Reverse Engineering Systems to Identify Flaws and Understand Behaviour

John Michael Anthony Foster

The University of Sheffield

Faculty of Engineering
Department of Computer Science

September 2020

A thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

Abstract

Accurate system models are applicable to many software engineering tasks. Despite their utility, models are often neglected during development. It is therefore desirable to reverse engineer them from existing systems. One way to do this is to record traces of the system and infer a model by generalising from this behaviour.

Unfortunately, the models inferred by current techniques often cannot represent how the data values associated with each action affect system behaviour. This raises the following questions. What kind of model do we need in order to show the interplay between behaviour and data? How can we infer such models from system traces? How can we infer functions to relate input data with subsequent outputs? How can we use our models once they have been inferred?

To answer these questions, the first contribution of this thesis is a new model definition designed to show the relationship between data and behaviour. Secondly, I present a technique to infer such models from system traces, and define a preprocessing step to infer functions that relate system inputs and outputs. I then empirically evaluate the models produced by my technique and compare them to those produced by a state-of-the-art tool. Finally, I show how the inferred models can be used to analyse properties of the systems they represent.

The results show that my technique infers models which are more accurate and intuitive than the current state of the art. My tool can also handle circumstances where the output of a system depends on data values not present in the traces, and can identify situations where the result of particular actions depends on specific data values. The models inferred by my tool can be used by existing verification tools to prove and refute properties of the underlying systems.

Acknowledgements

First and foremost, I would like to thank my supervisors, John Derrick and Achim Brucker for their advice and support throughout the PhD. Thanks must also go to Neil Walkinshaw, Siobhán North, Georg Struth, Andrei Popescu, and Dmitri Traytel for being so generous with their time and knowledge; to Graeme Smith and Ian Hayes for their hospitality in Australia; and to Ramsay Taylor, who inspired me to do a PhD in the first place.

I would also like to thank the members of the Verification and Testing groups at the University of Sheffield — especially Ibrahim Althomali, David Paterson, Tom White, Jonathan Julián Huerta y Munive, Abdullah Alsharif, George O'Brien, and Ben Clegg, as well as honorary members Alasdair Warwicker and George Hall — not only for their knowledge and encouragement, but also for making lunches and pub trips so enjoyable.

Finally, I want to thank my parents for their unconditional love and support which has enabled me to get this far, and the friends I've made in Sheffield along the way. While they make no direct contribution to this thesis, they have certainly made its creation more bearable.

Publications

Conference Papers

Michael Foster, Ramsay Taylor, Achim D. Brucker, and John Derrick. Formalising extended finite state machine transition merging. In *Formal Methods and Software Engineering*, pages 373–387. Springer International Publishing, 2018

Michael Foster, Achim D. Brucker, Ramsay Taylor, Siobhán North, and John Derrick. Incorporating data into EFSM inference. In *Software Engineering and Formal Methods*, pages 257–272. Springer International Publishing, 2019

Proof Artefacts

Michael Foster, Ramsay Taylor, Achim D. Brucker, and John Derrick. A formal model of extended finite state machines. *Archive of Formal Proofs*, 2020 (Accessed 19/09/2020). `http://isa-afp.org/entries/Extended_Finite_State_Machines.html`, Formal proof development

Michael Foster, Ramsay Taylor, Achim D. Brucker, and John Derrick. Inference of extended finite state machines. *Archive of Formal Proofs*, 2020 (Accessed 19/09/2020). `http://isa-afp.org/entries/Extended_Finite_State_Machine_Inference.html`, Formal proof development

Works in Progress

Ramsay Taylor, Michael Foster, and Siobhán North. Automatically verifying or refuting trace properties of extended finite state machines. *DRAFT*

Michael Foster, Neil Walkinshaw, and John Derrick. Using genetic programming to infer output and update functions for efsm transitions. *DRAFT*

Contents

Chapter 1
Introduction

Computer software is becoming increasingly important in people's lives. Because of this, there is an increasing demand for systems to be properly tested and, in some cases, formally verified. Accurate models of software behaviour are a valuable tool in this process. For example, they can be used to automatically generate test suites [68], and can act as oracles for regression testing [59]. For critical applications, a model may be a mandatory requirement to obtain certification.

There are numerous ways in which software can be modelled, and the required formality and level of detail varies greatly depending on the intended use. For example, a model of an aircraft autopilot system used for safety certification clearly needs to be much more comprehensive than a model designed to give an overview of a simple database management system to a client. The former may use specialist tools such as Simulink and X-Plane [126], where simple hand-drawn diagrams and natural language documents may be sufficient for the latter.

Formal models use notations with mathematically defined syntax and semantics. Such models are useful for reasoning about properties of systems, and can facilitate the use of automated verification tools. There are many formal modelling techniques available, including Z [134], VDM [62], and state machines. Informal models are less well-defined and may take the form of natural language documents or simple diagrams. Such models are generally used to give an overview of a system to non-technical stakeholders.

Despite their value, models are often neglected during development. This means that there is a lot of software for which no model exists. It is therefore desirable to be able to *reverse engineer* models from existing systems. There are numerous techniques in the literature to help with this [10, 16, 106, 152], the main idea being to observe how the system behaves and then make *generalisations* from these observations.

Automated inference techniques often produce some form of *finite state machine* model. These come in numerous flavours with differing levels of detail and computational power. At the bottom end, we can simply model what actions the software performs in which order. To add more detail, we can also include in our model any inputs and outputs that are associated with the various actions. Of course, the input to a particular piece of software often *determines* the output. The pinnacle of both detail and computational power is to model the exact transformation from input to output.

1.1 Motivating Example

To motivate this work, consider a simple vending machine which dispenses drinks. Customers first select the drink they desire, and then insert coins to pay for it. A running total of their value is displayed to the user on a small screen. Once sufficient payment has been inserted, the customer can dispense their drink by pressing the *vend* button. If the customer presses *vend* before they have inserted enough money, nothing is dispensed. Let us assume, for this example, that all drinks have the same price of one pound.

We would like a model of this system. Unfortunately, the original developers are unavailable and we do not have access to the source code. We must therefore try to infer a model of the system by observing its behaviour. As the machine operates, it records the actions it performs with their associated inputs and outputs. Sequences of such actions are commonly referred to as *traces.* Some traces of the drinks machine are shown in Figure 1.1. Here, actions have a *label*, which correspond to method names, and *inputs,* which correspond to method arguments. They may also produce observable *outputs* which represent method return values, or other externally observable behaviour such as text on a screen.

In Figure 6.1, I use the notation $\mathit{methodName}(i_1, i_2, \ldots)/[o_1, o_2, \ldots]$ such that $\mathit{coin}(50)/[100]$ represents the event *coin* being called with a single input of 50 and producing a single output of 100 which, in this case, represents the screen displaying this value. The *select* events produce no observable output, as is often the case with real vending machines. In the traces, events are delimited by commas, with individual traces being enclosed in angle brackets. Traces will generally take this form throughout this work.

$$\langle \mathit{select}(\text{“tea”}), \mathit{coin}(50)/[50], \mathit{coin}(50)/[100], \mathit{vend}()/[\text{“tea”}]\rangle$$
$$\langle \mathit{select}(\text{“tea”}), \mathit{coin}(100)/[100], \mathit{vend}()/[\text{“tea”}]\rangle$$
$$\langle \mathit{select}(\text{“coffee”}), \mathit{coin}(50)/[50], \mathit{coin}(50)/[100], \mathit{vend}()/[\text{“coffee”}]\rangle$$
$$\langle \mathit{select}(\text{“soup”}), \mathit{vend}(), \mathit{coin}(100)/[100], \mathit{vend}()/[\text{“soup”}]\rangle$$

Figure 1.1: Some sample traces of the simple vending machine.

