Preprint:

Vega, G., Groz, R., Oriat, C. et al. (3 more authors) (Submitted: 2024) Learning EFSM models with registers in guards. [Preprint - arXiv] (Submitted)

https://doi.org/10.48550/arXiv.2406.07040

Journal of Machine Learning Research 1–14, 2024 Learning and Automata (LearnAut) – ICALP 2024 workshop

Learning EFSM Models with Registers in Guards

Germán Vega GERMAN.VEGA@IMAG.FR
Roland Groz ROLAND.GROZ@UNIV-GRENOBLE-ALPES.FR
Catherine Oriat CATHERINE.ORIAT@UNIV-GRENOBLE-ALPES.FR
LIG, Université Grenoble Alpes, F-38058 Grenoble, France

Michael Foster M.FOSTER@SHEFFIELD.AC.UK
Neil Walkinshaw N.WALKINSHAW@SHEFFIELD.AC.UK
Department of Computer Science, The University of Sheffield, UK

Adenilso Simão ADENILSO@ICMC.USP.BR
Universidade de São Paulo, ICMC, São Carlos/São Paulo, Brasil

Abstract

This paper presents an active inference method for Extended Finite State Machines, where inputs and outputs are parametrized, and transitions can be conditioned by guards involving input parameters and internal variables called registers. The method applies to (software) systems that cannot be reset, so it learns an EFSM model of the system on a single trace.

Keywords: Active inference, Query learning, Extended Automata, Genetic Programming

1. Introduction

Automata learning by active inference has attracted interest in software engineering and validation since the turn of the century Peled et al. (1999) Hagerer et al. (2002). Although classical automata (in particular Mealy models) have been used in software testing for a long time Chow (1978), software engineering has evolved to richer models of automata, with parameters on actions, internal variables, and guards, as can be found in such formalisms as SDL, Statecharts or UML. Inferring such complex models, whose expressive power is usually Turing-complete, is challenging.

A significant step was the introduction of register automata Isberner et al. (2014). However, the practical application of those inference methods is limited by two main problems. Firstly, the expressive power of the inferred models is usually restricted (e.g. only boolean values or infinite domains with only equality). Secondly, most inference methods learn the System Under Learning (SUL) by submitting sequences of inputs from the *initial state*, thus requiring the SUL to be reset for each query, which can be costly, impractical or impossible.

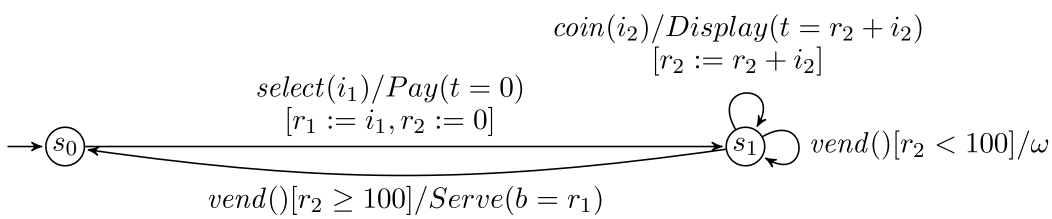
Recently, Foster et al. (2023) proposed an algorithm to infer EFSMs in the above circumstances. It uses an approach based on the hW algorithm by Groz et al. (2020) to infer the structure of the control state machine and to work around the absence of a reliable reset, and uses Genetic Programming Poli et al. (2008) to infer registers, guards and output functions from the corresponding data observations. This approach is, however, restrictive because guards on transitions can only involve input parameters and cannot be based on internal registers. Our challenge, therefore, is to provide a more general solution that not only infers the presence of registers and corresponding output functions but also enables us to include state transitions that depend upon register values. *Our paper proposes an algorithm that can infer EFSM models that include registers in guards.*

Addressing this problem raises a number of challenges. i) it is necessary to identify the underlying control state machine, by distinguishing what should (preferably) be encoded in states and what should be encoded in registers; ii) for any transition in the EFSM, it is necessary to identify any guards on inputs and the inferred registers, and to identify any functions that may affect the state of the registers or the values of the observable outputs; and iii) since no reliable reset is assumed, analyzing the influence of registers should be done from comparable configurations of the SUL, so it requires finding a way of coming back to a previously recognized configuration.

These challenges are tightly interdependent; any guards and output functions are highly dependent on the transition structure of the machine. Any inference process must generalise from the executions observed so far (the inferred model must be capable of recognising or accepting sequences that have not yet been observed). At the same time, it must avoid overgeneralization (i.e. accepting impossible or invalid sequences of inputs).

2. Running example and background models

We consider a straightforward running example that exhibits most features of the model yet is small enough to be fully illustrated in this paper. Our example is a vending machine, shown in Figure 1, where the choice of drink and the money paid into the machine are parameters. These are recorded by registers (internal variables), that influence later computations. Throughout the paper, we refer to the parameterized events (such as *select*(*tea*)) as concrete inputs or outputs and to the event types (such as *select*) as abstract ones.



(a)

$$\langle select(tea)/Pay(0), coin(50)/Display(50), vend()/\omega, coin(50)/Display(100), coin(50)/Display(150), vend()/Serve(tea)\rangle$$

(b)

Figure 1: Our vending machine EFSM and an example trace.

