Active Inference of Extended Finite State Models of Software Systems

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

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

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

Editors: François Coste, Faissal Ouardi and Guillaume Rabusseau

Abstract

Extended finite state machines (EFSMs) model stateful systems with internal data variables, and have many software engineering applications. It is possible to infer such models by observing system behaviour. Still, 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. We present an extension to the hW-inference algorithm that can infer EFSM models, with input and output parameters as well as guards and internal registers and their data update functions, from systems without a reliable reset. For the problem to be tractable, we require some assumptions on the observability and determinism of the system. The main restriction is that the control flow of the system must be finite, although data types could be infinite.

Keywords: Query learning, Extended Automata, Genetic Programming

1. Introduction

Accurate models of software behaviour are useful for a wide range of software engineering tasks, including checking system correctness Groce et al. (2002), identifying sequences of test inputs Choi et al. (2013), and comparing differences in behaviour between software versions Damasceno et al. (2019). 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 Lee and Yannakakis (1996).

Our research extends previous inference methods to derive EFSM models from interacting with a system without assuming it could be reset before applying each query. It is based on the combination of two methods that have been shown to be effective.

- The hW-inference algorithm Groz et al. (2018) in its preset (non-adaptive) form Groz et al. (2020). This is an active inference algorithm for FSM models that does not require resetting the system.

- The use of genetic programming methods to derive output and register update functions as well as guards, as presented in Foster (2020). This approach was developed in a passive inference context, and we interleave it with hW-inference.

The resulting method, which we call ehW-inference, incorporates the ability to infer internal registers and the constraints and functions determining how the data states within the system change in response to input values.

2. EFSM model

To illustrate our approach, we use a vending machine, modelled on Figure 1. 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 inserted, 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. This is illustrated in Figure 1.

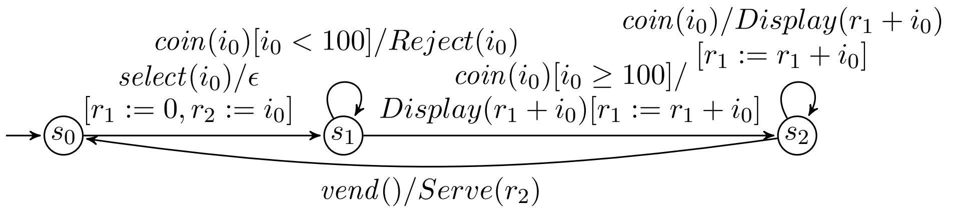
$coin(i_0)[i_0 < 100]/Reject(i_0)$
$select(i_0)/\epsilon$
$[r_1 := 0, r_2 := i_0]$
$coin(i_0)[i_0 \geq 100]/$
$Display(r_1 + i_0)[r_1 := r_1 + i_0]$
$coin(i_0)/Display(r_1 + i_0)$
$[r_1 := r_1 + i_0]$
s_0
s_1
s_2
$vend()/Serve(r_2)$

Figure 1: EFSM representing the vending machine.

An example execution is $\langle select(tea)/\epsilon, coin(50)/Reject(50), coin(100)/Display(100), coin(50)/Display(150), vend()/Serve(tea)\rangle$, in which inputs are separated from outputs by a "/" on the label of a transition. 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 can also store values in registers, which are typed variables. In Figure 1, r_1 will store the total value of coins inserted and r_2 will store the selected drink. This model is incomplete (no transition for *coin* in s_0). Still, we can complete it with self-loop transitions for inapplicable inputs (no state change): we label them with the special output symbol Ω (omitted in the graphical representation).

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.

3. Restrictions and overview of method

We assume that the system is deterministic and its control graph (represented by the automaton structure of the EFSM) is strongly connected (otherwise, we could not learn without reset). We also assume it is observable, i.e. any two distinct transitions sharing the same start state and input (type) have different output types (not just parameter values). We also assume that values assigned to registers should be visible at some point in the trace. Finally, guards cannot use registers, only input parameter values: this enforces a modelling style where stored values that play a role in the control graph are encoded in the states, so they can only take a finite number of values.

Just as hW-inference, our method discovers a homing sequence h and a characterizing set W Lee and Yannakakis (1996), except that they use fully parametric inputs such as $\mathit{coin}(50)$ but non-parametric output types such as D. The "backbone" of the algorithm proceeds initially as hW-inference with a single value for each input parameter (set I_1 of parametric inputs) to get the control structure (FSM) part of the model. Then it traverses the graph with new parameter values (sampling with a larger set I_S). In the course of sampling, it might discover new transitions if there is a guard on an input parameter. Finally, when the graph is strongly connected and has enough samples, the Genetic Programming search is started to generalise the guards, output and update functions and get an EFSM model. This EFSM model is submitted to an oracle that can return a counterexample to refine the model if needed, restarting the process with an updated h and W.

