partial – they are not *computational*. Approaches infer a state transition system with data-guards, but they do not include the *functions* that actually compute the changes on the data. As such, they can provide abstract summaries of possible program behaviours, but cannot, for example, be used to simulate an actual execution. Given an inferred model, it is only possible to determine whether or not a particular sequence of events (with corresponding data values) is valid / accepted. However, if the data values are *not* given, it is impossible to compute them.

In this paper we introduce an approach to augment inferred state machines with the functions that operate on the underlying data state. The approach can in principle be applied to any inferred Finite State Machine, and is not dependent upon a specific inference algorithm. The only constraint is that it is possible to map the transitions in the inferred model to the data values from which one wishes to infer the functions.

The approach operates in two phases. The first phase builds for each transition a 'training set'; for each variable a list of values are obtained for before and after the execution of the transition. In the second phase, Genetic Programming [23] is used to infer a function for each variable that is able to approximate the underlying computation. The result is a set of computational functions that can be mapped to each transition in the inferred model, so that the two can be used together to form a complete, 'computational' EFSM.

The specific contributions are as follows:

- A technique by which to post-process state machines that have been inferred from execution traces, to enhance them with data transformations.
- An openly available implementation.
- An evaluation of the accuracy of the inferred functions with respect to two published EFSMs.
- A proof-of-concept case study, showing how a model can be inferred from a Java class (the Apache Commons Math SimpleRegression class), and can be used as a basis for inference-driven testing.

Section II introduces the notions of EFSMs, EFSM inference, and Genetic programming. Section III introduces our technique by which to infer update functions. Section IV presents the quantitative evaluation of model accuracy.

[1] It is worth noting that there are various similar formalisms such as Abstract State Machines [15] or Event-B models [16] that can in this context be treated as equivalent to EFSMs.

Section V contains the inference-driven testing case study. Section VI discusses related work, and section VII presents our conclusions and gives an overview of our ongoing and future work.

II. Background

In this section we start with some brief preliminary definitions of EFSMs and the EFSM inference problem. This is followed by an illustration of how current techniques cannot compute the underlying data state. We then provide a brief introduction to the key concepts in Genetic Programming, upon which we will be developing the inference approach.

A. EFSMs and EFSM Inference

An EFSM is a conventional Finite State Machine that has been extended to include guards and (potentially) update functions over some data state. Here we present a slightly simplified definition of that given by Cheng and Krishnakumar [14] (we do not explicitly highlight output symbols, although these could trivially be added).

Definition 1 (Extended Finite State Machine (EFSM)): An EFSM M is a tuple $(S, s_0, E, V, \Delta, U, T)$.

- S is a set of states, $s_0 \in S$ is the initial state.
- E is defined as the set of events.
- V is a set of variables mapped to their corresponding values.
- $\Delta : V \rightarrow \{True, False\}$ is the set of *data guards*.
- U is a set of *update transformations* $V \rightarrow V$.
- T is a transition relation such that $T : S \times \Delta \times E \rightarrow S \times U$.

We use the subscript as a shorthand to refer to a particular component of an EFSM. For example, M_S refers to the set of states S in machine M. Also as a shorthand, we use V_{names} to refer to the names of a set of variables, and V_{vals} to refer to the set of values. □

The EFSM inference challenge considered in this paper is based on the notion of *traces* of events in a software system. These are defined as follows.

Definition 2 (Events and Traces): An *event* is a tuple $(l, Vars)$, where l is the name of a function and $Vars$ is a set of tuples (n, c) mapping variable names n to their concrete values c. A *trace* $t \in Tr$ is a finite sequence of events $\langle (l, Vars)_0, \ldots (l, Vars)_n \rangle$. □

The practical process of encoding a trace depends to an extent on the system under analysis, and upon the aspects of behaviour that are of interest. In some cases we might be given logs of system behaviour (c.f. log-analysis work by Ohmann *et al.* [22]). In other cases the system might be instrumented to focus on events and variables of interest [7].

The challenge is to infer an EFSM, given only a set of traces. The resulting EFSM should accurately predict the behaviour of the underlying system. One typical means of assessing accuracy (c.f. previous work [3]) is to present the inferred EFSM with a set of traces that were not used for the inference, which may or may not originate from the system under analysis, and to measure the proportion of those traces that are correctly classified as valid / invalid.

```
public class Recurse {

    public static void main(String[] args){
        int depth = Integer.parseInt(args[0]);
        recurse(depth);
    }

    private static void recurse(int depth) {
        System.out.println("recurse "+depth);
        if(depth == 0)
            baseCase();
        else {
            depth--;
            recurse(depth);
        }
    }

    private static void baseCase() {
        System.out.println("basecase");
    }

}
```

```
-------------                   -------------
recurse 3                       recurse 5
recurse 2                       recurse 4
recurse 1                       recurse 3
recurse 0                       recurse 2
basecase                        recurse 1
-------------                   recurse 0
recurse 2                       basecase
recurse 1                       -------------
recurse 0                       recurse 0
basecase                        basecase
-------------                   -------------
```

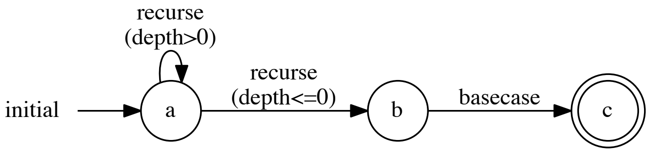


Fig. 1. A simple Java program that recurses for a given value (depth), four sample traces, and an EFSM inferred by current techniques (in this case MINT [3]).

B. Motivating Example

We motivate the problem with a small toy example, illustrated in Figure 1. Let us suppose that we wish to infer a model that captures the order in which the methods `recurse` and `basecase` are called. For each method we print out its name (i.e. `recurse` or `basecase`), along with any associated data values (i.e. `depth` for `recurse`). Running this on four inputs (depth=3,2,5, and 0) gives rise to the traces shown below the code.

A conventional EFSM inference technique applied to these traces will produce the model shown at the bottom of Figure 1. This one is produced by the MINT inference technique [3] (other inference techniques produce different transition systems or guards, ultimately accepting and rejecting different sets of traces). Ultimately however, the outputs consist of a state transition system, where transitions are (sometimes) accompanied by a guard.

Problem: State machines of the sort shown in Figure 1 are only partial. They specify *what* can happen (which sequences are possible), but fail to describe *how* the underlying data state is changed. In other words, they do not provide the update function U in definition 1.

The absence of update functions has two important ramifications. Firstly, the inferred model may falsely classify invalid traces as valid. For example, the trace $\langle(recurse, \{depth = 17\}), (recurse, \{depth = 0\}), (basecase, \{\})\rangle$ would be accepted by the inferred model, even though we know from looking at the code (and we can infer from the traces) that this sequence should not occur (*depth* cannot jump from 17 to 0).

The second (possibly more important) problem is that the model is not *computational*. It cannot be queried to predict what will happen. We cannot provide an initial value for depth (e.g. $depth = 17$), and predict the resulting path taken through the model, along with the corresponding changes to the data state. We cannot examine properties, e.g. whether the value of *depth* can ever be negative.