Figure 2 is taken from Foster et al. (2023). In that work, guards can only be influenced by input parameters, not registers, so coins are rejected if they are less than the price of the drink. There are two distinct states after drink selection (s_1 and s_2), one where insufficient money has been input, and one where the drink can be served. By allowing registers in guards, Figure 1(*a*) represents a more realistic vending machine where we can pay with smaller coins until we reach a sufficient amount. A single state after selection is enough to allow different transitions depending on the amount stored in a register r_2.

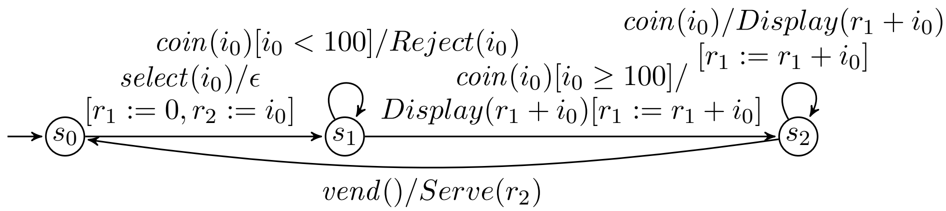


Figure 2: No register allowed in guard (from Foster et al. (2023)

The output ω of the $s_1 \xrightarrow{vend} s_1$ transition is a special output indicating that the transition neither produces visible output nor changes the system state. In a vending machine, this could model that the button cannot be pressed mechanically or is visibly ineffective.

Having named input and output parameters can also provide evidence that they share a common register value, as is the case for the t output parameter in Figure 1(a), which represents the total amount of money put into the system by coins for a selected drink. t is shared by output types *Pay* and *Display* and mapped to a single register r_2.

3. Assumptions for tractability

In this work, an EFSM is a tuple $(Q, \mathcal{R}, \mathcal{I}, \mathcal{O}, \mathcal{T})$ where Q is a finite set of states, $\mathcal{R}$ is a cartesian product of domains, representing the type of registers. $\mathcal{I}$ is the set of concrete inputs, structured as a finite set of abstract inputs I each having associated parameters and their domains P_I. Similarly $\mathcal{O}$ for concrete outputs based on O for abstract outputs. $\mathcal{T}$ is a finite set of transition, i.e. tuples (s, x, y, G, F, U, s') where $s, s' \in Q$, $x \in I$, $y \in O$, $G : P_I(x) \times \mathcal{R} \to \mathbb{B}$ 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.

Associated with an EFSM, its *control machine* is the non-deterministic finite state machine (NFSM) that results from abstracting from parameters (and therefore guards and registers). It defines the finite control structure of a given EFSM and is defined by a quadruple (Q, I, O, Δ), where Δ is the transition function $\Delta : Q \times I \times O \to Q$, which lifted to sequences of inputs outputs as $\Delta^* : Q \times (I \times O)^* \to Q$. The control machine for Figure 1(a) is shown in Figure 3.

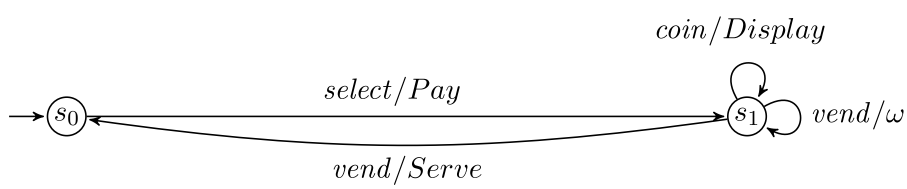


Figure 3: Control NFSM of the vending machine.

We assume that the SUL is semantically equivalent to an EFSM, which has the following properties:

- It is *deterministic* (at the concrete level), and its control NFSM is both *strongly connected* (since we do not assume a reset, we can only learn a strongly connected component) and *observable* (i.e. if an input triggers different transitions from the same state, they have different abstract outputs).
- Registers can only take values from input and output parameters, and are therefore observable: no hidden values influence the computation unnoticed. We further restrict them to storing only each parameter's last value (it could be extended to a bounded history). Note, however, that the guards and output functions can be arbitrarily complex, so this does not restrict the expressive power.

Our method also relies on some inputs and hints provided on the system, which we assume can be given, although they might be approximate. The main goal is to reduce the practical complexity of the inference. These will be shown on our example in Section 4. In particular, we assume we are given a characterization set $W \subset 2^{\mathcal{I}^+}$; and a homing sequence $h \in \mathcal{I}^*$, which also assigns a fixed value to all input parameters.

As the domains of input parameters can be infinite or very large (e.g. integers or floats), we will just use sample values in the learning process. We assume we are given (or pick) 3 levels of samples of concrete inputs $I_1 \subset I_2 \subset I_s$, with I_1 giving one concrete instance per abstract input, to infer the control machine, I_2 providing a few more concrete inputs to elicit guarded transitions, and I_s being a larger set of samples to have enough data for the generalisation process to infer the output functions and guards. No specific knowledge of the SUL is required to design those samples. We could pick either base values (e.g. for integers, 0 in I_1, plus 1 or -1 in I_2, and any further values in I_s) or just random values. The initial subsets will be extended when counterexamples provide new sample values that trigger so far unseen transitions.