If we have a sufficiently large sample of such traces, we can make generalisations about the behaviour of the system and infer a model which is able to *predict* how the system might behave when faced with new input sequences. Many existing inference techniques [10, 16, 98] infer finite state machine (FSM) models which cannot handle inputs or outputs. To infer such a model from the traces in Figure 1.1, we must either remove the data entirely or encode it within the actions by folding input and output values into the transition labels. Taking the latter approach to retain as much information as possible, we represent an event $\mathit{label}(i_0)/[o_0]$ as the atomic action $\mathit{label}\_i_0\_o_0$. The transformed traces are shown in Figure 1.2.

$$\langle \mathit{select\_tea}, \mathit{coin\_50\_50}, \mathit{coin\_50\_100}, \mathit{vend\_tea}\rangle$$
$$\langle \mathit{select\_tea}, \mathit{coin\_100\_100}, \mathit{vend\_tea}\rangle$$
$$\langle \mathit{select\_coffee}, \mathit{coin\_50\_50}, \mathit{coin\_50\_100}, \mathit{vend\_coffee}\rangle$$
$$\langle \mathit{select\_soup}, \mathit{vend}, \mathit{coin\_100\_100}, \mathit{vend\_soup}\rangle$$

Figure 1.2: Some sample traces of the simple vending machine.

The traces in Figure 1.2 can be transformed into a model which directly represents them. This is shown in Figure 1.3 but is not a very good model of the system for a number of reasons. Firstly, it is only able to handle the traces in Figure 1.2, so fails to meet our objective of *predicting* system behaviour for new traces. Secondly, the model is larger than it needs to be. Thinking about how a real drinks machine works, once we have selected tea, it should not matter whether we pay for it with two 50p coins or a single one pound coin. The resulting state should be the same. Thus, states q_7 and q_8 in the model actually represent the same state in the underlying system. This is a key concept in FSM inference and is discussed more in Subsection 3.4.2. An optimal FSM model which could be inferred from the traces in Figure 1.1 is shown in Figure 1.4.

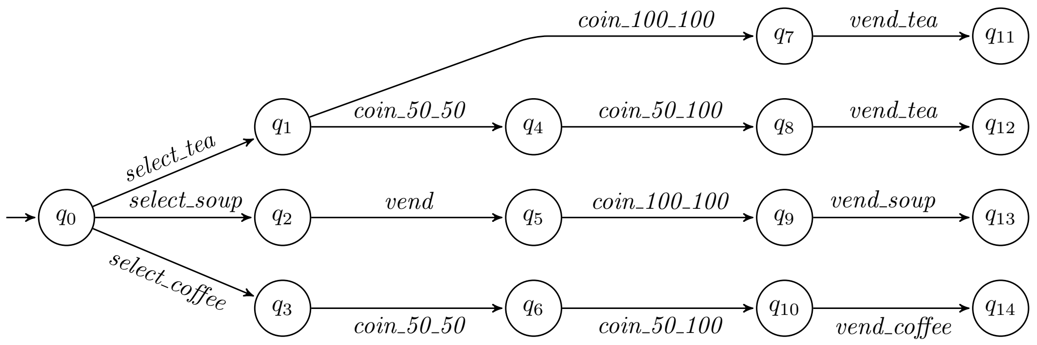
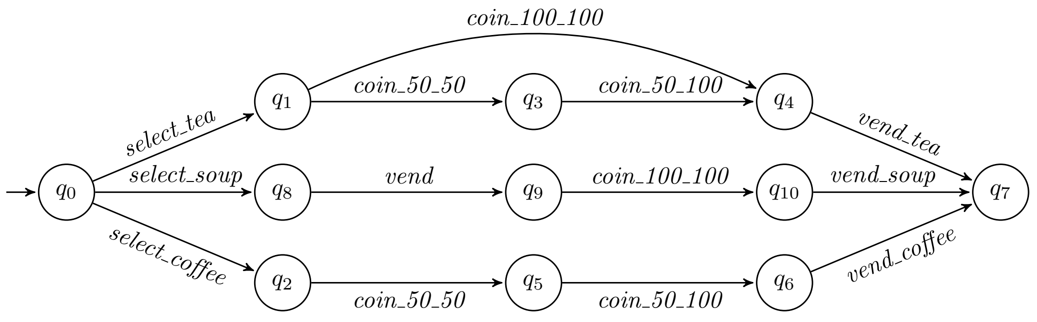


Figure 1.3: A model representing exactly the traces in Figure 1.2.

While the model in Figure 1.4 is smaller than the one in Figure 1.3, it is still a poor model of the system. In order to maintain the fact that the user receives the same drink they select, each drink must have its own distinct path through the system. This means that we must have three paths (one for each drink) which all represent the same top-level behaviour, i.e. selecting a drink, paying for it, and then receiving it. This is because we are unable to separate the idea of *control flow* — what events happen in which order — from *data* — the exact values flowing around the system. We are therefore forced to encode data within the control flow of the system, which makes our model a lot larger than we would like. It also means that adding new data to the system, for example an additional drink, causes an increase in the size and complexity of the model which is disproportionately large compared to the change in behaviour.

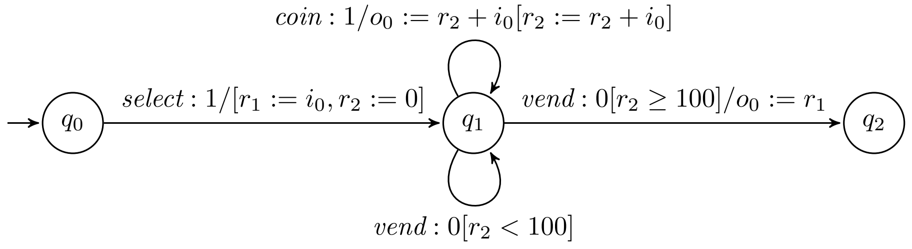


I want to believe

Figure 1.4: A classical FSM model of a simple drinks machine as could be inferred from the traces in Figure 1.1.

What we would really like is a model which can separate control flow from data. One such model is shown in Figure 1.5. This is an extended finite state machine (EFSM) model as detailed in Chapter 4. A detailed explanation of the syntax of transitions can be found in Chapter 4. For now, it surfaces to say that a transition from anterior state q_m to posterior state q_n has the general form $q_m \xrightarrow{\textit{label}:\textit{arity}\text{[guards]/outputs[updates]}} q_n$.

In Figure 1.5, the customer calls the *select* with an input which represents the drink they wish to purchase. This value is assigned to a *register*, r_1, and a second register, r_2, is initialised to zero. The model moves the model into state q_1, which represents a drink having been selected. From here, the customer may then either insert coins or trigger the *vend* action.



$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select : 1/[r_1 := i_0, r_2 := 0]$
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_0
q_1
q_2
$vend : 0[r_2 < 100]$

Figure 1.5: An EFSM model of the drinks machine.

The *coin* transition takes one input (which represents the value of the inserted coin in pence) and produces one output (which represents the machine displaying its current credit) which is assigned the value of r_2 plus the input. r_2 is then updated to the same value to keep track of it.

Both *vend* transitions have a guard on r_2. Here, there are two possible scenarios. Either the customer has inserted sufficient payment or they have not. If r_2 is less than 100, the customer has not yet paid enough so cannot receive a drink. If it is greater than or equal to 100, the customer has paid sufficiently, so can receive their drink. In the latter case, the model moves into state q_2, which represents a completed interaction.

There are several challenges we must overcome to infer a model like the one in Figure 1.5 from the traces in Figure 1.1. Firstly, the model in Figure 1.5 has *states* which must be identified. This challenge has been tackled in a number of ways in classical FSM inference but is made more complex here by the fact that EFSMs have registers which enable us to arbitrarily move information between control flow states and registers.

To infer a model by merging states, we must determine which states we wish to merge. This decision is usually made based on their outgoing transitions. Successful inference by state merging also relies on the merging of transitions which represent the same behaviour. The challenge with EFSMs is that transitions which express the same behaviour may not be exactly identical. We therefore need a way to determine when and how to merge such transitions.

Another key challenge is to *generalise* the literal inputs and outputs from the traces into functions that relate input and output. For example, the *coin* transition in Figure 1.5 has the output function $r_2 + i_0$. Each of the traces is a concrete *instance* of this behaviour. We need a way to automatically infer the function from the traces. This challenge is compounded by the fact that the model in Figure 1.5 makes use of *registers* which do not show up in the traces. If we are to successfully infer output functions, we must also identify when and how to use registers.

1.2 Purpose of the Work

System models have many uses including model-based testing [68, 139], to detect cyber attacks [143], and to aid the process of requirements capture [40]. Models can also help to provide an intuitive understanding of complex systems. While this project is mainly concerned with *how* to infer models rather than *what* they will be used for, it is important to keep in mind some notion of the intended applicability such that my techniques can be effectively evaluated.

