Incorporating Data into EFSM Inference

Michael Foster[1], Achim D. Brucker[2], Ramsay G. Taylor[1], Siobhán North[1], and John Derrick[1]

[1] Department of Computer Science, The University of Sheffield
Regent Court, Sheffield, S1 4DP, UK
{jmafoster1, r.g.taylor, s.north, j.derrick}@sheffield.ac.uk
[2] Department of Computer Science, University of Exeter, Exeter, UK
a.brucker@exeter.ac.uk

Abstract. Models are an important way of understanding software systems. If they do not already exist, then we need to infer them from system behaviour. Most current approaches infer classical FSM models that do not consider data, thus limiting applicability. EFSMs provide a way to concisely model systems with an internal state but existing inference techniques either do not infer models which allow outputs to be computed from inputs, or rely heavily on comprehensive white-box traces that reveal the internal program state, which are often unavailable.
In this paper, we present an approach for inferring EFSM models, including functions that modify the internal state. Our technique uses black-box traces which only contain information visible to an external observer of the system. We implemented our approach as a prototype.

Keywords: EFSM Inference · Model Inference · Reverse Engineering

1 Introduction

Accurate system models are applicable to a broad range of software engineering tasks. They can be used to automate the process of model-based testing [7,15], to detect cyber attacks [16], and to aid the process of requirements capture [4]. Despite their utility, system models can be neglected during development, if they are built at all. It is therefore desirable to reverse engineer models from existing systems. One way to do this is to record executions of the system and use these *traces* to infer a model.

There is abundant work on the inference of finite state machine (FSM) models from traces [2,10,18], much of which falls into the family of *state merging* algorithms. These begin by constructing the most specific automaton which accepts all of the observed traces, and iteratively consolidate the model by merging states in the FSM which are believed to represent the same program state. The resulting model, as well as being smaller than the original, is often more general. It is able to predict how the system might behave when faced with previously unseen traces. This is a key feature of model inference and differentiates it from automaton minimisation.

P.C. Ölveczky and G. Salaün (Eds.): SEFM 2019, LNCS 11724, pp. 1–17, 2019.

Classical FSMs cannot handle data so they struggle to represent systems that exhibit data-dependent behaviour, for example a vending machine which dispenses drinks selected by users. Here, the input of the *select* action determines the output of *dispense*. A classical FSM model of the system would require a separate path for each available drink, so would likely be rather large. Extended Finite State Machines (EFSMs) provide a richer model, featuring a persistent data state, which could be used to store the selected drink, but existing EFSM inference techniques [12,19] do not infer how the data state is used, nor can they capture the *causal* effect of input on output.

This paper presents a technique to infer EFSM models from system traces which explicitly capture this causal relationship. The main contributions are:

1. A technique which uses black-box traces (instead of the more commonly used white-box traces) to infer EFSM models which capture the causal relationship between inputs and outputs.
2. A prototype tool which implements this technique.

The rest of this paper is structured as follows. Section 2 introduces a motivating example and explains how state merging algorithms work. Section 3 presents our EFSM inference technique. Section 4 discusses how we introduce data registers to capture the causal relationship between input and output. Section 5 details how we implemented our technique as a prototype inference tool. Section 6 evaluates our technique with reference to the scenario presented in Section 2. Section 7 concludes the paper and discusses possible future works.

2 Background

Reverse engineering models from traces is an inference process which aims to make statements about the overall behaviour of a system by generalising from observations. Consider a simple vending machine which produces traces like those in Figure 1. Users first *select* a drink by providing its name as an input. The *coin* operation allows users to pay for their drink by inserting coins of a given value, displaying as output the total value inserted so far. Once sufficient payment has been inserted, the *vend* operation is triggered to dispense the drink.

In Figure 1, we use the notation $methodName(i_1, i_2, \ldots)/[o_1, o_2, \ldots]$ such that $coin(50)/[50]$ represents the event *coin* being called with a single input of 50 and producing a single output of 50. We delimit events with arrows and omit the outputs from events like $select(\text{`}coke\text{'})$ which do not produce any.

$$select(\text{`}coke\text{'}) \rightarrow coin(50)/[50] \rightarrow coin(50)/[100] \rightarrow vend()/[\text{`}coke\text{'}]$$
$$select(\text{`}pepsi\text{'}) \rightarrow coin(50)/[50] \rightarrow coin(50)/[100] \rightarrow vend()/[\text{`}pepsi\text{'}]$$
$$select(\text{`}coke\text{'}) \rightarrow coin(100)/[100] \rightarrow vend()/[\text{`}coke\text{'}]$$

Fig. 1: Exemplary traces of the vending machine.

To infer a classical FSM model from the traces in Figure 1, we must either remove the data entirely or encode it within the actions by folding input and output values into the transition labels. Taking the latter approach, we represent an event $label(i_1)/[o_1]$ as the atomic action $label\_i_1\_o_1$. The inference process begins by building an automaton which accepts exactly the observed traces. This is usually a tree-shaped automaton called a prefix tree acceptor (PTA), where traces with common prefixes share a common path through the model up to the point of divergence. Figure 2 shows a PTA representing the traces in Figure 1.

