Formalising Extended Finite State Machine Transition Merging

Michael Foster, Ramsay G. Taylor, Achim D. Brucker, and John Derrick

Department of Computer Science, The University of Sheffield
Regent Court, Sheffield, S1 4DP, UK
{jmafoster1, a.brucker, r.g.taylor, j.derrick}@sheffield.ac.uk

Abstract. Model inference from system traces, e.g. for analysing legacy components or generating security tests for distributed components, is a common problem. *Extended Finite State Machine* (EFSM) models, managing an internal data state as a set of registers, are particularly well suited for capturing the behaviour of *stateful* components however existing inference techniques for (E)FSMs lack the ability to infer the internal state and its update functions.
In this paper, we present the underpinning formalism for an EFSM inference technique that involves the merging of transitions with updates to the internal data state. Our model is formalised in Isabelle/HOL, allowing for the machine-checked validation of transition merges and system properties.

Keywords: Model Inference · State Machine Models · EFSM

1 Introduction

Accurate behavioural models of software systems are very valuable for development and maintenance. They are particularly useful during the testing phase where they have acted as oracles for regression testing [9] and can be used to automatically generate tests [10]. Such models are also useful in requirements engineering [5], aiding the understanding of systems.

Despite their value, models are often neglected during development. It is therefore useful to *reverse engineer* them from existing systems. There is substantial work on reverse engineering Finite State Machine (FSM) models from observations of systems including [11,13,18,20]. Most modern inference approaches begin by building a Prefix Tree Acceptor (PTA) [18], a tree-shaped automaton accepting exactly the traces observed. States and transitions are then merged where they are thought to represent the same system component.

The models produced by classical FSM inference struggle with complex systems, especially those exhibiting behaviour dependant on an internal state. Extended Finite State Machine (EFSM) inference is a promising solution to this

J. Song Dong and J. Sun (Eds.): ICFEM 2018, LNCS, pp. 1–15, 2018.
© 2018 Springer-Verlag. This is the author's version of the work. It is posted at https://www.brucker.ch/bibliography/abstract/foster.ea-efsm-2018 by permission of Springer-Verlag for your personal use. BibTeX, Word, EndNote, RIS

problem. EFSM models extend traditional FSMs by providing control flow decisions based on input values as well as persistent data storage [17]. Current EFSM inference approaches [15,19] tend to focus on guard expressions – functions that make control flow decisions based on input or data-state values – but overlook how individual transitions mutate the data state. The inference of data update functions is a key technical challenge in EFSM inference but significantly complicates the merging process.

The primary contributions of this work are as follows:

1. A formal process by which EFSM transitions with update functions may be merged.
2. The introduction of *contexts*, a scheme by which constraints on data values may be traced through EFSMs.
3. The use of contexts to prove properties and equivalence of EFSM models.

The rest of the paper is structured as follows: After a brief motivating example, Section 2 fixes our definition of EFSMs. Our formalism for merging EFSM transitions with update functions is introduced in Section 3, as is the concept of contexts. Section 4 discusses transition subsumption and how it is used in the merging process. Section 5 shows how contexts may be used to analyse properties of EFSM models. Finally, Section 6 concludes the paper, discussing related and future work.

1.1 Motivating Example

The inference process starts with a black-box system and observes its behaviour when presented with different inputs. From these observations, a model can be produced which reflects the observed behaviour. For example, consider a simple vending machine whose traces are exemplified in Figure 1.

In the *actual system* the *select* operation takes one parameter: the desired drink. The *coin* operation allows the user to insert coins to pay for their drink. The output of each *coin* operation is the total amount inserted so far. Once the value reaches 100, the *vend* operation triggers the drink to be dispensed. Pressing *vend* when the total coinage inserted is less than 100 yields no output.

$$select(coke) \rightarrow coin(50)/[50] \rightarrow coin(50)/[100] \rightarrow vend()/[coke]$$
$$select(coke) \rightarrow coin(100)/[100] \rightarrow vend()/[coke]$$
$$select(pepsi) \rightarrow coin(50)/[50] \rightarrow vend() \rightarrow coin(50)/[100] \rightarrow vend()/[pepsi]$$

Fig. 1: Some observed traces of a drinks machine in which an event has the format *label(arguments)/[outputs]*[1].

FSM inference processes use traces to construct a candidate model. This is done by constructing an initial PTA and iteratively merging states and transitions until an FSM model of the system similar to the one in Figure 2 is obtained.