For the purposes of this work, I aim to formulate tools and techniques which can produce human-readable models, specifically ones which give an understanding of how data is transformed throughout model execution. With such models, it is then possible to carry out verification activities such as model checking in order to verify (or refute) certain properties of the underlying systems as detailed in Chapter 9. Moreover, the work in this thesis takes place in a fully passive black-box scenario. That is, models are inferred entirely from traces provided at the start of inference without examining the inner workings of the system under inference or consulting an oracle.

1.3 Thesis Aims and Objectives

This thesis aims to advance the field of model inference from traces, specifically the inference of *extended finite state machines.* Thus, the thesis aims to answer the high level research question "What strategies can we apply to automatically infer extended finite state machine models from black-box traces?". The thesis answers this question by addressing the previously mentioned challenges with the following main objectives.

- To bring together desirable characteristics from the various existing EFSM definitions in the literature to form a new definition which is well suited to inference.
- To establish a technique which can be used to determine whether one EFSM transition can account for the behaviour of another such that they can be merged.
- To establish a state merging technique to infer extended finite state machine models, including functions to relate inputs and outputs and mutate the data state, from black-box software execution traces.
- To evaluate this technique with respect to a baseline and the current state of the art.
- To establish techniques to aid the process of the verification of properties of EFSM models once they have been inferred.

1.4 Contributions of the Thesis

C1: The formulation of a new EFSM definition that combines desirable characteristics from the literature – A new formal definition of EFSMs which incorporates the desirable aspects from various existing definitions into a single model definition which is also formalised in Isabelle/HOL (Chapter 4).
C2: The *subsumption in context* and *direct subsumption* relations – The formalisation of the *subsumption in context* relation which enables us to tell whether one EFSM transition accounts for the behaviour of another under certain circumstances, and the *direct subsumption* relation on top of this to tell whether one transition in a given EFSM accounts for another in a different model (Chapter 5).
C3: The use of direct subsumption to formulate a state merging algorithm for EFSM inference – A state merging technique to infer computational EFSM models from black-box execution traces, which uses the direct subsumption relation as a basis (Chapter 6).

C4: A technique to infer functions which relate inputs, outputs, and registers as well as transition guards – A technique to infer, from black-box traces, the output functions which relate inputs and outputs, including the use of internal registers. This technique can also be used to infer guards to distinguish transitions with value-dependent behaviour (Chapter 7).
C5: An empirical evaluation of my inference technique – An empirical investigation into the performance of my inference technique in comparison to a baseline approach and the current state of the art (Chapter 8).
C6: A framework to allow model checking and theorem proving to be used in tandem to prove properties of EFSMs – A framework of function definitions to aid the verification of models once they have been inferred (Chapter 9).

1.5 Thesis Structure

This thesis is structured as follows.

Chapter 2 begins with an overview of the different aspects of software execution which we can record. The chapter goes on to discuss the various FSM models that exist in the literature, as well as introducing a few alternative modelling techniques for completeness. Finally, the chapter discusses the field of refinement, and how this relates to model inference.

Chapter 3 provides background relating directly to model inference from traces. The chapter first introduces the inference challenge before outlining how the basic state merging approach works for classical FSMs and discussing the various techniques in the literature. Next, the chapter provides an extensive review of the state of the art of EFSM inference before discussing the limitations of existing techniques and gaps in the literature.

Chapter 4 presents my novel definition of EFSMs which combines desirable characteristics from various existing definitions in the literature. This definition is formalised in Isabelle/HOL to serve as a foundation for subsequent work and to allow various key properties to be proven.

Chapter 5 discusses how *contexts* are used to record the values of registers at various points during the execution of an EFSM model. The *subsumption in context* relation is defined as a way to determine whether one transition can account for the behaviour of another, given a particular register valuation. The chapter goes on to define the *direct subsumption* relation on top of this, which works at EFSM level to determine whether it is safe to merge a given pair of transitions. Finally, I briefly discuss the necessity of contexts when analysing system properties.

Chapter 6 uses the direct subsumption relation from Chapter 5 as the foundation of a state merging technique to infer EFSM models from black-box traces. As part of this technique, simple *heuristics* are used to recognise certain data usage patterns and abstract away concrete values in favour of generalised functions that *compute* output from input. This technique is implemented as a prototype tool using the Isabelle formalisation from Chapter 4 as a basis.

Chapter 7 presents a more general approach to infer output functions which is free of the limitations of the heuristics used in Chapter 6. I first introduce the concepts behind genetic programming before using these ideas as part of a technique to infer functions which relate inputs and outputs, and recognise when and how registers need to be used.

Chapter 8 provides a comprehensive empirical evaluation of my inference tool in the context of several case studies. The performance of my tool is compared to a baseline approach and to the current state of the art of EFSM inference.

Chapter 9 is concerned with the verification of models once they have been inferred. The chapter first briefly introduces the field of formal verification before discussing linear temporal logic (LTL) and how it can be used to specify properties of models. The chapter then goes on to discuss how Isabelle/HOL can be used to prove properties of EFSMs specified in LTL, and presents my framework of function definitions designed to make this process easier. Next, the chapter discusses how we can use a model checking tool to help us find counterexamples to untrue properties, and why this is useful. Finally, the chapter demonstrates the use of this in the context of two case studies.

Chapter 10 concludes the thesis with summaries of the contributions of each chapter, limitations, and ideas for possible future research directions.

Chapter 2

Background I - Finite State Machines

This chapter formally introduces events and traces, and defines the key concept of *prefix closure*. It also provides a gentle introduction to finite state machine models and, for completeness, briefly summarises some other models of computation. Finally, the field of *refinement* is introduced in the context of finite state machines.

2.1 Events and Traces

If we are to infer a model of a program, we must first observe how it behaves. For a completely black-box system, the only behaviour we can model is that which is observable to the outside world. We can record this observable behaviour in the form of a *trace* of program execution. A program trace lists the sequence of actions a program performs during its execution. Collections of such traces are commonly referred to as *logs* and can be very useful to system administrators for debugging and detecting malicious use of software systems. As will be shown in Chapter 3, program logs can also be used to infer models of system behaviour. This section provides a brief introduction to the theory of *traces*, originating from [110], and shows how this can be applied to real systems to produce observations from which we can infer a model.

2.1.1 Definitions

Finite sequences of symbols over a given alphabet Σ are referred to as *strings.* Elements of Σ can be thought of as observable *actions* which a system can perform, such that strings represent possible executions of the system. The set of all strings over Σ is denoted Σ^*. Subsets of Σ^* are referred to as *languages.* We can then form an intuitive definition of a *trace.* Before doing so, it is helpful to introduce some additional terminology which will be used throughout this work.

Definition 1. An *action* is an activity which can be performed by a system. *Inputs* (sometimes referred to as *arguments*) are additional data parameters provided to actions. *Outputs* are data values which are produced as a result of performing an action. An *event* is a triple made up of an action with its inputs and outputs.

Example 2.1.1. To clarify Definition 1, recall the motivating example from Section 1.1. The first trace contained an event *coin*(50)/[100]. Here, the action is *coin.* There is a single input of 50 and a single output of 100. The second trace contains an event *vend().* Here, there are no inputs or outputs so the action *vend* is itself an event. Throughout this work, I refer to such events as *atomic.*

Definition 2 provides a working definition of *traces.* Concurrent systems require a more complex definition, as in [110], but only sequential systems are considered in this work. Some examples of traces can be seen in Figure 1.1.

Definition 2. A *trace* of a system is a sequence of actions in its alphabet, Σ, together with any associated inputs or outputs.

For reasons which will become clear in Chapter 3, it is helpful for us to define an ordering on traces such that we can say whether one trace is "less than" another. To do this, we must first define the concatenation operation such that we can sequentially join traces together.

Definition 3. The *concatenation* of two traces t_1 and t_2, denoted $t_1 \cdot t_2$, is defined as the "append" operator on lists, in which # is the construction (or *Cons*) operator.

$$\langle\rangle \cdot y = y$$
$$\langle x \# xs \rangle \cdot y = x \# (xs \cdot y)$$

Example 2.1.2. Consider trace $t_1 = \langle a, b, c\rangle$ and $t_2 = \langle c, d, e\rangle$. Their concatenation $t_1 \cdot t_2 = \langle a, b, c, c, d, e\rangle$.