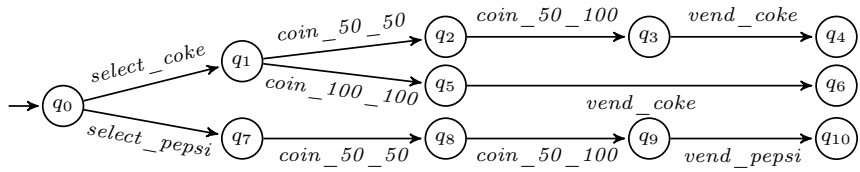


Fig. 2: A classical FSM PTA built from the traces in Figure 1, in which transition input and output data has been encoded into the transition labels.

We condense the PTA by merging states which we believe represent the same program state, based on the commonality of their outgoing transitions. In Figure 2, for example, q_3 and q_5 both have an outgoing *vend_coke* transition. The result of merging these two states has two nondeterministic outgoing *vend_coke* transitions. This does not make sense as we merged q_3 and q_5 because we believe their respective outgoing transitions represent the same behaviour, meaning that their destination states should represent the same program state. We merge these states (q_4 and q_6) so the two *vend_coke* transitions are no longer distinct. In this way, branches of a PTA are *zipped* together as we merge successive states.

When the model becomes deterministic again, we search for another pair of states, that might represent the same program state, to merge. This continues until no more pairs of states are believed to represent the same program state. An optimal FSM model which could be inferred from the traces in Figure 1 is shown in Figure 3. This is more concise than the PTA in Figure 2 but is still relatively large and cannot predict the behaviour of the system for unseen inputs.

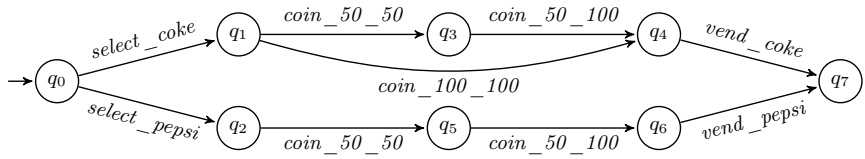


Fig. 3: A classical FSM representation of a simple drinks machine in which transition input and output data has been encoded in the transition labels.

We cannot expect to infer models of unobserved behaviour but it is not unreasonable to want to predict the outcome of applying the same *action* with different *data*. Classical FSM models cannot separate these, so the transitions *select_coke* and *select_pepsi* are different behaviours rather than two instances of *select* with different inputs. This means that small changes in behaviour, like adding additional drinks, have a disproportionate effect on model complexity. EFSMs are a promising solution to this problem. Numerous definitions exist in the literature [3,6,11] but all make use of similar features: parametrised guarded inputs, a persistent data state, and output expressed in terms of input. These features make EFSM models more expressive but also much harder to infer.

Previous work on EFSM inference [12,19] focusses on establishing concise transition guards which aggregate observed data values into a single transition. While this is a valuable contribution, the models inferred by these techniques fail to capture the fact that input *determines* subsequent output.

Example 1. For the traces in Figure 1, existing EFSM inference methods might produce a model similar to Figure 4. It is much smaller than the classical FSM in Figure 3, as we are now able to separate action from data. Here, we have a guard on the *select* transition which requires the first input, i_1, to be either *'coke'* or *'pepsi'*. This is mirrored by the output, o_1, of *vend*. All observed inputs and outputs of *coin* were greater than or equal to 50 so the guard reflects this.

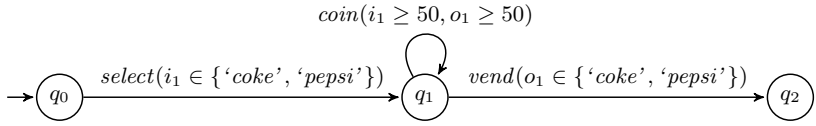


Fig. 4: An EFSM as might be inferred by existing methods. Here, transitions take the form $label(g_1, g_2, \ldots)$ in which inputs are denoted i_n and outputs o_n.

This model summarises the observed values but fails to show how output is computed from input — it is not *computational*. We know that the output of the *vend* transition is either *'coke'* or *'pepsi'* but cannot tell which we will get, much less that it is determined by the input to *select*. Inputs and outputs are both just treated as variables here so there is no explicit link between them. □

The EFSMs inferred by [12] and [19] use the program variables present in the traces but do not infer *how* individual transitions update variables. An ideal EFSM model of the traces in Figure 1 is shown in Figure 5, in which transitions are written $label : arity[guards]/outputs[updates]$. Here, we use a register, r_1, to record the selected drink, and another, r_2, to keep track of the money inserted. This allows us to *compute* the outputs of *vend* and *coin*. Techniques such as [17] attempt to infer fully computational models like this but rely on white-box traces to expose the values of internal variables. Since white-box traces are often unavailable, we would ideally like to use black-box traces, which only contain information available to an external observer of the system.

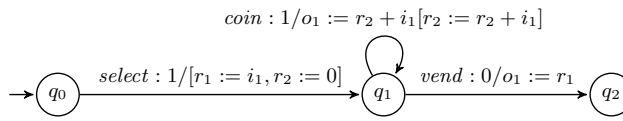


Fig. 5: The ideal EFSM model of the drinks machine.

