Inferring Extended Finite State Machine Models from Software Executions

Neil Walkinshaw
Department of Computer Science,
The University of Leicester, UK

Ramsay Taylor and John Derrick
Department of Computer Science,
The University of Sheffield, UK

***Abstract*—The ability to reverse-engineer models of software behaviour is valuable for a wide range of software maintenance, validation and verification tasks. Current reverse-engineering techniques focus either on control-specific behaviour (e.g. in the form of Finite State Machines), or data-specific behaviour (e.g. as pre/post-conditions or invariants). However, typical software behaviour is usually a product of the two; models must combine both aspects to fully represent the software's operation. Extended Finite State Machines (EFSMs) provide such a model. Although attempts have been made to infer EFSMs, these have been problematic. The models inferred by these techniques can be non deterministic, the inference algorithms can be inflexible, and only applicable to traces with specific characteristics. This paper presents a novel EFSM inference technique that addresses the problems of inflexibility and non determinism. It also adapts an experimental technique from the field of Machine Learning to evaluate EFSM inference techniques, and applies it to two open-source software projects.**

I. Introduction

Context: 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. Their use is becoming increasingly popular from a testing perspective, where they have been used as oracles for regression testing [1], to detect intrusions / anomalous behaviour [2], and to automatically generate test cases [3], [4]. Beyond testing, models can aid software comprehension [5] and requirements engineering [6].

This range of applications has given rise to a multitude of model inference techniques. Most techniques are dynamic; they take as input a set of traces of the system in question and aim to infer models that generalise upon these traces. These have largely focussed either on the inference of sequential models in the form of Finite State Machines (FSMs) [7], [8], [9], or as data pre-/post-conditions, as espoused by tools such as Daikon [1].

Such models tend to only provide a partial view of software behaviour. In practice, behaviour is often the result of interplay between the sequencing of events or function calls (as modelled by FSMs) and the values of the associated parameter or data-state variables (as modelled by invariants). This observation has prompted attempts to infer more complete models that are able to combine the two facets of behaviour. A pioneering technique that sought to address this issue was developed by Lorenzoli *et al.*[10], who devised the GK-Tails algorithm that combined the k-tails FSM inference algorithm [9] with the Daikon invariant detection system [1] to produce fully-fledged Extended Finite State Machines (EFSMs) [11] (FSMs with data-guards on the transitions). Though useful, current approaches suffer two drawbacks:

Non-determinism and lack of a connection between data constraints and sequential events: In current approaches, inferred data constraints carry out a descriptive role (summarising the data for individual transitions). They are not inferred in such a way that specific configurations of data are causally linked to subsequent control-events. As a consequence they fail to capture the explicit logical relationship between data and control. A walk through a machine can involve numerous points at which there are several possible paths to take for a given data-state.

Inflexibility: Current inference approaches are limited to highly specific combinations of control and data-inference algorithms. However, software systems can be very diverse, which demands flexibility. Depending on the domain, systems might differ substantially in the typical trace size and diversity, the proportions of spurious events, noise, numerical variables, boolean variables, or structured variables such as lists etc. These can require completely different learner configurations; no single, specific learning algorithm will be effective on all types of system.

Contribution: This paper presents a generalised inference technique for EFSMs to address the problems listed above. The specific contributions are as follows:

- An EFSM inference technique that is (1) capable of inferring EFSMs that are deterministic and (2) modular, allowing the use of a wide family of data classifier algorithms to analyse the data-state.
- An openly available proof of concept implementation that incorporates the WEKA classifier framework [12].
- A preliminary evaluation of our EFSM inference technique on two open-source modules - a worker-pool module from the Basho Riak distributed database implemented in Erlang, and an SMTP protocol implementation from the Oracle Java Mail framework.

II. Background

This section presents the core definitions that are to be used throughout the rest of this paper, namely traces, Finite State Machines, and Extended Finite State Machines. Finally, it provides a small motivating case study.

A. Definitions

As is the case with most behavioural model inference techniques, the technique presented in this paper takes as input a set of program traces. These are of a generic format, consisting of sequences of event or function labels accompanied by variable values [10]. They can be defined as follows:

Definition 1: ***Traces*** A *trace* $T =< e_0, \ldots, e_n >$ is a sequence of n trace elements. Each element e maps to a tuple (l, v), where l is a label representing the names of function calls or input / output events, and v is a vector of variable values (this may be empty).

The selection of which variables to trace depends on the tracing capabilities and the level of abstraction at which the trace is being recorded. This may vary according to the purpose of the model. The choice of trace abstraction makes no difference to the inference technique presented here. As with other approaches to this topic [10], [7], the mechanism and encoding used to collect the traces is left to the user, and depends to an extent on the intended purpose of the model.

Definition 2: ***Finite State Machine*** A Finite State Machine (FSM) can be defined as a tuple (S, s_0, F, L, T). S is a set of states, $s_0 \in S$ is the initial state, and $F \subseteq S$ is the set of final states. L is as defined as the set of labels. T is the set of transitions, where each transition takes the form (a, l, b) where $a, b \in S$ and $l \in L$. When referring to FSMs, this paper assumes that they are deterministic.

In intuitive terms, Extended Finite State Machines [11] augment conventional FSM with a memory. Transitions between states are not only associated with a label, but are also associated with a guard that represents conditions that must hold with respect to the variables in the memory:

Definition 3: ***Extended Finite State Machine*** An Extended Finite State Machine (EFSM) M is a tuple $(S, s_0, F,$ $L, V, \Delta, T)$, where S, s_0, F and L are defined as in a conventional FSM. V represents the set of data states, where a single instance v represents a set of concrete variable assignments as defined in Definition 1. The *data guard* set Δ is the set of data guards, where each guard δ takes the form $(l, v, possible)$, where $l \in L$, $v \in V$, and $possible \in \{true, false\}$. The set of transitions T is an extension of the conventional FSM version, where transitions take the form (a, l, δ, b), where $a, b \in S$, $l \in L$, and $\delta \in \Delta$. Unlike conventional FSMs, the transition system of an EFSM can be non-deterministic.

It should be noted that the EFSMs as defined here and by Lorenzoli *et al.* are abstracted versions of the EFSMs defined by Cheng and Krishnakumar [11]. In the original definition, variables in V are updated by an *update function*, the execution of which is subject to the satisfaction of the data guard. In our and Lorenzoli *et al.*'s case, the models are declarative, in the sense that they capture the sequences of events and variable values that are (or are not) possible, but do not necessarily explicitly capture how the individual variables are updated.

