



Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/177494/>

Version: Accepted Version

---

**Proceedings Paper:**

Foster, M., Derrick, J. and Walkinshaw, N. (2022) Reverse-engineering EFSMs with data dependencies. In: Testing Software and Systems. ICTSS 2021 : The 33rd IFIP International Conference on Testing Software and Systems, 10-11 Nov 2021, Virtual conference. Lecture Notes in Computer Science. Springer Nature, pp. 37-54. ISBN: 9783031046728. ISSN: 0302-9743. EISSN: 1611-3349.

[https://doi.org/10.1007/978-3-031-04673-5\\_3](https://doi.org/10.1007/978-3-031-04673-5_3)

---

This is a post-peer-review, pre-copyedit version of a paper published in Testing Software and Systems, Proceedings. The final authenticated version is available online at: [http://dx.doi.org/10.1007/978-3-031-04673-5\\_3](http://dx.doi.org/10.1007/978-3-031-04673-5_3).

**Reuse**

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

**Takedown**

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing [eprints@whiterose.ac.uk](mailto:eprints@whiterose.ac.uk) including the URL of the record and the reason for the withdrawal request.

# Reverse-Engineering EFSMs with Data Dependencies<sup>\*</sup>

Michael Foster<sup>1</sup>[0000-0001-8233-9873], John Derrick<sup>1</sup>[0000-0002-6631-8914], and  
Neil Walkinshaw<sup>1</sup>[0000-0003-2134-6548]

Department of Computer Science, The University of Sheffield  
Regent Court, Sheffield, S1 4DP, UK  
{m.foster, j.derrick, n.walkinshaw}@sheffield.ac.uk

**Abstract.** EFSMs provide a way to model systems with internal data variables. In situations where they do not already exist, we need to infer them from system behaviour. A key challenge here is inferring the functions which relate inputs, outputs, and internal variables. Existing approaches either work with white-box traces, which expose variable values, or rely upon the user to provide heuristics to recognise and generalise particular data-usage patterns. This paper presents a preprocessing technique for the inference process which generalises the concrete values from the traces into symbolic functions which calculate output from input, even when this depends on values not present in the original traces. Our results show that our technique leads to more accurate models than are produced by the current state-of-the-art and that somewhat accurate models can still be inferred even when the output of particular transitions depends on values not present in the original traces.

**Keywords:** EFSM Inference · Model Inference · Genetic Programming

## 1 Introduction

Reactive systems – systems that respond to their environment, their users, or other systems – are commonly modelled as Finite State Machines (FSMs). These offer an intuitive basis upon which to model and reason about the sequential behaviours of a wide range of systems from network communication protocols to GUIs, and form the foundation of many verification and testing techniques [24]. Reactive systems that incorporate data (where computation requires a memory, or where data can be supplied and received through inputs and outputs) can be represented as Extended Finite State Machines (EFSMs) [14]. Despite their utility, models can be neglected due to the pressures of system development.

The challenge of reverse-engineering FSMs and EFSMs has been the subject of a considerable amount of research. Where the field of FSM inference is mature and has produced many powerful approaches [11,22,38], current techniques to infer EFSMs tend to suffer from a variety of drawbacks. Some approaches

---

<sup>\*</sup> Michael Foster and Neil Walkinshaw are funded by the EPSRC CITCoM project.

produce results that are only partial, in that they do not infer *how* data variables change throughout execution [26,39], or lack internal data variables entirely [12,34]. Those approaches that do infer fully-fledged EFSMs are limited either in terms of their practical applicability [6,10,13,18,35], or accuracy [37].

Inferring accurate, complete EFSMs is particularly challenging when the update functions have interdependencies; when a function on one transition depends on a value computed by another transition. Empirical work by Androutsopoulos *et al.* [7] suggests that these are widespread, arising in around a third of the transitions in the models that they studied. Current inference approaches, such as MINT [37], cannot handle such interdependencies because they infer transition functions on an individual basis, without considering relationships to other transitions. A further problem in the case of MINT is that it is incapable of inferring variables that are not explicitly part of the execution trace. This means that it is not a truly black-box technique. Finally, its update functions are only inferred *after* the transition structure of the machine has been decided, which is often too late because the underlying structural inference algorithm (which is largely data-insensitive) can end up merging transitions together that should remain separate because they should have different update behaviours [22].

In this work we present a technique that addresses these limitations. The key contributions of this paper are as follows:

- An approach to infer update functions *before* any structural state machine inference has taken place, instead of afterwards, so that transitions with different update functions that should remain separate can be kept apart.
- An approach based on Genetic Programming to infer hidden variables (as part of update inference) using values observed in other transition update functions. This captures interdependencies *between* transitions, enabling the inference of state machines that are more precise than the state-of-the-art.
- An openly available [16] proof-of-concept implementation, along with the full experimental data-set and scripts.
- A small empirical study that assesses the accuracy of our approach in comparison to the state-of-the-art, with respect to two systems.

The rest of this paper is structured as follows. Section 2 introduces a motivating example and gives some necessary background. Section 3 explains the details of our technique, the implementation of which is discussed in Section 4. Section 5 evaluates our technique experimentally. Finally, Section 6 concludes the paper and discusses possible future work.

## 2 Background

This section defines EFSMs and traces, and gives an overview of the current state-of-the-art in EFSM inference. We then highlight the limitations of existing techniques. Throughout this work, we draw from a toy example of a simple vending machine. Users first *select* a drink. They then insert *coins*, with the total balance being displayed as output on a small screen. Once sufficient payment has been inserted, the machine *vends* the selected drink.

