Active Inference of EFSMs Without Reset[⋆]

Michael Foster[1][0000−0001−8233−9873], Roland Groz[2], Catherine Oriat[2], Adenilso Simao[3], Germán Vega[2], and Neil Walkinshaw[1]

[1] Department of Computer Science, The University of Sheffield, UK
{m.foster, n.walkinshaw}@sheffield.ac.uk
[2] Univ. Grenoble Alpes, Laboratoire d'Informatique de Grenoble (LIG), France
{roland.groz, catherine.oriat, german.vega}@univ-grenoble-alpes.fr
[3] ICMC, São Carlos, Brazil
adenilso@icmc.usp.br

Abstract. Extended finite state machines (EFSMs) model stateful systems with internal data variables, and have many software engineering applications, including system analysis and test case generation. Where such models are not available, it is desirable to reverse engineer them by observing system behaviour, but existing approaches are either limited to classical FSM models with no internal data state, or implicitly require the ability to reset the system under inference, which may not always be possible. In this paper, we present an extension to the hW-inference algorithm that can infer EFSM models, complete with guards and internal data update functions, from systems without a reliable reset, although there are currently some restrictions on the type of system and model.

1 Introduction

Accurate models of software behaviour are useful for a wide range of software engineering tasks, including checking system correctness [12], identifying sequences of test inputs [7], and comparing differences in behaviour between software versions [8]. Reactive systems — systems that respond to their environment, their users, or other systems — are commonly modelled as (Extended) Finite State Machines ((E)FSMs), and such models form the basis of many testing and verification techniques [16].

⋆ The authors acknowledge the support of ANR project PHILAE (ANR-18-CE25-0013) and ACHAR project from LIG. Michael Foster and Neil Walkinshaw are funded by the EPSRC CITCoM project. Adenilso Simao would like to thank the CEPID-CeMEAI/ICMC-USP (FAPESP grant 2013/07375-0).

Despite their value, models can be neglected during development, or may not exist at all. In such situations, we need to *reverse engineer* them from existing systems, and the task of inferring (E)FSM models has been the subject of a considerable amount of research. A popular strategy here is the minimally adequate teacher framework [3], in which a model is inferred by posing a series of *queries* to the system under inference (aka SUL, System Under Learning). However, existing inference techniques [14, 15, 23, 11] tend to implicitly require the ability to return the system to some known "initial" state from which to execute a trace, which is not always feasible. While inference approaches have been developed to minimise resets [20, 13], these do not give adequate consideration to how data values affect the behaviour of the SUL.

In this paper, we present the ehW-inference algorithm (the e standing for "extended") to infer EFSM models of systems with internal, data-dependent behaviour without the need for resets. Our main contributions are as follows:

- An extension to the hW-inference algorithm [13] called ehW-inference, which incorporates the ability to infer internal registers and the constraints and functions that determine how the data states within the system change in response to inputs.
- A "proof of concept" demonstration of ehW-inference being applied to a small example system.

The rest of the paper is structured as follows. Section 2 presents a motivating example and a brief overview of the relevant background upon which our contribution is based. Section 3 presents our ehW-inference algorithm. Section 4 provides a walk-through of the algorithm, showing how it can be applied to our motivating example from Section 2. Finally, Section 5 concludes the paper and discusses potential future work.

2 Background and related work

We first introduce a running example to illustrate the type of model we use and our inference method. We compare our model and approach with existing work. We then formally define EFSMs and discuss the semantics of the model and the operations that can be applied.

2.1 Running example

To illustrate our approach, we use a vending machine, modelled on Figure 1a. Starting from state s_0, a user can *select* a drink (e.g., tea or coffee), then insert a coin. The price of a drink is 100 (there are coins of values 20, 50, 100, and 200). The machine will reject any initial payment less than the value of the drink, but a user may choose to enter more coins. Every time a coin is accepted, the running total is displayed. After paying, the user can press a *vend* button to be served the selected drink, and the balance in excess of the cost of a drink will be reset. An example execution is shown in Figure 1b. The formal semantics of this are detailed in Section 2.3.

$$\begin{tikzcd}[column sep=huge]
{} \arrow[r] & s_0 \arrow[r, "{\mathit{select}(i_0)/\epsilon\ [r_1 := 0, r_2 := i_0]}"] & s_1 \arrow[loop above, "{\mathit{coin}(i_0)[i_0 < 100]/\mathit{Reject}(i_0)}"] \arrow[r, "{\mathit{coin}(i_0)[i_0 \geq 100]/\ \mathit{Display}(r_1 + i_0)[r_1 := r_1 + i_0]}"] & s_2 \arrow[loop above, "{\mathit{coin}(i_0)/\mathit{Display}(r_1 + i_0)\ [r_1 := r_1 + i_0]}"] \arrow[ll, bend left, "{\mathit{vend}()/\mathit{Serve}(r_2)}"]
\end{tikzcd}$$

(a) EFSM representing the vending machine.

$$\langle \mathit{select}(\mathit{tea})/\epsilon, \mathit{coin}(50)/\mathit{Reject}(50), \mathit{coin}(100)/\mathit{Display}(100), \mathit{coin}(50)/\mathit{Display}(150), \mathit{vend}()/\mathit{Serve}(\mathit{tea})\rangle$$

(b) An example execution of the simple vending machine, in which an event is denoted $\mathit{input}(\mathit{parameters})/\mathit{output}(\mathit{parameters})$.

Fig. 1: The vending machine EFSM and an example trace.

In Figure 1a, inputs are separated from outputs by a "/" on the label of a transition. As shown in Figure 1a, our models can have parametric inputs, such as *select*, which carries an enumerated type for the choice of drink, or *coin*, which carries the integer value of the coin. Outputs can also bear parameters: this is the case for all three outputs in our model (*Reject*, *Display*, and *Serve*, which we subsequently abbreviate to R, D, and S). Our models are capable of storing values in registers, which are typed variables. In our example, r_1 will store the total value of coins inserted and r_2 will store the drink that was selected.