3 Extending the Inference Process

In this section, we present our technique to infer EFSM models from traces. While there are many different EFSM representations in the literature, [3,11] we use the one we defined in previous work [6].

Definition 1. *An EFSM is a tuple,* (S, s_0, T) *where* S *is a finite non-empty set of states,* $s_0 \in S$ *is the initial state, and* T *is the transition matrix* $T : (S \times S) \rightarrow \mathcal{P}(L \times \mathbb{N} \times G \times F \times U)$ *with rows representing origin states and columns representing destination states. In* T*,* L *is a set of transition labels.* $\mathbb{N}$ *gives the transition* arity *(the number of input parameters), which may be zero.* G *is a set of Boolean guard functions* $G : (I \times R) \rightarrow \mathbb{B}$*.* F *is a set of* output functions $F : (I \times R) \rightarrow O$*.* U *is a set of* update functions $U : (I \times R) \rightarrow R$*.*

In G*,* F*, and* U*,* I *is a tuple* $[i_1, i_2, \ldots, i_m]$ *of values representing the inputs of a transition, which is empty if the arity is zero. Inputs do not persist across states or transitions.* R *is a mapping from variables* $[r_1, r_2, \ldots]$*, representing each register of the machine, to their values. Registers are globally accessible and persist throughout the operation of the machine. All registers are initially undefined until explicitly set by an update expression.* O *is a tuple* $[o_1, o_2, \ldots, o_n]$ *of values, which may be empty, representing the outputs of a transition.*

Syntactic sugar allows transitions from state S_m to state S_n to take the form

$$S_m \xrightarrow{label:arity[g_1,\ldots,g_n]/f_1,\ldots,f_n[u_1,\ldots,u_n]} S_n$$

The first part of the transition is an atomic *label* naming the event. This is followed by a colon and the *arity* of the transition. Guard expressions $g_1, \ldots, g_n$ are enclosed in square brackets. Next comes a slash, after which $f_1, \ldots, f_n$ define the outputs. Finally, update expressions $u_1, \ldots, u_n$, enclosed in square brackets, define the posterior data state. There should be at most one update function per register per transition to maintain consistency. For transitions without guards, outputs, or updates, the corresponding components are omitted.

Our inference process follows the same basic structure as classical FSM inference algorithms — we build a PTA and then iteratively merge states to form a smaller model. Our technique differs from classical FSM inference in two ways. Firstly, because of the more complex EFSM transitions, attempts to resolve the nondeterminism introduced by merging states might fail, meaning that two states which initially seemed compatible cannot actually be merged. This is not the case in classical FSM inference. We tackle this in Subsection 3.2. Secondly, the nondeterminism introduced by merging states cannot be resolved by simply merging destination states. We address this in Subsection 3.3.

3.1 PTA Construction

The first step is to construct a PTA from the observed traces in the same way as for classical FSM inference. Beginning with the empty EFSM, we iteratively attempt to walk each observed trace in the machine. When we reach a point where there is no available transition, we add one. While classical FSMs use an atomic label, EFSMs deal with data so we add guards to test for the observed input values, and outputs which produce the observed values. For example, the event $coin(50)/[50]$ causes the transition $coin : 1[i_1 = 50]/o_1 := 50$ to be added to the machine. The event label is *coin*. It takes one input, which must be equal to the observed input value of 50, and produces the literal output 50.

3.2 Merging States

Like in classical FSM inference, we use a predefined metric to order potential state merges by how strongly we believe that two states represent the same program state. The INFERENCESTEP function in Algorithm 1 merges the first (highest scoring) pair in the list of potential merges and calls RESOLVENONDETERMINISM (detailed in the Subsection 3.3) to resolve any resulting nondeterminism. If this succeeds, the merging process begins again with a new list of potential merges, continuing until no more states can be merged. If RESOLVENONDETERMINISM fails, this indicates that our belief of the two states representing the same program state was false, as we were unable to merge their respective behaviours. We then successively attempt to merge lower scoring state pairs until either one is successful or we run out of possibilities, at which point inference terminates.

Algorithm 1 The top level inference process.

```
 1: function LEARN(l, scoringMetric)
 2:     return INFER(MAKEPTA(l), scoringMetric)
 3: function INFER(efsm, scoringMetric)
 4:     switch INFERENCESTEP(efsm, SCOREMERGES(efsm, scoringMetric)) do
 5:         case None
 6:             return efsm
 7:         case Some new
 8:             return INFER(new, scoringMetric)
 9: function INFERENCESTEP(e, merges)
10:     switch merges do
11:         case []
12:             return None
13:         case ((s1, s2)#t)
14:             e' = MERGESTATES(s1, s2, e)
15:             switch RESOLVENONDETERMINISM(NONDETPAIRS(e'), e, e') do
16:                 case Some new
17:                     return Some new
18:                 case None
19:                     return INFERENCESTEP(e, t)
```

3.3 Resolving Nondeterminism by Merging Transitions

Classical FSM inference merges duplicate behaviours into a single transition by merging their destination states. Since FSM transitions with the same origin state are only nondeterministic if their labels are equal, there is no need to explicitly merge transitions. This happens "for free" when we merge their destination states. The two transitions then have the same label, origin, and destination so they are no longer distinct. With EFSMs, transitions which express the same behaviour may not be identical. Thus the merging of transitions becomes an explicit step in the algorithm. There is also the possibility that two nondeterministic transitions cannot be merged, which does not occur in classical FSM inference. For example, in Figure 6b, if r_1 holds value *'coke'*, there is no observable difference between the behaviour of the two *vend* transitions and they can be merged. If r_1 holds any value other than *'coke'*, there is an observable difference in behaviour and the transitions cannot be merged.