B. Motivating Example

To motivate this work, we consider a scenario where we need to reverse-engineer the behaviour of a mine pump controller [13]. The pump controls water levels in a mine, and is activated or deactivated depending on the levels of water. However, it also monitors the levels of methane in the mine, and must switch the pump off if the levels of methane become too high.

Let us consider the scenario where we are confronted with such a controller, the source code is unavailable, but we are able to obtain a trace of its behaviour - which gives us its actions, and the associated levels of water and methane and pump activity (i.e. whether it is on or off). Thus, we end up a trace of 2931 events, from which a snippet is shown below:

```
...
turn_on 73.44274560979447 596.7792240239261 false
low_water 28.898501010661718 584.0357656062484 true
switch_pump_off 28.898501010661718 584.0357656062484 true
turn_off 28.898501010661718 584.0357656062484 true
highwater 31.47476437422098 588.4568312662454 false
switch_pump_on 31.47476437422098 588.4568312662454 false
...
```

So the question we are faced with is this: Given such a trace, what is the sequential behaviour of the system? We know intuitively that the activities are based on the data, but how? In other words, what is the EFSM that can accurately capture this behaviour for us?

III. Basic Inference Techniques

EFSMs combine two complementary perspectives on software behaviour: (1) the FSM component that shows the order in which various events or functions can be executed and (2) the rules that relate the data state of the system to the execution of these events. Both of these model families have been the subject of a substantial amount of model inference research, both within the fields of Software Engineering and Machine Learning. The remainder of this section presents a high-level overview of these techniques, providing enough information for the next section to show how they can be combined in a modular way. It concludes with a short overview of the weaknesses of existing EFSM inference techniques that have attempted to combine FSM and data rule inference.

A. Evidence-Driven State Merging

The challenge of inferring an FSM from a set of example sequences has been an established research challenge since Gold's pioneering research on language identification in the late 60s [14]. Numerous inference techniques were developed, spurred by several competitions that encouraged novel techniques to infer models from training sets made available over the internet [15], [8]. Evidence-Driven State Merging is one of the most popular and accurate inference techniques to emerge. It won the Abbadingo competition [15] and was used as a baseline for the recent STAMINA competition [8]. We use the basic steps of the EDSM algorithm as a basis for our EFSM inference algorithm. Its basic functionality is described below. More in-depth descriptions of the individual components of the algorithm are available in Lang's original paper [15], or one of the selection of Software Engineering references that describe its use [6], [8] .

```
Data: S, s0, F, L, T, s1, s2, s3, s4, Traces
/* S, s0, F, L, T are as per definition 2, s0, ... s4 ∈ S,
   Traces is the set of input traces                    */
Result: A deterministic FSM consistent with Traces
1  infer(Traces) begin
2      (S, s0, F, L, T) ←generatePTA(Traces);
3      while (s1, s2) ← choosePairs((S, s0, F, L, T)) do
4          (S, F, T) ← merge(S, F, T, s1, s2);
5      end
6      return (S, s0, F, L, T)
7  end
   /* Function to merge s1 to s2 and ensure that the result
      is deterministic.                                    */
8  merge(S, F, T, s1, s2) begin
9      S ← S \ {s1};
10     F ← F \ {s1};
11     T ← changeSources(s1_out, s2, T);
12     T ← changeDestinations(s1_in, s2, T);
13     while (s3, s4) ←findNonDeterminism(S, T) do
14         (S, F, T) ←merge(S, F, T, s3, s4);
15     end
16     return (S, F, T)
17 end
```

Algorithm 1: Basic state merging algorithm.

Most FSM inference techniques (including the popular k-tails algorithm [9]) fit into the family of "state-merging" techniques. These are variants of the same underlying algorithm[1], which is shown in Algorithm 1. From here on, we use s_{out} to refer to the outgoing transitions from state s, and s_{in} to refer to the incoming transitions.Traces are as defined in Definition 1, but the data variable values are ignored (given that FSMs do not incorporate data).

The algorithm begins by composing the traces into the most specific possible correct FSM — the PTA — and then proceeds to merge states to produce progressively smaller and more generalised *hypothesis FSMs*. The key steps are introduced below:

line 2: the set of traces $Traces$ is arranged by the `generatePTA` function into an initial FSM in the form of a *prefix-tree acceptor (PTA)* [6]. This can be described intuitively as a tree-shaped state machine that exactly accepts $Traces$, where traces with the same prefix share the same path from the initial state in the PTA up to the point at which they diverge. Leaf states are added to F as final / accepting states. For an intuitive illustration, there is an example PTA in Figure 1

lines 3-5: The state merging challenge is to identify pairs of states in the current hypothesis FSM that represent equivalent states in the subject system, and to merge them. Starting off from the PTA, pairs of states are iteratively selected and merged, until no further pairs of equivalent states can be found, which indicates convergence at the final hypothesis FSM.

`choosePairs`: This function is responsible for choosing a pair of states that are most likely to be equivalent from a given FSM. The general process is described in Algorithm 2. The `selectPairs` function selects a set of state pairs in the current machine that are deemed to be suitable merge-candidates

[1]Unlike many similar Machine Learning techniques, we cannot presume the presence of *negative* examples – in our scenario we only have software traces to work from, which are by definition positive. Accordingly, the algorithms presented here omit negative traces.

```
Data: S, T, s1, s2, score, PairScores, t1, t2
1  choosePairs(S, T) begin
2      PairScores ← ∅;
3      foreach (s1, s2) ∈ selectPairs(S) do
4          score ← calculateScore(s1, s2);
5          PairScores ← PairScores ∪ {(s1, s2, score)};
6      end
7      return sortDescending(PairScores)
8  end
9  calculateScore(s1, s2) begin
10     score ← 0;
11     while (t1, t2) ← equivalentTransitions({s1, s2}) do
12         score ← score + 1;
13         score ← score+calculateScore(t1_dest, t2_dest);
14     end
15     return score
16 end
```

Algorithm 2: The EDSM candidate pair selection procedure

(using an approach known as the Blue-Fringe algorithm [15], [6]). These pairs are then scored by the `calculateScore` function. This operates by counting the number of transitions in the outgoing paths from the two states that share the same labels (this is always finite because one of the states is always the root of a tree [15]). Finally, the state pair with the highest score is deemed to be most likely to be equivalent, and is returned.