4. Inferring a Vending Machine Controller

We now illustrate the execution of ehW-inference in our example. We start with $h = \epsilon, W = \{\}, I_1 = \{\mathit{coin}(50), \mathit{select}(\mathit{tea}), \mathit{vend}\}$. As h and W are empty, we initially learn a single state automaton with each input X from I_1.

$$(s_0, \langle\rangle) \underbrace{\quad}_{h=\epsilon} \underbrace{\quad}_{w=\{\}} \underbrace{\xrightarrow{\mathit{coin}(50)/\Omega}{}_1}_{X=\mathit{coin}(50)} (s_0, \langle\rangle) \underbrace{\quad}_{w=\{\}} \underbrace{\quad}_{h=\epsilon} \underbrace{\xrightarrow{\mathit{select}(\mathit{tea})/\epsilon}{}_2}_{X=\mathit{select}(\mathit{tea})} (s_1, \langle \mathit{tea}, 0\rangle) \underbrace{\quad}_{w=\{\}}$$

$$\underbrace{\quad}_{h=\epsilon} \underbrace{\xrightarrow{\mathit{vend}/\Omega}{}_3}_{X=\mathit{vend}} (s_1, \langle \mathit{tea}, 0\rangle) \underbrace{\quad}_{w=\{\}} \underbrace{\xrightarrow{\mathit{coin}(100)/D(100)}{}_4}_{\mathit{sampling}} (s_2, \langle \mathit{tea}, 100\rangle)$$

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

$$(s_2, \langle \mathit{tea}, 100\rangle) \underbrace{\xrightarrow{\mathit{coin}(100)/D(200)}{}_5}_{h} (s_2, \langle \mathit{tea}, 200\rangle) \underbrace{\xrightarrow{\mathit{coin}(100)/D(300)}{}_6}_{w} (s_2, \langle \mathit{tea}, 300\rangle)$$

We now know that applying h with response D leads to q_0, characterized by $\mathit{coin}(100) \mapsto D$, but we have not yet learnt Δ (next state function) for $(q_0, \mathit{coin}(100))$, so we need to home again before proceeding. This leaves us still in q_0, so we can now learn a transition, with $\alpha = \epsilon$ (no transfer needed), and we use $X = \mathit{coin}(100)$ 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 *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 two-state EFSM model that will be passed to the GP generalisation. 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 $(s_2, \langle coffee, 100\rangle)$, so will respond with $D(150)$ whereas the conjecture would respond $R(50)$. Since output types D and R differ we add $coin(50)$ to W and restart the backbone with $h = coin(100)$ and $W = \{coin(100), coin(50)\}$.

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.

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

As before, we can then apply the generalisation procedure to infer a full 3-state EFSM model. The guard differentiating the two *coin* transitions is rather simplistic: $i_0 = 50$ vs $i_0 \neq 50$. Because of this, our oracle is able to return the counterexample $coin(20)/R(20)$ (step 68). This brings a new input parameter, 20, into play.

Looking at this counterexample, it differs only in terms of its data values, and there is no h or W non-determinism. We need only rerun our generalisation procedure on the newly extended trace. This gives the same model as before, but with guards $i_0 \leq 50$ and $i_0 > 50$, and the output $R(i_0)$ instead of $R(50)$. Given the domain of the input to *coin*, this is equivalent to the model in Figure 1, since there is no input between 50 and 100. Thus, we were able to learn an accurate model of our vending EFSM in just 68 steps.

5. Conclusion

We have developed a novel approach able to infer rich EFSM models of software systems without resetting. We are now trying to assess its performance on systems (models) of varying size and complexity.

Acknowledgments

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).

References

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.

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.

Michael Foster. *Reverse Engineering Systems to Identify Flaws and Understand Behaviour*. PhD thesis, University of Sheffield, September 2020. URL `https://etheses.whiterose.ac.uk/28568/`.

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.

Roland Groz, Nicolas Bremond, and Adenilso Simao. Using adaptive sequences for learning non-resettable fsms. In *ICGI 2018*, pages 30–43, Wroclaw, September 2018.

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.

D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8), 1996. doi: 10.1109/5.533956.