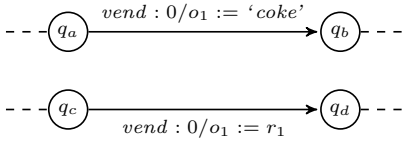
Algorithm 2 Resolving nondeterminism.

```
1: function resolveNondeterminism([], _, new)
2:     if deterministic(new) then
3:         return Some new
4:     else
5:         return None
6: function resolveNondeterminism(((from, (d1, d2), (t1, t2))#ss), old, new)
7:     destMerge ← mergeStates(d1, d2, new)
8:     switch mergeTransitions(old, destMerge, t1, t2) do
9:         case None
10:            resolveNondeterminism(ss, old, new)
11:        case Some merged
12:            newPairs ← nondetPairs(merged)
13:            switch resolveNondeterminism(newPairs, old, merged) do
14:                case Some new′
15:                    return Some new′
16:                case None
17:                    resolveNondeterminism(ss, old, new)
18: function mergeTransitions(old, destMerge, t1, t2)
19:     if directlySubsumes(old, destMerge, origin(t1, old), t2, t1) then
20:         return Some replaceTransition(destMerge, t1, t2)
21:     else if directlySubsumes(old, destMerge, origin(t2, old), t1, t2) then
22:         return Some replaceTransition(destMerge, t2, t1)
23:     else
24:         return None
25: function directlySubsumes(e1, e2, s1, s2, t2, t1)
26:     return (∀p.acceptsTrace(e1, p) ∧ getsUsTo(s1, e1, p) ⟹
                    acceptsTrace(e2, p) ∧ getsUsTo(s2, e2, p) ⟹
                    subsumes(t2, anteriorContext(e2, p), t1))
                ∧ (∃c.subsumes(t2, c, t1))
```

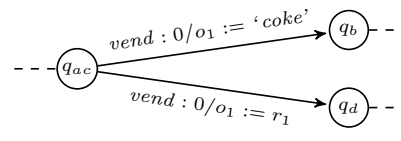
The resolveNondeterminism function takes a list of nondeterministic transition pairs and merges the destination states of the first pair. It then calls mergeTransitions to merge the transitions themselves. If this is successful, resolveNondeterminism recurses until all nondeterminism has been resolved. If the transition merge fails, nondeterminism might be resolved by merging a different transition pair. Successive attempts are made until either one is successful or there are no more potential merges. In the latter case, resolveNondeterminism fails, indicating that the original state pair should not have been merged.

When merging EFSM transitions, one must *account for* the behaviour of the other. This is conceptualised, for guarded transitions, as *subsumption* in [12] and extended to transitions with data updates in [6] which introduces *contexts* to record constraints on the values of inputs and registers during the execution of an EFSM, for example that a register holds a particular value. The idea of *subsumption in context* formalises the intuition that, in certain contexts, a transition t_2 reproduces the behaviour of t_1 and updates the data state in a manner consistent with t_1 meaning that t_2 can be used in place of t_1 with no observable difference in behaviour. For state s in an EFSM e, we say that a context c is *obtainable* if there exists a trace which is accepted by e, leaving it in state s, and produces c when executed.

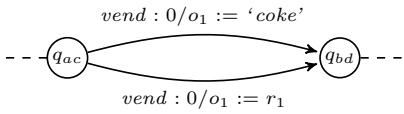
Example 2. Consider the EFSM fragments in Figure 6. Let us call transitions $q_a \rightarrow q_b$ and $q_c \rightarrow q_d$ in Figure 6a t_1 and t_2 respectively. Say that the inference process merges states q_a and q_c to form the model in Figure 6b. This results in nondeterminism between t_1 and t_2 which we would like to resolve.



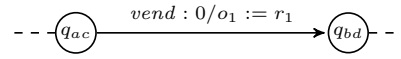
(a) Fragment of $\mathcal{M}_1$ before merging q_a and q_c.



(b) Fragment after merging q_a and q_c.



(c) Fragment after merging q_b and q_b to form $\mathcal{M}_2$.



(d) Fragment after merging the two transitions.

Fig. 6: The evolution of an EFSM fragment during the merging process.