## 2.1 Definitions

**Traces.** As systems execute, we can record the sequence of actions performed, along with any inputs and return values. Figure 1 shows some traces of our simple drinks machine. In our notation,  $coin(50)/[100]$  represents the event  $coin$  being called with the input 50 and outputting 100. We delimit events with commas and omit the outputs from events like  $select("tea")$  which do not produce any.

$$\begin{aligned} &\langle select("tea"), coin(50)/[50], coin(50)/[100], vend()/["tea"] \rangle \\ &\langle select("tea"), coin(100)/[100], vend()/["tea"] \rangle \\ &\langle select("coffee"), coin(50)/[50], coin(50)/[100], vend()/["coffee"] \rangle \end{aligned}$$

Fig. 1: Exemplary traces of the vending machine.

**EFSMs.** An EFSM is a conventional FSM that has been extended to explicitly model how a system handles data. While there are many different EFSM representations in the literature [14,25], our technique is designed to work with the inference process from [18], so we use that definition [17,18,19,20].

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

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

This differs from the traditional EFSM definition [14] in several ways. In [14], transitions take one literal input, produce one literal output. Our definition assigns each transition an explicit label and allows multiple inputs and outputs (or none at all). Transitions may also produce outputs as a function of input and register values, which allows transition behaviour to be *generalised*.

Definition 1 technically only affords each transition one guard, output, and update, but syntactic sugar allows a transition from state  $q_m$  to  $q_n$  to take the form  $q_m \xrightarrow{label.arity[g_1, \dots, g_g] / f_1, \dots, f_f [u_1, \dots, u_u]} q_n$  in which guards  $g_1, \dots, g_g$  are implicitly conjoined, output functions  $f_1, \dots, f_f$  are evaluated to produce a list of outputs, and update functions  $u_1, \dots, u_u$  are executed simultaneously. We use this notation throughout this work, for example in Figure 3.

## 2.2 Genetic Programming

The technique we present in Section 3 uses Genetic Programming (GP) [21] to infer expressions which relate sets of input-output pairs from the traces. We therefore provide a brief introduction to the essential notions of GP that we use in this work. For a more comprehensive overview, we refer the reader to [31].

In (tree-based) GP, candidate functions are represented as syntax trees in which branch nodes represent operators (“non-terminals”), and leaf-nodes represent variables and constants (“terminals”). GP is an approach to synthesise these functions by evolution. The basic loop is as follows and iterates for a fixed number of *generations* or until we find a function with optimal *fitness*.

1. Generate an initial population of random functions.
2. Evaluate each expression according to some *fitness function*.
3. Select the best individuals to continue to the next generation.
4. Create a new population by a process of crossover and mutation.
5. Repeat from step 2 until some stopping criterion is met.

The most important aspect of this for our purposes is the *fitness function*. This provides a metric for the suitability of candidate functions. Fitness is evaluated by executing each candidate function on all available inputs and then comparing the resulting set of outputs to the corresponding outputs in the trace data. For numerical values, the fitness function is taken as the average distance between the predicted and the actual values. For nominal outputs, the fitness is calculated as the proportion of instances where the outputs were correct.

Another key step in the algorithm is the creation of a new population by *crossover* and *mutation*. Crossover recombines desirable characteristics from individuals in the population. Mutation simulates the small changes in DNA which occur during natural reproduction, allowing us to introduce new characteristics.

## 2.3 EFSM Inference

Model inference enables us to make statements about the overall behaviour of a system by generalising from its *traces*. A popular way to do this [18,26,39] is to convert the traces into a tree-shaped model called a prefix tree acceptor (PTA) like in Figure 2. States in the PTA which are believed to represent the same program state are then merged, resulting in a smaller and more general model.

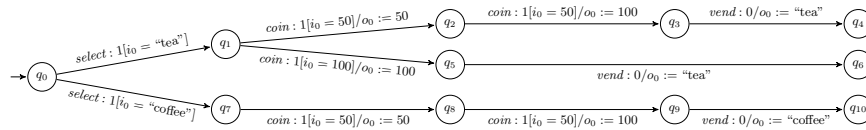


Fig. 2: The PTA of the traces in Figure 1.

As well as inferring the control flow, we also want to infer the functions that transform inputs into outputs. For example, in Figure 2, the output of each *vend*

event is the input of *select*, and the output of each *coin* event is a running total of the inputs. To express such behaviour, we must use the internal data state of the model. The EFSM in Figure 3 uses a register,  $r_1$ , to keep track of the inputs to *coin*, and uses a second register,  $r_2$ , to store the input of *select* for later use as the output of *vend*. These registers affect the behaviour of the model, but do not appear in its traces — they are *latent*.

## 2.4 Limitations of Existing Approaches

Figure 3 shows the ideal EFSM model of the drinks machine, but there are currently no techniques in the literature which can infer this effectively from the traces in Figure 1. A major obstacle to overcome is that registers  $r_1$  and  $r_2$  are latent variables, so their existence and usage must be inferred. One technique [18] allows users to provide *data abstraction heuristics* to facilitate this. To provide these heuristics, the user requires a prior understanding of the system, which means that this technique cannot be applied to any realistic inference scenario.

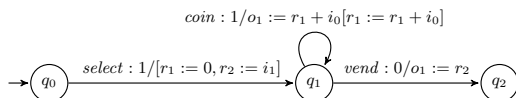


Fig. 3: The EFSM representing the traces in Figure 1.

MINT [37] is an alternative approach which uses GP to infer update functions for variables. This is done as a postprocessing step for existing models. Having inferred a model from a set of traces, the first stage of postprocessing is to execute the model on these traces. For each transition, the anterior and posterior variable values are recorded. These are then used as the inputs and outputs for GP to evolve individual update functions for each variable of each transition.