Having defined the concatenation operation for traces, we can then define a partial order relation as follows, which calls a trace t_1 "less than" another trace t_2 iff t_1 is a *prefix* of t_2.

$$t_1 \sqsubseteq t_2 \overset{\text{def}}{\iff} \exists \sigma \in \Sigma^*.\, t_1 \cdot \sigma = t_2$$

This leads us on nicely to the notion of *prefix closure,* which is a very important concept for model inference as it gives us the ability to arrange collections of traces in a trie — a data structure in which all descendants of a node share a common prefix. This is explained in more detail in Section 3.4.

Definition 4. The prefix closure of a language $\mathcal{L}$ is $\{s_1 \in \Sigma^* | \exists s \in \mathcal{L}.\, s_1 \sqsubseteq s\}$ with $\mathcal{L}$ being prefix closed if $\forall s \in \mathcal{L}. s' \sqsubseteq s \implies s' \in \mathcal{L}$.

Throughout this work, I assume that the trace languages of software systems are prefix closed. That is, if action a followed by action b followed by action c represents a valid interaction with the software, simply executing just action a, or action a followed by action b *without* action c, also represent valid interactions.

2.1.2 What to Record

Let us return to the running example of the simple drinks machine from Section 1.1, which is implemented by Java code in Figure 2.1. There are three attributes here: `selected` (which stores the selected drink), `value` (which keeps track of the amount of money inserted so far), and `PRICE` (which is the current price of the drinks in the machine in pence). There are also three public methods which represent the possible actions the machine can perform.

The `vend` method is particularly noteworthy because it exhibits value-dependent behaviour. That is, the return value depends on the value of a variable. If `value` is greater than or equal to `PRICE`, then the selected drink is returned. If not, the `null` value is returned, corresponding to the user not getting their drink because they have not yet inserted enough money.

If we want to produce traces of the program in Figure 2.1, we must first decide what it is that we wish to observe. At the lowest level of detail, we could simply record the sequences of methods (or *actions*) that are called. Traces of this form would look like $\langle$*select*, *coin*, *vend*$\rangle$. Such traces allow us to determine the *control flow* of the application (i.e. which events happen in what order) but do not give us a particularly comprehensive view of the system.

```
public class SimpleDrinksMachine {

        private String selected;
        private int value = 0;
        private final int PRICE = 100;

        public void select(String drink) {
                this.selected = drink;
        }

        public int coin(int value) {
                this.value += value;
                return this.value;
        }

        public String vend() {
                if (value >= PRICE) {
                        return selected;
                }
                return null;
        }
}
```

Figure 2.1: A Java implementation of the simple drinks machine.

Both the `select` and `coin` methods take an input. We could record these in the traces, as well as the method name. Traces of this form might look like $\langle select(\text{"tea"}),\ coin(50),\ vend()\rangle$ and allow us to see what kind of inputs each method is normally called with. From this, we can make generalisations, for example "the *coin* method normally gets called with a numeric argument which is greater than zero". Traces like these allow us to infer models which can be used as *runtime monitors* [100] to flag up unusual behaviour, for example if the `coin` method was called with a negative input.

To further increase the level of detail, we could record the return values of functions as event *outputs*. Traces of this form might look like $\langle select(\text{"tea"}),\ coin(100)/[100],\ vend()/[\text{"tea"}]\rangle$. As described in later chapters, a large enough collection of such traces might then allow us to infer a model in which output is symbolically computed from input. Such a model would allow us to *predict* how the system might behave when faced with previously unseen inputs. For example, if we had traces in which users selected "tea" and "coffee", we could predict what might happen if they were to select "soup".

Thus far, we have only considered traces which record information visible to an external observer of the system. Such traces are referred to as *black-box* traces. It is as if the system is encapsulated within a black box which cannot be opened. As a user interacting with the vending machine program, we know that we have called the `coin` method with a particular input argument, and we can record the value which was displayed to us when we did so. We do not know, however, that there is a variable called `value` which is mutated by our actions.

If we are able to look inside the "box", however, we can see that there are some `private` internal variables which we cannot access from outside the system. The values of these variables can also be included as part of the traces. Traces which include information that is hidden to outside observers are called *white-box* traces and might look like the one in Figure 2.2, which correspond to the EFSMs inferred by the techniques presented in [106] and [152]. These traces allow us to see how internal variables change throughout the execution of a program, again enabling us to predict how the system might behave when faced with unseen inputs [150].

$$
\begin{array}{l}
\langle \\
\ \mathit{select}(\mathtt{input} = \text{“tea”}), \\
\ \mathit{coin}(\mathtt{input} = 50, \mathtt{selected} = \text{“tea”}, \mathtt{value} = 0) \\
\ \mathit{coin}(\mathtt{input} = 50, \mathtt{selected} = \text{“tea”}, \mathtt{value} = 50) \\
\ \mathit{vend}(\mathtt{selected} = \text{“tea”}, \mathtt{value} = 100) \\
\rangle
\end{array}
$$

Figure 2.2: A white-box trace of the simple drinks machine.

We may choose to include additional information such as *timestamps* as part of the traces. The inclusion of timestamps allows us to infer roughly how long each event takes to execute, and may enable us to infer *timed automata* from the traces, like in [122]. Such models can then be used to detect when actions take longer than expected, which might indicate network problems or suspicious activity.

2.1.3 Obtaining Traces

Depending on the kind of traces we wish to obtain, there are a number of different methods and utilities available. This section summarises a few of these for completeness, although this thesis is more concerned with what can be done with traces once they have been obtained and works under the assumption that trace data at a suitable level of abstraction is readily available.

Most web servers implement some form of logging functionality by default. That is, information about requests sent to and from the system is written to a log file. These log files are primarily intended to help system administrators to detect suspicious activities and to diagnose issues. If these log files can be transformed into meaningful traces, then they can be used to infer models. It is also worth mentioning packet sniffing tools such as WireShark,[1] which enable users to monitor network packets and inspect the data they contain. For networked systems, such utilities may be sufficient to obtain trace data.

The main challenge with both of these techniques is the distributed parallel nature of networked systems. Busy web servers can receive millions of requests per second from many users, which makes the task of discerning individual execution traces extremely difficult.

For local systems, several utilities are available on Linux to record kernel calls made by programs. One such utility is `strace`,[2] which produces black-box traces listing the system calls made (with their arguments and return values) during program execution. Auxiliary utilities such as `b3`[3] build on top of `strace` to provide more structured traces.

[1]`https://www.wireshark.org` (Accessed 29/08/20)
[2]`https://strace.io` (Accessed 29/08/20)
[3]`https://github.com/dannykopping/b3` (Accessed 29/08/20)

If we can modify the source code of the system as part of the tracing process, most programming languages have some sort of logging functionality which can be baked into the program to provide an arbitrary level of detail about the state of the program as it runs. This is best done during development, though, as it can be an extremely arduous process if done post-hoc. It also requires the code to be redeployed, which is not always possible on a live system.

Some languages, for example Java, provide basic functionality to ease the process of post-hoc logging in the form of *aspect oriented programming*, [93]. This allows logging code to exist separately to the program code and, in some cases, does not require recompilation, which can be helpful for live systems that cannot be taken down.

2.1.4 Negative Traces

Traces which come from real systems are necessarily *positive* — they describe behaviour which the system is capable of performing. Negative traces are the opposite of this: they describe behaviour which the system *cannot* perform. Where positive traces must be accepted by the inferred model, negative traces must be rejected, although notions of acceptance and rejection differ depending on the model.

Example 2.1.3. In our simple vending machine example from Section 1.1, traces in which a user receives a drink before inserting coins are negative traces because the system does not exhibit this behaviour. Such a trace might be $\langle \mathit{vend}()/[\text{“tea”}],\ \mathit{coin}(50)/[50]\rangle$. Another class of negative traces is those where the customer receives a different drink to the one they selected. Such traces might look like $\langle \mathit{select}(\text{“tea”}),\ \mathit{coin}(100)/[100],\ \mathit{vend}()/[\text{“coffee”}]\rangle$.