Although simple, this example illustrates the various inference challenges that we are faced with. We are not able to observe the register state when interacting with the machine. We do not know how many (if any) registers exist, or how they affect the sequential behaviour and output parameters of the machine. The only data visible to us are the input and output parameters. There is no "reset" function. We do not presume the prior existence of some representative set of example executions from which we can seek to derive the underlying model. The only thing we know is the signature of the interface (inputs and outputs) so that we are able to interact with the system.

2.2 Related work

Although there are several existing EFSM inference approaches in the literature, none of them has the capability of addressing this combined set of challenges. One technique [10] allows users to provide *data abstraction heuristics* to facilitate the introduction of registers during the inference process, but this requires the user to have a prior understanding of the system, which means that this technique cannot be applied to truly black-box inference scenarios.

Another technique, MINT [23], uses genetic programming (GP) infer update functions for variables. However, MINT cannot discover data dependencies between different transitions, for example, between *select* and *vend* in Figure 1a, nor can it discover internal registers like r_1 and r_2 in Figure 1a. Work presented in [9, 11] overcomes this by allowing the GP to introduce latent registers to output expressions and inferring update functions in a second pass of GP.

The above techniques are *passive*; they infer models from a predefined set of traces. There are also many *active* inference techniques in the literature [2, 5, 22], which infer models by querying the SUL, but these techniques only support updates in the form of simple assignments, or they do not support updates at all [17]. Register updates in terms of anterior values, such as the *coin* transitions in Figure 1a, are beyond them. These techniques also implicitly require that the SUL can be reset to a known state from which to execute the queries, which may not always be viable.

Another group of approaches [4, 21] phrase the EFSM inference problem as an instance of SAT. The solution is then a set of boolean variables, which together represent the automaton. Unfortunately, these approaches only consider boolean data values and do not support internal variables, so have limited applicability.

2.3 Definitions

Extended Finite State Machines State machine inference approaches such as Angluin's $L*$ method [3] infer deterministic automata, which do not incorporate data. In this paper, we use EFSMs [6], which do.

Definition 1. *An EFSM M is a tuple*[4] $M = (S, \mathcal{R}, I, O, P_I, P_O, T)$ *where S is a finite set of states, $\mathcal{R}$ is a cartesian product of domains, representing the type of registers. A domain is a set of values, such as int, $float$ or string. I is a finite set of (abstract) inputs. O is a finite set of (abstract) outputs. P_I is a mapping from I to a product of domains which are the type of parameters of the inputs. The type can be empty if the input has no parameter. P_O is a mapping from O to a product of domains , which are parameters of the outputs. Outputs may also have no parameter. T is a finite set of transitions.*

Each transition $t \in T$ is a tuple (s, x, y, G, F, U, s') where $s, s' \in S$, $x \in I$, $y \in O$, $G : P_I(x) \times \mathcal{R} \to \mathbb{B}$ is the transition guard, $F : P_I(x) \times \mathcal{R} \to P_O(y)$ is the output function that gives the value of the output parameters, $U : P_I(x) \times \mathcal{R} \to \mathcal{R}$ is the update function that gives the value of the registers after the transition.

Given an EFSM M as above, its *control FSM* is the FSM M' defined as $M' = (S, I, O, T')$ where $T' = \{(s, x, y, s') \mid \exists t \in T, t = (s, x, y, G, F, U, s')\}$.

An EFSM is *deterministic* iff for any state s and input x, any value of the registers $(r_0, ...r_k)$, and any value of input parameters $(p_0, ...p_j)$, there is at most one transition $t \in T$ such that $G((p0, ...p_j), (r_0, ...r_k)))$ holds. An EFSM is *complete* iff under the same conditions there is at least one such transition.

Semantics A *trace* is a sequence of events, as exemplified in Figure 1b, where an *event* is an instance of the observable part of a transition. As in Figure 1b

[4] Our definition is more detailed than Cheng and Krishnakumar's [6] to enable internal register variables and externally visible data parameters to be distinguished. We also do not have an initial state as this does not make sense for no-reset inference.

we denote this as $i(v)/o(v')$, for example the event $coin(100)/Display(100)$. For each event, we have $i \in I, o \in O, v \in P_I(i)$ and $v' \in P_O(o)$. Further, we refer to $i(v) \in \mathcal{I}$, as a *parametrized input* (or concrete input) for $(i, v) \in I \times P_I(i)$ and $\mathcal{I} = \bigcup_{i \in I} \{i\} \times P_I(i)$. Similarly, we call $o(v') \in \mathcal{O}$ a *parametrized output* and have $(o, v') \in O \times P_O(o)$ and $\mathcal{O} = \bigcup_{o \in O} \{o\} \times P_O(o)$. We denote the absence of an observable output by ϵ, which does not bear parameters.

As an EFSM executes a trace, transitions update registers and move the model between states. A *configuration* of an EFSM is a pair (s, r) of a state s and an n-tuple of values r, representing the values of each register $r_1, \ldots, r_n$. For example, when executing the trace in Figure 1b, after performing the $select(tea)$ event, we have the configuration $(s_1, (tea, 0))$. An *execution step* of the EFSM from (s, r), denoted as $(s, r) \xrightarrow{i(v)/o(v')} (s', r')$, is such that $\exists(s, i, o, G, F, U, s') \in T, G(v, r) \land v' = F(v, r) \land r' = U(v, r)$. An *execution* of the EFSM from (s, r) is a sequence of execution steps such that the posterior configuration of each step is the anterior configuration of the next step.