Figure 4 shows a model MINT might infer of the traces in Figure 1. Crucially, the *longitudinal dependency* between *vend* and *select* is missed. There are two reasons for this. Firstly, MINT infers data updates per-transition, so cannot discover relationships between different transitions. Secondly, a variable is required to store the input to *select* for later reuse. Figure 3 uses  $r_2$  for this, but MINT only considers variables that appear in the traces, so has no way to facilitate the relationship. The technique we present in Section 3 overcomes these limitations to enable Figure 3 to be effectively inferred from the traces in Figure 1.

This work tackles the problem of *passive* inference — inferring a model using only the traces provided — but there is also much literature on *active* inference [8]. Here, the learner asks questions about the system under inference of the form “Is this trace acceptable behaviour?”. These are often answered by running the proffered traces directly on the system under inference. There are many active EFSM inference techniques [5,6,10,13,35], but these do not support arithmetic operations in data updates, only simple assignments. Functions that update

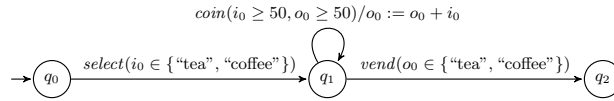


Fig. 4: An EFSM of the traces in Figure 1 as might be inferred by MINT. MINT has no notion of outputs, so  $o_0$  here represents an internal register.

registers in terms of their anterior values, such as the *coin* transition in Figure 3, are beyond them. Another limitation of all active learners is the requirement to run arbitrary traces to answer queries, which may not always be viable.

Another group of approaches [12,33] rephrase 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.

### 3 Inferring Output and Update Functions

This paper addresses the challenge of inferring EFSMs from truly black-box systems where we cannot inspect the internal state or ask arbitrary queries. Instead, we must reason about the system purely in terms of the observable behaviour recorded in a fixed set of its traces. The key challenge here is to infer the necessary internal variables that enable us to capture functionality where there is a dependence on some input data that might have been provided several steps previously, without relying on the visibility of the internal data state.

Our approach works by inferring the key internal variables and the functions that update them during an execution. This allows for “longitudinal” dependencies, where an input is provided at one point (e.g. the user selects a drink), and referenced several steps later in the machine (e.g. the machine dispenses the drink, but only after the user has paid for it). As established by Androutsopoulos *et al.* [7], such dependencies are common in EFSM specifications.

To infer these functions accurately, we cannot adopt the approach of existing techniques such as MINT, which infer the transition functions as a postprocessing step *after* the transition structure has been inferred. The process of state merging leads to a loss of information which is vital to track these longitudinal dependencies. Hence, the situation in Figure 4, where the inferred model allows for the undesirable situation where a user selects tea but receives coffee.

To avoid this information loss, our approach operates as a *pre*processing technique. We take advantage of the detailed trace-by-trace information in the PTA before it is merged, inferring internal variables and associated update functions directly from the prefix tree. We therefore retain flexibility as our approach does not impose any restrictions on the state merging algorithm that is subsequently used to infer the model structure. Our approach tackles three interdependent inference challenges: (1) the functions to compute output from transition inputs, (2) the registers needed to support this, and (3) the functions required to update register values to ensure they hold the correct values when evaluated.

**Algorithm 1:** Outline of our GP preprocessing technique.

---

```

Input: A set of traces  $T$ 
Output: A prefix tree  $pta$ 
// Generate a PTA from the traces using the conventional approach.
1  $pta \leftarrow \text{BUILDPTA}(T)$ ;
/* Group transitions by their structure (label and arity) and history (to
   restrict the inference challenge for each group to the same context). */
2  $groups \leftarrow \text{GROUPTRANSITIONS}(pta)$ ;
3 for  $g_1 \in groups$  do
    /* Use GP to infer functions that accurately predict outputs for group,
       including the ability to infer the presence of memory registers that
       can be presumed to contain any missing values if required. */
4      $fun \leftarrow \text{INFEROUTPUTFUN}(g_1)$ ;
    // Replace literal outputs with inferred functions.
5      $newPTA \leftarrow \text{REPLACELITWITHFUN}(pta, g_1, fun)$ ;
    // Infer updates to the inferred memory registers.
6     for  $r_n \in fun.latentVars$  do
7         for  $g_2 \in groups$  do
8              $newPTA \leftarrow \text{INFERUPDATEFUNS}(g_2, \text{TARGETVALUES}(pta, r_n))$ ;
    // Check that inferred functions are compatible with traces.
9     if  $\text{ACCEPTS}(newPTA, T)$  then
10         $pta \leftarrow newPTA$ ;
    /* Combine functions for transitions that were put into separate groups
       because they had different histories (line 2). */
11     $pta \leftarrow \text{STANDARDISE}(pta)$ ;
12  $pta \leftarrow \text{DROPGUARDS}(pta)$ ;
13 return  $\text{RESOLVENONDETERMINISM}(pta)$ ;

```

---

Algorithm 1 outlines our technique. We first group related transitions in the PTA together (line 2). We then infer output and update functions for each group using GP. This works in two steps. In the first step (line 4), the GP infers functions to compute output from input and identifies the use of registers if required (addressing challenges 1 and 2 above). In the second step (lines 6-8) it ensures that these registers are correctly updated by other transitions in the PTA before they are evaluated. The rest of this section elucidates the process.

*GroupTransitions* (line 2) forms groups of transitions that represent the same behaviour. Transitions are grouped together if they have the same *structure*, i.e. the same label, arity, and produce the same number and types of outputs. The PTA in Figure 2 has three structural transition groups: *select*, *coin*, and *vend*.