[1] The output component is omitted if none is produced.

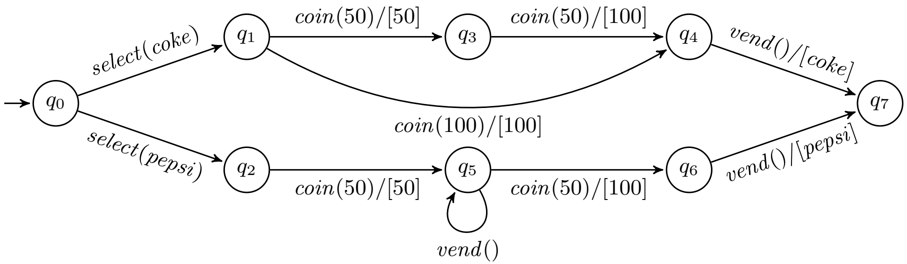


Fig. 2: A classical FSM model reflecting the traces in Figure 1

The problem with FSM inference: Classical FSM inference processes produce models of the system like the one in Figure 2. Note that transition labels are atomic so, for example, the *select*(*coke*) transition does not represent an event with label *select* and input *coke*, rather the transition is labelled by the literal string "select(coke)" making it a completely separate entity from the transition "select(pepsi)". This is a major problem as it means that information such as the selected drink and accrued funds must be encoded as part of the control state. Increasing product choice or the coins accepted quickly causes an explosion in model size disproportionate to the change in observable behaviour.

EFSM inference: The FSM in Figure 2 looks promising but is flawed because of its atomic labels. It is preferable to generate an EFSM model such as the one in Figure 3. Here, the selected drink is stored in a register r_1 for later use in the output of the *vend* transition. A second register r_2 (initialised with 0 by the select transaction) keeps track of the money inserted so far. Drinks are only dispensed once this value reaches 100. This enables customers to pay for their drink with any coin in any order. This is a much more concise and faithful model of the real system.

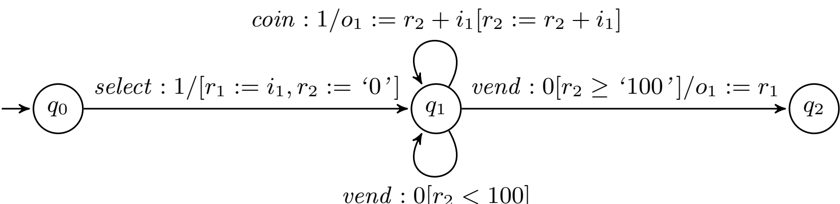


Fig. 3: An EFSM model of the simple vending machine in which transitions have the general form *label* : *arity*[*guards*]/*outputs*[*updates*][2]

[2] Where a particular transition lacks guards, outputs, or updates, the relevant components are omitted.

State of the art: EFSM inference techniques have been developed to produce models with parameterised guarded inputs and a separate data state. Notable works include the GK-tails algorithm [15] and the MINT algorithm [19].

GK-tails builds on top of the well established k-tails [2] algorithm. Each transition is annotated with a set of variable values at the current point of execution. When transitions are merged, their sets of variable values are also merged. The algorithm uses Daikon [9] to infer properties of variables which are used to ascertain whether a pair of states is compatible for merging.

The MINT approach [19] also has strong foundations but uses classifiers to determine, based on current data values, the labels of subsequent events. A key difference to GK-tails is that here data values are globally accessible so the classifiers have more data to work with. Classifiers are used not only to determine the validity of transition merges, but to detect and resolve nondeterminism.

While these techniques are valuable contributions and perform well for certain tasks, both fall short in that they fail to capture *how* data values are changed by individual transitions and are therefore unable to generate the EFSM in Figure 3. Including data update functions as part of each transition significantly complicates the process of transition merging. This work presents a method of comparing two transitions to assess their compatibility for merging. The actual inference process is the intended subject of future work.

2 Extended Finite State Machines

To define our method, we first need to fix the format of our EFSM model. Various EFSM models are presented in the literature [4,14] as well as similar ideas under different names [3,8]. Since the aim is to automatically infer data update functions, our model affords them a more detailed treatment, combining desirable aspects of various existing models. As with classical FSM models, EFSMs are usually presented graphically like in Figure 3.

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.