We also assume we have 2 subsets of the registers $R_w \subseteq R_g \subseteq R$ such that R_g are the only registers that can be used in guards, and R_w are the registers used in guards that may be traversed when applying sequences from W from any state. These can be overapproximated with $R_w = R_g = R$ in the worst case. However, if R_w contains output registers, then the set of reachable values when applying inputs only from I_1 must be finite.

When we exercise a system, we learn *instances* of transitions with concrete values for input and output parameters. For each transition of a control machine, we may have several sample values. We collect these in a structure $\Lambda : Q \times I \times O \to 2^{\mathcal{R} \times \mathcal{I} \times \mathcal{O} \times \mathcal{R}}$, which associates abstract transitions from Δ with the observed concrete inputs and outputs and register configurations.

A *sampled FSM*, which our backbone algorithm in Section 4 will learn, is a quintuple $(Q, \mathcal{I}, \mathcal{O}, \Delta, \Lambda)$. Our generalization process will then infer an EFSM from such a sampled machine. The sampled machine for Figure 1 is shown in Figure 3.

Finally, we assume we are in the MAT framework from Angluin (1988), where an oracle can provide counterexamples: this can easily be approximated by a random walk on the inferred machine (NFSM or EFSM) where (abstract, resp. concrete) outputs can be compared with the observed responses from the SUL.

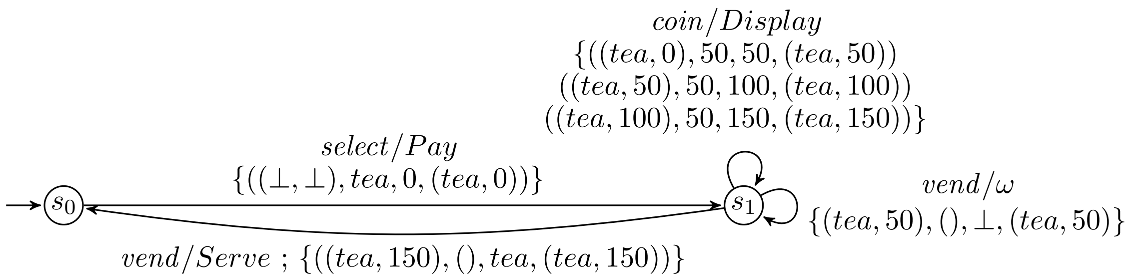


Figure 4: Sampled FSM with trace from Figure 1(*b*)

4. Execution of method on running example

At its core, our *ehW*-inference algorithm (see Annex) adapts the Mealy *hW*-inference algorithm by Groz et al. (2020), the so-called backbone, to learn the control structure of the EFSM on abstract inputs and outputs. The fact that registers can influence guards has two implications. First, from a given state, the same concrete input can produce two different abstract outputs. Second, and more importantly, characterisations (abstract responses to W) may be influenced by the register configuration (at least registers from R_w).

Therefore during the learning process, we can only conservatively infer that we are in the same state if we observe the same characterisation *from the same register configuration*. So in this new algorithm, the state space of the inferred NFSM is : $Q \subset 2^{W \to O^+} \times \mathcal{R}_w$. Another difference is that to be able to incrementally characterise a state (or learn a transition from it) we need to reach it with the same register configuration. For this reason, we require a stronger notion of homing that always resets the registers, and there is additional complexity for transferring to the next state or transition to learn (as reflected in Algorithm 3).

In our example from Figure 1(*a*), there are four observable input/output parameters (i_1 for choice of drink, i_2 for the value of the coin, t for the total amount already inserted, and b for the drink served). So there are at most 4 registers needed (let us name them i_1, i_2, b, t), and the algorithm will track the last value of each.

We pick $W = \{select(coffee)\}$. The abstract output sequence *Pay* will characterise s_0 whereas the sequence Ω (inapplicable input) characterises s_1. The only guard in our example is on input *vend*, so our W will never traverse it. But to show the robustness of the approach, let us assume we do not have this precise information, and think the choice of drink could play a role (which might be the case if drinks had different prices). So we pick $R_w = \{i_1\}$. As for other guards, let us assume we have no clue, so we pick $R_g = R = \{i_1, i_2, b, t\}$.

We pick as register homing sequence $h = coin(100).vend.select(coffee)$. Notice that for this simple example there are other shorter homing sequence, but we need one that resets the registers from R_g (the chosen h will reset values of i_1 to *coffee*, i_2 to 100 and t to 0). Notice also that this is actually a resetting sequence to s_1, but this may not be the case for more complex examples.