`merge`: The merge function merges $s1$ into $s2$ by first removing $s1$ from S and (if $s1 \in F$) replacing $s1$ with itself in F. It continues by redirecting all transitions $s1_{in}$ to $s2$, and changes the source states of all transitions $s1_{out}$, making $s2$ their new source state. The resulting transition system is checked for non-determinism, and this is eliminated by recursively merging the targets of non-deterministic transitions. A more detailed description of this process can be found in Damas *et al.*'s description [6].

B. Data Classifier Inference

To extend the inference approaches beyond FSMs, and to produce EFSMs it is also necessary to learn succinct rules of behaviour over the data parameters present in the traces.

The process that is referred to here as "data classifier inference" refers to a broad range of techniques that seek to identify patterns or rules between variables from a set of observations, and to map these to a particular outcome or 'class'. The possible classes could be $\{true, false\}$ if the aim is to infer whether a given set of variable values is possible or not, or more elaborate, e.g. $\{rain, sun, overcast\}$ if the aim is to predict the weather from a set of factors.

A huge variety of techniques have been developed in the Machine Learning domain, spurred by problems in domains as diverse as Natural Language Processing, Speech Recognition, and Bioinformatics. A part of the reason for this diversity is that techniques can contain specific optimisations for their target domain. Amongst the hundreds of different techniques, core techniques include Quinlan's C4.5 Decision Trees inference algorithm [16], ensemble-learning approaches such as AdaBoost [17], and Bayesian inference techniques such as the simple naive Bayes approach [18].

Classifier inference techniques start with a sample of observations (referred to here as a 'data trace'), which map a set of

variable values to their respective 'classes', where a class is some categorical outcome. The goal is to produce some form of a decision procedure that is able to correctly predict the class of a set of unseen variable values. More formally, a data trace and the classifier inference problem can be defined as follows:

Definition 4: ***Data trace:*** Given a variable domain V and a set of classes C, a *data trace* T_D of size n can be defined as a set $\{(v_0, c_0), \ldots, (v_n.c_n)\}$, where a vector of variable assignments v_i is mapped to its corresponding class c_i.

Definition 5: ***Data classifiers and the inference problem:*** A *data classifier* $DC : V \rightarrow C$ is a function that maps samples of variable assignments to their respective classes. The inference problem can be characterised as inferring a classifier DC from a trace T that generalises upon the observations in the trace to classify further elements correctly when $v \notin T$.

As with FSM inference techniques, the choice of data rule inference technique depends on a multitude of factors (e.g. noisiness and potential bias in the data, the need for efficiency versus accuracy, readability of the inferred model etc.), and has itself been the subject of extensive research within the Machine Learning community [19]. Several comprehensive implementation frameworks have emerged to facilitate experimentation, such as the Java WEKA API [12] used for our work, which includes implementations of approximately 100 of the most of the established inference techniques.

C. Current EFSM Inference Approaches

The area of software model inference is vast, and has been active for the past 40 years [9]. Aside from the wealth of FSM inference and data-specification inference techniques, there are several techniques that have sought to combine the two. Notable examples include Dallmeier *et al.*'s ADABU technique [20] and Lorenzoli *et al.*'s Gk-tails algorithm [10]. Both have been generally evaluated on Java programs, but can in principle be applied to arbitrary program traces.

ADABU obtains models by obtaining a detailed record of the data-state at each point in the execution. It then adopts a simple, prescribed mapping to obtain abstract data characterisations at each point. For example, numerical values x are mapped to $x > 0$, $x == 0$, or $x < 0$, and objects are mapped to `null` or `instance of c` for some `c`. These are then combined into an abstract state machine according to the sequence in which these abstract states appear in the traces. The result is not strictly-speaking an EFSM (transitions do not have guards), but is mentioned here because it still combines data with control, and formulates this as a state machine.

For Gk-tails the basic idea is to build on top of the normal state-merging approach described above (they choose the well established k-tails variant [9]). However, each state transition is labelled not only with a label, but also with the set of variable values that correspond to the execution at that point. As transitions are merged into each other during the state merging process, they map to increasing numbers of data values. The GK-tails algorithm uses Daikon [1] to infer rules that link the variables together for each transition. These rules are then factored in to the selection of state pairs (the `choosePairs` function above) to prevent incompatible pairs of states from being merged. This is an important contribution; it shows how data from traces can factor in to the inference process, and has influenced the development of our algorithm, which we will describe in the following section.

Although they have proven to be successful for certain tasks, both approaches are hampered by key limitations. The first problem is flexibility. Both approaches are tied to very specific forms of data abstraction: the ADABU data-abstraction is hard-coded, and in the case of GK-tails it is tied to Daikon. However, it is well established from the field of Machine Learning that in practice different algorithms excel (and under perform) according to the specific characteristics of a given data set. It is implausible that a data abstraction technique would be well suited for for arbitrary sets of software traces. Some software behaviour may vary according to subtle numerical computations, others might vary according to specific configurations of boolean and string variables. These require different learners to yield the most accurate models.

The second problem is specific to GK-tails (since it infers EFSMs, whereas ADABU does not). Data rules that summarise the guards on a transition are inferred on a per-transition basis. This has two downsides. The first is the sheer volume of trace data that is required; each transition (in the ultimate model) needs to have been executed with a sufficiently broad range of data to yield a model that is accurate in its own right. The second is non-determinism. Since each transition data model is inferred independently, it is possible (and probable) that the resulting model contains states where there a given data set-up can lead to multiple outgoing transitions.

Models inferred by Gk-tails still have an intrinsic value for several purposes, and the inaccuracies discussed above can often be tolerated depending on the purpose of the model. For example, such models can still be useful if the model is intended as a descriptive overview of system behaviour. However, for purposes where the model might be required for classifying correct / incorrect behaviour, or to simulate a software system, models have to be more precise and deterministic.