A little syntactic sugar allows an EFSM transition from anterior state S_m to posterior state S_n to take the general 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* which is the name of the event. This is followed by a colon and the *arity* of the transition, a natural number indicating the number of input parameters taken. Guard expressions g_1 to g_n are enclosed in square brackets. Next comes a slash, after which expressions f_1 to f_n define the outputs. Finally, update expressions u_1 to u_n, enclosed in square brackets, define the posterior data state. There should be at most one update function per register per transition in order to maintain consistency. For transitions without guards, outputs, or updates, the corresponding components are omitted.

Guard expressions take the current data state and a tuple of inputs and are *satisfied* if the specified conditions are met. If this is the case, the EFSM is said to have *accepted* the input. A transition cannot be taken if its guard is not satisfied. Guards operate over literals, inputs, and registers, the latter two collectively being referred to as "variables". Literals are enclosed in single quotes in order to distinguish them from variable names. Numeric values are assumed to be parsed automatically when required. Absence of a guard corresponds to the literal guard *true* which accepts any input with any data state.

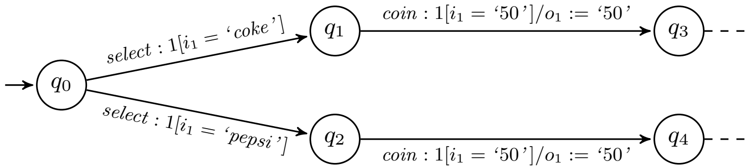
Functions to compute the outputs and updates use expressions over literals and variables evaluated from the *anterior data state*. Assignment syntax _ := _ is used to identify the value being computed. As with guards, literal values are enclosed in single quotes and numeric values are parsed automatically. Registers not explicitly updated by a transition remain unchanged and are initially undefined, so cannot be used before they have been assigned.

3 A Formalism for Merging EFSM Transitions

During the inference process, a PTA is generated containing fragments such as in Figure 4a. States with similar outgoing transitions are then merged to create a more concise model. This will likely introduce nondeterminism to the model, which can be resolved by merging the destination states of offending transitions and then the transitions themselves, arriving at something like the fragment shown in Figure 4b. This section describes a method for merging EFSM transitions, introducing the concept of *contexts* as a record of constraints on the possible values of variables and expressions.

3.1 Method Overview

Our method uses the idea of subsumption (adapted from [15]) together with *contexts*, our scheme for recording constraints on the data state of EFSMs. The method of merging transitions with identical origin and destination states can be



(a) A fragment of the PTA built from the traces in Figure 1

q_0 — $select : 1/[r_1 := i_1]$ → $q_{\{1,2\}}$ — $coin : 1[i_1 = \text{‘50’}]/o_1 := \text{‘50’}$ → $q_{\{3,4\}}$

(b) After merging state q_1 with q_2. The resulting nondeterminism is resolved by merging q_3 with q_4 and then merging the *coin* transitions.

Fig. 4: An EFSM model fragment before and after merging states and transitions

roughly described as follows. Firstly, the transitions must have the same label and arity, otherwise they represent different behaviours and cannot be merged. Next, the guard of one transition should be implied by that of the other. In this way, one transition accepts a subset of inputs of the other. In cases where both transitions may be taken, their output should be identical otherwise there is an observable difference between the transitions and they cannot be merged. Additionally, the data updates performed by the two transitions should be consistent with each other such that the output of subsequent transitions is not affected by the merge. If these conditions are met, the transition with the more specific guard is said to be an *instance* of the one with the more general guard. It can therefore be trivially deleted without affecting the observable behaviour of the model.

To implement this method, we need to define a way to determine when transitions are merged. Subsumption allows us to do this but it is necessary to introduce the idea of contexts to relate the internal data state of the system to observable output values of transitions.

3.2 Contexts

One may be tempted to use observational equivalence when merging transitions since two transitions exhibiting the same observable behaviour can be thought of as equivalent. Two transitions are *observationally equivalent* if, when presented with the same input, they produce the same output. The transitions may make different updates to the data state but register values are not directly observable so the difference is hidden. Since registers may be used as part of the output of subsequent transitions, the use of observational equivalence on a per-transition basis is likely to cause an observable difference at a later point in model execution. While observational equivalence must certainly be maintained, it is not a strong enough criterion for transition merging.

A stronger test would be *trace equivalence.* Trace equivalence extends observational equivalence to a sequence of inputs. If two EFSMs produce identical

output sequences for all given sequences of inputs, they are trace equivalent. Since more than one transition is considered, differences in update functions may manifest themselves if affected register values are used in subsequent output. The problem is that trace equivalence is only in terms of concrete traces so is not conducive to the *generalisation* of transitions.