We merged states q_a and q_c because we believe that their respective outgoing transitions express the same behaviour. This means that their respective desti-

nation states should represent the same program state, so we merge q_b with q_d to form $\mathcal{M}_2$, shown in Figure 6c. We then ask if one transition accounts for the behaviour of the other such that they can be merged. This means that in every situation where we could have taken t_1 in $\mathcal{M}_1$, we should now be able to take t_2 in $\mathcal{M}_2$ with no observable difference in behaviour, or vice versa. If r_1 holds value *'coke'*, then t_2 accounts for the behaviour of t_1. □

In Example 2, it is unlikely that r_1 will always hold the value *'coke'* in state q_{ac} but we only need t_2 to account for the behaviour of t_1 in situations where it could be taken in $\mathcal{M}_1$. This means that traces which got us to q_a in $\mathcal{M}_1$ must, when run in $\mathcal{M}_2$, produce contexts in which t_2 subsumes t_1, i.e. contexts in which $r_1 =$ *'coke'*. If this is the case, we say that t_2 *directly subsumes* t_1. This is not presented in [6] and is expressed as the first conjunct of the DIRECTLYSUBSUMES function in Algorithm 2. The second conjunct says that there must exist a context in which t_2 subsumes t_1, which accounts for models with unreachable states, from which any transition would otherwise directly subsume any other transition.

The MERGETRANSITIONS function can only merge transitions where one directly subsumes the other. If this is not the case, then neither can be used in place of the other without risking some observable difference in the behaviour of the model. In this case, MERGETRANSITIONS fails, returning **None**.

4 Introducing Registers

The technique in Section 3 allows us to infer deterministic EFSM models from traces by merging transitions where one subsumes the other, but we cannot yet fully capture the causal relationship between input and output. To achieve this, we must infer the use of *internal variables* which store information about the current state for later use. This section explains how we do this.

Example 3. The EFSM in Figure 7 is the best model of the traces in Figure 1 that our technique can infer so far. It is, essentially, an EFSM version of Figure 3. While this is a more *accurate* view of the system — transitions are now expressed as events with parameters rather than atomic actions — it is no more *expressive*.

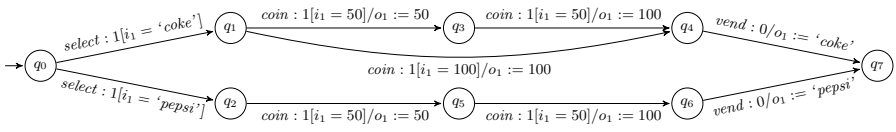


Fig. 7: An EFSM model inferred from the traces in Figure 1.

The model contains two pairs of identical *coin* transitions which we could merge by *zipping* the path $q_1 \to q_3 \to q_4 \to q_7$ with $q_2 \to q_5 \to q_6 \to q_7$ as discussed in Section 2. We cannot do this, though, as it requires the transitions

$vend : 0/o_1 := \text{'coke'}$ and $vend : 0/o_1 := \text{'pepsi'}$, which have different literal outputs, to be merged. Since there is always an observable difference in their behaviour, neither *vend* transition directly subsumes the other so they cannot be merged. This means we cannot condense Figure 7 any further.

Looking at the bigger picture, the two *vend* transitions do actually exhibit the same behaviour. Both produce, as output, the input of the initial *select* transition. If we could abstract away the concrete inputs and outputs, we could infer a smaller and more general model of the system. □

To this end, we allow the MERGETRANSITIONS function to attempt to introduce internal variables if neither transition directly subsumes the other. The aim here is not to create a "one size fits all" magic oracle, rather to provide a number of smaller heuristics, each of which focusses on a particular *data usage pattern.* We pass a list of heuristics to MERGETRANSITIONS as an additional argument, each of which either successfully returns an EFSM, or fails. If no direct subsumption occurs between two transitions, the heuristics are applied in the supplied order until either one of them succeeds or there are no more left to apply. This approach makes the tool extensible and gives users a degree of control over the characteristics of the final model as they can choose to provide or withhold particular heuristics. If neither transition directly subsumes the other and none of the heuristics are successful, the transition merge fails.

The fact that a heuristic successfully produces an EFSM does not guarantee the model to be acceptable. For example, the heuristic which always returns the empty EFSM resolves any nondeterminism (since a model with no transitions is trivially deterministic) but is clearly unacceptable. We must therefore be suspicious of solutions offered by heuristics if we want our inference process, as a whole, to always return an acceptable model of the original traces.

This leads to the question how to define whether or not a model is acceptable. Clearly a minimum requirement for models inferred from traces is that they reproduce all of the observed behaviour. Since the original set of traces is finite, we can simply run each one through the model and compare the output to the original. We run this sanity check after each state-merge to ensure that the model still reflects the observed behaviour. If this is not the case, the model is discarded as if the state merge had failed. The remainder of this section details some heuristics which are relevant to our running drinks machine example.

4.1 The Store and Reuse Heuristic

An obvious candidate for generalisation is the "store and reuse" pattern. This manifests itself in Example 3 when the input of *select* is subsequently used as the output of *vend*. Recognising this usage pattern allows us to introduce a *storage register* to abstract away concrete data values and replace two transitions whose outputs differ with a single transition that outputs the content of the register.

The first step is to find *intratrace* matches — instances of data reuse *within* traces. We walk each trace in the current EFSM, recording when the output of a transition matches the input of an earlier transition, to obtain a set of matches

for each trace in the form $\{((transition, inputIndex), (transition, outputIndex))\}$. We then look to see if any of the matches concern the transitions we are trying to merge. If so, we attempt to *generalise* these transitions. This consists of introducing a fresh register to act as storage, adding an update to this register, and dropping the restriction on the relevant input value. The value of this register then becomes the output of the second transition. For example, we generalise the pair $((select : 1[i_1 = \text{`}coke\text{'}], 1), (vend : 0/o_1 := \text{`}coke\text{'}, 1))$ to $((select : 1/[r_1 := i_1], 1), (vend : 0/o_1 := r_1, 1))$, where r_1 does not already occur in the EFSM.