Finally, we pick $I_1 = \{coin(100), select(coffee), vend\}$. We always pick I_1 such that it includes all concrete inputs from h and W, so as to be able to follow transitions when walking the graph of the NFSM to transfer to the next transition to learn. And $I_2 = I_s = I_1 \cup \{coin(50), coin(200), select(tea)\}$.

$$\frac{?}{s0} \underbrace{\xrightarrow{\Omega} 1 \xrightarrow{\Omega} 2 \xrightarrow{Pay(0)} 3}_{h=coin(100).vend.select(coffee)} \frac{?}{s1} \underbrace{\xrightarrow{\Omega} 4}_{W=select(coffee)}$$

$$\frac{?}{s1} \underbrace{\xrightarrow{Display(100)} 5 \xrightarrow{Serve(coffee)} 6 \xrightarrow{Pay(0)} 7}_{h=coin(100).vend.select(coffee)} \frac{?}{s1} \underbrace{\xrightarrow{\Omega} 8}_{W=select(coffee)}$$

$$\frac{?}{s1} \underbrace{\xrightarrow{Display(100)} 9 \xrightarrow{Serve(coffee)} 10 \xrightarrow{Pay(0)} 11}_{h=coin(100).vend.select(coffee)} \frac{q_1}{s1}$$

Figure 5: Homing tails characterisations (internal SUL states shown only for reference)

Figure 5 shows how we first learn the homing tails. In steps 1-4 (steps are denoted $\rightarrow n$ in Figure 5), we learn that $H(\Omega.\Omega.Pay) = q_1 = (\Omega, \mathit{coffee})$, i.e. that after seeing the abstract output sequence $\Omega.\Omega.Pay$ the state we reach is characterised by the abstract output sequence Ω when W is executed in R_w configuration $i_1 = \mathit{coffee}$. Similarly, in steps 5-8 we learn that $H(Display.Serve.Pay) = (\Omega, \mathit{coffee}) = q_1$. After a new homing (steps 9-11), we observe a previously recorded output sequence, so we know the current state (q_1) and we can start learning its transitions.

$$\frac{q_1}{s1} \underbrace{\xrightarrow{Display(100)} 12}_{X=coin(100)} \frac{?}{s1} \underbrace{\xrightarrow{\Omega} 13}_{W} \frac{q_1}{s1} \underbrace{\xrightarrow{\Omega} 14}_{X=select(coffee)} \frac{q_1}{s1} \underbrace{\xrightarrow{Serve(coffee)} 15}_{X=vend} \frac{?}{s0} \underbrace{\xrightarrow{Pay(0)} 16}_{W} \frac{?}{s0}$$

$$\underbrace{\xrightarrow{Display(100)} 17 \xrightarrow{Serve(coffee)} 18 \xrightarrow{Pay(0)} 19}_{h} \frac{q_1}{s1} \underbrace{\xrightarrow{\omega} 20 \xrightarrow{Display(100)} 21 \xrightarrow{Serve(coffee)} 22}_{transfer=vend.coin(100).vend}$$

$$\frac{q_2}{s0} \underbrace{\xrightarrow{\Omega} 23}_{X=coin(100)} \frac{q_2}{s0} \underbrace{\xrightarrow{Pay(0)} 24}_{X=select(coffee)} \frac{?}{s1} \underbrace{\xrightarrow{\Omega} 25}_{W} \frac{q_1}{s1}$$

$$\frac{q_1}{s1} \underbrace{\xrightarrow{Display(100)} 26 \xrightarrow{Serve(coffee)} 27}_{transfer=coin(100).vend} \frac{q_2}{s0} \underbrace{\xrightarrow{\Omega} 28}_{X=vend} \frac{q_2}{s0}$$

Figure 6: NFSM learning (internal SUL states shown only for reference)

For each learnt state, we try to infer a transition using each of the concrete inputs in I_1. We start from the current state q_1 with input $coin(100)$, see steps 12-13 in Figure 6, and we discover a self loop. In step 14 we discover that input *select* is not allowed in the state, so we stay in the same state and can immediately learn the next input. Finally, in step 15 we discover that input *vend* leads to a newly discovered state $q_2 = (Pay, \mathit{coffee})$. Thus $\Delta(q_1, vend, Serve) = q_2$, and $\Lambda(q_1,\ vend,\ Serve) = ([100,\ \mathit{coffee},\ 100,\ \mathit{coffee}],\ vend,\ Serve(\mathit{coffee}), [100, \mathit{coffee}, 100, \mathit{coffee}])$. Note that after applying W from q_2 we do not know the current state in the trace (shown as "?"), so we need to home again.

At this point we try to come back to state q_2 to fully learn its transitions. First we home to a known state (steps 17-19 in Figure 6). This leads to q_1, from which we have learnt all inputs from I_1, so we follow a path in the partially inferred NFSM automaton to transfer (steps 20-22 in Figure 6) to q_2. Compared to Mealy learning, there is an additional complexity because we need to reach not just a node of the graph, but also a specific register configuration. In step 20 in Figure 6, we optimistically try the shortest path (follow transition *vend*), but that fails because we observe a new abstract output ω for the same input. Thus we learnt that *vend* from q_1 triggers a guarded transition with at least two possible abstract outputs. Since the new output is ω we know it did not change the state, so we are still in q_1 and need to transfer to q_2. To reach that goal, we need to reach the register configuration $[100, \mathit{coffee}, 100, \mathit{coffee}]$ that we know enabled *vend*. The transfer sequence of steps 21 and 22 does this. After reaching q_2, we are able to learn the transitions for input *coin*(100) (step 23) and *select*(*coffee*) (step 24). Finally we need to do a new transfer to learn transition *vend* (step 28), and we have an NFSM that is complete with respect to I_1.