What needs to be used is *contextual equivalence*. This relates possible register values to observable output and is a generalisation of trace equivalence. Consider the transitions $vend : 0/o_1 := \text{'coke'}$ and $vend : 0/o_1 := r_1$. At first glance, the two transitions look quite different, but there is a circumstance where they are observationally equivalent: when r_1 holds the value *'coke'*. In this *context*, there is no observable difference between the two transitions.

In our EFSM inference method, transitions have three contexts during their evaluation. The exact values of registers may not always be known but guard and update expressions allow certain constraints to be inferred. If a transition is taken then its guard must have been satisfied. A transition with guard $i_1 = 50$ may only be taken when i_1 holds the value fifty. If an update expression then assigns the value of i_1 to a register, it is now known that the value of that register is fifty. Similarly, if the value of a register is known to be greater than five before a transition is taken and an update function increments it by five then it is now known that the value of that register must be greater than ten.

Definition 2. *A* context *is a mapping from expressions in terms of inputs and registers to constraints on their values.*

The exact typing is dependent on the types of the inputs and registers. For integers, a context is a mapping from operations on integers (addition, multiplication, etc.) to constraints such as less than, greater than, and equality. When working with lists, contexts map operations such as concatenation, length, and folding to appropriate constraints.

Contexts are written as maps enclosed in double square brackets and use "curried" notation to record constraints on the values of expressions. For example $[\![r_1 + i_1 \mapsto = 6]\!]$ represents the context where the value of $r_1 + i_1$ is equal to six. The key is "$r_1 + i_1$" and the constraint is "$= 6$". This corresponds to the guard $r_1 + i_1 = 6$. Most constraints can be viewed as guards, the exceptions being the literals *true* and *false* which represent unrestrictedness and inconsistency respectively.

Uninitialised register values map to a special "undefined" constraint. This constraint is not satisfiable as it is impossible to access a register without a value, but cannot lead to inconsistent reasoning, since unassigned registers cannot be used in computation. This is not the same as having an explicitly unrestricted register which has been assigned a value about which nothing is known. Inputs which are not explicitly constrained map to literal *true* since nothing about their value is known but if presented, they are known to have a value.

A transition t will have three contexts during its evaluation. The *anterior context*, $A(t)$, is the set of constraints which is known before the transition is taken. This context contains only expressions concerned with registers since the

transition has not yet received any input. The *medial context*, $M(t)$, is the set of constraints immediately after the guard has been applied. This includes constraints on input values as these are currently in scope. The *posterior context*, $P(t)$, is the set of constraints after the update function has been executed. This does not include constraints on inputs since they do not persist. This context forms the anterior context for the next transition. In this way, contexts flow through an EFSM tracing constraints on register values.

Constraints do not relate variables. The reason for this is best illustrated with an example. If r_1 is known to be greater than i_1 for a particular transition, but nothing is known about the value of i_1 then nothing meaningful is known about r_1 either. To say that it is greater than an unknown value is meaningless since there is always a possible valuation of the pair such that the property holds. Only when something more concrete about one of the two variables is known can this constraint be of use.

3.3 Computing Contexts

Algorithm 1 describes how to compute the posterior context of a transition. Line 2 applies the guards of the transition to the anterior context to form the medial context. If the medial context is consistent, the update functions can be applied, looking up constraints from the medial context as necessary. Any constraints on expressions involving input values are then removed. To say that a context is consistent is to say that all of the constraints are satisfiable simultaneously. An inconsistent medial context means that the guard was not satisfied and the transition cannot be taken with the given anterior context.

Algorithm 1 Computing the posterior context

```
1: function POSTERIOR(Transition t, AnteriorContext c)
2:     c' ← MEDIAL(c, t.guards)
3:     if CONSISTENT(c') then
4:         return APPLYUPDATES(c', t.updates)
5:     else
6:         return FALSE
7:     end if
8: end function
```

A key step when building the medial context is the rearrangement of expressions. If a guard states that r_1 must be greater than i_1 then it is also the case that i_1 is less than r_1. Both must be added to the context at this stage as the constraint affects both variables. Similarly, if it is known that $r_1 = i_1 + r_2$ then it is also known that $i_1 = r_1 - r_2$ and that $r_2 = r_1 - i_1$. If constraints on two of the variables are known then constraints on the third can be calculated.