When multiple transition pairs generalise to the same thing, *between* multiple traces, we call this an *intertrace* match. Finding intertrace matches indicates that the same kind of behaviour occurs across multiple traces, potentially with different data values. This provides evidence in favour of generalising and merging transitions in the model.

4.2 The Increment and Reset Heuristic

Another usage pattern is "increment and reset". In our drinks machine example, the *coin* action outputs the sum of the previous *coin* inputs. This allows customers to use multiple coins to pay for their drink and to observe the total value they have inserted so far. Correctly identifying this usage pattern is not an easy problem to solve, but a naive heuristic is not difficult to implement.

The idea here is that if we want to merge two transitions with identical input values and different numeric outputs, for example $coin : 1[i_1 = 50]/o_1 := 50$ and $coin : 1[i_1 = 50]/o_1 := 100$, then the behaviour must depend on the value of an internal variable. We implement a heuristic which, when faced with such a merge, drops the input guard and adds an update to a fresh register, in this case summing the current register value with the input. For this to work, we must ensure that the register is initialised before our modified transitions are taken. To do this, we augment transitions incident to the origin state with an update function which sets the relevant register to zero. This is the "reset" part of the heuristic which ensures that the register is defined before it is used. A similar principle can be applied to other numeric functions such as subtraction.

4.3 The Same Register Use Heuristic

Heuristics operate on a per-merge basis so it is possible that multiple registers may be introduced to serve the same purpose at different points during the inference process. It is therefore important to recognise this and consolidate register usage to allow transitions which implement the same behaviour with different registers to be merged.

Consider, for example, the transitions $coin : 1/o_1 := r_1 + i_1[r_1 := r_1 + i_1]$ and $coin : 1/o_1 := r_2 + i_1[r_2 := r_2 + i_1]$. Both transitions use a single register and are identical up to the name of this register so it is possible that r_1 and r_2 are just two different names for the same register. We therefore try to "merge" the two registers by renaming r_1 to r_2, or vice versa.

5 Implementation

The next task is to code up our technique into an executable program. Unfortunately, some parts of our technique, most notably the DIRECTLYSUBSUMES function, cannot be effectively computed. This section details how we tackled this to produce a prototype inference tool using Isabelle/HOL [14] (henceforth referred to as just "Isabelle"), a proof assistant and programming environment.

Isabelle allows datatypes and functions to be specified using a Haskell-style syntax, so we can use Isabelle to write programs and to prove that these programs satisfy certain properties. From previous work [6], we already had a formalisation of EFSMs in Isabelle with various proofs. We used this as a starting point for our implementation to avoid the duplication of work. A strength of using Isabelle for implementation is that functions can be expressed at a high level of abstraction, meaning that our Isabelle code is almost identical to the pseudocode in Algorithms 1 and 2.

Since Isabelle code is not directly executable, the built-in *code generator* [8] can be used to automatically convert Isabelle functions and datatypes to runnable code in a number of conventional programming languages. The code is not particularly well optimised but, assuming correctness of the code generator, properties which hold for the Isabelle formalisation also hold for the generated code. Once we had encoded our technique in Isabelle, we used the code generator to automatically create an executable Scala implementation. This, along with our formalisation, is available at https://github.com/aca13jmf/efsm-inference.

Of course, the code generator cannot generate code for non-computable functions like DIRECTLYSUBSUMES. This leaves us with gaps in our implementation which must be implemented manually. For these, the `code_printing` statement provides the ability to replace functions with custom implementations in the target language. Surprisingly, we were only faced with two problematic functions.

The first of these, NONDETPAIRS, provides details of nondeterministic transitions in an EFSM. For each state, it checks if there is a choice between any pair of outgoing transitions. This involves checking if the conjunction of their guards is satisfiable. We leveraged an existing SMT solver, Z3 [13], to do this for us by converting the guards to an appropriate format at runtime.

Coping with the non-executability of DIRECTLYSUBSUMES was more challenging. This function checks subsumption for all traces which get us to a particular state. The problem here is that there could be an infinite number of traces so we cannot use exhaustive search. Direct subsumption can be proven by induction over traces, on a case by case basis, but this is laborious. We cannot reasonably ask users to do this each time the inference process needs to know whether one transition directly subsumes another.

The solution to this lies in the fact that the inference process only encounters transitions from the original PTA and those introduced by the heuristics. If we can use Isabelle to prove direct subsumption for the various different *families* of transitions the inference process will come across, then the task of checking direct subsumption at runtime becomes a pattern matching exercise. For example, if we merge two states with a pair of identical outgoing transitions, we need to

check if a transition directly subsumes itself. Clearly every transition is able to account for its own behaviour, so it does not make sense to check this on a per-merge basis. We proved the *general case* in Isabelle so that at runtime we can simply check to see if the two transitions we are attempting to merge are equal. If they are, then we have direct subsumption. We applied this approach to the other patterns that occur when using the heuristics detailed in Section 4.