Negative traces can be just as helpful as positive ones when trying to infer models, as they help to mitigate *overgeneralisation.* While it is desirable to generalise beyond the observed behaviour, it is clearly not desirable to do this excessively. Excessive overgeneralisation occurs when the inferred model is capable of exhibiting spurious behaviour. An example of this is given in Chapter 3. If we have some examples of such behaviour, we know when this point has been reached and can avoid making inference steps which introduce such behaviour to the model.

The main disadvantage of negative traces is that they generally need to be handcrafted. The number of such traces is then limited by the time and patience of the person tasked with writing them, as well as their knowledge of the system.

2.1.5 Characteristic Samples

If we are to use traces to infer a model which is faithful to the underlying system, we clearly need sufficiently many traces to demonstrate the program's functionality. Ideally we would have all the traces that the program is capable of producing, but this is often infeasible as there could be infinitely many. We must therefore work with a representative sample.

Example 2.1.4. Consider again the drinks machine from Figure 2.1. If we only have traces in which the customer selects tea, this may lead us to believe that tea is the only product sold. The machine may also be capable of dispensing coffee, soup, or a brimming glass of spiders, but we have no way of knowing this if “tea” is the only drink we have seen be selected.

We can call a set of traces *characteristic* of a program's behaviour if there is no trace which could be added that would reveal new behaviour [44, 72]. That is, the model that is inferred from the collection of traces would not be changed by the addition of a new one. The obvious question this raises is "How do we know when we have a characteristic sample?". This idea is explored in [39], which presents the idea of k-completeness. The idea here is that a log is k-complete if there is no trace which changes the model inferred from the log by the k-tails inference algorithm [16]. The paper then goes on to define a metric called k-confidence which is an estimate of how likely it is that a given log is k-complete. Experimental results presented in [39] show that the metric is "highly reliable" but it is not particularly well-used in the literature.

2.2 Computational Models

A model is a representation of a system which describes certain aspects of its functionality. Models are used in a wide variety of domains and have many uses. Computational models model computation. They show *what* a computer program should do without necessarily giving details of *how* this should be performed. Such models can be used to generate software tests [68, 139], to detect cyber attacks [143], and to aid the process of requirements capture [40].

In this work, we are concerned with formal models of software, specifically finite state machines (FSMs). The term "state machines" is used to describe a family of models based around *states* and *events*, the idea being that the model moves from state to state as it responds to a sequence of events. Models with a finite number of such states are called "*finite* state machines". State machines exist in a multitude of forms and have been around for many decades. The original idea is attributed to McCulloch and Pitts [111] but there have been contributions from many other famous names such as Turing [141], Minsky [61], Mealy [112], and Moore [116].

FSMs show the progression of events and the branching points during the execution of a piece of software. The effect of each action is determined by the current state, which is itself determined by the previous actions. Thus, the state can be used to store historical information about previous events. This ability is a key feature of state-based models. The remainder of this section introduces the various different kinds of finite state machine model, roughly ordered by computational power.

2.2.1 Labelled Transition Systems

The simplest form of state machine is a *labelled transition system* (LTS). Here, we have a set of states, a set of actions, and a transition function which tells us the resulting state for each action from each state. There are no inputs or outputs here, so events and their corresponding transitions consist solely of action labels. They are *atomic.*

Definition 5. An LTS [49] is a tuple (Q, q_0, Σ, T) where

Q is a finite non-empty set of states.

$q_0 \in Q$ is the initial state.

Σ is a set of actions (or *events*) called the *alphabet.*[4]

$T \subseteq S \times A \times S$ is a transition relation.

Transitions usually take the form $p \xrightarrow{a} q$ rather than (p, a, q).

[4]Certain definitions of LTSs also include τ actions, representing unobservable internal system behaviour. The LTSs in [49] are designed in terms of observable trace behaviour, so do not include such actions.

Example 2.2.1. Consider a variant of our simple drinks machine in which a user first inserts a coin and then selects either "tea" or "coffee". This can be expressed as an LTS as follows.

$$(\{q_0, q_1, q_2, q_3\}, q_0, \{coin, tea, coffee\}, \{q_0 \xrightarrow{coin} q_1, q_1 \xrightarrow{tea} q_2, q_1 \xrightarrow{coffee} q_3, \})$$

This LTS models a single drinks machine transaction. In the initial state q_0, the only valid action is *coin*. This moves the model into state q_1, from which the user can then choose *tea* or *coffee*. If the user chooses *tea*, the model moves to state q_2. If the user chooses *coffee*, the model moves into state q_3. There are no actions available from either q_2 or q_3, signifying that after the user has inserted a coin and chosen a drink, the transaction is finished.

As with all FSMs, LTSs can be expressed diagrammatically without loss of formality. This is often aesthetically preferable so will be done henceforth wherever possible. The diagrammatic representation of the LTS from Example 2.2.1 is shown in Figure 2.3. The correspondence between the two is as follows. The states $\{q_0, q_1, q_2, q_3\}$ are shown as labelled circles. The initial state, q_0, is marked as such by an unconnected incoming arrow. The transition relation is represented by connecting pairs of states with an arrow labelled with the action. The alphabet, Σ, can be inferred by enumerating the transition labels. This notation is common to most diagrammatic FSM representations and will be used throughout this work.

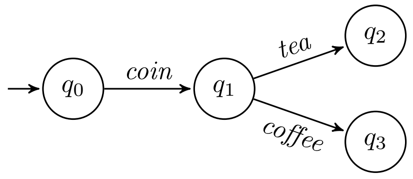


Figure 2.3: A diagram of the drinks machine LTS in Example 2.2.1.

2.2.2 Deterministic Finite Automata

Deterministic Finite Automata (DFAs) are similar to LTSs but are constructed such that each state has exactly one outgoing transition for each action in the alphabet. This makes them *deterministic* and *complete.* DFAs also introduce the idea of *accepting* (or *final*) states such that they can accept (or reject) a particular string.

Definition 6. A DFA is a 5-tuple (Q, q_0, Σ, T, F) where

Q is a finite set of *states.*

$q_0 \in Q$ is the *initial state.*

Σ is the *alphabet.*

$T : Q \times \Sigma \to Q$ is the *transition function.*

$F \subseteq Q$ is the set of *accepting states.*

Definition 7. A model is *complete* if, from every state, it is able to respond to every action in Σ. That is, every state has at least one transition for each action. A model is *deterministic* if, in every state, there is at most one transition for each action in Σ.

The notion of *complete* models is quite important. A complete model can respond to every event in every state, meaning that it can process every string in Σ^*, even if it does not accept it. Models which are not explicitly complete can be made implicitly so by adding a “sink state” for missing transitions.

Example 2.2.2. Recall the simple drinks machine from Example 2.2.1. We may wish to distinguish when a particular interaction has finished. To do this, we can use *accepting states*. Figure 2.4 is an incomplete DFA representation of the simple drinks machine. Here, q_2 and q_3 represent the user having inserted a coin and received a drink. These are completed transactions so are made accepting (or *final*) states, denoted by a double circle. In q_0 no transaction, valid or otherwise, has yet occurred. State q_1 represents the user having inserted a coin but not yet selected a drink. The transaction is not yet complete here, so these are not an accept states.

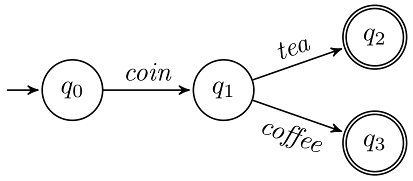


Figure 2.4: An NFA representing the simple drinks machine.

While the model in Figure 2.4 is *deterministic*, it is not *complete* as it cannot respond to every action in Σ from every state. There is no transition for the *tea* action from state q_0, for example. To make our model complete, we must add some additional transitions. Clearly these transitions do not represent valid actions, so they cannot go to any existing state in the model. We must therefore add an extra state for the additional invalid transitions to go to. It should be impossible to escape from this state as, once the model has done something invalid, it cannot return to validity. The error state therefore has three reflexive transitions, one for each action. Such error states are referred to as *sink states* and will become important to this work in Chapter 9. This is shown in Figure 2.5.