IV. The EFSM Inference Algorithm

This section presents our EFSM inference algorithm. It adopts a similar template to the approach proposed by Lorenzoli *et al.* [10]. It also builds upon established state-merging techniques, and also works by attaching data values to transitions. However that is where the similarity ends. Instead of relying on a single data model inference approach, the algorithm proposed here is *modular*; it enables the incorporation of arbitrary data classifier inference techniques (there are over fifty in the WEKA library used by our reference implementation). Secondly, by using classifiers, the data rules and their subsequent control events are explicitly tied together. Finally, instead of inferring rules on a per-transition basis, a set of global data rules are inferred (e.g. some classifiers

Data: $EFSM, \Delta, k, c, DataTrace, s1, s2, t1, t2, Vars$
/* Here A is shorthand for the collection $(S, s_0, F, L, \Delta, T)$ as per definition 3. Components of A are denoted by subscript (e.g. A_S). */
/* $s_1, s_2 \in S$ */
/* $t1, t2 \in T$, */
/* $DataTrace$ is a trace as in definition 4. */
/* $Vars$ is a one-to-many mapping from transitions in T to trace elements in $Traces$. k is an (optional) integer ≥ 0 representing a minimum merge score. */
Result: An EFSM consistent with $Traces$

```
1  Infer(Traces, k) begin
2      DataTraces ← prepareDataTraces(Traces);
3      Δ ← inferClassifiers(DataTraces);
4      (A, Vars) ←generatePTA(Traces, Δ);
5      while (s1, s2) ← choosePairs(A, Δ, k) do
6          (A', Vars') ← merge(A, (s1, s2), Vars));
7          if consistent(A', Δ, Vars') then
8              A ← A';
9              Vars ← Vars';
10         end
11     end
12     return A
13 end
14 merge(A, s1, s2, Vars) begin
15     A_S ← A_S \ {s1};
16     A_F ← A_F \ {s1};
17     A_T ← changeSources(A_T, s1_out, s2);
18     A_T ← changeDestinations(A_T, s1_in, s2);
19     while (t1, t2) ←equivalentTransitions(A_T, s2, A_Δ) do
20         if (t1_dest == t2_dest) then
21             Vars(t2) ← Vars(t2) ∪ Vars(t1);
22             A_T ← A_T \ {t1};
23         else
24             (A, Vars) ← merge(A, (t1_dest, t2_dest), Vars);
25         end
26     end
27     return (A, Δ)
28 end
```

Algorithm 3: EFSM Inference Algorithm

infer them as if-then-else constructs), so they are able to take advantage of all of the available data, instead of just those data points that are attached to individual transitions.

The section starts off with a description of the algorithm itself. This is followed by a description of the state-merging algorithm that underpins the inference of the transition system. Finally, a small example is included to illustrate the key steps, and to provide an intuition of the final machine.

A. Inference Algorithm

The inference algorithm builds upon the state-merging baseline in Algorithm 1. In simple terms, there is an extra step beforehand, which is the inference of a set of classifiers. Each classifier corresponds to a label in the trace (e.g. the signature of a method in a Java trace). From a set of inputs to the method, the classifier serves to predict the next label in the trace (i.e. the name of the next method to be called). In the rest of the inference algorithm, the purpose of most functions remains the same, however this time their behaviour is modulated by the classifiers. Also, there is a new function `consistent`, which ensures that the model that is produced at each iteration is consistent with all of the classifiers.

The algorithm is shown in Algorithm 3, and the key steps are illustrated in Figure 1. For an intuitive understanding, it is easiest to start with Figure 1, and to then trace the equivalent steps in Algorithm 3. The following description will draw upon both. Starting with a set of initial traces (top left), the algorithm starts by processing them to create one 'data traces' per label (shown to the right in the diagram). This adds a 'class' variable to each data point, showing for every data configuration what the label of the subsequent event is. These data traces can then be used to infer a set of classifiers (moving to the right in the diagram). The examples used here are decision trees (such as those produced by the C4.5 algorithm [16]), but they could be any suitable representation of a decision procedure, such as neural nets or if-then-else clauses [12].

At this point, the initial set of traces is re-analysed, but this time to produce a Prefix Tree Acceptor (PTA) [8] (line 4 in the algorithm, bottom left in the diagram). There are however two important differences from the prefix trees used in the conventional algorithm. In this PTA, transitions are labelled not only with the name of a function, but also with the sets of data variable values that correspond to each transition (in the algorithm this mapping is represented by $Vars$). For example, in Figure 1, the transition $0 \xrightarrow{mult} 1$ would map to $\{(x = 4, y = 2, res = NaN), (x = 1, y = 1, res = NaN), (x = 2, y = 0, res = 0)\}$. Secondly, a pair of states (a, b) only share the same prefix in the PTA if the inferred classifiers yield identical predictions for every data-configuration in the prefix of a as they do for b. This means that the PTA represents the most specific EFSM that exactly represents the given set of traces.

Together, the embellished PTA and the data classifiers are used as a basis for iterative state-merging, following a similar iterative loop to the original state-merging procedure in Algorithm 1. The key differences are as follows.

The `choosePairs` function (line 5) takes several additional variables into account. k is an optional parameter to represent the minimum score before a pair of states can be deemed to be equivalent. A pair of transitions can only be deemed to be equivalent if their attached data values lead to the same predictions. The other additional parameter is self-explanatory; the set of classifiers Δ is required to compute the equivalence of transitions for the computation of the scores.

The `merge` function (line 6 - shown in full in lines 14-28) differs from the generic version. The is similar in nature to the previous differences and is centred on the `equivalentTransitions` function. This function does not not just deem two transitions to be equivalent if the labels are the same, but also factors in the equivalence of the attached data values, as determined by the relevant data classifiers. So although the initial steps of generating the initial merge remain identical (see Figure 1), the difference lies in how any non-determinism in the resulting transition structure is handled. In the generic FSM inference approach in Algorithm 1, non-determinism is simply eliminated by recursive merging. Here a similar recursive merging process takes place, however (as with the PTA generation), transitions are not merely compared to each other in terms of their labels, but also in terms of their attached data.