Latent variables can lead to side effects [7] such that transitions with the same structure may be subject to different data states depending on where in a trace they occur. To provide a degree of uniformity for the GP, we only place transitions into the same group if they share the same *history*.

To account for contiguous blocks of the same event, we cannot simply look at the previous transition. Instead, we look backwards in time until we find one which is *structurally different*. For example, the most recent structurally different transition of all the *coin* transitions in Figure 2 is *select*. Consecutive structurally identical transitions (like  $q_1 \xrightarrow{\text{coin}} q_2 \xrightarrow{\text{coin}} q_3$  in Figure 2) represent the same behaviour so have consistent side effects. By contrast, if our simple drinks machine had a *refund* event to reset the balance to zero, *coins* inserted after this event would be grouped separately to those that follow *select*.

*InferOutputFun* (line 4) takes a set of input/output pairs and uses GP to infer a function to relate them. The key challenge here is getting the GP to work with *latent variables*; registers like  $r_1$  and  $r_2$  which are absent from the traces. As mentioned in Section 2, this is not something which MINT can do. To infer Figure 3 from the PTA in Figure 2, we need to be able to do this.

To introduce new registers, we simply add them to the set of variables used by the GP, but this causes a problem. As discussed in Section 2, the fitness of candidate functions is assessed by executing them on the inputs from the traces. Unfortunately, register values do not appear in the traces, so we cannot evaluate functions involving them. Our solution is to look to the inputs and outputs in the traces, and assign each register the value that yields the closest output to the target. The justification for this is that register values are usually either set to a particular input or observed as an output. Full details can be found in [17].

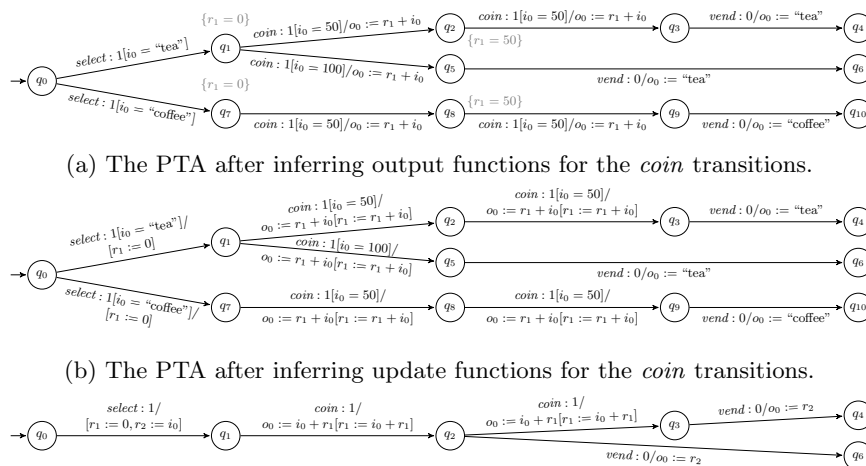
Latent variables give the GP a lot of freedom when evolving expressions, so we want to minimise their use. We expect transitions like *coin* to use their non-latent inputs as part of the output, so want to find an expression involving them if we can. Thus, we penalise the fitness of expressions which use latent variables without using all the non-latent ones. In situations where ignoring inputs is the correct solution, expressions cannot achieve optimal fitness but, since our GP has a set maximum number of generations, this will not stop it from terminating.

To further limit the use of latent variables, we first call the GP without them. If this fails, we add one latent register to the set of variables and run GP again. If either attempt is successful, `REPLACELITWITHFUN` (line 5) replaces the literal outputs of the transition group with the inferred function. For example in Figure 2, the output behaviour of the *coin* transitions generalises to  $i_0 + r_1$ . Replacing the concrete outputs with this function gives Figure 5a. If the GP fails both attempts, we keep the literal outputs from the PTA. We could continue adding registers until the GP succeeds, but we here choose to stop after one.

*Update Function Inference* (lines 7-9). To ensure that registers introduced by `INFEROUTPUTFUN` hold the correct values when evaluated, we walk each trace in the PTA annotating each state with target register values, as illustrated in Figure 5a. These are propagated backwards so every state in the prefix path has a target value. This is what allows  $r_2$  in Figure 3 to be initialised by *select*. Without it, registers could only be initialised immediately prior to use, which would not allow us to discover longitudinal relationships between transitions.

Starting at the root of the PTA, we call GP again (without latent variables) for each group. The “inputs” are the transition input values and the anterior register value, if defined. The “output” is the target register value. For example, in  $q_2$  and  $q_8$  of Figure 5a, we need  $r_1$  to hold the value 50. The input to the respective incident *coin* transitions is 50, and the anterior value of  $r_1$  must have been zero. Thus,  $r_1 := r_1 + i_0$  works as possible update, as shown in Figure 5b.

*Accepts* (lines 9-10). After inferring output and update functions for a transition group, we check to ensure that the new PTA still accepts the original traces and produces the correct outputs. If not, we must reject our inferred functions and default back to the literal outputs from the PTA for that particular group.



(c) The PTA after inferring output and update functions for all transitions, dropping guards, and resolving nondeterminism.

Fig. 5: Preprocessing the PTA in Figure 2.

*Standardise (line 11)* takes a PTA and attempts to “standardise” output and update functions between transitions with the same structure that were grouped separately due to their histories. For example, in our *refunding* vending machine from earlier, we want our two groups of *coin* transitions to have the same output and update functions. The full details of this process can be found in [17].