C. Genetic Programming

This paper will present an approach that uses a technique called Genetic Programming (GP) [23] to address the issues discussed above. GP has recently found numerous applications in Software Engineering, and has been suggested as a possible technology for Reverse-Engineering by Harman *et al.* [**?**]. We conclude this section by providing a brief, generic introduction to the essential notions in GP. This is necessarily brief for space reasons, and we only refer to the GP concepts that we have chosen to use in this work. For a broader, more comprehensive overview the reader is referred to Poli *et al.*'s field guide [25].

In (tree-based) GP, candidate programs are formulated as abstract syntax trees, where branch nodes correspond to 'non-terminals' representing functions, and leaf-nodes represent atomic values or variables (terminals). GP is an approach to synthesise these programs by evolution. The basic loop is as follows (terms in italics will be elaborated below):

1) Generate an initial population of programs as random compositions of non-terminals and terminals.
2) Execute each program and evaluate it according to some *fitness function*.
3) *Select* the best programs from the population.
4) Create a new population by a process of *cross-over* and *mutation*.
5) Repeat from step 2 until some stopping criterion is met.

Fitness function: The fitness function provides a metric for the accuracy of the candidate program. Fitness is evaluated by executing a candidate on all available inputs, and by comparing the resulting set of computed outputs to the outputs in the trace data. If the output is numerical, the fitness function is taken as the average absolute distance between the predicted and the actual values. For nominal outputs the fitness is calculated as the proportion of instances where the outputs were identical.

Selection: Step 3 is responsible for selecting good candidates from the population, so that they can be fed into the next generation. A popular approach, which we adopt here, is Tournament Selection [25]. Groups of candidates are chosen at random, and the best individual is chosen to be fed to the next generation. In our case the selection process is *elitist*, this means that the best individual from one generation is always preserved for the next one.

Crossover and Mutation: The candidates that were selected in step 3 are subjected to a mixture of crossover and mutation (the frequency at which they occur is given in probabilistic terms). We choose to use the most common form cross-over called *subtree-crossover* [25]. Mutation is carried out by selecting a random node in a tree and changing it.

Arbitrary crossover or mutation can easily lead to nonsensical programs - for example by using String terminals with a function that expects integer parameters. Strongly-typed GP [25] prevents this from happening by ensuring that every terminal and non-terminal has a declared type. In a strongly-typed GP, every crossover or mutation operation is constrained so that the result fits the type-constraints.

Termination and result: The loop either terminates once a candidate has been identified that cannot improve in terms of fitness, or once the number of iterations hits a given limit. The resulting programs are usually expressed in prefix-notation – for example the expression '(x+1)/2' becomes '/(+(x,1),2)'.

III. Inferring EFSM Update Functions with GP

This paper presents a technique that uses GP to solve the problem of missing update-functions discussed in Section II-B. We start with a description of the technique itself. this is followed by a walk-through on a small example, and a discussion of the corresponding implementation.

A. The Inference Algorithm

The technique starts from a state transition system and a corresponding set of traces (it is assumed that the former has been inferred from the latter). The technique then operates by generating for each transition, and for each variable on that transition, a training set that can be supplied to the GP algorithm. The GP algorithm is then used to infer the corresponding functions. The technique is summarised in Algorithm 1. The steps are explained in more detail below.

Phase 1 - Building training sets: Every variable at a given transition has its own training set. This training set is built by identifying every trace that traverses the transition, and by identifying the values in the trace immediately prior to the transition, and the value of the variable in question once the transition is traversed.

In the algorithm this process is illustrated between lines 2 and 6. All of the traces in *Tr* are processed as follows. For each trace element $(l, Vars)_i$, the corresponding transition *Trans* is identified in the inferred machine (as returned by the `walk` function in line 4). For each variable n in the subsequent trace element $(l, Vars)_{i+1}$, a 'training set' is constructed that maps the set of variable assignments $Vars_i$ to the subsequent value of n in $(l, Vars)_{i+1}$ (referred to as c in the algorithm).

Algorithm 1: Inferring update functions

```
Input: An inferred EFSM $EFSM = (S, s_0, E, V, \Delta, U, T)$ (without update
       functions) and a set of traces $Tr$
Output: $EFSM$ enhanced with update functions
// For a set of variables V, $V_{names}$ returns a set of
   variable names.
// Initialise an empty map ($TD : (T \times V_{names}) \rightarrow 2^{(V \times V)}$).
1  $TD \leftarrow initialiseMap()$;
   // For each trace...
2  foreach $\langle (l, Vars)_0, \ldots (l, Vars)_n \rangle \in Tr$ do
       // For every trace-element bar the last one...
3      for $i \leftarrow 0$ to $n - 1$ do
           // Obtain transition corresponding to position i
              in the trace
4          $Trans \leftarrow walk(EFSM, \langle (l, Vars)_0, \ldots (l, Vars)_i \rangle)$;
           // For each variable n and value c in the
              subsequent trace elemement
5          foreach $(n, c) \in Vars_{i+1}$ do
               // Create a training item, using the current
                  value of n
6              $Training \leftarrow (Vars_i(n), c)$;
               // Add that to the training set for the
                  identified transition
7              $TD_{Trans,n} \leftarrow TD_{Trans,n} \cup \{Training\}$;

   // For each transition, infer and store update functions
      for each variable.
8  foreach $t \in EFSM_T$ do
9      foreach $n \in Vars_{names}$ do
           // Obtain the training data
10         $Train \leftarrow TD_{t,n}$;
           // Use GP to infer a function from the data.
11         $Func \leftarrow gp(Train)$;
           // Add function to the set of update functions
              in E for transition t.
12         $EFSM_{U,t} \leftarrow EFSM_{U,t} \cup Func$;

13 return $EFSM$;
```

Phase 2 - Inferring the functions: For each transition in the T and for every variable in $Vars_{names}$, the corresponding training set is obtained from TD. This is used to infer a function by genetic programming (the `gp` function in line 11 - described in more detail below). The result is added to the set of functions for that transition in $EFSM$. As a result, the EFSM has, for each transition, a function corresponding to each variable that computes the subsequent value of the variable from the set of variables at that point. The update function for a transition consists of the simultaneous execution of all of the functions for each individual variable, leading to a fully updated data-state.

In principle, any symbolic regression approach could be applied. Ultimately, the goal is to find a program that is able to transform the input in a training set (a set of variable values at some transition) into a target value (the value of a given variable at the subsequent transition). For our implementation we have chosen the form of GP that was described in Section II-C. We use a strongly typed tree-based GP. We use Tournament selection to identify candidates for recombination, and use sub-tree crossover and subtree-mutation to accomplish this [25].

The non-terminals and terminals selected for our implementation are shown in Table I. These were selected to provide a reasonable spread of functionalities that could, in our mind, combine to approximate a reasonably broad range of behaviours (of course others could be chosen instead, and there are unlimited options in this respect [25]). Most of the non-terminals are self-explanatory, elementary mathematical operations. However, there are two operators that stand out. The cast(x:I) and cast(x:D) operators will respectively cast an integer expression into a double and vice-versa. This is to cater for the situation where there are multiple variables of both types, and where the value of a variable of one type may affect the computation of a variable of another type.