A configuration (s, r) is reachable from an arbitrary initial configuration (s_0, r_0) if there exists an execution ending in (s, r). An EFSM is *strongly connected* iff, given a reachable configuration (s, r) and a state s', there exists an execution from (s, r) ending in state s'.

Depending on the nature of the SUL, certain inputs may be invalid from a given state (e.g., a button in a GUI might be rendered inactive). For such systems, the underlying EFSM is inherently incomplete. To denote that input i is not available from state s, we use a special output symbol Ω, and "complete" the EFSM with transitions of the form $(s, i, \Omega, \top, \{\}, \{\}, s)$, which leave the model configuration unchanged.

Operations Similar to FSM functions associated with transition triggers (state and input), we define the output function λ and configuration update δ for an EFSM as follows.

$$\lambda((s, r), (x, v)) = \begin{cases} (y, v'), & \text{if } \exists(s, x, y, G, F, U, s') \in T, G(v, r) \land v' = F(v, r) \\ \Omega, & \text{otherwise} \end{cases}$$

$$\delta((s, r), (x, v)) = \begin{cases} (s', r'), & \text{if } \exists(s, x, y, G, F, U, s') \in T, G(v, r) \land r' = U(v, r) \\ (s, r), & \text{otherwise} \end{cases}$$

These will be lifted to sequences of parametrized inputs in the usual way, and we also define $\delta_s((s, r), (x, v))$ as the first element of $\delta((s, r), (x, v))$.

We also define projections that abstract from *output* parameters. For $(o, v) \in O \times P_O(o)$, $\pi(o, v) = o$. Projections are lifted to sequences of parametrized outputs. Moreover, we slightly abuse the notation and consider that, when applied to a parametrized input, $\pi(i, v) = (i, v)$. Thus, when applied to a trace in $(\mathcal{I}\mathcal{O})^*$, the projections will result in a trace in $(\mathcal{I}O)^*$.

2.4 Inferring Functions with Genetic Programming

When inferring an EFSM, there are two dimensions to the inference challenge. On the one hand, there is the challenge of inferring the potential sequential behaviours of the model. On the other hand, there is the task of inferring the "data-state" of the machine – of inferring the presence of registers, and of how they and output parameters are updated during execution.

One approach adopted in previous EFSM inference approaches [9, 11, 23] is Genetic Programming (GP) [19]. Here, a GP engine is supplied with the elementary components of a function — operators and operands — as well as a sample of input and corresponding output values. This takes the form of a table where columns represent the different variables and rows represent different execution instances. Candidate functions are typically represented by their parse tree, which is the representation we use for our technique in Section 3. The GP engine then searches through different combinations of operators and operands with the aim of finding one which is able to approximate the given set of data-points. This search follows the principles of Genetic Algorithms; solutions are combined and mutated iteratively, and the best solutions are chosen according to a fitness function, in this case the error-margin between the observed outputs and the outputs computed by the inferred functions.

3 The *ehW*-inference Algorithm

Our goal is to infer EFSM models, complete with guards and data transformations, of black-box systems which we cannot arbitrarily reset. This section presents our *ehW*-inference algorithm and the assumptions associated with it.

3.1 Assumptions

EFSMs introduce a particular inference challenge as the same set of behaviours can be modelled in a variety of ways. For example, conditional behaviour can either be encoded as guards on states, or can be directly encoded into separate states. We subsequently assume that the SUL can be modelled by an EFSM that has the required properties. This constrains the style of EFSM inferred and also fundamentally assumes that there is a finite state "control" model.

For our investigation of the *ehW*-inference algorithm, we have started from a relatively restrictive set of assumptions about the target model. These are collectively intended to ensure that the SUL is controllable, and that its transitions between different states are observable.

Connectivity The control FSM of the EFSM is strongly connected. In a state machine without a reset, we assume that the inference process is always able to reach any state from any other state. Otherwise, we would only be able to infer a strongly connected component.

Determinism The EFSM is deterministic. This is a classical and essential assumption in inference approaches from traces, to be able to recognize different configurations by the fact that they yield different observations.

Observability An EFSM is *observable* iff any two distinct transitions $t = (s, x, y, G, F, U, s')$ and $t' = (s, x, y', G', F', U', s'')$ that share the same starting state and (abstract) input have different (abstract) outputs, i.e. $y \neq y'$.

We also introduce two assumptions which allow us to infer guard, output, and update functions.

Register domain observability Values assigned to registers by update functions should be visible at some point as an input or an output parameter. This need not occur at the transition where the register assignment occurs.

Guard visibility Guards can only use input parameters of the transition, and not registers. In other words, registers can only contribute to the computation of parameter outputs (e.g. to display the total value of coins inserted in our example). They cannot, however, be used to condition state transitions.

Guard visibility is a limitation of our current *ehW*-inference algorithm. Where a system produces outputs that depend on internal stored variables (such as, counters), the EFSM our algorithm builds would have as many different states as reachable values of the (vector of) variables. This means that, for the algorithm to infer an EFSM for a system whose decisions are based on internal variables, the system would need a finite control state space.

3.2 Homing and Characterizing

Since we do not presume the existence of a reset function, the *ehW*-inference algorithm must compensate for this during the learning process. We achieve this with the help of "homing" and "characterizing" sets. Intuitively, a homing sequence is an input sequence whose tail state is uniquely determined by the observed output sequence. A characterizing set is a set of input sequences that provide a unique response for every state in the system, thus enabling each state to be uniquely identified. Previous work [13] has shown how these notions can be incorporated into a learning setting to enable the inference of conventional FSMs without reset functions. To enable this here, we provide definitions of homing sequences and characterizing sets that are specific to EFSMs.