*Generalisation (lines 12-13)*. Currently, the model has symbolic output and update functions, but each transition still has its literal input guards. We want our final model to be more responsive, so we drop these guards at this stage. This can introduce nondeterminism, which is undesirable in a PTA as trace prefixes no longer necessarily share a common path. In fact, this nondeterminism is simply an indication that the model contains duplicated behaviour and is easily resolved by merging states and transitions [18]. This results in Figure 5c. This is smaller than Figure 2 as the top and bottom branches are “zipped” together.

Having processed the PTA, we then perform the conventional state merging process [18] to produce Figure 3. This perfectly models the drinks machine, capturing data dependencies using internal registers. To the best of our knowledge, this is the first technique to infer such relationships using only system traces.

## 4 Implementation

We built our implementation on two frameworks. For the GP component, we significantly enhanced the GP implementation used for MINT [37]. For the underlying PTA, we built upon our Isabelle/HOL state-merging framework [16,18]. This section provides details of these enhancements and adaptations.

## 4.1 Genetic Programming

The original GP implementation [37] follows the basic steps outlined in Subsection 2.2. An initial population is first generated by randomly combining terminals and non-terminals<sup>1</sup> to form syntactically valid expressions. These are then evolved through crossover and mutation, with only the best surviving to the next generation. Our main addition was a fitness function to enable latent registers to be introduced, as discussed in Section 3. In addition, several other changes were necessary to improve the performance of the GP in this new context.

The mutation operator used in [37] simply replaces a random node with a new random subtree, but we found that this failed to produce satisfactory outcomes. We created a richer set of mutation operators inspired by a different open-source GP implementation [1] which offers more scope for useful mutations during evolution (details in [17]). To further enhance the impact of mutation operators, we also took inspiration from Doerr *et al.* [15] and apply up to three mutations in sequence as making occasional large changes to individuals has been shown to help escape local optima and avoid premature convergence.

Another implementational issue we faced was *bloat* [23]. While [37] applies some basic simplification to expressions, it still yielded more complex expressions than were desirable, often including redundant operations like `+0`. To mitigate this, we used Z3 [30] to simplify our expressions. This can reveal semantic duplicates in the population, which become identical when simplified. We replace these duplicates with new random individuals to keep the population distinct and diverse. To further manage bloat, we also use *lexicographic parsimony pressure* [27] to break ties in fitness, favouring smaller expressions over larger ones.

## 4.2 PTA Preprocessing

Our technique is designed to work with the inference tool from [18], which is implemented in Isabelle/HOL with executable Scala code exported using Isabelle’s code generator. To incorporate algorithm 1, we defined the functions in Isabelle and then automatically generated the Scala code using the code generator. While [18] uses Isabelle to verify certain aspects of transition merging, we here use Isabelle purely for compatibility reasons. Rather than formalising our GP in Isabelle, we specified `INFEROUTPUTFUN` and `INFERUPDATEFUN` abstractly and hooked their Scala counterparts into our Java implementation from Subsection 4.1 using a thin wrapper function.

The implementation for this work comes to around 1000 lines of Java on top of [37] to implement our GP (`INFEROUTPUTFUN` and `INFERUPDATEFUN`) and an additional 551 lines of Isabelle code (translating to 2010 lines of automatically generated Scala code) and 496 lines of manually written Scala code on top of [18] to implement the rest of algorithm 1. All of this code is available at [16].

---

<sup>1</sup> The inference tool we use here [18] currently supports only `+`, `-`, and `×` for integers, and literal assignment for strings, although our GP has broader support [16,37].

## 5 Evaluation

This section describes a small experiment where we compare our approach against MINT [37] (the current state-of-the-art of passive EFSM inference). For our technique to be successful, we want to infer models which can correctly predict system outputs for unseen traces. We also want our technique to be robust to data values being absent from the traces. Our research questions are as follows.

**RQ1** Does the processing of the PTA by our technique prior to state merging lead to more accurate models than the current state-of-the-art?

**RQ2** How robust is our technique to latent variables?

### 5.1 Methodology

**Metrics.** Both our RQs are concerned with model accuracy. To evaluate this, we use one set of traces (the *training set*) to infer a model and then use another set of traces (the *test set*) to compute various accuracy metrics. In this evaluation, we use the following two metrics, in which the *accepted prefix* is the first part of the trace, where the outputs produced by the model match those of the system.

$$\text{Sensitivity} = \frac{\text{number of accepted positive traces}}{\text{total number of positive traces}} = \frac{\text{Accepted}}{\text{prefix length}} = \frac{\text{length of accepted prefix}}{\text{length of trace}}$$

Sensitivity is the proportion of positive traces in the test set accepted by a model. This is often paired with *specificity*, which is the proportion of negative traces rejected by the model. Here though, we are more concerned with whether our models correctly calculate the output values in response to the given inputs than whether it can correctly classify traces as positive or negative. Our models produce outputs in response to inputs, so traces are only accepted if the correct outputs are produced. Thus, there is much less risk of overgeneralisation here, making specificity an inappropriate metric for this evaluation.

**Subject systems.** To illustrate the performance of our technique, we evaluate it on the two published models summarised below. The first is a lift door controller published in [32] and used in [17,37] to evaluate inference tools. The second system [2] is a Java accompaniment to [28] based on the game Space Invaders.

System	States	Variables	Transitions	Traces/Events
LIFTDOORS [17,37,32]	6	<i>timer</i>	10	348/9333
SPACEINVADERS [17]	4	<i>x, aliens, shields</i>	7	100/2580

Our work is motivated by the fact that existing EFSM inference approaches do not consider the possibility of internal variables which do not appear explicitly as transition inputs. Thus, we chose our subject systems for their use of these variables, and the *relationships* between data values used by different transitions, rather than for the complexity of any individual function. We also chose systems which differ in terms of the number of state variables as this is identified in [37] as being a factor which has a significant effect on the accuracy of MINT.