TABLE I
NON-TERMINALS AND TERMINALS CHOSEN FOR OUR GP IMPLEMENTATION

Non-Terminals	
Double	add(x:D,y:D), subtract(x:D,y:D), multiply(x:D,y:D), divide(x:D,y:D), power(x:D,y:D), root(x:D, y:D), cast(x:I)
Integer	add(x:I,y:I), subtract(x:I,y:I), multiply(x:I,y:I), divide(x:I,y:I), power(x:I,y:I), cast(x:D)
Terminals	
Double	0.0, 0.5, 1.0, 2.0, all variable names in *Vars* of type double
Integer	0, 1, 2, all variable names in *Vars* of type integer
Strings	All variable names in *Vars* of type String, all String values observed in the traces.
Booleans	All variable names in *Vars* of type Boolean, `true`, `false`.

The implementation is modular, allowing for the easy addition of terminals and non-terminals. It is clear from the current selection that the emphasis is placed on numerical variables. For string and boolean variables we simply include terminals that consist of other string variable names and the values observed in the traces. However, this is merely a reflection of the types of system we have been experimenting with; it would be straightforward to add String non-terminals (e.g. concatenation, sub-string selection, etc.) to carry out more extensive String operations, or to include operations on other types such as lists (see Section VII).

B. Walk-through on Running Example

To illustrate the algorithm, we return to the example from Figure 1. Let us start with the first trace (input=3). Iterating through it, the `walk` function (line 4) applied to $\langle (recurse, \{depth = 3\}) \rangle$ returns transition $a \xrightarrow{depth>0.0} a$. Now, we look to the subsequent trace element $((recurse, \{depth = 2\}))$. For each variable (in this case just `depth`) we build a training set, taking the current set of variable assignments ($depth = 3$) and mapping it to the value at the next element (2).

Repeating this the other elements in this trace and the others, for transition $t = a \xrightarrow{depth>0.0} a$, we end up with:

$$\begin{aligned} TD_{t,depth} = \{ & (depth = 3, depth = 2), \\ & (depth = 2, depth = 1), \\ & (depth = 1, depth = 0), \\ & (depth = 5, depth = 4), \\ & (depth = 4, depth = 3)\} \end{aligned}$$

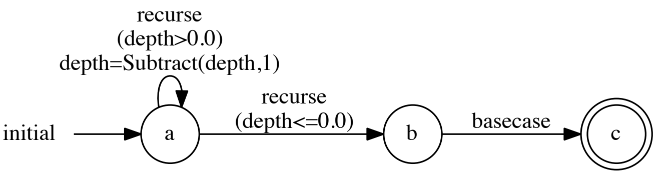


Fig. 2. Inferred model from traces in Figure 1, with update function inferred by GP. The Identity functions on transitions $a \xrightarrow{recurse} b$ and $b \xrightarrow{basecase} c$ are omitted for readability.

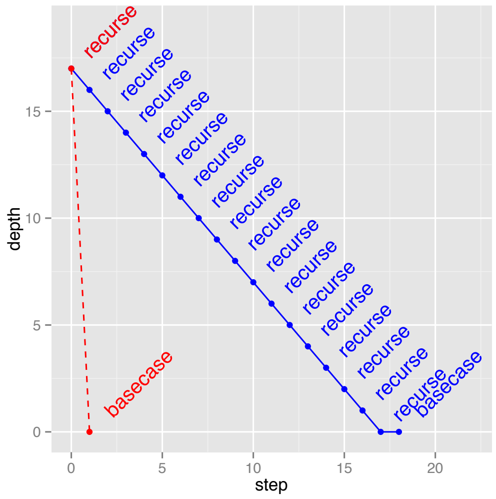


Fig. 3. Illustration of valid and invalid trace for initial configuration of $depth = 17$.

If we repeat the same exercise for the subsequent transition $s = a \xrightarrow{depth \leq 0.0} b$, we end up with the less interesting training set: $TD_{s,depth} = \{(depth = 0, depth = 0)\}$. Similarly, for the final transition $u = b \xrightarrow{basecase} c$ $TD_{u,depth} = \{(depth = 0, depth = 0)\}$.

Moving on to the second phase, we run the `gp` function on each training set. For the first transition, this gives us `depth=subtract(depth,1)`. For the other two transitions, it simply gives us the simple identity function `depth=depth`. The resulting EFSM is shown in Figure 2.

Simulating Behaviour: The enhanced inferred model in Figure 2 enables us to simulate the behaviour of the model from a given initial configuration. In Section II-B we noted that the (incorrect) trace $\langle(recurse, \{depth = 17\}), (recurse, \{depth = 0\}), (basecase, \{\})\rangle$ would be accepted by the conventional model shown in Figure 1. However, our model enhanced with update functions shows clearly that this is not possible. Figure 3 plots the value of `depth` as computed by the enhanced model, starting with $depth = 17$. The dashed line represents the invalid trace.

It is not necessarily the case that any inferred model can be entirely simulated from just its initial data configuration. If the system is reactive – i.e. some of its labels or data variables are external inputs, then these will still be required to compute the underlying data state.

TABLE II
SUBJECT SYSTEMS

System	States	Trans.	Vars.	Traces/Events
LiftDoors [27], [28]	6	12	1	381/10,461
CruiseControl [29], [30]	5	17	5	381/9,776

C. Implementation

The technique has been implemented as an extension to our MINT EFSM inference tool [3]. The source code, executables, and links to all experimental materials used in this paper are available online[2]. The implementation of the GP framework and the extension to the existing FSM / EFSM inference tools required approximately 5000 additional lines of code.

The GP extension is implemented as a 'decorator' for the conventional inference process. This emphasises the point that this approach is not tied to a specific EFSM inference technique. We happen to use it in conjunction with our MINT EFSM inference approach, but it can just as easily be activated for conventional FSMs inferred, for example, by k-Tails [6], or EDSM [12], which are also implemented in our MINT tool.

One practical problem that tends to arise with GP is the issue of 'bloat' [26], where the size of generated programs can rapidly increase. The extent to which this is a problem depends on the purpose of the inferred models. If they are to be used for simulation or test-generation (as we intend to use them), then readability is not an issue. It is however an issue if the models are for human consumption. To attenuate this problem, we include some basic (but often effective) rewriting routines. For example, GP can often produce large sub-trees that ultimately just produce a constant value. In such cases, we replace the sub-tree with a single node representing the constant.

IV. Evaluation of Function Accuracy

Our work has been motivated by the fact that conventional inferred (E)FSMs lack update functions. Without these, they cannot be used to accurately model how the underlying data state changes during the course of a computation. In this section we seek to evaluate the accuracy of the inferred data model by comparing execution trace data (which was not used as part of the inference) to the data predicted by our model. The details of the methodology are presented below, followed by results and discussion. The data (the traces, reference models, etc.) are available online [3] (this file also contains the material for the Apache Commons Math example in the next section).