A sequence $h \in \mathcal{I}^*$ is *homing* iff $\forall(s, s', r, r') \in S^2 \times R^2, \pi(\lambda((s, r), h) = \pi(\lambda((s', r'), h) \Rightarrow \delta_s((s, r), h) = \delta_s((s', r'), h)$. This means that the sequence of (non-parametrized) outputs uniquely defines the state reached at the end of the sequence. Thus, by applying h and observing the outputs, it is possible to ascertain the state reached at the end of h. A set $W \subset \mathcal{I}^*$ is *characterizing* iff $\forall(s, s') \in S^2, s \neq s', \exists w \in W, \forall(r, r') \in R^2, \pi(\lambda((s, r), w)) \neq \pi(\lambda((s', r'), w))$. Thus, a state can identified by the (uniquely) response to every sequence in W.

3.3 Inputs and Data Structures

We assume we are given:

– An input set I with associated parameters P_I.
– A SUL whose behaviour can be modelled by an EFSM with these inputs, satisfying the assumptions in Section 3.1, to which we can apply sequences of parametrized inputs and observe the corresponding parametrized outputs.

- A tentative homing sequence $h \in \mathcal{I}^*$. This may be empty (ϵ).
- A tentative characterizing set $W \subset \mathcal{I}^*$. This may be the empty set.
- For each domain of input parameters, an ordered list of values. We further require that those values must include all values appearing in h and W, and that they should appear at the beginning of the lists of their domains. We denote I_1 a set of inputs parametrized with at least the first value in each domain. And we denote I_s the set of all (sampled) parametric inputs that can be created using the provided list of values.

As we are in the active inference setting, we also require a means of interacting with the SUL, commonly referred to as a driver [13] or a mapper [2, 1]. This serves as a bridge between the inference engine and the system, and can be used to map low level inputs and outputs from the software to more abstract tokens better suited to the inference process, for example to convert network packets into a more abstract or readable format in line with the desired modelling style. It is therefore critical for inference that any such abstractions applied by the interface fulfil the assumptions set out in Section 3.1 as this is how the inference engine will perceive the system. However, since this work is more concerned with establishing the theoretical foundations of *ehW*-inference, we do not give this further consideration here.

During learning, the trace observed at any given moment is represented by ω. This is extended whenever we apply an input and observe the corresponding output. The algorithm will record deduced information in the following sets:

- $Q \subset 2^{W \to O^+}$ denote states, defined by their characterization. Each state is named by traces recording its responses to the input sequences from W.
- $\Delta : Q \times \mathcal{I} \to Q$ and $\Lambda : Q \times \mathcal{I} \to 2^{\mathcal{O}}$ record transitions. With guard visibility, the abstract output sequence and state reached from a given state by applying a parametrized input sequence is unique and does not depend on the value of registers, so $\Delta(s,(x,v)) = s'$ and $(y,v') \in \Lambda(s,(x,v))$ iff $\exists(s,x,y,G,F,U,s') \in T, \exists r \in \mathcal{R}, G(v) \wedge v' = F(v,r)$. Δ, Λ are lifted to sequences, and as usual, for an empty input sequence ϵ, the output sequence is also ϵ and $s' = s$.

3.4 *ehW*-inference Backbone

The *ehW*-inference algorithm, detailed in Algorithm 1, has four core parts. The first part, called the *backbone*, follows what is in-effect the basic *hW*-inference backbone algorithm [13]. Our backbone infers an FSM transition structure using only a single parameter value for each input, so we do not need registers or transition guards. The second part of the algorithm, *sampling*, is responsible for traversing the inferred structure using different parameter values for the inputs, the aim being to gather data for part three of our algorithm, *generalisation*, which infers the registers, output functions, and transition guards within an EFSM. The final part, *counterexample processing*, again comes from [13] and involves searching for inconsistencies between the conjecture model and the SUL.

Algorithm 1 ehW algorithm

```
1: Inputs: $I$, $I_1 \subset I_s \subset \mathcal{I}$, $h \in I_s^*$, $W \subset I_s^*$
2: repeat
3:     $Q, \Delta, \Lambda, q \leftarrow \bot$
4:     repeat
5:         $\omega, Q, \Delta, \Lambda \leftarrow$ BACKBONE $(I_1, h, W, \Delta, \Lambda)$ ▷ Learn the structure
6:         for $t \in \Delta, i \in I_s$ do ▷ Apply sampling to learn data values for generalisation
7:             if $\pi(t(i)) \notin \Lambda(t(i))$ then
8:                 break
9:     until No inconsistency
10:    repeat
11:        $M \leftarrow$ GENERALISE$(\omega, h, Q, I, O, P_I, P_O, \Delta, \Lambda)$ ▷ Generalise into an EFSM
12:        $(\omega, CE) \leftarrow$ GETCOUNTEREXAMPLE$(M, SUL)$
13:    until $\neg$ (CE is a data CE)
14:    if CE found then ▷ Present conjecture to oracle and process resulting counterex.
15:        $(W, I_1, I_s) \leftarrow$ PROCESSCOUNTEREXAMPLE
16: until no counterexample found
17: return $M$
```

Backbone It is the job of the backbone hW-inference algorithm (line 5) to identify the basic control structure of the model. This is done in the same way as in [13], but using only one input parameter for each abstract transition. The basic idea is to learn states by first applying the homing sequence h to reach a known location and then sequences in W to distinguish the destination state. Transitions are learned by applying (parametrised) events in I and then sequences in W to discover the destination.

The backbone runs until we end up with a graph structure that contains a strongly connected component using a single input parameter value for each transition. Then, we can run sampling, call the generalisation procedure, and ask the oracle for a counterexample, as explained later. However, this is predicated on the fact that the h and W sets provided are correct – i.e. that h is genuinely a homing sequence, and that W is characterizing. If this is not the case, this will manifest itself through various inconsistencies which, when detected, indicate that the h or W sets need to be extended and the backbone restarted.