To an extent, the values above mask the complexity of the two systems. LIFTDOORS has only one system variable, but this is shared between and modified by every transition. Despite the heavy data dependencies of LIFTDOORS, it is SPACEINVADERS which is the more complex case study. There are three state

variables here, and the system is much more reactive. All but one of the transitions in this model emanate from the same state, giving a much greater variation in the traces produced by this system. By contrast, LIFTDOORS has only one or two transitions from each state, so the traces are more constrained.

**RQ1 (assessing accuracy).** This RQ asks whether our *preprocessing* approach infers more accurate models than the *postprocessing* used by MINT. This work focuses on *latent variables* which do not appear in traces, allowing us to infer models from traces that only contain information observable from outside the system without probing the internal program state. MINT, however, is not applicable to this scenario. To compare it to our technique, we must work with traces where the output of each transition depends only on its input.

To evaluate the accuracy of our models, we followed the standard procedure of creating a training set and a test set, where the former is used to infer a model and the latter is used to compare the predictions made by the model to the ground truth. For LIFTDOORS, we used the same traces [3] as [37]. For SPACEINVADERS, we modified the code to log certain actions [17] and obtained traces by manually playing the game. For each system, we then took random samples of 60 traces and divided them in half to form the training and test sets, each of 30 traces. These are available online alongside our implementation [16].

The accuracy of our inferred models depends on the selection of training traces and the random seed passed to the GP. To control for these we ran the inference tools with 30 seeds each for 30 sets of traces. Thus, we inferred a total of 900 models for each technique. As well as the random seed, our GP technique has a number of configurable parameters. These are the population size,  $\mu$ , the number of new individuals per generation,  $\lambda$ , and the number of generations. Here, we use  $\mu = 100$ ,  $\lambda = 10$ , and 100 generations. MINT has a similar set of configurable parameters, all of which we left at their default values.

We anticipate that both techniques will perform well here but that our technique will outperform MINT. Because we infer output and update functions prior to merging, they play an active role in the inference process and help to shape the structure of the final model. By contrast MINT infers functions after state merging, when this structure has already been determined. MINT also requires every event to report the value of every variable, so can be led astray by superfluous information. Our technique does not require this, so is more targetted.

**RQ2 (assessing robustness to missing variables).** This RQ investigates how robust our technique is to variables being absent from the traces. Such variables indicate potential dependencies between the data values of different transitions. To investigate this, we took the training and test sets we used for RQ1 and elided one input at a time. For SPACEINVADERS, we also elided combinations of two variables. Thus, we are no longer in the purely functional domain: the output of certain events depends on values which are missing from the traces. The main challenge here is inferring the correct use of internal registers as part of the output functions, and then inferring suitable updates to facilitate this.

MINT has no notion of hidden variables, so is simply not applicable to systems where we cannot inspect the internal data state during execution. Thus,

we must evaluate our technique in isolation. We anticipate that obfuscating variables will lead to a drop in the accuracy of the models produced by our technique since the GP has less information to guide it and must use latent variables in expressions, which gives it much more freedom to produce esoteric functions which do not properly generalise. It must also infer update functions in addition to output functions, which gives it extra opportunities to make mistakes.

## 5.2 Results and Discussion

The raw data from all of our experiments is available online [4]. The distributions of accuracy values, in terms of sensitivity and accepted prefix-lengths, are shown in Figure 6. We will proceed to answer both RQs in terms of these box-plots.

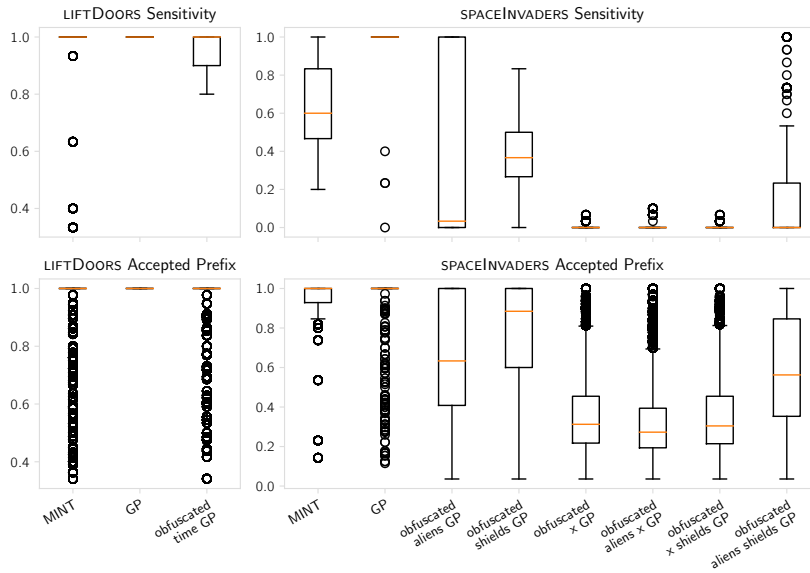


Fig. 6: Accuracy metrics for the two systems.

**RQ1 (assessing accuracy).** This RQ concerns the GP and MINT plots. Figure 6 shows that our technique (GP) achieves a perfectly accurate model in all but a few outlying cases of `SPACEINVADERS`. MINT performs comparably for `LIFTDOORS` but only achieves a median sensitivity of 0.6 for `SPACEINVADERS`.