A. Methodology

a) Subject Systems: The details of the systems we chose for this study are shown in Table II. These systems were

[2] https://bitbucket.org/nwalkinshaw/efsminferencetool/overview

[3] http://www.cs.le.ac.uk/people/nw91/Files/ICSMEData.zip

chosen because they differ in terms of the number of variables that constitute their data states, a factor that we anticipate will have a significant effect on the accuracy of our approach. The first is an implementation of a Lift-door controller, the EFSM for which was published by Strobl *et al.* [27], and used by Androutsopoulos *et al.* [28] in her work on EFSM slicing. Since there was no existing implementation of this, we generated a simple Java implementation that exactly reflects the behaviour set out in the model. The second is a more complex automotive cruise-control system, where the implementation and model are available on the Software Artefact Infrastructure repository [29], and has been used in previous research on state-based testing [30].

To an extent the values in the table mask the gulf in complexity between the two models. The behaviour represented by the LiftDoors model is less reactive to external inputs; transitions are mainly triggered automatically by the state of the internal variable and only occasionally are triggered by an external stimulus (e.g. interrupting the closure of the doors).

The CruiseControl system in contrast is highly reactive. Although it only has 5 states, all inputs are always possible at each state. Behaviour is always triggered by external stimuli (e.g. pushing the clutch or the brake pedal, the time for which they are pushed), combined with the current internal state variables (e.g. speed of the car). Furthermore, the behaviour often depends on how long an input has been administered for (e.g. how long the brake pedal has been pressed). This gives rise to a much greater variability of behaviour in CruiseControl than in LiftDoors.

b) Generating traces: Neither model is accompanied by an existing set of traces. For both systems, traces from the systems were obtained by simulating their use. This was achieved by setting up a test harness with a transition model of the system, where inputs to the system were chosen by random walks through the model. In the case of the CruiseControl model, we introduced a slight probabilistic bias to some transitions to encourage complete coverage of the model. CruiseControl inputs were also associated with a time variable, recording how long the input had been provided for (since this is known to have an effect on the internal state).

Given that both systems could potentially run for an infinite number of steps (and that neither had final states), we chose to limit the trace-length. This was achieved by setting the length to a random number between 2 and 50 (picked from a uniform distribution). The final sets of traces are available online.

c) Evaluating accuracy: We evaluate the accuracy of our inferred update functions by selecting traces that have not been used in the inference process, and comparing the data values predicted by our inferred models against the actual data states contained in the traces.

In this evaluation we do not assess the accuracy of the underlying state transition model. This is inferred before our post-processing technique is applied (in our case using the MINT inference algorithm [3]). We do however discuss the relationship between the (in-)accuracy of the state transition structure and our inferred functions in Section IV-C.

To avoid the inherent bias of selecting just a single trace from our set of traces, we adopt a procedure inspired by k-folds Cross Validation [31]. The set of traces is partitioned into k 'folds'. In our case we choose $k = 10$, which has been shown to be the best setting for several generic Machine Learning evaluation tasks [31]. Then, over k iterations, the traces belonging to $k - 1$ folds are used to infer a model, and the traces belonging to the remaining fold are used to evaluate the model by investigating how good the model is at predicting them – in our case, by seeing how well the model predicts the data values contained in the traces. The output is a set of traces where, for each trace, we also obtain the values predicted by an inferred model (where the trace was omitted from the training set).

There is a large amount of stochasticity in the GP algorithm, which gives rise to possibility that a particular model could be down to good or bad fortune. To attenuate this risk, we repeat every k-folds exercise 30 times, using different random seeds.

The accuracy measurement comes down to a comparison between the data values that are attached to an actual trace (which is not used during the model inference) and the equivalent values that are predicted by our inferred model. For a given evaluation trace, we obtain the values from our model by stripping any data values from the trace that are meant to be computed by the model. We then use the stripped trace to 'walk' through the inferred model, using the inferred functions to compute the corresponding variable values instead. This gives rise to two time-series – the set of target values and the set of inferred values.

To measure the agreement between these time series we calculate the Root Mean Square Error (RMSE) – a metric that is commonly used to evaluate model accuracy in Machine Learning. Given two time series x_1 (a sequence of reference values) and x_2 (the values produced by an inferred model), both of length n, The RMSE measures the mean error produced by the model (small values are desirable). It is calculated as follows:

$$RMSE = \sqrt{\frac{\sum_{t=0}^{n}(x_{1,t} - x_{2,t})^2}{n}}$$

This is scale-dependent. Since we want to assess the accuracy of our (GP) models without considering the specific scales of the variables they compute, we calculate the Normalised RMSE (NRMSE). This is computed by dividing the RMSE by the range of values observed in the reference data set, leading to a value between 0 and 1. In this case, zero represents the best possible case (no error at all), and 1 represents the worst case (continuously large errors):

$$NRMSE = \frac{RMSE}{max(x_1) - min(x_1)}$$

For boolean variables we calculate the binary error rate [32]. This is defined as $\frac{fp+fn}{tp+tn+fp+fn}$, where *tp* represents instances where both values are *true*, *tn* where both values are *false*, and *fp* and *fn* represent a true-false and false-true disagreement respectively. A high score (both are within the limits $[0, 1]$)

indicates that, for a given trace, there is a high error rate (i.e. a low rate of agreement).

To provide some insight into the relationships between variables we also present some of the time-series in a similar vein to Figure 3. For the sake of readability (and since we are focussing on data values), we leave out the textual annotations of the labels at each point.

B. Results

1) LiftDoors: The error measured for the computed values for the (only) variable `t` in LiftDoors was relatively low. The mean RMSE was 1.35 (out of a total value range of 10), so the mean NRMSE was 0.13. The spread of NRMSE values is shown in the histogram at the top of Figure 4.

Figure 4 contains two time-series plots that show the variable values through the course of two specific executions. These were chosen because their RMSE scores are close to the mean RMSE measured across all traces, and can thus be considered to be reasonably representative. Blue solid lines represent the traced variable values, whereas red dashed lines represent the corresponding values computed by the inferred models.

From these series, two remarks can be made. Firstly, deviations from the expected score tend to be localised to specific functions. For example, in both series, it is apparent that the functions computing `t` for the `waittimer` and `closingDoor` were slightly inaccurate. Secondly (and unsurprisingly) the inaccuracy incurred by the miscalculation of a value at one point can propagate for several steps. For example, in the first example the value of `t` is miscalculated at the second instance of `fullyOpen`, and remains incorrect for the subsequent four steps (`timeout, closingDoor, closingDoor, closingDoor`).

2) CruiseControl: The mean NRMSE values for all of the variables in CruiseControl are given in the box-plot in Figure 5. It should be noted that the first four plots show the distribution for numerical error in NRMSE, whereas the final plot shows the error-score for binary values (see Section IV-A). The scores should also be interpreted with care; NRMSE is computed in relative terms to the value-range of the variable (in parentheses below each box). If a variable (such as Speed – as discussed below) has occasional very large values, the larger range can artificially deflate the overall NRMSE score.

Although the functions inferred for CruiseControl offer an approximation of the expected values, their accuracy tends to vary from one variable to the next. Specifically, Figure 5 indicates that the accuracy of the functions inferred for distance and speed are much better than for brake, throttle, and ignition.