Sampling The main purpose of sampling (lines 6–8) is to enrich the set of values for each transition so as to be able to generalise concrete values into symbolic output and update functions (Line 11). The basic idea is to fire every transition learned by the backbone hW-inference algorithm with every input parameter in its domain[5] and observe the corresponding output. This then forms the training set for GP. However, in doing this, we may observe inconsistencies between (abstract) transitions.

[5] Variable domains do not have to be finite, although we can obviously only execute finite samples of infinite domains. Where counterexamples require a data value not

For example, in learning the simple drinks machine, the backbone may use 50 as its input to the *coin* transitions. In this case, the algorithm can only observe *Reject*(50) as an output, since the first coin to be input must be 100 or greater. During sampling, we then observe *coin*(100)/*Display*(100). This is inconsistent with what we have observed so far as *Reject* and *Display* clearly represent different output behaviour. Thus, we have discovered an inconsistency and can return to backbone hW-inference inference with an updated h and W.

Inconsistencies Inconsistencies (line 9) can be detected as soon as we apply a sequence and observe differing output symbols from those expected from the partial machine. These can manifest themselves in various ways.

If h is not homing, it is possible that the same response leads to two different states, which would be considered by the algorithm as a single one. This can give rise to apparent non-determinism, which we call h-ND inconsistency. h-ND inconsistencies occur when we have observed previously $h/a.\beta/v.x/y$ and then apply $h/a'.\beta/v'.x'/y'$ s.t. $\pi(a.v) = \pi(a'.v')$ yet $\pi(y') \neq \pi(y)$ and ($x = x'$ or $y = \Omega$ or $y' = \Omega$).

Since we assume we are learning an observable EFSM satisfying guard visibility, the difference in outputs implies the control states s and s' reached after h/a and h/a' are different. Thus, h is not homing, and extending it with the prefix of β up to the first differing output symbol will distinguish two more states.

Similarly, if W is not characterizing, two different control states could be confused as a single one; the algorithm would associate outgoing transitions and sequences from those two states to the single reconstructed one.

A W-ND inconsistency occurs when we previously observed $h/a.\alpha/u.\beta/v.x/y$ and then observe $h/a'.\alpha'/u'.\beta/v'.x'/y'$ (where $\alpha, \alpha' \in \mathcal{I}^*$, $\beta \in \mathcal{I}^*$ and $x, x' \in \mathcal{I}$) s.t. $\pi(a) \neq \pi(a')$ or $\alpha \neq \alpha'$ but $\Delta(H(\pi(a)), \alpha) = \Delta(H(\pi(a')), \alpha')$, $\pi(v') = \pi(v)$ and $\pi(x') = \pi(x)$ yet $\pi(y') \neq \pi(y)$ and ($x = x'$ or $y = \Omega$ or $y' = \Omega$). In this case, $\Delta(H(\pi(a)), \alpha)$ and $\Delta(H(\pi(a')), \alpha')$ can in fact be distinguished by $\beta.x'$, so $\beta.x'$ can be added to W.

As in the case of the FSM hW-inference algorithm, we can remark that all states traversed while applying β can be distinguished by some suffix of β. We can extend W with any such suffix that is not yet in W. However, we would refrain from adding all suffixes into W as the cardinal of W acts as a multiplicative factor on the complexity of the learning [13].

Generalisation The role of generalisation (Line 11) is to infer symbolic output and update expressions which account for the concrete output and update values observed during the backbone and sampling phases. In essence, we want to take the collected values for each (i, o) pair of input and output types and infer a general expression F for that pair. However, a complication is that the output values may be influenced by the values of unobservable registers. We need our

in the observed sample, the oracle (Section 3.6) is free to include include these, and they are added to the sampled domain as part of counterexample processing.

technique to infer this, along with any updates U to the registers to ensure they evaluate to the correct values. Additionally, where the model contains data-dependent behaviour, we need to infer symbolic guards to distinguish this. This enables us to predict how the inferred model might behave when faced with unseen inputs. To do this, we apply a technique based on GP, similar to in [9, 11]. We present the details of this in Section 3.5.

Counterexamples Once we have found a strongly connected FSM and generalised it to an EFSM, we look for a discrepancy between outputs from the EFSM and the SUL. We first need to synchronise the EFSM with the SUL which can be done by re-running the past trace on the EFSM model from the earliest occurrence of a homing sequence. The trace can be extended using one of the usual strategies (such as random walk, bounded model checking) to look for discrepancies. As soon as an output differs between EFSM and SUL, the extended trace is returned as a counterexample.

3.5 Generalisation

We here lay out the details of our generalisation step described above. The goal here is take the concrete data values observed in the backbone and sampling phases and infer symbolic expressions which account for them, thereby enabling the model to be used to predict the output from the system when executed with unseen input parameters. We may also need to infer symbolic guards to distinguish data-dependent behaviour.

To do this, we apply a technique based on GP, similar to in [9, 11]. This is shown in Algorithm 2, which defines the GENERALISE function from Algorithm 1. There are five main steps. First, we convert the abstract data structures of the backbone algorithm into an initial EFSM (line 2). Next, we group together instances of transitions that we would like to generalise to the same F and U (line 3). For each group, we use GP to produce an output function which satisfies the observed input/output pairs (line 5). This may introduce a new register to the model for which we need to infer updates (lines 7–9) to ensure it holds the correct value when evaluated. Finally, we drop literal input guards on transitions (e.g., $i_0 = 50$, line 13) and resolve any resulting nondeterminism (line 14).

EFSM Construction The first step of generalisation, EFSM (line 2), is to convert the abstract data structures to a concrete EFSM model where transitions guard for the observed input parameters and produce the observed concrete outputs. This is a fairly trivial process, except that we must drop all events in ω before the first occurrence of the current (lastly used) h as we do not know where we are, meaning we cannot reliably group transitions from before this point.