These results are not surprising. When we preprocess with GP, we generalise concrete data values from the traces to symbolic functions. This is not a particularly difficult task in the purely functional scenario, and our GP is correct in all but a few outlying cases of `SPACEINVADERS`. This then enables many states to be merged, leading to a very accurate model. While MINT also uses GP to infer functions, it does so *after* the structure of the model has been inferred. It

also tries to infer transition guards during inference to aggregate the observed data values (where our technique simply discards them). This is a particular problem for systems like SPACEINVADERS with multiple variables as these often cause MINT to infer spurious guards, leading to an inaccurate model structure.

**RQ2 (assessing robustness to missing variables).** This RQ concerns all plots except MINT. Since MINT has no notion of latent variables, it is simply not applicable here. Figure 6 shows that obfuscating the system timer for LIFTDOORS has a relatively small effect on the accuracy of the models inferred by our system, but that the effect of obfuscation is much greater for SPACEINVADERS.

Again, this is not surprising. The two contributing factors here are the two calls to GP detailed in Section 3. Here, we must use latent variables in the output functions as the result depends on variables absent from the traces. This gives the GP much more freedom when inferring functions, so it is more likely to be incorrect. We also need to infer update functions for each latent variable. This was not necessary in RQ1, so there is an extra opportunity to make mistakes.

For both subject systems, the main cause of inaccuracy is a failure to *recognise* events rather than a failure to adequately calculate output from input. This is because, if our GP makes a mistake or fails to come up with a function, the dropping of transition guards in the generalisation step is detrimental to state merging. More states must remain separate and share the underlying functionality between them. This leads to models that are both larger and less reactive.

For SPACEINVADERS, the variable we obfuscate has a huge effect on the accuracy of the model. The *aliens* and *shields* variables do relatively well when obfuscated, but the  $x$  variable leads to very poor models. There are two reasons for this. Firstly, most events in the traces are movement events, which depend on  $x$ , so any mistake with these is given much more opportunity to reveal itself. Secondly, because our technique introduces one register per transition group and there are two movement events (left and right) which both mutate  $x$ , our technique struggles to work out what is going on here.

**Discussion.** While Figure 6 shows that our technique infers more *accurate* models than MINT, it does not show what these models actually look like. Figure 7 shows a model of SPACEINVADERS inferred by our technique in the purely functional setting of RQ1. This model concisely shows the behaviour of the system. By contrast, most of the models inferred by MINT are too large to effectively display here. The same is true for LIFTDOORS. Where our technique drops the guards on transitions before state merging, MINT tries to infer guards to aggregate the observed data values. These are often overly specific, which leads to cluttered and chaotic models, even if they are accurate in terms of traces.

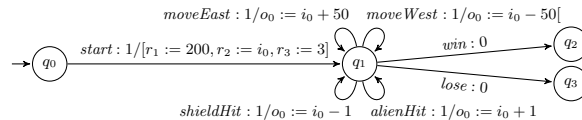


Fig. 7: A model of SPACEINVADERS inferred by our technique.

### 5.3 Threats to Validity

This evaluation cannot be used to (and does not aim to) draw general conclusions about the accuracy or scalability of either our technique or MINT. Our main aim here is illustrate each technique performs “out of the box”, its applicability, and factors which affect model accuracy. There are, however, certain aspects of the study that must be taken into account when reviewing the results.

**Choice of systems.** For this study we used two fully specified EFSMs. Although these present us with valuable insights here, it will require a larger, more diverse selection of systems to produce more generalisable results.

**Selection of parameters.** We did not spend time optimising the configuration parameters used by either our technique or MINT. This avoids the threat of overfitting values to these subject systems, biasing the results in favour of either technique, but opens up the threat that there may be more suitable configurations. A more specific selection of parameters may lead to more accurate results, but parameter optimisation falls outside the intended scope of this investigation.

## 6 Conclusion

This paper presented a GP-based technique to infer functions that relate inputs, outputs, and internal variables of EFSM models. We use this as part of preprocessing step for the inference process to generalise the initial PTA before merging states. To the best of our knowledge, this is the first technique to do this in a truly black-box setting. Our results indicate that our technique leads to more accurate models than those inferred by MINT [37], the current state-of-the-art.

A key aspect of our technique is the ability to infer output functions involving variables which do not appear in the traces, and update functions to ensure these variables hold the correct values when evaluated. While eliding variables from the traces reduces the accuracy of the models we can infer, our technique still improves upon MINT, which cannot be applied at all in this scenario.

There are many applications of GP [21], but [37] is the only work which applies it to EFSM inference. Work in [9] applies similar techniques to learn *feature models*, but these do not model control flow. Work in [10,18] considers latent variables but relies on simple heuristics, which limits applicability. Active techniques such as [6,13,35] build on the  $L^*$  algorithm [8] to infer EFSMs with data updates, but these techniques rely on submitting queries about the system under inference. Our technique is entirely passive, using only on the traces provided. The field of process mining [36] has also produced various techniques to infer models from traces. The main focus, though, is on control flow rather than data usage. Research carried out in [29] considers the data perspective, but does not attempt to infer models which can predict system behaviour for new traces.

One possible line of future work is to increase the set of operations for the GP, including the ability to handle floating-point numbers. This would make our technique applicable to a broader range of systems. Another line of work would be a more comprehensive evaluation involving more systems, which would enable us to draw more general conclusions about accuracy and scalability.