One apparent explanation for this difference is the fact that brake, throttle and ignition are all inputs, which are (in our case) controlled by a pseudo-random algorithm. Although the inputs follow certain probabilistic patterns to ensure that the controller is put through its paces, they also exhibit a lot of random behaviour. Accordingly, their behaviour can rarely be easily inferred from the rest of the state of the system, which explains the lower accuracy for these variables. However, for state and distance, which *can* at least be approximated from the inputs and the state of the system, the accuracy of the inferred functions is markedly improved.

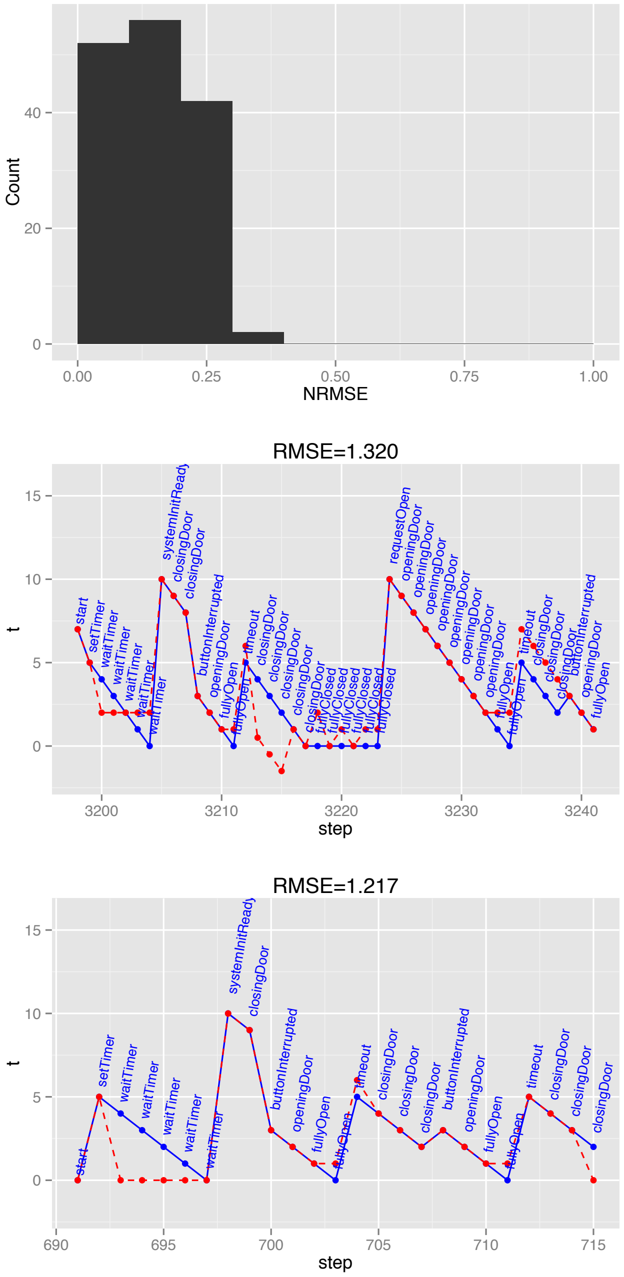


Fig. 4. Histogram of NRMSE for LiftDoors, with two examples of trace trajectories.

The presence of multiple variables, the fact that some of these variables are ‘noisy’, and the larger number of possible events make it harder to infer accurate models. This is illustrated in the distance plot in Figure 5, which is broadly representative. Although the value of the inferred

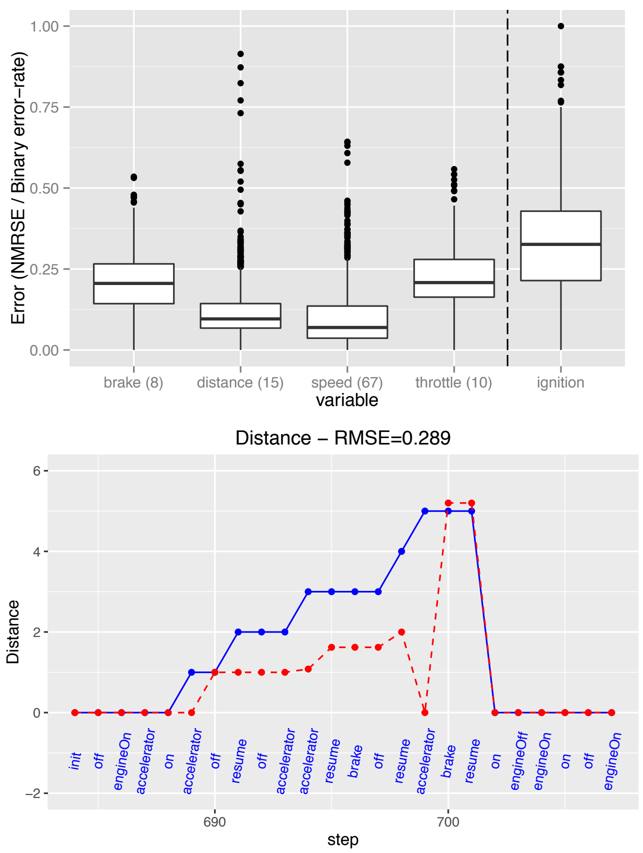


Fig. 5. NRMSE distributions, with sample trajectory inferred for the distance variable.

values broadly increase and decrease at the right points, the extent to which they do can be inaccurate (this is often because the value is calculated from other variable values which, if noisy, can be easily mis-calculated).

C. Discussion

From the two case studies used here, we are able to draw some high-level insights into the accuracy of the approach. Both systems have state transition systems that are of a similar complexity, and were inferred from trace sets with the same number of traces, where the length of the traces was also similar. Nonetheless, LiftDoors is substantially more accurate than CruiseControl. There are two key reasons for this:

1) **State variables:** There is just one state variable in the LiftDoors example, whereas there are five in CruiseControl. This means that, for each function inferred by GP, there are many more terminals (variables) to be taken into account. This in turn raises the probability that a mistake is made – that spurious variables are used within a transition function.
2) **Breadth of sequence possibilities:** In the LiftDoors machine, each state has at most two different types of subsequent event, and usually only one. In other words, a set of traces will tend to consist of very similar sequences of events. This is not the case for CruiseControl. Every type of input is possible from every state. This means that any set of traces over this model will be much more heterogeneous. In this case the state machine inference algorithm will struggle to match sequences of events that are in fact equivalent. This in turn implies less training data that can be used by the GP algorithm to infer the transition functions, producing less accurate outputs as a result.

There are several potential means by which to address the problems posed by these issues. These are discussed in the context of our future work, in Section VII.

D. Threats to Validity

The study used herein cannot be used to (and does not aim to) draw general conclusions about the accuracy of the technique. It does however aim to provide the reader with a reasonable idea of how the technique performs ‘out of the box’. There are however several parameters of this study that must be taken into account when reviewing the results, which we briefly discuss here.

Choice of systems: For this study we used two fully specified EFSMs. Although these present us with valuable insights here, it will require a larger, more diverse selection of systems to produce more generalisable results.

Selection of parameters: There are many parameters to our approach. For the GP there is the choice of terminals and non-terminals, the cross-over, mutation, tournament size, and population size parameters. Then there are the EFSM inference parameters [3] such as the choice of state-merging heuristic, and any minimum merging thresholds. For all of these we avoided deliberate bias by simply using the default settings in our tool. Of course, a more comprehensive study will need to control for these parameters, to establish their effect. If anything, the current results can be seen as an under-approximation of the ‘ideal’ performance of the system. A more system-specific selection of parameters would, depending on the system, probably lead to much more accurate results.