Once states and transitions have been merged, the `merge` function has the additional responsibility of reallocating the data values from the source transitions to the target ones. This

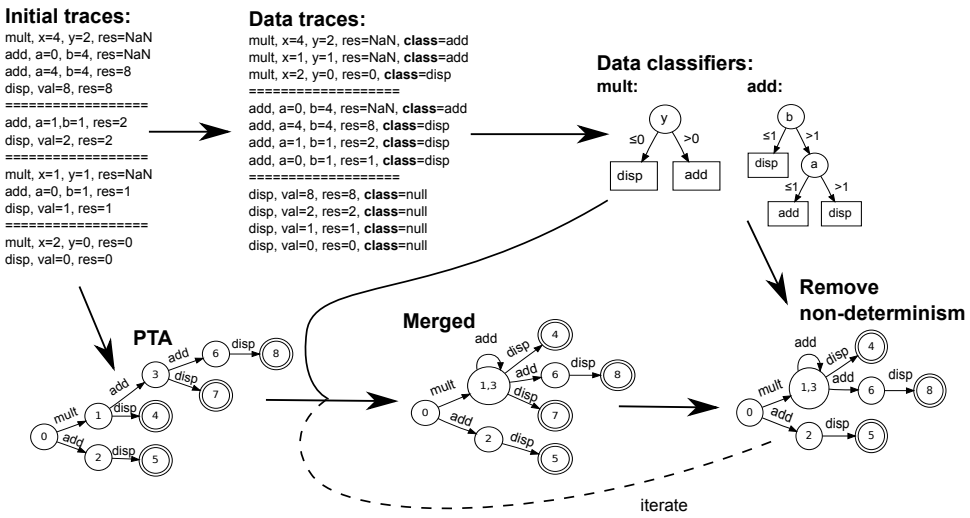

Figure 1. Illustration of the key inference steps with respect to traces from an imaginary calculator system. The classifiers used in this example happen to be decision trees, but could assume any representation depending on the chosen data rule inference algorithm.

is achieved by updating the map $Vars$ from transitions to data variables (line 21). The resulting merged model and updated mappings from transitions to data are returned as $(A', Vars')$.

Finally, once the merge has been processed, the resulting model is checked by the function `consistent`. This makes sure that the merged machine A' attached data $Vars'$ are consistent with the classifiers in Δ. For each transition $t \in A'_T$, the corresponding the data variable values are obtained from $Vars'(t)$. This is provided to the corresponding classifier in Δ, which returns a predicted subsequent label (i.e. the name of the method to be executed subsequently). This is checked against the transition structure of A'. If the target state of t does not have an outgoing transition with the predicted label, `consistent` returns `false`.

If A' is consistent with Δ and Var', the 'current' state machine A is updated to A', Var is updated to Var', and the whole process iterates until no further merges can be found. If consistent returns `false`, the current merge is ignored, and the next merge is attempted. This whole process continues until no more merges can be identified, when the process returns (A, Δ).

B. The Mine Pump Case Study

This section is concluded with a small example to provide an intuition of what the inferred machines look like, and to show how the data-rules and state machines can be interpreted as an EFSM. To do this we recall the example of the mine pump controller, as briefly discussed in Section II-B. We have a trace of 2931 events (illustrated previously), and we wish to better understand the underlying behaviour by inferring an EFSM. The full set of traces for this example and the other evaluation material, along with the proof of concept program are available from the project URL[2].

[2]http://www.cs.le.ac.uk/people/nwalkinshaw/efsm/

The following output is obtained by running our tool, using the WEKA J48 decision tree learner (the C4.5 algorithm [16]) to infer the underlying data relations. Although the tool is equipped to visualise the data-guards and state transitions in a GUI, we show them separately here, to provide a better intuition of what an inferred EFSM looks like. The resulting decision trees (inferred using the WEKA default settings) are shown below:

```
====== MODEL FOR:critical ======
pump = true: switch_pump_off
pump = false
|   methane <= 607.2162
|   |   methane <= 602.640094: not_critical
|   |   methane > 602.640094
|   |   |   water <= 31.699162: not_critical
|   |   |   water > 31.699162: critical
|   methane > 607.2162
|   |   water <= 74.875037: highwater
|   |   water > 74.875037: critical

====== MODEL FOR:not_critical ======
: switch_pump_on

====== MODEL FOR:highwater ======
methane <= 598.525559: switch_pump_on
methane > 598.525559: critical

====== MODEL FOR:turn_off ======
methane <= 597.355089: highwater
methane > 597.355089: not_critical

====== MODEL FOR:turn_on ======
methane <= 590.815697: low_water
methane > 590.815697
|   water <= 35.75304: low_water
|   water > 35.75304: critical
```

The decision trees be read as a set of if-then-else rules, where the conditions are on the values of the variables, and the outcomes represent the next event. For example, looking at the `highwater` tree, if `methane <= 598.525559` the model predicts that the next event is `switch_pump_on` , otherwise the next event is `critical`. For the `not_critical` tree, there are no conditions on the

variables, the next event is always `switch_pump_on`.

These data models already provide some basic constraints on the sequential behaviour of the system. They show which combinations of variable values lead from one function to another. However, they fail to provide a macroscopic view of the order in which the events can occur. This is provided by the state transition diagram, shown in Figure 2.

Together, the data-models and the state transition system form the EFSM. The inference algorithm ensures that the state transition system fully obeys the inferred data-models; if a state has an incoming event with a model that predicts a given event X, it will always have an outgoing transition for X.

Every transition corresponds to a particular "case" in the data model - a set of conditions in the model that yield a specific outcome. So the model is completely deterministic. Although there are states in Figure 2 with multiple outgoing transitions with the same label (e.g. state 0), the choice between which transition to take is decided by the data values at state 0. This is difficult to visualise without the GUI, but can in most cases be derived from the data-models by looking at the data-models. For state 0, the `critical` transition to state 113 occurs when `pump = false, methane > 607.2162, water <= 74.875037` , because this is the set of conditions that leads to the event `highwater` .

The model is intuitively useful. It is relatively compact, reducing 2931 trace events to a transition system of 17 states, accompanied by a reasonably compact description of the associated data rules. It also gives a good intuition of how the system behaves. The outgoing trajectories from state 0 correspond to what happens when (a) there are critical methane and water levels (state 113), (b) there are just critical methane levels (state 1027) and (c) there is just high water (state 1). It shows the order of events involved in switching the pump on and off, when these can occur, and how the different states are interrelated.

V. Preliminary Evaluation

The EFSM inference algorithm was implemented in Java, and is freely available[2]. For the evaluation the implementation has been applied to traces of test executions from modules in two systems. These are used to explore the accuracy of the models, and to illustrate the scalability of the approach with respect to realistic traces.

A. Experimental Setup

1) Subject systems: The `SMTPTransport` class in Oracle JavaMail[3] provides the functionality to send emails by SMTP. This choice was inspired by its use by Dallmeier *et al.* [21]. The test cases used were its own JUnit tests, coupled with the test sets for Apache Commons Mail[4] (which is built upon Java Mail). The Erlang `Poolboy` module in the Basho Riak distributed database[5], which implements a process for pooling connections. As test cases we used its own set of E-unit test cases.