$$\frac{q_2}{s0}\;\underbrace{\xrightarrow{\mathit{Pay}(0)} 29}_{\mathit{transfer}=\mathit{select}(\mathit{coffee})}\;\frac{q_1}{s1}\;\underbrace{\xrightarrow{\mathit{Display}(50)} 30}_{s=\mathit{coin}(50)}\;\frac{q_1}{s1}\;\underbrace{\xrightarrow{\mathit{Display}(250)} 31}_{s=\mathit{coin}(200)}\;\frac{q_1}{s1}$$

$$\underbrace{\xrightarrow{\mathit{Display}(350)} 32 \xrightarrow{\mathit{Serve}(coffee)} 33 \xrightarrow{\mathit{Pay}(0)} 34 \xrightarrow{\mathit{Display}(100)} 35 \xrightarrow{\mathit{Serve}(coffee)} 36}_{\mathit{transfer}=\mathit{coin}(100).\mathit{vend}.\mathit{select}(\mathit{coffee}).\mathit{coin}(100).\mathit{vend}}\;\frac{q_2}{s0}$$

$$\frac{q_2}{s0}\;\underbrace{\xrightarrow{\mathit{Pay}(0)} 37}_{s=\mathit{select}(\mathit{tea})}\;\frac{q_1}{s1}$$

$$\underbrace{\xrightarrow{\mathit{Display}(100)} 38 \xrightarrow{\mathit{Serve}(\mathit{tea})} 39}_{\mathit{CE}=\mathit{coin}(100).\mathit{vend}}$$

Figure 7: EFSM sampling (internal SUL states shown only for reference)

Once we have found the control structure (NFSM), we walk the graph to try incrementally each input from $I_s = I_1 \cup \{\mathit{coin}(50), \mathit{coin}(200), \mathit{select}(\mathit{tea})\}$ in each transition, so as to collect enough samples for the GENERALISE procedure to infer guards and output functions to build the EFSM. Notice that if we have previouosly learnt that an input is not allowed in a given state (Ω transitions), there is no need to sample further parameters for this transition.

Figure 7 shows the sampling phase for our example, for state $(\Omega, \mathit{coffee})$ we sample *coin*(50) in step 30 and *coin*(200) in step 31. For state $(\mathit{Pay}, \mathit{coffee})$ we sample *select*(*tea*) in step 37. Notice that the most complex part is to compute the transfer sequence to reach the R_g register configuration that enables a given transition. However, in the worst case, as the state was reached from some homing that resets all R_g registers, there always exists a path from some homing that leads to this previously visited state and register configuration.

In more complex examples, after sampling it is possible to identify states that are equivalent in the NFSM structure and compatible with the collected samples. These states can be merged in the EFSM, as we can infer, given the evidence, that they correspond to different register configurations of the same SUL state (this is done by the REDUCEFSM procedure).

Once the control structure (NFSM) is ascertained, we can use genetic programming to generalise from the collected samples to get the EFSM as was done in Foster et al. (2023). This is in fact simpler than in Foster et al. (2023) because, as discussed in Section 3, we know the values of the internal registers (and how they are updated) at each point in the trace. Thus, we only need to infer output functions and guards.

Notice in our example that even after sampling and generalisation, we have no evidence that the parameter of *Serve* can be anything other than *coffee*. We rely on the oracle to provide further counterexamples (see steps 38-39) that can be used to refine the functions. After step 39, the GENERALISE procedure produces exactly the EFSM of Figure 1(a) (modulo renaming).

5. A more complex example where W traverses guards

To illustrate the need for other key elements of our method, we highlight them on a slightly more complex example that entails problematic features. In Figure 8, the provided $W =$