V. Case Study: Application to Regression Testing

In this section we provide a small case study to (1) show that the approach can be applied to reverse-engineer Java APIs for larger frameworks, and (2) provide an illustration of how inferred models can be used for regression testing, as in the vein of growing body of work [1], [2], [33]: We use the inferred functions to derive assertions that can be used as oracles, and illustrate how the model can be used to identify test-inputs that explore previously unexplored behaviour.

As our subject system, we select the `stat.regression.SimpleRegression` class from the Apache Commons Math library[4]; this was selected because it is reasonably complex, is accompanied by a test set, and is ‘updateable’; calls made to a SimpleRegression object can

[4] https://commons.apache.org/proper/commons-math/

change its state in a variety of ways and it calculates its results on the fly. The `SimpleRegression` API[5] suggests that the `addData(double,double)` is a key function that enables the user to incrementally add observations as (x,y) coordinates to the regression data set. Here we show how the inferred model can be used to formulate new test cases that can be added to the existing tests.

a) Inferring the Model: There are 53 JUnit tests (53 separate usages of the SimpleRegression class, though some of these are wrapped in the same formal unit test). We obtained the traces of the sequences and the data by executing the tests with the Daikon Chickory tool [18] and recording the trace file. There are a total of 1689 trace elements within the traces. Importantly, the traces also include 52 variables that are affected by (and in turn affect) the sequencing of method calls (this is including object attributes, method parameters and return values). Let us consider the scenario where we are equipped only with the test sets (or their traces), but we have no knowledge of how the `SimpleRegression` class behaves.

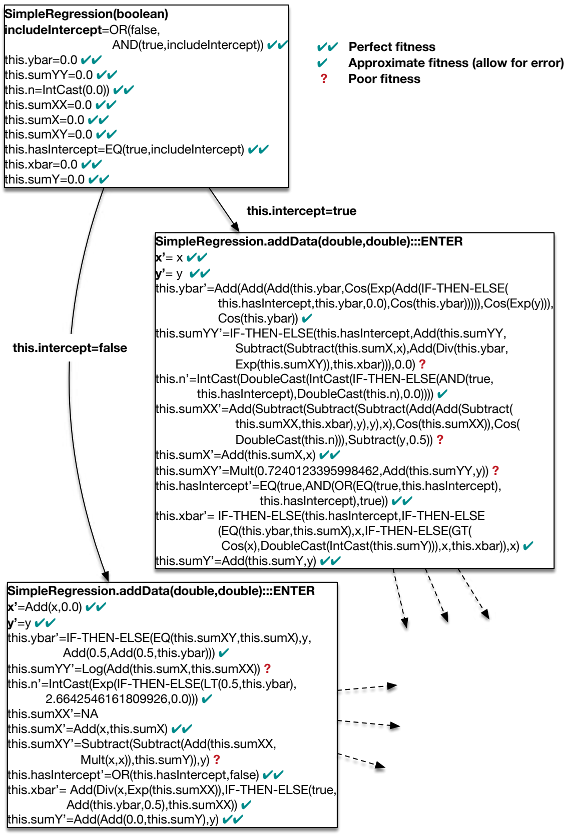
The inferred transition system (by MINT) contains 537 states and 705 transitions. The full state machine (along with the Daikon dtrace file) can be downloaded along with the traces we used in our previous experiments (see footnote 3).

b) Identifying Inputs: The sequences required to reach the `addData` function can be established by adopting typical EFSM-testing approaches [35]. The transitions that correspond to the `addData` entry-point are identified (there are 34 such transitions in the model). We then identify the sequence of inputs corresponding to the shortest path from the start-state to the source states for these transitions. This sequence is in effect using the history from previous test executions to set up the system into a suitable state so that `addData` can be tested.

c) Identifying test-oracles: For each transition, we extract the inferred functions. Functions that failed to produce a low fitness score are omitted. The rest can be used as assertions. For functions where the fitness score was not perfect, any assertions should allow for an error-margin (as a heuristic, we tend to allow for an error around the fitness function score).

We can then refer to the functions that were inferred for the exit-points to show *how* `addData` has behaved for the tests that have executed so far – in effect these are post-conditions. Our goal is then to identify inputs that will confound these post-conditions – this would mean that the behaviour we are eliciting is different from the tests from which the model was inferred.

d) Illustration: In Figure 6, the transition diagram provides an illustration of the information that is inferred, with respect to two of the most direct paths from the initial node to the `addData` function. To save space, elements of the model have been re-coded (we have left out function-exit points which are in the trace produced by Daikon, but have no bearing on the behaviour, and we have re-written long GP functions to shorter equivalent versions where possible). The inferred functions represent what the state of the system should be *after* the function has been invoked. Variables post-execution are hyphenated. Functions that achieved perfect fitness are given two 'ticks', those that achieved an approximate fitness (in this case ≤ 5 for doubles) are given a single tick, and others are