4 Subsumption and Generalisation

In this section we use contexts to solve the problem of merging transitions in EFSMs. The inference process begins by observing the outputs of a system when presented with particular inputs. An initial PTA is constructed to reflect this behaviour and states are iteratively merged to create a more concise and general model of the system.

States with similar outgoing transitions, such as q_1 and q_2 in Figure 4a, are good candidates for merging but the resulting model is often nondeterministic. This can be resolved by merging subsequent states and transitions, however this requires some notion of transition equality and generalisation. The idea of subsumption presented in [15] deals nicely with guards but does not consider data update functions. With the help of a running example, this section uses contexts to extend the idea of subsumption to output and update functions, ensuring that observational equivalence is maintained when transitions are merged.

Observe the EFSM in Figure 5 and note transitions $q_1 \to q_2$ and $q_2 \to q_2$ labelled with *coin* which will be referred to as c_1 and c_2 respectively. The merging process now merges state q_1 with state q_2 into a new state, $q_{\{1,2\}}$, which introduces nondeterminism to the model since there are two outgoing *coin* transitions, c_1 and c_2, from $q_{\{1,2\}}$ either of which may be taken when i_1 is 50. This can be resolved by merging the two transitions into one.

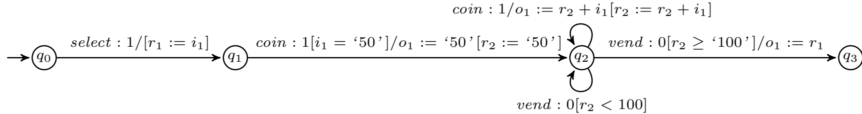


Fig. 5: An EFSM with a transition to be merged

Transitions may not always be compatible for merging so how exactly do we know if two transitions *are* compatible? Firstly they must have the same label and arity, otherwise they represent different behaviour. Lorenzoli *et al.* [15] discuss the idea of subsumption of guards, where one transition *subsumes* another if its guard is *more general*. Applying this principle to the example has the transition with no guard (corresponding to the literal guard *true*) subsuming the transition with guard $i_1 = \text{‘50’}$. This looks promising but the outputs and updates need to be considered too. The principle of subsumption must therefore be extended to take these into account.

Definition 3. *Transition t_2 can be said to* subsume *transition t_1 if*

1. *The guard of t_1 implies that of t_2*
2. *In the cases where it is possible to take t_1, the output of t_2 is identical*
3. *The posterior data state of t_2 is consistent with that of t_1*

The general idea is similar to refinement [6], the aim being to widen the precondition and reduce nondeterminism. Subsuming transitions are allowed to accept *more* inputs but under circumstances where either transition may be taken, it is important that the output of both is identical. If this is not the case, the two transitions are observably different and cannot be merged. Even though data registers are not directly observable – it is not possible to ask "what is the value of register r?" – they may be used as part of output functions so have the potential to affect observable behaviour of future transitions. It is therefore important that any register updates performed by the subsuming transition are consistent with those performed by the one being subsumed.

Contexts are used in our inference method when determining if one transition subsumes another because they help to place restrictions on the values of expressions. Algorithm 2 describes the conditions which must be satisfied for transition t_2 to subsume t_1. Line 2 checks to see if each condition in the medial context of t_2 is implied by that of t_1. In other words, that each condition in $M(t_2)$ is more general than its counterpart in $M(t_1)$. The second conjunct, on line 3, ensures that the output of both transitions is equal in every case where it is possible to take t_1. This is the check for observational equivalence. The conjunct on line 4 ensures that the posterior context of t_2 is more specific than that of t_1 in the cases where t_1 may be taken. This means that the restrictions on the values of expressions in $P(t_2)$ are at least as tight as those in $P(t_1)$. The final conjunct, on line 5, enforces that the posterior context of the subsuming transition is consistent whenever that of the subsumed transition is. This ensures that subsuming transitions don't perform spurious updates involving previously uninitialised registers.