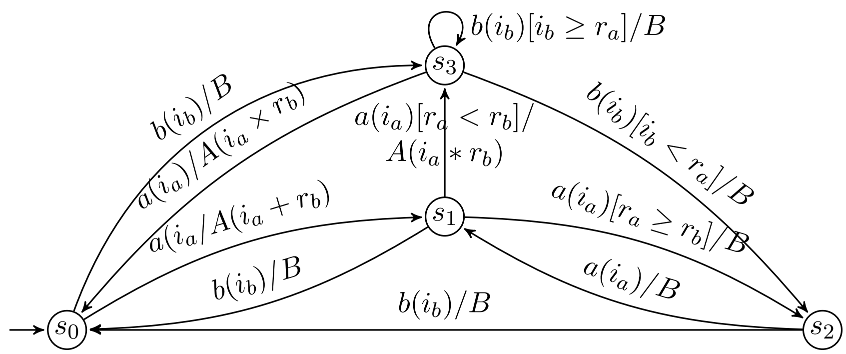


Figure 8: $W = \{b(0).a(0)\}$ from state s_3 can yield $A.B$ or $B.A$

$\{b(0).a(0)\}$ can have two outcomes when applied from state s_3: $A.B$ (if $r_a \leq 0$) or $B.A$ (otherwise); and $B.A$ is also the output for W from state s_1. Actually, $\{b(0).a(0)\}$ is not even fully characterizing, but just as hW-inference, the algorithm is relatively robust and can infer the correct structure from approximately correct h and W. There are 3 registers linked to input and output parameters, which we name r_a, r_b, r_A. $R_g = \{r_a, r_b\}$. We overapproximate $R_w = \{r_a, r_b\}$ even though in fact r_a is the only register really traversed when we know the SUL. When we apply the algorithm with $I_1 = \{a(0), b(0)\}$ and $I_2 = I_1 \cup \{a(1), b(1)\}$ there will be different states in the NFSM corresponding to state s_3, namely $(A.B, [0, 0])$, $(A.B, [0, 1])$, $(B.A, [1, 0])$, $(B.A, [1, 1])$. But we shall also have several NFSM states corresponding to a single other state, for instance $(B.B, [0, 0])$, $(B.B, [1, 0])$ for state s_0. The backbone algorithm comes up with an 11-state NFSM, but the REDUCEFSM procedure would merge redundant (equivalent) copies so we can get a sampled machine with only 4 states at step 310. This number of steps was caused by our overapproximation on R_w that led to exploring 11 states and 28 transitions. The whole inference process reached 372 steps for that machine when we added sample values $\{a(-5), b(-5)\}$ to I_s (before reducing) to get the exact data functions.

6. Conclusion

In this paper, we investigated how to infer EFSM models that include registers in guards. By allowing registers to be used in guards, the inference method presented here should be applicable to many systems. There are still a number of issues we want to investigate to consolidate it. First, we assumed here that we could be given correct h and W for the SUL. But just as in hW-inference, we can look for inconsistencies that could reveal the need to expand h or W, which could even be empty (not provided) at the start. Extending h as in hW-inference is straightforward, but there are more options for W. Another challenge is to find optimal transfer sequences. In our current preliminary implementation, we fall back on homing in all cases before transferring, and using a cached access table to states and configurations, but shorter transfers would reduce the length of the inference trace. Finally, we would like to assess the method and its scalability. It is classical to work with randomly generated machines, but for EFSMs, there are many parameters to define what a random EFSM should be. Although there are benchmarks for finite automata[1], it is not easy to find strongly connected models with parameterized I/O behaviour. And automatically assessing that the inferred model is correct can also be challenging, since in the general case, equivalence of two EFSMs is undecidable.

References

Dana Angluin. Queries and concept learning. *Machine learning*, 2(4), 1988.

T.S. Chow. Test software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

Michael Foster, Roland Groz, Catherine Oriat, Adenilso da Silva Simão, Germán Vega, and Neil Walkinshaw. Active inference of efsms without reset. In Yi Li and Sofiène Tahar, editors, *Formal Methods and Software Engineering - 24th International Conference on Formal Engineering Methods, ICFEM 2023, Brisbane, QLD, Australia, November 21-24, 2023, Proceedings*, volume 14308 of *Lecture Notes in Computer Science*, pages 29–46. Springer, 2023. doi: 10.1007/978-981-99-7584-6_3. URL https://doi.org/10.1007/978-981-99-7584-6_3.

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.

Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *FASE*, pages 80–95, 2002. URL citeseer.ist.psu.edu/hagerer02model.html.

Malte Isberner, Falk Howar, and Bernhard Steffen. Learning register automata: from languages to program structures. *Machine Learning*, 96(1), 2014.

Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *Proceedings of FORTE'99*, Beijing, China, 1999.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. lulu.com, 2008. ISBN 978-1-4092-0073-4. URL http://www.gp-field-guide.org.uk/.

[1] http://automata.cs.ru.nl/Overview#Mealybenchmarks

Algorithm 1 Homing into a known state

Function Home(T, r, h) ▷ *Apply h until we can know the state reached*
  repeat
    $(T, a, r) \leftarrow$ Apply(T, r, h) ▷ *Apply homing seq h, observe response a, update registers*
    let $\eta = \pi(a) \in O^*$ ▷ *State reached by homing is associated to η. π is abstraction*
    if *$H(\eta)$ is undefined for some $w \in W$* **then** ▷ *Learn characterization of the tail state*
      $(T, y, r) \leftarrow$ Apply(T, r, w)
      $H(\eta) \leftarrow H(\eta) \cup \{w \mapsto \pi(y)\}$
    else ▷ *We know the state reached after h/a*
      let $q = (H(\eta), \rho_w(r, \epsilon))$ be the state reached at end of h ▷ *$\rho_w(r, \sigma)$ first applies register updates from i/o sequence σ to r, then projects on R_w*
      $Q \leftarrow Q \cup \{q\}$; $A(\pi(a))(q, r) \leftarrow \epsilon$ ▷ *A records known access to configuration (q, r) from homing tail*
  until $q \neq \perp$
  return (T, q, r)