Transition Grouping We use the name GROUPTRANSITIONS (line 3) to be consistent with [9, 11], but what we are actually doing here is grouping events

Algorithm 2 Outline of our generalisation.

```
1: function GENERALISE($\omega$, $h$, $Q$,$I$,$O$,$P_I$,$P_O$,$\Delta$,$\Lambda$)
2:     $efsm \leftarrow$ EFSM($Q, I, O, P_I, P_O, \Delta, \Lambda, h, \omega$) ▷ Convert the abstract data structures into a concrete EFSM
3:     $groups \leftarrow$ GROUPTRANSITIONS($\Lambda, \omega$)
4:     for $g_1 \in groups$ do
5:         $fun \leftarrow$ INFEROUTPUTFUN($g_1$) ▷ Use GP to infer functions that predict outputs for each group, introducing registers if required.
6:         $newEFSM \leftarrow$ REPLACELITWITHFUN($efsm, g_1, fun$) ▷ Replace literal outputs with inferred functions.
7:         for $r_n \in fun.latentVars$ do ▷ Infer updates for any new registers.
8:             for $g_2 \in groups$ do
9:                 $newEFSM \leftarrow$ INFERUPDATEFUNS($g_2$, TARGETVALUES($newEFSM, r_n$))
10:        if ACCEPTS($newEFSM, \omega$) then ▷ Check that inferred functions are compatible with traces. If not the $efsm$ remains unchanged.
11:            $efsm \leftarrow newEFSM$
12:        $efsm \leftarrow$ STANDARDISE($e$) ▷ Unify transition groups split by history.
13:    $efsm \leftarrow$ DROPGUARDS($efsm$)
14:    $efsm \leftarrow$ RESOLVENONDETERMINISM($efsm, \omega$)
15:    return $efsm$
```

in ω by their corresponding transition in Δ as this is known here. These groups then form the training sets for GP. As in [9, 11], though, there is the additional need to split groups by their *history* (the preceding groups) to account for any side effects of other transition groups on unobserved register values.

Output and Update Inference Output functions are inferred using GP as detailed in [9, 11]. In short, GP uses a series of crossover and mutation operations to combine a predefined set of operators and operands into an expression which maps the observed input parameters to the observed output parameters as discussed in Section 2.4. This may introduce new registers to the model for which update expressions must be inferred to ensure that they hold the correct values when they are evaluated. Details of this process can be found in [9, 11].

Standardisation Where groups are split by their respective histories, the GP may infer different output and update functions for the separate subgroups. Because we know these subgroups are in fact instances of the same transition, we need to unify the output and update expressions of the various subgroups. This is what standardisation does. Full details of this are published in [9, 11].

Resolution of Nondeterminism Having inferred symbolic output and update expressions, we can now drop the guards (line 13) which prevent transitions from responding to unobserved input parameters. As in [9, 11], this leads to nondeterminism which must be resolved. There are two potential sources of this. The first is duplicated behaviour, which is introduced to the model when we sample different data values for the same (abstract) transition. As in [9, 11], this can be trivially resolved by merging the offending (concrete) transitions, which should be identical because of the standardisation step.

The other source of nondeterminism, which is not considered in [9, 11], is data-dependent behaviour. This cannot be resolved by merging as the behaviours are different. Algorithm 3 shows how we resolve this by calling GP a third time to infer guard functions that distinguish pairs of nondeterministic transitions. For each nondeterministic pair of transitions (line 3), we walk the trace in the model (lines 6 - 12) recording the input and register values when either transition is taken (lines 8 and 10). We then call GP to find a boolean guard expression which evaluates to *true* for one transition and *false* for the other.

Algorithm 3 Resolving nondeterminism.

```
 1: function resolveNondeterminism($efsm, \omega$)
 2:     while $efsm$ is nondeterministic do
 3:         $t_1, t_2 \leftarrow$ getNondeterministicPair($efsm$)
 4:         $took_1, took_2 \leftarrow \emptyset, \emptyset$ ▷ Initialise the training sets for each transition
 5:         $state, registers \leftarrow$ initialise($efsm$)
 6:         for $event \in \omega$ do ▷ Walk the trace in the model.
 7:             if correspondsTo($event, t_1$) then ▷ If we took $t_1$, add the inputs and registers to the training set for $t_1$.
 8:                 $took_1 \leftarrow took_1 \cup (event.inputs, r)$
 9:             if correspondsTo($event, t_2$) then ▷ If we took $t_2$, add the inputs and registers to the training set for $t_2$.
10:                 $took_2 \leftarrow took_2 \cup (event.inputs, r)$
11:             $r \leftarrow$ updateRegisters($r, event$)
12:             $s \leftarrow$ updateState($s, event$)
13:         $guard \leftarrow$ inferGuard($took_1, took_2$) ▷ Call GP to infer a guard to distinguish the two transitions based on the input and register values they were fired with.
14:         $efsm \leftarrow$ addGuardToTransitions($guard, t_1, t_2$)
```

3.6 Oracle Procedure

With nondeterminism resolved, we can then present the EFSM to the oracle, which attempts to extend the trace to end with an output from the SUL which differs from the conjecture EFSM. If the output type is different, then we have observed an inconsistency as described above, and need to revise the structure of the control FSM. However, if the difference is only on the output parameter values, this means the functions computed by the generalisation were incorrect. We call this a *data counterexample*. To resolve data counterexamples, we can simply rerun our generalisation procedure with the new data. As in other approaches that learn models from unbounded black box systems, only an approximate oracle can be implemented. If the oracle cannot find a counterexample, the model is assumed to be equivalent to the system.