Algorithm 2 Transition subsumption in context

1: **function** SUBSUMES(Transition t_2, t_1, AnteriorContext c)
2: **return** $\forall x.\ \textsc{medial}(t_1, c)[x] \implies \textsc{medial}(t_2, c)[x] \land$
3: $\quad\quad \forall i\ r.\ \textsc{canTake}(t_1, i, r) \implies \textsc{outputs}(t_1, i, r) = \textsc{outputs}(t_2, i, r) \land$
4: $\quad\quad \forall x.\ \textsc{posterior}(t_2, \textsc{medial}(t_1, c))[x] \implies \textsc{posterior}(t_1, c)[x] \land$
5: $\quad\quad \textsc{consistent}(\textsc{posterior}(t_1, c)) \implies \textsc{consistent}(\textsc{posterior}(t_2, c))$
6: **end function**

The concept of subsumption is used in our method when merging transitions since, in a given pair of nondeterministic transitions, one will often subsume the other. Algorithm 3 describes the process in detail. Lines 1 - 4 describe the simplest merging case. If one transition subsumes the other directly, the subsumed transition can be trivially deleted without causing a contextual difference.

Lines 5 - 10 describe the case where one transition subsumes the other in a different context, for example if a register held a particular value. The OBTAINANTERIOR function tries to modify update functions of incoming transitions to accommodate this. Usually this involves assigning a value to a previously

Algorithm 3 Merging two transitions

Input: Transition t_1, t_2, AnteriorContext c, EFSM e, State s
1: **if** SUBSUMES(t_1, t_2, c) **then**
2: t_2 can simply be deleted, leaving t_1 as the result of the merge
3: **else if** SUBSUMES(t_2, t_1, c) **then**
4: t_1 can be deleted, leaving t_2 as the result of the merge
5: **else if** $\exists c'$. SUBSUMES$(t_2, t_1, c') \vee$ SUBSUMES(t_1, t_2, c') **then**
6: **if** OBTAINANTERIOR(c', e, s) **then**
7: Delete either t_1 or t_2 as appropriate
8: **else**
9: The transitions cannot be merged
10: **end if**
11: **else**
12: The transitions cannot be merged
13: **end if**

undefined register. If this is achieved, one transition then subsumes the other and we are back to the simple case.

If it is not possible to modify update functions of incoming transitions, for example if a relevant register value is already set by an update function, then no subsumption exists and the merge fails. This is also the case if no anterior context exists in which one transition may subsume the other directly.

Example 1. Let us now apply our method to the running drinks machine example from Figure 5 and carry out the process with the nondeterministic *coin* transitions from state $q_{\{1,2\}}$. Intuitively, c_2 should subsume c_1 because it has no guard. Running Algorithm 3 should verify this.

The anterior context of both transitions is $[\![r_1 \mapsto \mathit{true}]\!]$ since the only way to reach state $q_{\{1,2\}}$ from the initial state is to take the *select* transition which assigns a value to r_1 but places no restriction on it. The guard of c_1 gives $M(c_1) = [\![r_1 \mapsto \mathit{true}, i_1 \mapsto = 50]\!]$. Since there is only one guard expression which restricts a single variable, i_1, to a literal value, there is no rearranging step here.

The medial context of c_2 is equal to the anterior context since c_2 has no guard. There is no explicit restriction on i_1 in this context so, as discussed in Section 3, it's constraint is literal *true*. In this case, since $\mathit{true} \implies \mathit{true}$ and $= 50 \implies \mathit{true}$ condition one of Algorithm 2 has been met.

Now to investigate condition two. The only case where it is possible to take c_1 is when $i_1 = 50$. In this case the output of c_1 is literal 50. Transition c_2 cannot produce an output since r_2 has not been initialised, hence condition two fails. This means that c_2 does not subsume c_1 directly.

If, in the anterior context, r_2 was equal to zero then the outputs of the two transitions would be identical in the case where $i_1 = 50$. The posterior contexts of the two transitions would also be identical and consistent, satisfying conditions three and four of Algorithm 2. In this case, the addition of the update $r_2 :=$ '0' to the *select* transition produces the desired anterior context, allowing transition c_2 to subsume c_1. This may take place without breaking contextual equivalence

since r_2 was previously undefined in the posterior context of *select*. With the new anterior context, c_2 subsumes c_1 directly meaning that c_1 may be trivially deleted, resulting in the EFSM in Figure 3. □

When faced with two transitions to merge, it may be the case that neither subsumes the other directly but there exists a transition which subsumes both. Consider the transitions $coin : 1[i_1 = \text{'20'}]/o_1 := \text{'20'}[r_2 := \text{'20'}]$ and c_1. Clearly they are instances of the same behaviour but neither subsumes the other. If presented with a candidate for a subsuming transition, contexts may be applied in the same way to establish the validity of the candidate. How such candidates are obtained is outside the scope of this paper and is the intended subject of future work but the method presented here can be used to validate such candidates.