SimpleRegression(boolean)
includeIntercept=OR(false,
AND(true,includeIntercept)) ✓✓
this.ybar=0.0 ✓✓
this.sumYY=0.0 ✓✓
this.n=IntCast(0.0)) ✓✓
this.sumXX=0.0 ✓✓
this.sumX=0.0 ✓✓
this.sumXY=0.0 ✓✓
this.hasIntercept=EQ(true,includeIntercept) ✓✓
this.xbar=0.0 ✓✓
this.sumY=0.0 ✓✓
✓✓ Perfect fitness
✓ Approximate fitness (allow for error)
? Poor fitness
this.intercept=true
this.intercept=false
SimpleRegression.addData(double,double):::ENTER
x'= x ✓✓
y'= y ✓✓
this.ybar'=Add(Add(Add(this.ybar,Cos(Exp(Add(IF-THEN-ELSE(
this.hasIntercept,this.ybar,0.0),Cos(this.ybar)))))),Cos(Exp(y))),
Cos(this.ybar)) ✓
this.sumYY'=IF-THEN-ELSE(this.hasIntercept,Add(this.sumYY,
Subtract(Subtract(this.sumX,x),Add(Div(this.ybar,
Exp(this.sumXY)),this.xbar))),0.0) ?
this.n'=IntCast(DoubleCast(IntCast(IF-THEN-ELSE(AND(true,
this.hasIntercept),DoubleCast(this.n),0.0)))) ✓
this.sumXX'=Add(Subtract(Subtract(Subtract(Add(Add(Subtract(
this.sumXX,this.xbar),y),y),x),Cos(this.sumXX)),Cos(
DoubleCast(this.n))),Subtract(y,0.5)) ?
this.sumX'=Add(this.sumX,x) ✓✓
this.sumXY'=Mult(0.7240123395998462,Add(this.sumYY,y)) ?
this.hasIntercept'=EQ(true,AND(OR(EQ(true,this.hasIntercept),
this.hasIntercept),true)) ✓✓
this.xbar'= IF-THEN-ELSE(this.hasIntercept,IF-THEN-ELSE
(EQ(this.ybar,this.sumX),x,IF-THEN-ELSE(GT(
Cos(x),DoubleCast(IntCast(this.sumY))),x,this.xbar)),x) ✓
this.sumY'=Add(this.sumY,y) ✓✓
SimpleRegression.addData(double,double):::ENTER
x'=Add(x,0.0) ✓✓
y'=y ✓✓
this.ybar'=IF-THEN-ELSE(EQ(this.sumXY,this.sumX),y,
Add(0.5,Add(0.5,this.ybar))) ✓
this.sumYY'=Log(Add(this.sumX,this.sumXX)) ?
this.n'=IntCast(Exp(IF-THEN-ELSE(LT(0.5,this.ybar),
2.6642546161809926,0.0))) ✓
this.sumXX'=NA
this.sumX'=Add(x,this.sumX) ✓✓
this.sumXY'=Subtract(Subtract(Add(this.sumXX,
Mult(x,x)),this.sumY)),y) ?
this.hasIntercept'=OR(this.hasIntercept,false) ✓✓
this.xbar'= Add(Div(x,Exp(this.sumXX)),IF-THEN-ELSE(true,
Add(this.ybar,0.5),this.sumXX)) ✓
this.sumY'=Add(Add(0.0,this.sumY),y) ✓✓

```
@Test
public void addDataInterceptTrueYbarTest() {
    boolean includeIntercept = true;
    SimpleRegression testobj =
        new SimpleRegression(includeIntercept);
    Assert.assertEquals(testobj.ybar, 0.0,0D);
    Assert.assertEquals(testobj.hasIntercept,
        (includeIntercept == true));
    //Other assertions from constructor
    Random r = new Random();
    double x = r.nextDouble()+r.nextInt(20);//15.6
    double y = r.nextDouble()+r.nextInt(20);//5.2
    double expected = (testobj.ybar + Math.cos
        (Math.exp(testobj.hasIntercept?testobj.ybar+
        Math.cos(testobj.ybar):0.0+Math.cos(testobj.
        ybar))))+Math.cos(Math.exp(y))+
        Math.cos(testobj.ybar);
    testobj.addData(x,y);
    Assert.assertEquals(testobj.ybar, expected,5D);
}
```

Fig. 6. Extract from `SimpleRegression` state machine, with a sample derived JUnit test. For space reasons the test is specific to the ybar attribute. To run, the SimpleRegression attributes have to be made visible (protected).

[5] https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/stat/regression/SimpleRegression.html

appended with a question-mark.

The test case has in this case been constructed by hand as a literal translation from the inferred functions, but this could feasibly be automated. The invocations to SimpleObject follow the sequence given by the path in the state machine. After each invocation, assertions are inserted to reflect the inferred functions (for space reasons we focus only on the `ybar` attribute). For `addData`, `x` and `y` are parameters. If we choose any of the values that have been used in the training set (e.g. 15.6 and 5.2 had been used in the Apache tests), the test will pass; the value of `testobj.ybar` will be within a small delta of the value predicted by our inferred function (assigned here to `expected`).

If we want to identify new test cases that do not re-execute behaviour that has already been explored, we can focus on identifying values of `x` and `y` that 'break' the inferred assertions. This can (for example) be elicited in our case by attempting random values, as is the case in our example. For example, $x = 9.731, y = 13.606$ (which will appear if the test case in Figure 6 is executed) contradict the assertion. In a typical inference-driven testing cycle [33], [36], these new inputs would be assimilated into the test set, and the model would be inferred afresh, repeating the cycle until no contradictory inputs can be found.

Qualitative Remarks

One notable property of the inferred models is that, depending on the given set of traces, the inferred model may become very large, with several transitions only supported by few trace elements. This means that the inferred functions for these transitions can be highly specific. This is especially the case when traces have large label-sets, as is the case with the `SimpleRegression` model. In this case 49% of the inferred functions are only inferred for a single data-point.

For cases such as the automated testing scenario, this is not a problem (if anything, the task of finding new test cases is facilitated). It could however be problematic if the inferred model is intended for documentation or human consumption; for such cases it is recommended that any inferred functions are annotated alongside the number of data points from which they were inferred, so that they can be interpreted accordingly. In our future work, we intend to adapt the underlying state-merging algorithms to guide them towards solutions where each transition is supported by a large number of data-points, to facilitate the inference of functions (we suspect that this would lead to more accurate models in any case).

VI. Related Work

The area of Genetic Programming is also a thriving area of research since the early nineties [23]. Although GP has never been used to infer data-functions in EFSMs, it has been used before to infer the underlying state transition structures. Brave [37] used GP to evolve programs that contained the necessary instructions to construct accurate state machines from traces, by splitting states and changing their properties. This technique however pre-dated the EDSM state-merging algorithm [12] which underpins our technique and is generally agreed to represent the state-of-the-art.

The authors are only aware of one line of work that attempts to infer fully computational automata. Howar *et al.*[38] have built upon Angluin's $L*$ model inference algorithm [39] to infer Register Automata (state machines that, similarly to EFSMs have an internal data state, and accept data parameters as input). Given that their algorithm is based upon $L*$, it relies upon a different learning setting to the one considered in this paper. It relies upon the ability to submit tests to the system being inferred (which can often reach a substantial number). It also relies upon the availability of an 'oracle', a mechanism that has the ability to determine whether a hypothesised model provided by the learner is correct and, if not, to provide a counter-example. The setting considered in our work is entirely passive; once the traces have been provided, there is no more input required.

There has been a some work on inferring state machines from source code (as opposed to dynamic traces) [40], [41]. This work is however predicated on the availability of the source code (and often a corresponding static analysis framework such as a symbolic execution engine). The work we have presented in this paper requires neither, and can work from traces alone.

VII. Conclusions and Future Work

This work has presented a technique that can, given a state machine and a set of traces, infer the state-transition functions for each transition in the state machine. In this way, inferred state machines can be made to be *computational*. They can not only be used to state whether or not a sequence is or is not possible. They can be used to compute the data values at each step as well.

Our preliminary results indicate that the technique is reasonably accurate. However, establishing this in detail will require a larger more in-depth experimental study, where the various potentially confounding factors are more tightly controlled. This forms part of our ongoing work.

The results from our case studies indicated that (unsurprisingly) multiple variables, coupled with a breadth of activity at every state in the system – could have an impact on model accuracy. These are problems that have arisen in various guises in Machine Learning and in Software Engineering. There are two immediate options that can be used to address them. The variable-issue can be potentially addressed by filtering-out potentially confounding variables, for which there are a variety of algorithms [42]. The second problem, which has been advocated before in the inference of conventional state machines, is to accept additional sequential constraints from a user, to guide the state machine inference [4], [11].

The authors have made only very little effort to fine-tune the selection of non-terminals and terminals in the GP framework. There is an ongoing effort to experiment with the use of new operators, and constructs such as loops [25] to produce functions that are more accurate, and applicable to a broader range of systems.

References

[1] Chen, Qi Alfred, Zhiyun Qian, and Z. Morley Mao. ”Peeking into your app without actually seeing it: Ui state inference and novel android attacks.” 23rd USENIX Security Symposium (USENIX Security 14). 2014.