Note that our EFSM may not be strongly connected if, during inference, we get “trapped” in a state. In this case, the oracle may extend the set of input parameter values (as well as output parameter values) make it possible to reach states that are not reachable with the current set of values. The oracle may also extend the sets of input and output parameter values to elicit data counterexamples which could not be revealed otherwise.

4 Inferring a Vending Machine Controller

We now illustrate the execution of ehW-inference on our running example from section 2.1. We start with $h = \epsilon, W = \{\}, I_1 = \{coin(50), select(tea), vend\}$. As h and W are empty, the backbone will just learn a “daisy” (single state) automaton with each input X from I_1.

$$(s_0, \langle\rangle) \underbrace{\quad}_{h=\epsilon} \underbrace{\quad}_{w=\{\}} \underbrace{\xrightarrow{coin(50)/\Omega}_{1}}_{X=coin(50)} (s_0, \langle\rangle) \underbrace{\quad}_{w=\{\}} \underbrace{\quad}_{h=\epsilon} \underbrace{\xrightarrow{select(tea)/\epsilon}_{2}}_{X=select(tea)} (s_1, \langle tea, 0\rangle) \underbrace{\quad}_{w=\{\}}$$
$$\underbrace{\quad}_{h=\epsilon} \underbrace{\xrightarrow{vend/\Omega}_{3}}_{X=vend} (s_1, \langle tea, 0\rangle) \underbrace{\quad}_{w=\{\}} \underbrace{\xrightarrow{coin(100)/D(100)}_{4}}_{sampling} (s_2, \langle tea, 100\rangle)$$

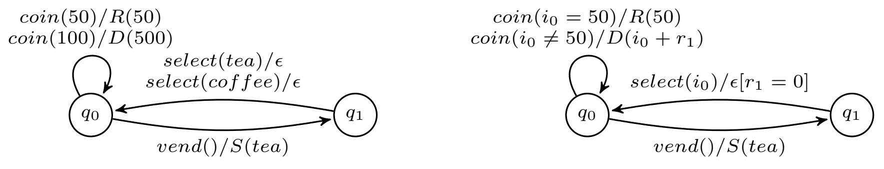
We sample with $I_s = I_1 \cup \{coin(100), select(coffee)\}$. As soon as we apply $coin(100)$, we spot nondeterminism, leading us to revise $h = coin(100)$, $W = \{coin(100)\}$ and $I_1 = \{coin(100), select(tea), vend\}$. We restart the backbone.

$$(s_2, \langle tea, 100\rangle) \underbrace{\xrightarrow{coin(100)/D(200)}_{5}}_{h} (s_2, \langle tea, 200\rangle) \underbrace{\xrightarrow{coin(100)/D(300)}_{6}}_{w} (s_2, \langle tea, 300\rangle)$$

We now know that applying h with response D leads to q_0, characterized by $coin(100) \mapsto D$, but we have not yet learnt Δ for $(q_0, coin(100))$, so we need to home again before proceeding. This leaves us still in q_0, so we can now learn a transition from it, with $\alpha = \epsilon$ (no transfer needed), and we use $X = coin(100)$ as it is used in h and W.

$$(s_2, \langle tea, 300\rangle) \underbrace{\xrightarrow{coin(100)/D(400)}_{7}}_{h} \underbrace{\xrightarrow{coin(100)/D(500)}_{8}}_{X} \underbrace{\xrightarrow{coin(100)/D(600)}_{9}}_{w} (s_2, \langle tea, 600\rangle)$$

We just learnt $\Delta(q_0, coin(100)) = q_0$, so we know we remain in q_0, and can continue learning other inputs. Thus, we learn that the transition for *select* is an Ω self-loop transition, and *vend* outputs $Serve(tea)$ and goes to a state where W gives Ω. Thus we learn a new state $q_1 = \{coin(100) \mapsto \Omega\}$. After further steps to learn all transitions from state q_1 and sampling with inputs from I_s, we reach step 25 where we have found a two state machine with q_0 (a merged state of s_1 and s_2 in the SUL) and q_1 (corresponding to state (s_0), and transitions on all inputs from I_s. This graph is strongly connected, so the backbone ends with



(a) Control FSM with values from Λ. (b) EFSM after generalisation.

Fig. 2: Conjecture built from $\omega, h, \Delta, \Lambda$ after 25 steps.

the model in Figure 2a. The generalisation will infer a two state EFSM, shown in Figure 2b, and the algorithm will ask the oracle for a counterexample.

A simple counterexample is obtained by sending *coin*(50) to the SUL, which at this point is in configuration $(s_2, \langle \textit{coffee}, 100\rangle)$, so will respond with $D(150)$ whereas the conjecture would respond $R(50)$. Since output types D and R differ, this is not a data counterexample but W-ND, so we add *coin*(50) to W and restart the backbone with $h = \textit{coin}(100)$ and $W = \{\textit{coin}(100), \textit{coin}(50)\}$.

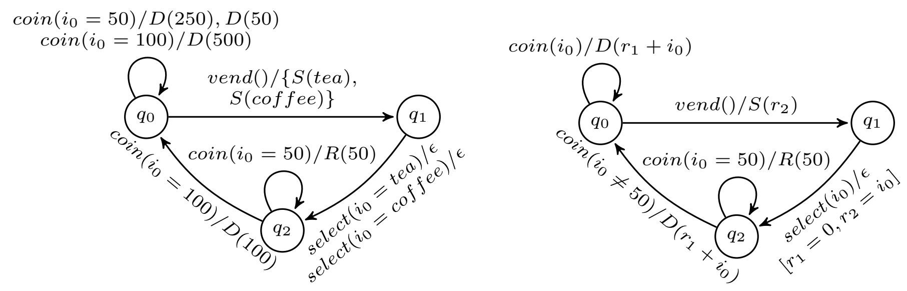
As h has not changed, we can implement a dictionary (as proposed in previous learning methods [18]), viz. the outputs of any input sequence of the form $h\alpha Xw$ that was previously applied (at some point) on the SUL can be assumed to be valid and can be reused to fill structures without reapplying the input sequence. However, as W changed, we need to completely re-learn the set of states Q.