## References

1. <https://github.com/lagodiuk/genetic-programming>, accessed 2020-02-03
2. [http://www.doc.ic.ac.uk/~jnm/book/book\\_applets/concurrency/invaders](http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency/invaders), accessed 2020-05-15
3. <http://www.cs.le.ac.uk/people/nw91/Files/ICSMEData.zip>, accessed 2020-04-15
4. <https://doi.org/10.15131/shef.data.15172969>
5. Aarts, F.: Tomte : Bridging the gap between active learning and real-world systems. Ph.D. thesis, Radboud University Nijmegen (2014)
6. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.: Automata learning through counterexample guided abstraction refinement. In: FM 2012: Formal Methods. pp. 10–27. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
7. Androutsopoulos, K., Gold, N., Harman, M., Li, Z., Tratt, L.: A theoretical and empirical study of efsm dependence. In: 2009 IEEE International Conference on Software Maintenance. pp. 287–296 (2009)
8. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* **75**(2), 87–106 (1987)
9. Arcaini, P., Gargantini, A., Radavelli, M.: Achieving change requirements of feature models by an evolutionary approach. *Journal of Systems and Software* **150**, 64–76 (2019)
10. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines using domains with equality tests. In: Fundamental Approaches to Software Engineering. vol. 4961 LNCS, pp. 317–331. Springer Berlin Heidelberg (2008)
11. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers* **C-21**(6), 592–597 (1972)
12. Buzhinsky, I., Vyatkin, V.: Automatic inference of finite-state plant models from traces and temporal properties. *IEEE Transactions on Industrial Informatics* **13**(4), 1521–1530 (2017)
13. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Learning extended finite state machines. In: Software Engineering and Formal Methods, pp. 250–264. Springer, Cham (2014)
14. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Proceedings of the 30th International Design Automation Conference. pp. 86–91. ACM Press (1993)
15. Doerr, B., Le, H.P., Makhmara, R., Nguyen, T.D.: Fast genetic algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference. p. 777–784. Association for Computing Machinery (2017)
16. Foster, M.: EFSM inference (2020), <https://github.com/jmafoster1/efsm-inference>
17. Foster, M.: Reverse Engineering Systems to Identify Flaws and Understand Behaviour. Ph.D. thesis, The University Of Sheffield (2020)
18. Foster, M., Brucker, A.D., Taylor, R., North, S., Derrick, J.: Incorporating data into EFSM inference. In: Software Engineering and Formal Methods. pp. 257–272. Springer International Publishing (2019)
19. Foster, M., Brucker, A.D., Taylor, R.G., Derrick, J.: A formal model of extended finite state machines. *Archive of Formal Proofs* (2020), [https://isa-afp.org/entries/Extended\\_Finite\\_State\\_Machines.html](https://isa-afp.org/entries/Extended_Finite_State_Machines.html), Formal proof development
20. Foster, M., Taylor, R., Brucker, A.D., Derrick, J.: Formalising extended finite state machine transition merging. In: Formal Methods and Software Engineering. pp. 373–387. Springer International Publishing (2018)

21. Koza, J.R.: Genetic programming: On the programming of computers by means of natural selection. MIT Press (1992)
22. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In: Grammatical Inference. pp. 1–12. Springer Berlin Heidelberg (1998)
23. Langdon, W.B.: Quadratic bloat in genetic programming. In: Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation. p. 451–458. GECCO'00, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2000)
24. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proceedings of the IEEE **84**(8), 1090–1123 (1996)
25. Lorenzoli, D., Mariani, L., Pezzè, M.: Inferring state-based behavior models. In: Proceedings of the 2006 international workshop on Dynamic systems analysis. p. 25. ACM Press (2006)
26. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 13th international conference on Software engineering. p. 501. ACM Press (2008)
27. Luke, S., Panait, L.: Lexicographic parsimony pressure. In: Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation. pp. 829–836. Morgan Kaufmann Publishers Inc. (2002)
28. Magee, J., Kramer, J.: State models and Java programs. Wiley Hoboken, 2<sup>nd</sup> edn. (2006)
29. Mannhardt, F.: Multi-perspective process mining. Ph.D. thesis, TU Eindhoven (2018)
30. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer Berlin Heidelberg (2008)
31. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. <http://www.gp-field-guide.org.uk> (2008)
32. Strobl, F., Wisspeintner, A.: Specification of an elevator control system. Tech. rep., TUM (1999), <https://www.broy.in.tum.de/publ/papers/elevator.pdf>
33. Ulyantsev, V., Tsarev, F.: Extended finite-state machine induction using sat-solver. In: 2011 10th International Conference on Machine Learning and Applications and Workshops. vol. 2, pp. 346–349 (2011)
34. Ulyantsev, V., Buzhinsky, I., Shalyto, A.: Exact finite-state machine identification from scenarios and temporal properties. Int J Softw Tools Technol Transfer **20**(1), 35–55 (2016)
35. Vaandrager, F., Midya, A.: A Myhill-Nerode theorem for register automata and symbolic trace languages. In: Theoretical Aspects of Computing. pp. 43–63. Springer International Publishing (2020)
36. Van Der Aalst, W.: Process mining. Commun. ACM **55**(8), 76–83 (2012)
37. Walkinshaw, N., Hall, M.: Inferring computational state machine models from program executions. In: 2016 IEEE International Conference on Software Maintenance and Evolution. pp. 122–132. IEEE (2016)
38. Walkinshaw, N., Lambeau, B., Damas, C., Bogdanov, K., Dupont, P.: STAMINA: A competition to encourage the development and assessment of software model inference techniques. Empirical Software Engineering **18**(4), 791–824 (2013)
39. Walkinshaw, N., Taylor, R., Derrick, J.: Inferring extended finite state machine models from software executions. Empirical Software Engineering **21**(3), 811–853 (2016)