5 Analysing System Properties

Another benefit of introducing contexts is the following. Having created an EFSM model of a system, it is possible to use it to prove properties of that model. With the drinks machine example, it is desired that a user will always receive the drink they originally selected. Another desirable property, for the proprietors at least, is that customers only receive their drinks if they have inserted enough money. Contexts allow us to prove properties like these.

Example 2. Consider the drinks machine model in Figure 3. Looking only at the labels, as would be provided by a classical FSM model, it appears to be possible to go straight from q_1 to q_2 without inserting any coins. The trace $select(coke) \rightarrow vend()/[coke]$ seems like a valid option, meaning that a user could get their drink for free. Contexts help to show that this is not the case.

The *vend* transition can only be triggered from state q_1. The only way to reach this state from the initial state is to do a *select* transition. This transition produces a posterior context of $[\![r_1 \mapsto true, r_2 \mapsto = 0]\!]$. Triggering *vend* with this anterior context will only allow the one which dispenses nothing to fire, since r_2 holds value zero which is less than 100. The only way to obtain a drink from *vend* is if r_2 holds a value greater than or equal to 100. The only transition from q_1 with an update function which increases r_2 is the *coin* transition. This means that the customer must insert at least one coin to receive their drink. □

The exact proof strategy varies depending on the property being proven but the general idea is to use the constraints of a particular context to prove that a transition may or may not be taken. In the case of Example 2, the guard of the *vend* transition with the desired output cannot be satisfied with an anterior context in which the value of r_2 is less than 100.

Another technique is to analyse update functions to see if any have the potential to affect variables of interest in the desired way. In Example 2, the variable of interest is r_2 and needs to be increased. The *coin* transition has no guard so may be taken with any anterior context and produces a posterior context with

r_2 incremented by the value of the input. Assuming that coins have a positive value, this increases the value of r_2. The destination state is equal to the origin, so the transition may be taken again if the input value was insufficient.

Contexts can also help prove observable equivalence of EFSM models. Consider the EFSM shown in Figure 6, an alternative model of the drinks machine in Figure 3. Contexts can be used to prove equivalence of the two models.

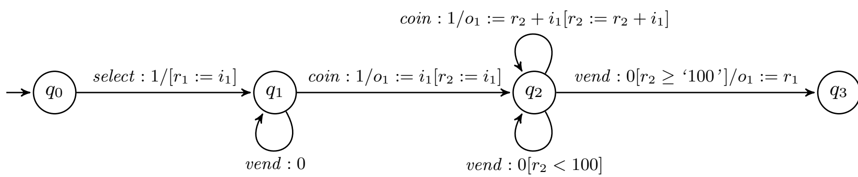


Fig. 6: A model which is observationally equivalent to the one in Figure 3

The idea here is similar to bisimulation with the aim being to form a relation between the states of two machines such that for all inputs, if one machine in a given state can accept an input, the other machine accepts the same input and produces the same output. The models must not only be trace equivalent but also contextually equivalent since register values may be used as part of output functions, potentially exposing differences in the data state. The model in Figure 6, M_1, can be proven to be contextually equivalent to the model in Figure 3, M_2, as follows.

Example 3. Starting both machines off in their respective initial states, it is only possible to do a *select* transition. Both machines are now in their respective q_1 states from which it is possible to do a *coin* or a *vend* transition. The context of M_1 at this point is $[\![r_1 \mapsto \mathit{true}]\!]$ and the context of M_2 is $[\![r_1 \mapsto \mathit{true}, r_2 \mapsto = 0]\!]$. Both machines can do an unguarded *coin* transition to produce the context $[\![r_1 \mapsto \mathit{true}, r_2 \mapsto \mathit{true}]\!]$. Both may do a *vend* transition which outputs nothing and leaves the context and state unchanged. M_1 also has a second outgoing *vend* transition but this may not be taken as r_2 is less than 100.

After having done a *coin* transition, M_1 is in state q_2 and M_2 is in state q_1. Subsequent *coin* transitions leave the state and context unchanged but allow a choice of either *vend* transition since nothing is known about the value of r_2. The guards on the two transitions are mutually exclusive so determinism is maintained. If r_2 is greater than or equal to 100 then adding further coins is futile but continues to be observationally equivalent. Alternatively, the *vend* transition which outputs the selected drink may be taken. This is the value of r_1 in both machines, set as the input of the *select* transition and not changed so identical inputs produce identical outputs. If r_2 is less than 100 then the *vend* transition in both cases leaves the state unchanged but produces a posterior context of $[\![r_1 \mapsto \mathit{true}, r_2 \mapsto < 100]\!]$. Subsequently, the same *vend* transition may be repeated indefinitely or another *coin* transition may be taken. □

These are just some of the ways context can be used to prove properties of systems. A full methodological breakdown is left for future work.