[2] Choi, Wontae, George Necula, and Koushik Sen. ”Guided gui testing of android apps with minimal restart and approximate learning.” OOPSLA'13 - ACM SIGPLAN Notices. Vol. 48. No. 10. ACM, 2013.

[3] Walkinshaw, N., Taylor, R., Derrick, J., ”Inferring extended finite state machine models from software executions”, Empirical Software Engineering (2015).

[4] Damas, Christophe, et al. ”Generating annotated behavior models from end-user scenarios.” Software Engineering, IEEE Transactions on 31.12 (2005): 1056-1073.

[5] Comparetti, Paolo Milani, et al. ”Prospex: Protocol specification extraction.” Security and Privacy, 2009 30th IEEE Symposium on. IEEE, 2009.

[6] Biermann, Alan W., and Jerome A. Feldman. ”On the synthesis of finite-state machines from samples of their behavior.” Computers, IEEE Transactions on 100.6 (1972): 592-597.

[7] Ammons, G., Bodk, R., & Larus, J. R. (2002). Mining specifications. POPL'02, ACM Sigplan Notices, 37(1), 4-16.

[8] Cook, J. E., & Wolf, A. L. (1998). Discovering models of software processes from event-based data. ACM Transactions on Software Engineering and Methodology (TOSEM), 7(3), 215-249.

[9] Walkinshaw, N., Bogdanov, K., Holcombe, M., & Salahuddin, S. (2007, October). Reverse engineering state machines by interactive grammar inference. In Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on (pp. 209-218). IEEE.

[10] Lo, D., & Khoo, S. C. (2006, November). SMArTIC: towards building an accurate, robust and scalable specification miner. In Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (pp. 265-275). ACM.

[11] Walkinshaw, N., & Bogdanov, K. (2008, September). Inferring finite-state models with temporal constraints. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (pp. 248-257). IEEE Computer Society.

[12] Lang, Kevin J., Barak A. Pearlmutter, and Rodney A. Price. ”Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm.” Grammatical Inference. Springer Berlin Heidelberg, 1998. 1-12.

[13] Beschastnikh, I., Brun, Y., Abrahamson, J., Ernst, M. D., & Krishnamurthy, A. (2015). Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. Software Engineering, IEEE Transactions on, 41(4), 408-428.

[14] Cheng K, Krishnakumar A (1993) Automatic functional test generation using the extended finite state machine model. In: 30th Conference on Design Automation. ACM, pp 8691

[15] Boerger, E., & Staerk, R. (2012). Abstract state machines: a method for high-level system design and analysis. Springer Science & Business Media.

[16] Abrial, J. R., & Hallerstede, S. (2007). Refinement, decomposition, and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae, 77(1-2), 1-28.

[17] Lorenzoli, Davide, Leonardo Mariani, and Mauro Pezz. ”Automatic generation of software behavioral models.” Proceedings of the 30th international conference on Software engineering. ACM, 2008.

[18] Ernst, Michael D., et al. ”The Daikon system for dynamic detection of likely invariants.” Science of Computer Programming 69.1 (2007): 35-45.

[19] Krka, I., Brun, Y., & Medvidovic, N. (2014, November). Automatic mining of specifications from invocation traces and method invariants. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 178-189). ACM.

[20] Lo, D., Maoz, S., & Khoo, S. C. (2007, November). Mining modal scenario-based specifications from execution traces of reactive systems. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (pp. 465-468). ACM.

[21] Lo, D., Mariani, L., & Santoro, M. (2012). Learning extended FSA from software: An empirical assessment. Journal of Systems and Software, 85(9), 2063-2076.

[22] Ohmann, Tony, et al. ”Behavioral resource-aware model inference.” Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. ACM, 2014.

[23] Koza, John R. Genetic programming: on the programming of computers by means of natural selection. Vol. 1. MIT press, 1992.

[24] Harman, Mark, William B. Langdon, and Westley Weimer. ”Genetic programming for Reverse Engineering.” WCRE. Vol. 13. 2013.

[25] Poli, Riccardo, William B. Langdon, Nicholas F. McPhee, and John R. Koza. A field guide to genetic programming. Lulu. com, 2008.

[26] Langdon, William B. ”Quadratic bloat in genetic programming.” Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation. Morgan Kaufmann Publishers Inc., 2000.

[27] Strobl, Frank, and Alexander Wisspeintner. ”Specification of an elevator control system - an autofocus case study.” Institutsbericht, Technische Universitaet Muenchen, Institut fuer Informatik (1999).

[28] Androutsopoulos, Kelly, David Clark, Mark Harman, Robert M. Hierons, Zheng Li, and Laurence Tratt. ”Amorphous slicing of extended finite state machines.” Software Engineering, IEEE Transactions on 39, no. 7 (2013): 892-909.

[29] Do, Hyunsook, Sebastian Elbaum, and Gregg Rothermel. "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." Empirical Software Engineering 10.4 (2005): 405-435.

[30] Briand, Lionel C., Yvan Labiche, and Yihong Wang. ”Using simulation to empirically investigate test coverage criteria based on statechart.” Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, 2004.

[31] Kohavi, Ron. ”A study of cross-validation and bootstrap for accuracy estimation and model selection.” In Ijcai, vol. 14, no. 2, pp. 1137-1145. 1995.

[32] Sokolova, Marina, and Guy Lapalme. ”A systematic analysis of performance measures for classification tasks.” Information Processing & Management 45, no. 4 (2009): 427-437.

[33] Fraser, Gordon, and Neil Walkinshaw. ”Assessing and generating test sets in terms of behavioural adequacy.” Software Testing, Verification and Reliability 25.8 (2015): 749-780.

[34] Walkinshaw, N., Bogdanov, K., Derrick, J., & Paris, J. (2010). Increasing functional coverage by inductive testing: a case study. In Testing Software and Systems (pp. 126-141).

[35] Petrenko, Alexandre, Sergiy Boroday, and Roland Groz. ”Confirming configurations in EFSM testing.” Software Engineering, IEEE Transactions on 30.1 (2004): 29-42.

[36] Weyuker, Elaine J. ”Assessing test data adequacy through program inference.” ACM Transactions on Programming Languages and Systems (TOPLAS) 5.4 (1983): 641-655.

[37] Brave, S. (1996, July). Evolving deterministic finite automata using cellular encoding. In Proceedings of the 1st annual conference on genetic programming (pp. 39-44). MIT Press.

[38] Howar, F., Steffen, B., Jonsson, B., & Cassel, S. (2012, January). Inferring canonical register automata. In Verification, Model Checking, and Abstract Interpretation (pp. 251-266). Springer Berlin Heidelberg.

[39] Angluin, D. (1988). Queries and concept learning. Machine learning, 2(4), 319-342.

[40] Walkinshaw, N., Bogdanov, K., Ali, S., & Holcombe, M. (2008). Automated discovery of state transitions and their functions in source code. Software Testing, Verification and Reliability, 18(2), 99-121.

[41] Sen, Tamal, and Rajib Mall. ”Extracting finite state representation of Java programs.” Software & Systems Modeling 15.2 (2016): 497-511.

[42] Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. The Journal of Machine Learning Research, 3, 1157-1182.