Algorithm 2 Backbone procedure with guards on registers

1 **Function** BACKBONE$(T, I_1, I_s, h, W, Q, \Delta, \Lambda)$
2 **Initializing:** $H \leftarrow \emptyset$, $J \leftarrow I_1$, $q \leftarrow \bot$, $r \leftarrow \bot$
3 **repeat**
4 **if** $q = \bot$ **then** ▷ *We do not know where we are*
5 $(T, q, r) \leftarrow$ HOME(T, r, h)
6 $(q', r', X, Y, r_1) \leftarrow$ TRANSFER$(q, r, \Delta, \Lambda, I_1, J \setminus I_1)$ ▷ *Target next transition to learn/sample*
7 **if** *such a path cannot be found* **then**
8 goto line 30 ▷ *Graph not connected, try connecting with* I_s
9 **if** $Y = \Omega$ *of* ω **then** ▷ $\pi(X)$ *is not enabled in* q', r'
10 $\Lambda(q', \pi(X), \omega) \leftarrow \{r', X, Y, r_1\}$, $\Delta(q', \pi(X), Y) \leftarrow q'$
11 $q \leftarrow q'$, $r \leftarrow r'$ ▷ *We continue learning from the same state*
12 **else** ▷ $(q', r') - X/Y \rightarrow (\mathbf{q''}, \mathbf{r_1}) \rightarrow w/\xi \rightarrow (q''', r_2)$
13 Let $q'' = \Delta(q', \pi(X), \pi(Y))$ ▷ q'' *(and so* Δ*) might be a "state under construction"*
14 **if** q'' *is fully defined* **then** ▷ *We are sampling new values of a known transition with same abstract output*
15 $\Lambda(q', \pi(X), \pi(Y)) \leftarrow \Lambda(q', \pi(X), \pi(Y)) \cup \{(r', X, Y, r_1)\}$, ▷ *Note we should not have a different* Y' *with same* r' *unless (W-)inconsistency*
16 $q \leftarrow \Delta(q', \pi(X), \pi(Y))$, $r \leftarrow r_1$
17 **else** ▷ *Learn tail of transition from* q' *on input* X
18 for some $w \notin dom(\pi_1(q''))$,
19 $(T, \xi, r_2) \leftarrow$ APPLY(T, r_1, w)
20 $\Lambda(q', \pi(X), \pi(Y)) \leftarrow \Lambda(q', \pi(X), \pi(Y)) \cup \{(r', X, Y, r_1)\}$
21 $\pi_1(q'')(w)) \leftarrow \pi(\xi)$ ▷ *This updates* Δ. π_1 *is first element of couple*
22 **if** $dom(\pi_1(q'')) = W$ **then**
23 $Q \leftarrow Q \cup \{(\pi_1(q''), \rho_w(r_1, \epsilon)\}$
24 $A \leftarrow$ UPDATEACCESS$(T, q'', \rho_w(r_1, \epsilon))$ ▷ *We record the shortest access to* (q'', r_1) *from* (q, r) *or the last homing in* T
25 $q \leftarrow \Delta^*(q', \pi(X.w), \pi(Y.\xi))$ if defined else $q \leftarrow \bot$
26 $r \leftarrow r_2$
27 **else**
28 $q \leftarrow \bot$
29 **if** $\Delta^-(\Delta, \Lambda, h, a, A)$ *(where* h/a *was latest homing in* T*) is defined and contains a complete strongly connected component (Scc)* **then** ▷ Δ^- *trims the* Δ *graph from* h/a *by cutting transitions that cannot be accessed with* A *and* Λ
30 $J \leftarrow I_s$
31 **until** Δ, Λ *are complete over* I_s *on a Scc*
32 **return** T, Q, Δ, Λ

Algorithm 3 Transferring from q, r to next transition to learn

Function Transfer$(q, r, \Delta, \Lambda, I_1, I_2)$ ▷ *We look for deterministic transfer, either through unguarded transitions, or when registers and input determine known transitions*

  ▷ *To reach a given input configuration r', we record the list of parameter values that can be set by unguarded transitions while building the path, and can adapt the parameter values at the end to set values to r'*

  ▷ *First we look for a short transfer with a bounded search ($k \geq 0$ is the bound, tailorable), and if that fails, we resort to a path from re-homing*

  Find short(-est) $\alpha \in (\mathcal{I} \times O)^k$ and $X \in I_1$ with $\Delta^*(q, \pi(\alpha)) = q'$ and $\rho_w(r, \alpha) = \pi_2(q') = \rho_w(r', \epsilon)$ s.t. $\pi_1(\Delta(q', \pi(X), *))$ is partial, ▷ *First try to learn new input from a state, in its reference configuration*

  ▷ *"partial" means either not defined at all, or there is an output whose characterisation is partial; if Ω was the output for $\pi(X)$ it is not partial regardless of r' and X; if it was ω for a given r' and X, we can still look for a different r' or X.*

  or $\exists r' \neq \pi_2(q'), Y \neq Y'$ s.t. $\rho_g(r, \alpha) = \rho_g(r', \epsilon)$ and $\{(\pi_2(q'), X, Y, *), (r', X, Y', *)\} \subset \Lambda(q', \pi(X), \pi(Y))$ and $\pi_1(\Delta(q', \pi(X), \pi(Y')))$ is partial ▷ *or a guarded transition*

  if *previous fails* **then** ▷ *(otherwise) no transition to learn in current Scc*

    Find α and $X \in I_2$ s.t. $\Lambda(q', \pi(X), *)$ does not contain $(*, X, *, *)$ ▷ *transfer to a transition to be sampled*