Traces of test cases in Java were collected by developing a tracing-aspect in AspectJ[6], and traces for Erlang modules were collected with an automated instrumentation system developed using the Wrangler refactoring API [22]. No manual abstraction was used (to avoid biasing the results). The tracers recorded trace-data for any invocations of interface functions to the module of interest. The recorded data included all parameter inputs, outputs, and (for Java) all instance variable values.

2) Measuring accuracy: EFSM inference techniques are intrinsically difficult to evaluate. Current approaches are 'model-based', in the sense that they rely upon some reference model that can be used as a basis for computing accuracy [23], [24], [21], [25]. This requirement for hand-crafted models can however be restrictive in terms of the size and complexity of the model against which a technique can be evaluated.

In Machine Learning this problem is common - there are typically no gold-standard reference models. One of the most popular evaluation techniques that can be used in such a situation (which we adopt for this study) is a technique known as *k-folds cross validation* [26]. Here, the simplifying assumption is made that the given set of examples (test cases in our case) collectively exercise the full range of behaviour from which we are inferring a model. The set is randomly partitioned into k non-overlapping sets. Over k iterations, all bar one of the sets are used to infer a model, and the remaining set is used to evaluate the model according to some metric (discussed below). For each iteration a different set is used for the evaluation. The final accuracy score is taken as the average of the k accuracy scores.

Of course, given the probability that test sets are not "rich enough", the accuracy score cannot be interpreted as an *absolute* score of the accuracy of the model with respect to the system in question. As with all dynamic analysis techniques, an incomplete test set will yield an incomplete model [21]. However, it can be used in our experimental setting to compare the *relative* performance of different model inference configurations. If we go further and accept that the test set represents a reasonable sample of typical program behaviour, then the resulting scores can be interpreted as being at least indicative of the actual accuracy score.

There are several metrics by which to assess 'accuracy', such as Precision, Recall (also known as Sensitivity) and Specificity. All are computed from the sets of true-positives (TP), true-negatives (TN), false positives (FP) and false-negatives (FN). We chose Sensitivity ($TP/(TP + FN)$), Specificity ($TN/(TN + FP)$), and BCR, which is the mean of the two [25].

3) Negative traces: To compute the true and false positive / negatives, it is not only necessary to have a useful base-sample of program traces, but also a set of 'negative' traces - traces that do *not* belong to the system in question (but

[3]http://www.oracle.com/technetwork/java/javamail/index.html
[4]http://commons.apache.org/proper/commons-email/
[5]http://basho.com/riak/
[6]http://eclipse.org/aspectj/

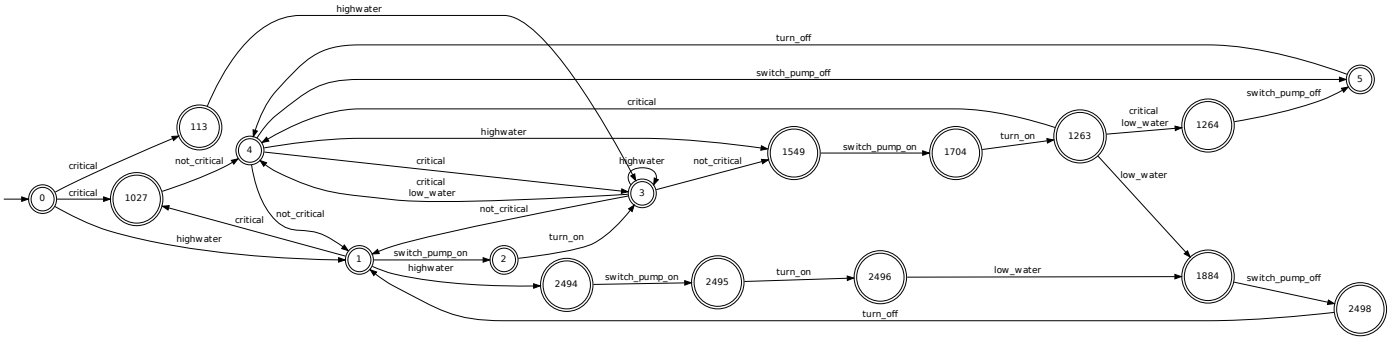

Figure 2. Inferred state transition diagram for mine pump controller

remain characteristically similar in terms of their contents). To obtain these negative test cases we adopted an approach inspired by the work of Dallmeier *et al.* [21]. We provided the same inputs to the program that were used to obtain the 'positive' traces, but mutated the program first. For this we use established mutation frameworks: MAJOR [27] for Java, and for Erlang a mutation testing tool based on the Wrangler refactoring API [22]. For each trace from the mutated program we manually inspect a text-diff of the trace with respect to the original to make sure that the mutant is not 'equivalent' - i.e. that it actually changes the behaviour. All of the traces used for the experiment are available on the project web page [refprojURL].

Once the traces are obtained, we use the positive traces for the k-folds cross-validation, and add the negative traces to the evaluation set at each iteration. We set k for the k-folds cross validation to 5, to ensure that there was an adequate sized evaluation set at each iteration. For both systems we controlled the experiment by attempting every combination of data classifier inference algorithm and minimum merging score k (not to be confused with the fold number).

The details of the experimental variables are as follows. We used eight classifier inference algorithms implemented in WEKA [12], using the default WEKA settings for each algorithm. The algorithms were selected to fit a broad range of numerical or discrete systems, and are: C4.5 [16] (listed as J48), NNGE, Naive-Bayes [18], AdaBoost [17], AdditiveRegression, JRIP (also known as RIPPER), M5 and M5Rules. We varied the minimum merge score k from 0 (only rely on data) to 1 (rely on data, but there must be at least one suffix that is the same for two states to be merged). Each configuration was repeated six times with different random seeds. For each system this resulted in 98 configurations. Since we were using $k = 5$ for the cross validation (five models were inferred per configuration), this meant that overall 490 models were inferred per case study. The experiments were executed on a 1.4GHz Intel Core2 Duo laptop with 2 GB of memory running Ubuntu Linux 12.04.