Since h is homing and W is now characterizing, this application of the backbone will discover all the states of the SUL in 17 extra steps (up to step 43), and all transitions on inputs from I_1 when we reach step 59. Sampling makes it possible to learn the last transition, *coin*(50) from state $s1$ at step 67, leaving the model shown in Figure 3a.

$$\underbrace{\xrightarrow{coin(100)/D(200)} 62}_{h}\ \underbrace{\xrightarrow{coin(50)/D(250)} 63}_{s}\ \underbrace{\xrightarrow{vend/S(tea)} 64}_{\alpha}\ \underbrace{\xrightarrow{select(coffee)/\epsilon} 65}_{s}\ \underbrace{\xrightarrow{coin(50)/R(50)} 66}_{X}\ \underbrace{\xrightarrow{coin(50)/R(50)} 67}_{w2}$$

As before, we can then apply the generalisation procedure to infer a full EFSM model. This is shown in Figure 3. As can be seen from the two $q_2 \xrightarrow{coin}$ transitions, the guard distinguishing them is rather simplistic. Because of this, our oracle is able to return the counterexample $\textit{coin}(20)/R(20)$ (step 68). This brings a new input parameter, 20, into play.

This counterexample differs only in terms of its data values, and there is no h or W nondeterminism. Thus, it is a data counterexample, indicating we need only rerun generalisation on the newly extended trace. This gives the same model as in Figure 3b, but with guards $i_0 \leq 50$ and $i_0 > 50$ where we previously had $i_0 = 50$ and $i_0 \neq 50$, and the output $R(i_0)$ instead of $R(50)$. Given the input domain of *coin*, this is equivalent to Figure 1a as there is no coin with a value between 50 and 100. Thus, we learnt an accurate model of our vending EFSM by executing just 68 events, although this is dependent on the stochastic outcome of GP. Running the algorithm again using a different random seed for



(a) Control FSM with values from Λ. (b) EFSM after generalisation.

Fig. 3: Conjecture built from $\omega, h, \Delta, \Lambda$ after 67 steps.

GP may produce different generalisations to Figures 2b and 3b, so may require additional steps to infer the target model.

5 Conclusions and Future Work

In this paper we have presented the *ehW*-inference algorithm. It is based on the *hW*-inference algorithm by Groz *et al.* [13], which enables the inference of conventional FSMs from systems without a reset functionality. However, we incorporate into this the GP-driven capability to infer registers and functional relationships between data-states, based on work by Foster *et al.* [11].

Our future work will go in two primary directions. Firstly, our approach currently operates under several relatively restrictive assumptions (Section 3.1). Some of these assumptions may be relaxed, and our future work will set out to establish the extent to which this is the case. Secondly, we have so far only presented a single running example, without delivering any insight into the scalability of the approach. This will be investigated in a more comprehensive empirical study, with models of varying size and complexity.

References

1. Fides Aarts. *Tomte : Bridging the gap between active learning and real-world systems*. PhD thesis, Radboud University Nijmegen, 2014.
2. Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM 2012: Formal Methods*, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
3. Dana Angluin. Queries and concept learning. *Machine learning*, 2(4), 1988.
4. Igor Buzhinsky and Valeriy Vyatkin. Automatic inference of finite-state plant models from traces and temporal properties. *IEEE Transactions on Industrial Informatics*, 13(4), 2017.
5. Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning extended finite state machines. In *Software Engineering and Formal Methods*. Springer, Cham, 2014.

6. Kwang-Ting Cheng and Avinash S Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *30th ACM/IEEE Design Automation Conference*. IEEE, 1993.
7. Wontae Choi, George Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices*, 48(10), 2013.
8. Carlos Diego N Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simao. Learning to reuse: Adaptive model learning for evolving systems. In *International Conference on Integrated Formal Methods*. Springer, 2019.
9. Michael Foster. *Reverse Engineering Systems to Identify Flaws and Understand Behaviour*. PhD thesis, University of Sheffield, September 2020.
10. Michael Foster, Achim D. Brucker, Ramsay Taylor, Siobhán North, and John Derrick. Incorporating data into EFSM inference. In *Software Engineering and Formal Methods*. Springer International Publishing, 2019.
11. Michael Foster, John Derrick, and Neil Walkinshaw. Reverse-engineering EFSMs with data dependencies. In *IFIP International Conference on Testing Software and Systems*. Springer, 2022.
12. Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2002.
13. Roland Groz, Nicolas Bremond, Adenilso Simao, and Catherine Oriat. hW-inference: A heuristic approach to retrieve models through black box testing. *Journal of Systems and Software*, 159, 2020.
14. Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012.
15. Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1), 2014.
16. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8), 1996.
17. Shang-Wei Lin, Étienne André, Jin Song Dong, Jun Sun, and Yang Liu. An efficient algorithm for learning event-recording automata. In *Automated Technology for Verification and Analysis*, pages 463–472. Springer Berlin Heidelberg, 2011.
18. Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, 2003.
19. Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. lulu.com, 2008.
20. Ronald L Rivest and Robert E Schapire. Inference of finite automata using homing sequences. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, 1989.
21. V. Ulyantsev and F. Tsarev. Extended finite-state machine induction using sat-solver. In *2011 10th International Conference on Machine Learning and Applications and Workshops*, volume 2, 2011.
22. Frits Vaandrager and Abhisek Midya. A Myhill-Nerode theorem for register automata and symbolic trace languages. In *Theoretical Aspects of Computing*. Springer International Publishing, 2020.
23. Neil Walkinshaw and Mathew Hall. Inferring computational state machine models from program executions. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016.