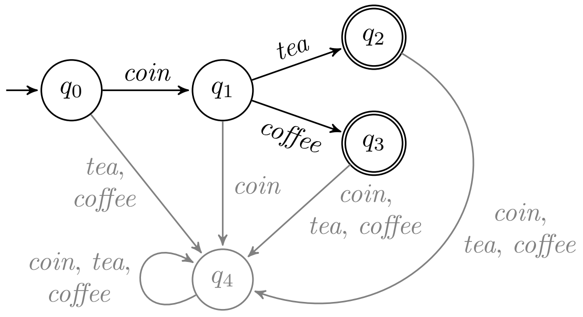


Figure 2.5: A DFA representing the simple drinks machine, in which the sink state and transitions representing invalid actions appear in grey.

There also exists a nondeterministic variant of the DFA — the *nondeterministic* finite automaton (NFA), which is defined as above but with $T : Q \times \Sigma \to \mathcal{P}(Q)$. This allows each state to have multiple outgoing transitions for any given action, or even none at all. This means that NFAs do not have to be either deterministic or complete. If they reach a point during execution where there is no corresponding transition, the model simply "dies" and the string is rejected. Additionally, models may include ϵ-transitions, which fire spontaneously without being explicitly called. Thus, the alphabet, Σ, also contains ϵ, which represents the empty character. Note though, that NFAs are not equivalent to LTSs because they have explicit accepting states and ϵ-transitions, which LTSs do not have.

Despite their different definitions, DFAs and NFAs are equivalent in computational power. That is, there does not exist a language which is accepted by an NFA for which we cannot construct an equivalently accepting DFA, and vice versa. Every DFA is an NFA by construction and for every NFA, there exists a DFA which accepts the same language. As will be discussed in Section 2.3, there are stronger definitions of equivalence which can distinguish between the two kinds of model.

It is not reasonable to be able expect to infer complete models from traces because, as discussed in Subsection 2.1.5, we cannot know whether the set of traces used to infer the model sufficiently represents the program. If we have never seen a trace in which the user inserts more than one coin, we do not know what the appropriate behaviour is. Mostly this is not an issue as we are attempting to infer a model of the behaviour that the system can exhibit. If we have no evidence to suggest that the system can perform a certain behaviour, there is no reason to include it in the model. This will become important, though, when we begin to explore properties of models in Chapter 9.

2.2.3 Inputs and Outputs

Most inference techniques in the literature attempt to infer either DFAs, NFAs, or LTSs, which I will collectively refer to as "classical FSMs" throughout the rest of this work. While classical FSMs show what actions can be performed from which states, they do not tell the full story. When we interact with software systems, we are usually just as interested in the inputs and outputs as we are the actions themselves. The atomic transitions of classical FSMs do not have the capability to model input-output behaviour, so we are in need of a more powerful model.

Mealy machines [112] are one way of modelling systems which produce output. In addition to the input alphabet Σ, there is also an output alphabet, Λ, and an output function such that every input yields an observable output as well as moving the model from state to state. In some formalisations, the transition and output functions are combined into one function of the form $Q \times \Sigma \to Q \times \Lambda$. Moore machines [116] are an alternative model where outputs are associated with states rather than transitions, so are not directly determined by input. Neither of these models natively includes either input parameters or the capacity for internal data storage, so they are unsuitable for modelling systems which make intensive use of these features.

Parametrised finite state machines (PFSMs) are a class of model with parametrised actions. That is, they are capable of handling events with inputs and outputs. Numerous definitions exist in the literature [101, 131], but they all fall into the same family. In Definition 8 (reproduced from [101]), the output is expressed as a function of the input parameters and the current state.

Definition 8. A PFSM is a tuple $(Q, I, O, D_I, D_O, T, q_0)$ where

Q is a finite set of states.
I is a finite set of inputs.
O is a finite set of outputs.
D_I is a set of input parameter values.
D_O is a set of output parameter values.
T is a set of transitions.
$q_0 \in Q$ is an initial state.

A transition $t \in T$ is described as $t = (q, q', i, o, p, f)$, where

$q \in Q$ is a source state.
$q' \in Q$ is a target state.
$i \in I$ is an input.
$o \in O$ is an output.
$p \subseteq D_I$ is a predicate on input parameter values.
$f : p \to D_O$ is an output parameter function.

Example 2.2.3. Figure 2.6 shows a PFSM representing the drinks machine. Here, the *coin* transition represents the insertion of a coin of value v. The *tea* and *coffee* transitions all take in a parameter l, which is a boolean value representing whether there is enough tea or coffee left to make the selected drink. If there is, the drink is dispensed. If not, there is no output.

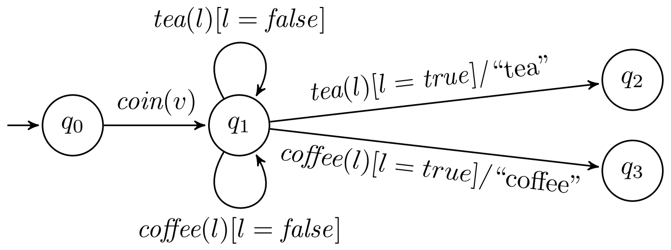


Figure 2.6: A PFSM representing two possible traces of the drinks machine.

While the model in Figure 2.6 is much closer to the underlying implementation than the previous models, the value of the input to *coin* has no bearing on the behaviour of the model. This means that the cost of a drink can vary wildly depending on what coins the customer has on them. Let us say that the price of a drink is one pound. We can then place a guard on the *coin* transition so it can only go to state q_1 if the value is greater than or equal to this. Otherwise, it simply loops on state q_0 and returns the coin to the customer. This is still not the correct behaviour, though, as if the customer has smaller change to the value of a pound, they cannot purchase a drink even though they have enough money to. To model the customer paying with multiple coins, we need to keep track of the value of the coins inserted so far.

We could do this by adding extra states to represent the various monetary values, and transitions from each of these states representing the insertion of a new coin. This would require an extra 99 states to be added to the model, one for each value up to a pound. From each state, we would have an outgoing transition for each denomination of coin going to the appropriate state. The obvious problem here is that this model would be very large and difficult to understand, especially considering the simplicity of the modelled behaviour.

We would really like to have an internal variable which represents the total inserted value, which the *coin* transition can updated. This allows us to push some of the control state into a separate *data state*, making the model much smaller and more intuitive. Unfortunately, the PFSMs from Definition 8 do not have internal variables, therefore further extension is needed.

2.2.4 Extended Finite State Machines

Every imperative programming language provides some notion of *variables* to store and manipulate values. This idea can be applied to PFSMs by providing them with simple memory which transitions can update during model execution. Various extended finite state machine (EFSM) definitions are presented in the literature [32, 105], as well as similar ideas under different names [20, 29, 56], but the basic idea is to allow information to be stored for later use, either as part of output, or in transition guards. One such definition is *register automata* from [29], reproduced in Definition 9. Like in PFSMs, transitions take in input parameters (which can be guarded) but now there is also a set of *registers* which act as simple memory.

Definition 9. A *register automaton* (RA) [29, Definition 4.1] is a tuple $(L, l_0, \chi, \Gamma, \lambda)$, where

- L is a finite set of states (called *locations* in [29]).
- $l_0 \in L$ is the initial state.
- λ maps each $l \in L$ to $\{+, -\}$, essentially corresponding to F in Definition 6.
- χ maps each state $l \in L$ to a finite set $\chi(l)$ of registers.
- Γ is a finite set of transitions, each of form $\langle l, \alpha(p), g, \pi, l' \rangle$, where
  - $l \in L$ is a source state.
  - $l' \in L$ is a target state.
  - $\alpha(p)$ is a parametrised action signature.
  - g is a guard over p and $\chi(l)$.
  - π (the *assignment*) is a mapping from $\chi(l')$ to $\chi(l) \cup \{p\}$.

The semantics of an RA are defined in [28] as follows. A *state* of an RA $A = (L, l_0, \chi, \Gamma, \lambda)$ is a pair (l, ν) where $l \in L$ and ν is a mapping from registers to their values. A *step* of A, $(l, \nu) \xrightarrow{\alpha(d)} (l', \nu')$ moves A from state (l, ν) to state (l', ν') by responding to event $\alpha(d)$ if there is a transition $(l, \alpha(p), g, \pi, l')$ such that d satisfies the guard under valuation ν and ν' is the updated evaluation with with $\nu'(x_i) = \nu(x_j)$ if $\pi(x_i) = x_j$, otherwise $\nu'(x_i) = d$ if $\pi(x_i) = p$.