    $q' = \Delta^*(q, \pi(\alpha))$, $r' = \rho(r, \alpha)$

  if *all previous fails* **then** ▷ *bounded search failed, resort to homing*

    $(T, a, r) \leftarrow$ Apply(T, r, h) and update Λ on the way (if transitions are in Q)

    if $\Delta^-(\Delta, \Lambda, h, \pi(a), A)$ *no longer contains unsampled states and transitions* **then**

      return *no path found* ▷ *all transitions in reachable Scc already sampled on I_2, transfer fails*

    Look by BFS for shortest sequence α in $\Delta^-(\Delta, \Lambda, h, \pi(a), A)$, pick $A(\pi(a))(q', r') = \alpha$ and X s.t. $\pi_1(\Delta(q', \pi(X), *))$ is partial or has guarded transition to partial state, or failing that has a transition to be sampled

  ▷ *Here q', r', α, X are defined. If Δ was partial in q, then $\alpha = \epsilon$, $q' = q$*

  $(T, \beta, r') \leftarrow$ Apply(T, r, α) and update Λ on the way

  if $\beta \neq \pi_o(\alpha)$ **then** ▷ *Transfer stopped prematurely on differing output*

    let $\beta = \beta'.o'$, $\alpha = \alpha'.X.\alpha''/\beta'.o.\beta''$ s.t. $\pi_o(\alpha') = \beta', o' \neq o$

    $Y \leftarrow o', r_1 \leftarrow r', r' \leftarrow \rho(r, \alpha'), q' \leftarrow \Delta^*(q, \alpha')$ ▷ *We found a new guarded transition*

    ▷ *If h or W were not correct, we should handle inconsistency if after α' we have same input configuration r for o and o'*

  else ▷ $(q, r) - \alpha(')/\beta(') \rightarrow (\mathbf{q}', \mathbf{r}') - X/Y \rightarrow (q'', r_1) \rightarrow w/\xi \rightarrow (q''', r_2)$

    $(T, Y, r_1) \leftarrow$ Apply(T, r', X)

  $A \leftarrow$ UpdateAccess(T, q', r') ▷ *We record the shortest access to (q', r') from (q, r) or the last homing in T*

  return (q', r', X, Y, r_1)

Algorithm 4 Main ehW algorithm

Input: $I_1 \subset I_2 \subset I_s \subset \mathcal{I}$, $W \subset I_1^*$, $R_w \subset R_g \subset R$
$h \in I_1^+$, s.t. $\rho_g(\perp, h)$ has a value for each register in R_g
Initializing: $T \leftarrow \epsilon$ ▷ *T is the learning trace*
repeat
  $Q, \Delta, \Lambda \leftarrow \emptyset$, $I_o \leftarrow I_1$
  repeat
    $T, Q, \Delta, \Lambda \leftarrow$ BACKBONE$(T, I_1, I_o, h, W, Q, \Delta, \Lambda)$
    handle inconsistencies on the way to update h, W, I_1 ▷ *if h & W not trustable*
    for *I_o incrementally ranging from I_1 to $I_1 \cup R_w(I_2)$* **do**
      ▷ *Sampling on $R_w(I_2)$ first to have the smallest set of register configurations to compare, hence smallest number of redundant states*
      $T, Q, \Delta, \Lambda \leftarrow$ BACKBONE $(T, I_1, I_o, h, W, Q, \Delta, \Lambda)$
    $I_o \leftarrow I_1 \cup R_g(I_2)$
    $T, Q, \Delta, \Lambda \leftarrow$ BACKBONE $(T, I_1, I_o, h, W, Q, \Delta, \Lambda)$
    for *I_o incrementally ranging from I_1 to $I_1 \cup R_w(I_2)$ then $\cup R_g(I_2)$* **do**
      $(T, CE) \leftarrow$ GETNFSMCOUNTEREXAMPLE$(T, Q, \Delta, \Lambda, SUL, I_o)$ ▷ *Ask for a CE using only inputs from I_o*
      if *CE found* **then**
        $(W, I_1, I_s, Q, \Delta, \Lambda) \leftarrow$ PROCESSCOUNTEREXAMPLE ▷ *If W changed, this resets Q, Δ, Λ*
        ▷ *At this point, I_s would not be changed*
        continue to start of repeat BACKBONE loop
    $I_o \leftarrow I_s$ ▷ *If R_g is correct, I_s will not change NFSM, just feed* GENERALISE
  until BACKBONE *terminates with no inconsistency*
  $(Q, \Delta, \Lambda) \leftarrow$ REDUCEFSM(Q, Δ, Λ) ▷ *Reduce, not Minimize, as there is no unique minimum*
  repeat
    $M \leftarrow$ GENERALISE$(T, h, Q, I, O, P_I, P_O, \Delta, \Lambda)$
    $(T, CE) \leftarrow$ GETCOUNTEREXAMPLE(M, SUL)
  until $\neg$ *(CE is a data CE)*
  if *CE found* **then**
    $(W, I_1, I_s, Q, \Delta, \Lambda) \leftarrow$ PROCESSCOUNTEREXAMPLE ▷ *If W modified, Q, Δ, Λ are reset*
until *no counterexample found*
return M