Different Literal Outputs. If two transitions have outputs which always differ, for example $vend : 0/[o_1 := \text{`}coke\text{'}]$ and $vend : 0/[o_1 := \text{`}pepsi\text{'}]$, then there is always an observable difference in behaviour. Along similar lines, transitions which produce different numbers of outputs are always distinguishable. In both of these cases neither transition directly subsumes the other.

Drop Guard Add Update. The "store and reuse" heuristic exchanges a concrete-value guard on an input for an assignment to a fresh storage register. For a pair of transitions, in which one has been generalised and the other has not, for example $select : 1/[r_1 := i_1]$ and $select : 1[i_1 := \text{`}coke\text{'}]$, if we can ascertain that the relevant register (in this case r_1) is undefined in the origin state, then the general transition directly subsumes the specific one.

Register Output. The "store and reuse" heuristic also replaces a literal output with the content of a register. For a generalised transition to subsume an ungeneralised one, it suffices to show that the relevant register holds the original output value in all relevant contexts which can be obtained in the origin state.

Increment and Reset. The pattern introduced by the "increment and reset" heuristic are more subtle. This heuristic drops a literal guard and introduces an update which *mutates* the datastate. We end up testing whether a transition of the form $coin : 1/o_1 := r_2 + i_1[r_2 := r_2 + i_1]$ subsumes a transition of the form $coin : 1[i_1 = n]/o_1 := m$. Neither transition can account for the behaviour of the other here as only one transition changes the data state. The updates are not *consistent* with each other. This means that the increment and reset heuristic only tends to be successful towards the end of the inference process when it is able to replace many transitions of the form $coin : 1[i_1 = n]/o_1 := m$ at once.

Having proved direct subsumption for the various transition families, our executable DIRECTLYSUBSUMES function simply steps through the cases until one matches. If none of the cases match, we have no choice but to ask the user but, for the heuristics detailed in this paper, this is not required. If additional heuristics were used that introduced new kinds of transitions to the model, further cases might be required to avoid queries to the user but, depending on the difficulty of the proofs, this would not be particularly arduous.

5.1 Checking Context Properties

In some of the patterns above, we require obtainable contexts to satisfy certain properties. Even though these are much simpler properties than subsumption, we still cannot exhaustively search all traces, nor can we expect a user to provide an inductive proof for each instance. Instead, we use SAL[3], a model checker with

[3] http://sal.csl.sri.com/

a similar representation to our own EFSM model. This allows us to automatically verify simple properties like "register r is always undefined in state s" in milliseconds. We do sacrifice some of the safety of an inductive proof, but doing so enables us to completely automate the process. Model checkers only work with finite datatypes, so we can only check a finite subset of all possible inputs. The larger this subset, the more confident we can be of the validity of a merge, but we must balance this with performance. If we are able to check traces over a suitable subset of inputs, then we can be reasonably confident that transition merges made as a result of this are safe.

6 Evaluation

When presented with the traces in Figure 1, our technique infers the machine in Figure 5 which we described as "ideal" in Section 2. There are many different metrics which could be used to assess this model including size and complexity, predictive power, observance of original behaviour, and correct classification of legal and illegal behaviours. This section provides evaluation and discussion of both the model and the inference process with reference to these metrics.

A common evaluation metric of classical FSM inference techniques [10,18] is the classification of legal and illegal behaviour. This is not suited to techniques that work only with observations of system behaviour which are, by definition, legal behaviours. It is unreasonable to evaluate such techniques with respect to illegal behaviour as examples of this are not available to the inference process.

The main aim of an automated inference is to create models that are easy to understand. This makes smaller models with fewer transitions more desirable. The model in Figure 5 is both small and simple as it has only three states and three transitions. The original PTA has ten of each. Our model is also smaller than the classical FSM in Figure 3 which has seven states and nine transitions.

Inferred models should, of course, exhibit all of the originally observed behaviour. This holds for our technique by definition since, at each stage of inference, the new machine is checked to ensure that it accepts all of the originally observed traces. The model in Figure 5 accepts all of the traces in Figure 1 and produces all of the originally observed outputs.

An important difference between inference and minimisation is that inference aims to generalise from the observed behaviour. The model we inferred exhibits the same top-level behaviour no matter what drink the user selects or what values of coins the user pays for their drink with. While this inevitably leads to models which *over generalise* the observed behaviour, it enables us to *predict* how the system might behave when faced with unseen inputs.

An advantage of our model over the one in Figure 4 is that our model is able to *compute* outputs from inputs. For any sequence of inputs to *coin*, we are able to predict the value of the output rather than simply placing constraints on it.

7 Conclusions and Future Works

This work presents a technique to infer EFSM models from black-box system traces. Building on [6] we have now shown how to infer computational EFSM models from traces by using heuristics which recognise data usage patterns. We defined *direct subsumption* and used it to help us merge transitions. We formalised our technique in Isabelle/HOL and exported it to executable Scala code using Isabelle's built-in code generator where possible.