Example 2.2.4. Figure 2.7 shows an RA representation of the drinks machine. The user first inserts a *coin* of value v, which is assigned to register r_1. The user can then insert more *coin*s, with the new value overwriting r_1. The *tea* and *coffee* transitions take the parameter l, as before, but now there are additional guards. If the value of r_1 is less than 100 (i.e. a pound), then the respective $q_1 \to q_1$ transition is taken, which produces no output. If the value of r_1 is greater than or equal to a pound, then either the $q_1 \xrightarrow{tea} q_2$ or $q_1 \xrightarrow{coffee} q_3$ transition is taken, depending on the action. Since there is no explicit notion of output here, these transitions set the value of r_2 to the specified drink, representing dispensing it to the user.

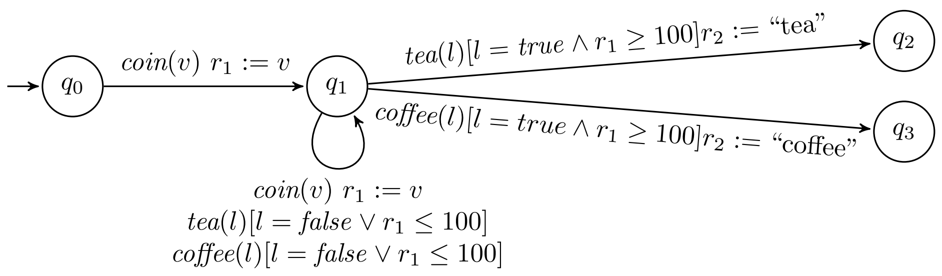

q_0
$coin(v)\ r_1 := v$
q_1
$tea(l)[l = true \wedge r_1 \geq 100]r_2 :=$ "tea"
q_2
$coffee(l)[l = true \wedge r_1 \geq 100]r_2 :=$ "coffee"
q_3
$coin(v)\ r_1 := v$
$tea(l)[l = false \vee r_1 \leq 100]$
$coffee(l)[l = false \vee r_1 \leq 100]$

Figure 2.7: An RA representing the simple drinks machine.

The problem here is that the RAs in [28] cannot perform arithmetic functions, so we are still unable to maintain a *running total.*. This is a major limitation of the RAs in [28] and will be resolved in Chapter 4, when I introduce my own EFSM definition.

Example 2.2.5. Consider the RA in Figure 2.8, which models the controller of a set of lift doors. Here, there is one register, r_1, which represents the timer used by the system. The timer is initially set by the *setTimer* action, which takes one input parameter, i, and assigns it to r_1. The system then does i *waitTimer* actions, subtracting one from the timer each time. When the timer reaches zero, the system is ready and r_1 is set to ten. The doors begin closing and the system outputs ten *closingDoor* events, subtracting one from the timer each time until the timer reaches zero, at which point the doors are *fullyClosed.*

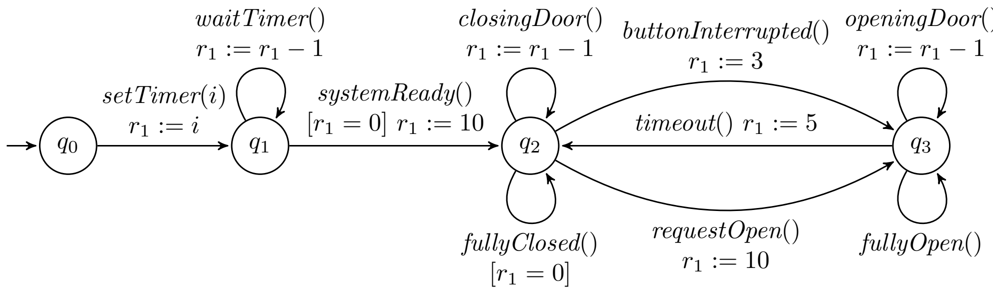


Figure 2.8: An RA representing a simple lift door controller.

Alternatively, the user can interrupt this process by triggering the *buttonInterrupted* event, which sets the timer to three and causes the doors to open again. Additionally, a user outside of the lift can request entry by triggering the *requestOpen* event, which sets the timer to ten and causes the doors to open. Once the doors are *fullyOpen*, the lift waits a few seconds before a *timeout* event sets the timer to five and causes the doors to begin closing again. We will explore this system again in more detail in Chapter 7.

This system is much more suited to an RA than the drinks machine from Example 2.2.4 as there are no explicit outputs here. There is no need to display the value of the time to the user at any point during the system's execution. Thus, the fact that RAs do not have distinct outputs is not a problem here.

The fact that RAs have no explicit outputs is a weakness in the model. Other EFSM definitions exist which do produce output, but the field of EFSMs is somewhat disparate. There are many definitions, all with their own strengths and weaknesses. Definitions are often formulated with a specific application in mind, so are not generally well suited to other tasks. This motivated me to formulate my own EFSM formalism which combines various desirable characteristics from existing definitions from the literature. This definition is presented in Chapter 4.

2.2.5 The Necessity for Registers

As their name suggests, register automata use *registers.* These play the role of variables in imperative programming languages. Transitions may assign and modify registers arbitrarily throughout the execution of the model. An alternative programming paradigm is *functional programming.* Here, full referential transparency is maintained by expressing outputs purely in terms of inputs. Variable values cannot be arbitrarily mutated. This makes for a very different programming experience.

```erlang
-module(drinks).
-export([machine/3]).

machine(null, null, Price) ->
  receive
    {select, [I0]} ->
      machine(I0, 0, Price)
  end;
machine(Selected, Value, Price) when Value >= Price ->
  receive
    {coin, [I0]} ->
      io:format("~p~n", [Value + I0]),
      machine(Selected, Value + I0, Price);
    {vend, []} ->
      io:format("~p~n", [Selected])
  end;
machine(Selected, Value, Price) when Value < Price ->
  receive
    {coin, [I0]} ->
      io:format("~p~n", [Value + I0]),
      machine(Selected, Value + I0, Price);
    {vend, []} ->
      machine(Selected, Value, Price)
  end.
```

Figure 2.9: The drinks machine program written in Erlang.

Figure 2.9 shows an implementation of the same drinks machine from Figure 2.1 in Erlang, a functional programming language. Here, the drinks machine is represented as a process which receives different messages. Instead of having globally accessible variables representing the selected drink, the money inserted, and the price, these variables must be passed around the program as parameters if they are to be kept track of. For example, upon receiving the `coin` message, the `machine` process restarts itself with `Value + I0`, where `I0` represents the parameter representing the value of the inserted coin.

While "variables" do not change their value here, the effect of restarting the `machine` process with new parameters has the same effect. That is, if we inspect the values of `Selected`, `Value`, and `Price` before and after a `coin` event, we will see that the value of `Value` increases by the input parameter. Register automata, and (E)FSMs in general, cannot be "restarted" in the same way that functional processes can. In order to represent the change of state caused by the `coin` event, there needs to be a state which can change. This is what registers allow us to do.

If we were forced to express output purely in terms of input, we would need to supply the variables as an inputs to each transition. This would make traces of the drinks machine look like $\langle$*select*("tea"), *coin*(50, 0, "tea")/[50], *coin*(50, 50, "tea")/[100], *vend*("tea")/["tea"]$\rangle$, in which the first input to *coin* is value of the coin, the second is the total amount inserted so far, and the third is the selected drink. While it is perfectly possible to do this, we have effectively taken the internal `Selected` and `Value` state variables and made them external. This is not a particularly good representation of a real drinks machine as customers should not have to keep reminding it of the drink they selected every time they insert a coin.

An alternative approach, which classical FSM models take, is to use the *control flow* state of the model to store data implicitly. As mentioned in Subsection 2.2.3, this can lead to very large and unwieldy models. Mutable registers allow us to arbitrarily move information between the control flow and data states, which can be helpful when working with large (and potentially infinite) state spaces. In our vending machine example, there is nothing to stop a customer inserting £1000 in 1p coins. To represent the accumulating value without registers, we would need 100000 states, one for each inserted coin. Using a register to keep track of this instead allows us to represent any number of coin insertions with a single control flow state and a reflexive transition, which makes for a much smaller model that more accurately captures the behaviour of the underlying system.