4) No baseline: This study is entirely exploratory, in the sense that it does not compare the accuracy of the models against a baseline technique. The reason for this is that there are no comparable baseline techniques that the authors are aware of. As mentioned in the related work, there are other techniques that produce EFSMs and conventional FSMs without data. However, the specific nature of these machines are so fundamentally different that it makes no sense to attempt to compare their accuracy. The EFSMs produced by Lorenzoli *et al.* [10] are non-deterministic (the models are mainly intended to be descriptive). However, when used to reason about their languages, it is possible that a given sequence might be both possible and impossible in the same machine, which hampers the comparison to our deterministic EFSMs in terms of accuracy. Conventional FSMs (e.g. as produced by k-tails [9]) do not incorporate data, but surely the data guards are a fundamental aspect for evaluating EFSMs. In both cases, a path through the model means something completely different, and as such attempting to draw a quantitative comparison would give few valid insights.

	`Poolboy` (Basho Riak)			`SMTPTransport` (Java Mail)		
k	**Classifier**	**BCR**	**ms**	**Classifier**	**BCR**	**ms**
0	NNGE	0.688	1604	JRIP	0.728	1877
	Bayes	0.665	1816	Bayes	0.715	663
	JRIP	0.616	353	J48	0.677	1137
	AdaBoost	0.613	321	AdaBoost	0.673	969
	J48	0.592	1320	NNGE	0.598	786
1	Bayes	0.715	180	JRIP	0.986	1386
	NNGE	0.704	398	Bayes	0.986	484
	JRIP	0.691	398	NNGE	0.986	584
	AdaBoost	0.67	138	AdaBoost	0.967	1312
	J48	0.65	107	J48	0.967	426

Table I
TOP FIVE CONFIGURATIONS PER SYSTEM, FOR EACH k.

B. Results

Since there is not enough space to present the results here in full, they can be downloaded[2]. The results are summarised in Figure 3. The key column is BCR (the harmonic mean of Sensitivity and Specificity). A BCR value of 0.5 indicates that the ability of the model to correctly determine whether a trace is valid or not is no better than a random guess.

The results, together, show that if you choose a suitable learner (regardless of the value of k), the inferred EFSM will achieve an accuracy value of approximately 0.7 or more. There is however a substantial variance in the accuracy, depending on the system, the choice of learner, and k. These are discussed further below.

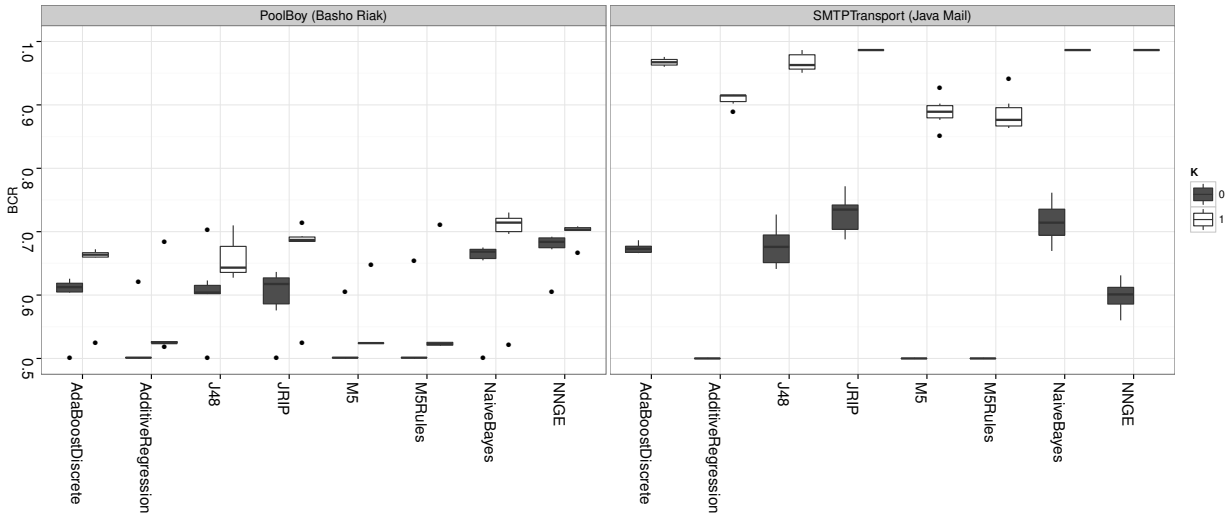

Figure 3. Overview of results. Box plots show an overview of all BCR scores, in terms of individual data classifier algorithms and values of k.

The first point is that the performance largely depends upon the selection of a suitable data classifier inference algorithm. This underscores our motivation for the paper as discussed in the introduction. There is no single combination of inference algorithms that will uniformly perform well for every type of system. For both systems, all of the configurations that involved the learners M5, M5Rules, and AdditiveRegression performed badly, especially for the PoolBoy system. This is because these numerical learners were not suited to the largely string-based trace data, which meant that their classification accuracy was very poor. However, other learners such as J48, NaiveBayes, and NNGE all performed relatively well on both systems.

The difference in performance between the two systems is apparent. The accuracy results on `PoolBoy` are significantly lower than on `SMTPTransport`, and the effect of setting $k = 1$ is much less pronounced. The reason for this comes down to a combination of factors: differences in the range of behaviours exhibited by the two systems, and the 'learnability' of their data constraints. `SMTPTransport` has a much narrower range of sequential behaviours, which makes its states easier to infer; this also explains why setting $k = 1$ has such a pronounced effect. Secondly, `SMTPTransport` makes use of basic data variables (e.g. numbers and booleans) from which models are often easy to infer. `PoolBoy`, being an Erlang program, makes use of complex nested lists and tuples, which are often difficult to untangle, and which rarely form the basis for meaningful data models.

Finally, there is the time factor. Perhaps surprisingly, there is no correlation between the time taken and the accuracy of the final result. Often inaccurate models take a lot longer to infer. The use of data classifiers does not necessarily lead to a substantial overhead in execution time, as shown by the `SMTPTransport` results when compared to k-tails. One reason for this is that suitable classifiers are good at ruling-out poor merges before they are attempted (saving the time), whereas unsuitable classifiers can lead to time-consuming merging operations that ultimately end up being inconsistent, and so have to be dismissed.

a) Threats to validity and discussion: It has to be emphasised that this study is only preliminary, and that these results can therefore only be interpreted as indicative. There are several other factors that a study on a larger number of systems will have to control (to be addressed in future work). For example, the number and diversity of test cases needs to be taken into account [21], [3], as does the type of system (the extent to which its behaviour is dependent on data, and the number of variables involved).