Most modern inference techniques fit into two categories. Active techniques such as [1,5,9] make use of an oracle, usually the end-user, to guide the inference by classifying traces as either possible or impossible. Assuming the availability of such an oracle, active techniques produce good quality models but are quite labour intensive. By contrast, passive methods such as [2,10,18] sacrifice the oracle in favour of complete automation. These techniques infer models from traces of the system under inference so, unlike active methods, they often do not have access to examples of impossible system behaviour in the form of *negative traces* which the system, by definition, is unable to produce.

Classical FSM models use atomic transitions which cannot separate actions from data. They must encode data within the control flow, so struggle with systems that exhibit data-dependent behaviour. EFSM models feature parametrised inputs, guarded transitions, and a persistent data state so are much better suited to modelling data-dependent behaviour. Existing EFSM inference techniques [11,19] focus on inferring transition guards but do not infer models which capture the *causal* relationship between input and output. Attempts have been made to infer computational models [17], but these rely on white-box traces to expose the inner system state. Such traces are often unavailable so the inference of computational EFSM models from black-box traces is a key challenge in EFSM inference. This work presents such a technique.

Future work includes the implementation of further heuristics, such as one to recognise boundary conditions which separate behaviour. Additionally, the tool needs to be run on larger case studies to investigate how well it scales.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). http://doi.org/10.1016/0890-5401(87)90052-6
2. Biermann, A.W., Feldman, J.A.: On the Synthesis of Finite-State Machines from Samples of Their Behavior. IEEE Transactions on Computers **C-21**(6), 592–597 (1972). http://doi.org/10.1109/TC.1972.5009015
3. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: 30th ACM/IEEE Design Automation Conference. pp. 86–91. IEEE (1993). http://doi.org/10.1145/157485.164585
4. Damas, C., Lambeau, B., Dupont, P., Van Lamsweerde, A.: Generating annotated behavior models from end-user scenarios. IEEE Transactions on Software Engineering **31**(12), 1056–1073 (2005). http://doi.org/10.1109/TSE.2005.138

5. Dupont, P., Lambeau, B., Damas, C., Van Lamsweerde, A.: The QSM algorithm and its application to software behavior model induction. Applied Artificial Intelligence **22**(1-2), 77–115 (2008). http://doi.org/10.1080/08839510701853200
6. Foster, M., Taylor, R.G., Brucker, A.D., Derrick, J.: Formalising extended finite state machine transition merging. In: Sun, J., Sun, M. (eds.) Formal Methods and Software Engineering. pp. 373–387. Springer International Publishing, Cham (2018)
7. Fraser, G., Walkinshaw, N.: Behaviourally adequate software testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. pp. 300–309. IEEE (2012). http://doi.org/10.1109/ICST.2012.110
8. Haftmann, F., Bulwahn, L.: Code generation from isabelle/hol theories. Part of the Isabelle documentation (2013), http://isabelle.in.tum.de/dist/Isabelle2017/doc/codegen.pdf
9. Isberner, M., Howar, F., Steffen, B.: The ttt algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification. pp. 307–322. Springer International Publishing, Cham (2014). http://doi.org/10.1007/978-3-319-11164-3_26
10. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar, V., Slutzki, G. (eds.) Grammatical Inference: 4th International Colloquium, ICGI-98 Ames, Iowa, USA, July 12–14, 1998 Proceedings, pp. 1–12. Springer, Berlin, Heidelberg, Berlin, Heidelberg (1998). http://doi.org/10.1007/BFb0054059
11. Lorenzoli, D., Mariani, L., Pezzè, M.: Inferring state-based behavior models. In: Proceedings of the 2006 international workshop on Dynamic systems analysis - WODA '06. p. 25. ACM Press, New York, New York, USA (2006). http://doi.org/10.1145/1138912.1138919
12. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering. pp. 501–510. ICSE '08, ACM, New York, NY, USA (2008). http://doi.org/10.1145/1368088.1368157
13. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL, Lecture Notes in Computer Science, vol. 2283. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). http://doi.org/10.1007/3-540-45949-9, http://link.springer.com/10.1007/3-540-45949-9
15. Taylor, R., Hall, M., Bogdanov, K., Derrick, J.: Using behaviour inference to optimise regression test sets. In: Nielsen, B., Weise, C. (eds.) Testing Software and Systems. pp. 184–199. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). http://doi.org/10.1007/978-3-642-34691-0_14
16. Valdes, A., Skinner, K.: Adaptive, model-based monitoring for cyber attack detection. In: Debar, H., Mé, L., Wu, S.F. (eds.) Recent Advances in Intrusion Detection. pp. 80–93. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). http://doi.org/10.1007/3-540-39945-3_6
17. Walkinshaw, N., Hall, M.: Inferring Computational State Machine Models from Program Executions. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 122–132. IEEE (2016). http://doi.org/10.1109/ICSME.2016.74

18. Walkinshaw, N., Lambeau, B., Damas, C., Bogdanov, K., Dupont, P.: STAMINA: a competition to encourage the development and assessment of software model inference techniques. Empirical Software Engineering **18**(4), 791–824 (2013). http://doi.org/10.1007/s10664-012-9210-3
19. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empirical Software Engineering **21**(3), 811–853 (2016). http://doi.org/10.1007/s10664-015-9367-7