6 Conclusions

This paper presents contexts, a way of recording constraints on data values at different points during the execution of an EFSM model. The concept of subsumption is extended to EFSM transitions which include data update functions and is used as part of a technique to merge EFSM transitions. Contexts also aid in proving certain properties of EFSM models, notably equivalence of models. Algorithms 1 and 2 have been formalised in Isabelle/HOL [16] and together with representations of EFSMs and contexts have been used to validate possible transition merges and prove the properties of the drinks machine example discussed in Section 5. It is the intention of the authors to submit these theory files to the AFP (https://www.isa-afp.org/).

The task of inferring a model from a set of software execution traces has been an active area of research since the 1960s [11]. Most inference algorithms fit into one of two categories: active and passive. Active techniques such as [1,7,12] allow the user to guide the inference process by categorising possible actions as possible or impossible from the current state. Most modern techniques (including the one presented in this work) tend to be more passive, inferring a generalised system model from observed system traces without reference to the user.

Classical FSM inference techniques produce models with atomic labels which struggle with systems exhibiting value-dependent behaviour. EFSM models feature parametrised inputs and a separate data state which solves this problem. EFSM inference techniques such as [14,19] build on classical techniques to infer EFSMs from program execution traces by state and transition merging. These approaches do not attempt to infer register update functions so do not have to consider the merging of transitions which feature them. The inference of register update functions is a key challenge in EFSM inference, so a technique to merge such transitions is required. This work presents such a technique.

Future work includes the identification and prioritisation of potential EFSM state and transition merges as well as the provision of candidate transitions as discussed in Section 4. The inference of register and input types from traces is also an area of interest.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**, 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 (Jun 1972). http://doi.org/10.1109/TC.1972.5009015
3. Börger, E., Stärk, R.: Abstract State Machines. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). http://doi.org/10.1007/978-3-642-18216-7

4. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: International Design Automation Conference (DAC). pp. 86–91. ACM Press, New York, New York, USA (1993). http://doi.org/10.1145/157485.164585
5. 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 (Dec 2005). http://doi.org/10.1109/TSE.2005.138
6. Derrick, J., Boiten, E.A.: Refinement in Z and Object-Z. Springer-Verlag, 2nd edn. (2014). http://doi.org/10.1007/978-1-4471-0257-1
7. 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 (Feb 2008). http://doi.org/10.1080/08839510701853200
8. Eilenberg, S.: Automata, Languages, and Machines. Academic Press, Inc., Orlando, FL, USA (1974)
9. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering **27**(2), 99–123 (2001). http://doi.org/10.1109/32.908957
10. Fraser, G., Walkinshaw, N.: Behaviourally adequate software testing. In: International Conference on Software Testing, Verification and Validation. pp. 300–309 (Apr 2012). http://doi.org/10.1109/ICST.2012.110
11. Gold, E.M.: Language identification in the limit. Information and Control **10**(5), 447–474 (1967)
12. 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)
13. 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, pp. 1–12. Springer, Berlin, Heidelberg, Berlin, Heidelberg (1998). http://doi.org/10.1007/BFb0054059
14. Lorenzoli, D., Mariani, L., Pezzè, M.: Inferring state-based behavior models. In: International Workshop on Dynamic Systems Analysis (WODA). p. 25. ACM Press, New York, New York, USA (2006). http://doi.org/10.1145/1138912.1138919
15. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: International Conference on Software Engineering (ICSE). p. 501. ACM Press, New York, New York, USA (2008). http://doi.org/10.1145/1368088.1368157
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic, vol. 2283. Springer-Verlag, Berlin, Heidelberg (2002). http://doi.org/10.1007/3-540-45949-9
17. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. IEEE Transactions on Software Engineering **30**(1), 29–42 (Jan 2004). http://doi.org/10.1109/TSE.2004.1265734
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 (Aug 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 (Jun 2016). http://doi.org/10.1007/s10664-015-9367-7
20. Weyuker, E.J.: Assessing Test Data Adequacy through Program Inference. ACM Transactions on Programming Languages and Systems **5**(4), 641–655 (1983)