In k-folds cross validation it is assumed that the trace set as a whole is 'representative' of software behaviour in general (this is a universal assumption of dynamic analysis techniques). Of course, the accuracy results can therefore only be interpreted in this light. If the trace sets are incomplete, the models are equally incomplete. One means by which to reduce the possibility of incomplete trace samples is to adopt a model-guided test input generation strategy, as proposed by Dallmeier et all [21], and in previous work by the authors [3].

Finally, it is important to note that quantifying accuracy only gives a partial view of the value of the technique. As was shown with the small mine-pump example, the inferred model and data constraints have a qualitative value that is difficult to quantify. They can provide a useful insight into the behaviour of a system. Even inaccurate models can play a useful role in understanding a system [1], or in assessing the extent to which a system has been tested [3].

VI. Conclusions and Future Work

This paper has introduced a new algorithm for the inference of EFSMs. It enables the combination of established state-

merging techniques with arbitrary data classifier inference algorithms, to infer models that more fully capture the behaviour of a software system. The approach does not rely on source code analysis, and has accordingly been demonstrated with respect to two systems written in entirely different programming languages and paradigms.

One of the contributions of the paper is a novel evaluation methodology that does not rely on the prior existence of hand-crafted models. This combines the k-folds cross validation method with program mutation (which is used to identify 'negative' examples). Our preliminary results from the experiments indicate that the algorithm is capable of returning accurate models if it uses suitable data classifiers. This latter point is especially important, and supports one of the key motivations for this work; there is no single combination of learners that will perform uniformly well for arbitrary software systems. The algorithm presented here offers the flexibility to incorporate different data classifier algorithms, depending on the characteristics of the data. In the immediate future we will carry out a more extensive, systematic evaluation. This will seek to assess some of the factors that were not controlled in this study, and will also look at qualitative factors, such as readability.

Alongside the work on further evaluation, we intend to use the models to build upon our early research on combining model inference with test generation. So far, this has concentrated solely on simple finite state machines [28], [29], [4] and data classifiers [3]. It is envisaged that the ability to incorporate these richer models will lead the the ability to produce more more rigorous test sets.

Finally, there remains the fact that the EFSMs inferred here (and by other techniques) are missing an important component. Although they produce state machines with guards, they are missing the actual data functions that transform the data state at each transition. Although this has been the subject of previous work by the authors [30], the technique proposed at the time relied on source code analysis. Our future work will investigate alternative dynamic analysis techniques that do not require source code (and are not restricted to specific languages), by exploring the use of techniques such as Genetic Programming.

REFERENCES

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 1–25, 2001.

[2] A. Valdes and K. Skinner, "Adaptive, model-based monitoring for cyber attack detection," in *Recent Advances in Intrusion Detection*. Springer, 2000, pp. 80–93.

[3] G. Fraser and N. Walkinshaw, "Behaviourally adequate software testing," in *ICST 2012*. IEEE, 2012, pp. 300–309.

[4] R. Taylor, M. Hall, K. Bogdanov, and J. Derrick, "Using behaviour inference to optimise regression test sets," in *ICTSS*, 2012, pp. 184–199.

[5] J. Cook and A. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 215–249, Jul. 1998.

[6] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde, "Generating annotated behavior models from end-user scenarios," *IEEE Trans. Software Eng*, vol. 31, no. 12, 2005.

[7] C. Lee, F. Chen, and G. Roşu, "Mining parametric specifications," in *ICSE*, 2011, pp. 591–600.

[8] N. Walkinshaw, B. Lambeau, C. Damas, K. Bogdanov, and P. Dupont, "STAMINA: a competition to encourage the development and assessment of software model inference techniques," *Empirical Software Engineering*, pp. 1–34, 2012.

[9] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behaviour," *IEEE Transactions on Computers*, vol. C, no. 21, pp. 592–597, 1972.

[10] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *ICSE*. ACM, 2008, pp. 501–510.

[11] K. Cheng and A. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *IDAC*. ACM, 1993, pp. 86–91.

[12] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, pp. 10–18, November 2009.

[13] J. Kramer, J. Magee, M. Sloman, and A. Lister, "Conic: an integrated approach to distributed computer control systems," *IEE Proceedings*, vol. 130, no. 1, pp. 1–10, 1983.

[14] E. M. Gold, "Language identification in the limit," *Information and Control*, vol. 10, pp. 447–474, 1967.

[15] 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 *ICGI*, V. Honavar and G. Slutzki, Eds., vol. 1433, 1998, pp. 1–12.

[16] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1993.

[17] Y. Freund and R. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," in *Computational learning theory*. Springer, 1995, pp. 23–37.

[18] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[19] S. Weiss and I. Kapouleas, "An empirical comparison of pattern recognition, neural nets and machine learning classification methods," *Readings in machine learning*, pp. 177–183, 1990.

[20] V. D. C., Lindig, A. Wasylkowski, and A. Zeller, "Mining object behavior with adabu," in *Proceedings of the 2006 international workshop on Dynamic systems analysis*. ACM, 2006, pp. 17–24.

[21] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 243–257, 2012.

[22] H. Li and S. Thompson, "A User-extensible Refactoring Tool for Erlang Programs," University of Kent, Tech. Rep., 2011. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2011/3171

[23] D. Lo and S.-C. Khoo, "QUARK: Empirical assessment of automaton-based specification miners," in *WCRE*. IEEE Computer Society, 2006, pp. 51–60.

[24] D. Lo, L. Mariani, and M. Santoro, "Learning extended FSA from software: An empirical assessment," *Journal of Systems and Software*, vol. 85, no. 9, pp. 2063–2076, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2012.04.001

[25] N. Walkinshaw and K. Bogdanov, "Automated comparison of state-based software models in terms of their language and structure," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 2, 2013.

[26] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *International joint Conference on artificial intelligence*, vol. 14, 1995, pp. 1137–1145.

[27] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a java compiler," in *ASE*, 2011, pp. 612–615.

[28] N. Walkinshaw, J. Derrick, and Q. Guo, "Iterative refinement of reverse-engineered models by model-based testing," in *FM*, 2009, pp. 305–320.

[29] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris, "Increasing functional coverage by inductive testing: A case study," in *ICTSS*, 2010, pp. 126–141.

[30] N. Walkinshaw, K. Bogdanov, S. Ali, and M. Holcombe, "Automated discovery of state transitions and their functions in source code," *Softw. Test., Verif. Reliab.*, vol. 18, no. 2, pp. 99–121, 2008.