2.2.6 Abstract State Machines and Event-B

In addition to state machines, there are a number of alternative formalisms with similar expressive power. Abstract state machines (ASMs) can be thought of as a collection of "**if** *condition* **then** *updates*" rules which transform "abstract states" [20]. The *condition* (or guard) is simply a predicate which, if true, allows the transition to fire. If it is not satisfied, the transition cannot fire. The *updates* are a list of functions of the form $f(v_1, \ldots, v_n) := x$ which are applied simultaneously when the transition fires. These updates are interpreted as updating the value of function f at the indicated parameters $v_1, \ldots, v_n$, for example $\mathit{status}(\mathit{cell}) = \mathit{alive}$. This can be thought of, in more conventional C-like syntax as $\mathit{cell.status} = \mathit{alive}$.

The semantics of ASMs dictate that, at each step, *all* rules for which the guard passes are applied. This is slightly different from how nondeterminism works in other FSM-style models where, if multiple transitions may be taken, one is arbitrarily chosen or multiple "copies" of the machine are spawned. Here, there is no explicit separation between control flow state and data variables, however, it is not hard to see that this separation could be artificially implemented with a *cfstate* variable. Thus, ASMs can be thought of as an equivalent model to the more powerful variants of EFSMs.

ASMs are particularly well suited to *model checking* and tools such as SAL [46], which I make use of in Chapter 9, use a very similar representation. The originally intended use, however, appears to be for the incremental development of systems from requirements capture to executable code through an informal refinement process [20].

Event-B models are similar to ASMs, again consisting of a set of "**if** *condition* **then** *updates*" rules. Event-B is considered an extension to the B-Method [127], the idea being to first specify an abstract system model and then refine it down to executable code. Event-B has tool support for this in the form of Rodin [6]. Since Event-B models are so similar to ASMs, they too can be thought of as an equivalent model to EFSMs.

2.3 Refinement

Abstract specifications tend to leave a lot of behaviour unspecified, especially the implementational details of how the high level goals are achieved. Refinement is the process of transforming an abstract specification into a concrete implementation.

Example 2.3.1. Consider the following abstract specification of the simple drinks machine.

A user first inserts a coin to pay for their drink before selecting either "tea" or "coffee" to dispense their chosen beverage.

There is much unspecified behaviour here. What denominations of coin does the machine take? How much does a drink cost? Can the machine run out of tea or coffee? During the process of refinement, decisions would be made to reduce this unspecified behaviour. This is known as the *reduction of nondeterminism.*

Refinement is about comparing the behaviour of systems. If an implementation $\mathcal{I}$ refines a specification $\mathcal{S}$, then the behaviours of $\mathcal{I}$ are *consistent* with those of $\mathcal{S}$. This can be quantified in terms of what we can *observe.* For high level specifications, we do not need to be concerned about behaviours we cannot see. We can therefore form the relation that if $\mathcal{I}$ refines $\mathcal{S}$, the set of observable behaviours of $\mathcal{I}$ is a subset of those of $\mathcal{S}$.

The notion of *consistency* is an important one in refinement. Any refinement of an abstract specification must capture all the specified behaviour. For example, the abstract specification of our simple drinks machine in Example 2.3.1 states that users first insert coins and then select their drink. Any implementation in which the user selected their drink before inserting coins would not be a refinement of the specified system.

2.3.1 Nondeterminism

Nondeterminism is central to refinement and refers to a system behaving differently on different runs when being run under the same conditions. We have already seen nondeterministic finite state machines in Section 2.2. Let us now establish a formal definition in terms of LTSs.

Definition 10. An LTS (Q, q_0, Σ, T) is *nondeterministic* if $\exists p. \exists a. \exists q. \exists q'. (p, a, q) \in T \wedge (p, a, q') \in T$. That is, if there is a state with multiple outgoing transitions for the same label.

Example 2.3.2. Consider the LTS in Figure 2.10, which is a nondeterministic representation of the simple drinks machine from Figure 2.3. Here, the initial *coin* event determines whether the user will receive tea or coffee. Since users of the machine have no explicit control over which *coin* transition is taken when they insert their payment, they do not have a say in whether they receive tea or coffee. This is very different to the model in Figure 2.3, where the decision is only made after payment has been inserted.

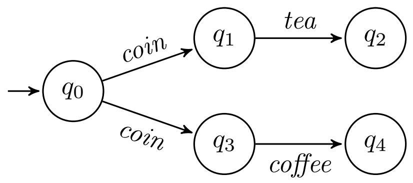


Figure 2.10: A nondeterministic representation of the drinks machine from Figure 2.3.

Despite the user having no control over which drink they receive, there is no *trace* of the system which can distinguish the model in Figure 2.10 from that in Figure 2.3. That is to say that the set of *observable behaviours* of the two models is identical.

Nondeterminism is not just about there being multiple possible paths through a given model, but also about unspecified behaviour. In Figure 2.10, it is not specified which *coin* transition should be taken in response to the action. This leaves the model with a *choice* which affects what it can do later. In Figure 2.3, there is no such choice, meaning that the ability to choose *tea* or *coffee* is retained until the last possible moment.

2.3.2 Trace Refinement

The simplest observation we can make of a system is to record the sequences of events it performs in the form of traces. Further to Definition 2, it is helpful to define traces more formally in terms of *processes.* From this, we can build a refinement relation.

Definition 11. A finite sequence of events σ is a *trace* of process p if $\exists q.\, p \xrightarrow{\sigma} q$, i.e. if p can respond to every event in the sequence. If σ is a trace of p, then p is said to *accept* σ.[5]

Example 2.3.3. The event sequence $\langle \mathit{coin}, \mathit{tea} \rangle$ is a trace of the LTS in Figure 2.3. The event sequence $\langle \mathit{tea}, \mathit{coin} \rangle$ is not.

Definition 11 is consistent with our assumption of prefix closure. That is, if a sequence of events σ is a trace of process p, then all prefixes of σ are also traces of p. It is also helpful to extend Definition 11 to models with inputs and outputs. This is done by strengthening the condition such that, in addition to being able to *respond* to every event in the sequence, the model must also produce the correct outputs from the given inputs. From our informal definition that $\mathcal{I}$ refines $\mathcal{S}$ if the set of observable behaviours of $\mathcal{I}$ is a subset of those of $\mathcal{S}$, we can thus form a definition of *trace refinement.*

[5]Here, we are working in terms of LTSs, which do not have explicit accepting states. For models such as DFAs, which do have explicit accept states, we would need the additional restrictions to handle this.

Definition 12. Let $\mathcal{T}(p)$ be the set of traces of process p. We say that p is refined by q if $\mathcal{T}(q) \subseteq \mathcal{T}(p)$, i.e. if q has less observable behaviour than p.

Example 2.3.4. If we note down the sets of traces of the LTSs in Figures 2.3 and 2.10 then it becomes apparent that they refine each other, as their traces are equivalent.

$$\mathcal{T}(\text{Figure 2.3}) = \mathcal{T}(\text{Figure 2.10}) = \{\langle\rangle, \langle coin\rangle, \langle coin, tea\rangle, \langle coin, coffee\rangle\}$$

From Definition 12, we can see that the empty process, shown in Figure 2.11 refines every other process because $\mathcal{T}(\text{Figure 2.11}) = \emptyset$. This comes as a result of the fact that trace refinement preserves *safety properties* but not *liveness properties.*

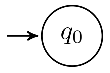


Figure 2.11: An LTS representing the empty process.

Definition 13. *Safety properties* are informally described as properties which require that "nothing bad ever happens". By contrast, *liveness properties* require that "something good eventually happens".

Example 2.3.5. Consider a simple traffic light controlled pedestrian crossing. One safety property we might like to have of this system is that the light is never green for both pedestrians and cars at the same time. A liveness property is that, if a pedestrian presses the button to cross, the cars are eventually stopped to allow them to do so.

From Definition 13, it is clear that if nothing ever happens, as with the empty process, then nothing *bad* can ever happen. Nothing *good* will happen either, but the system is at least safe. Ideally though, we would like to be able to use the system for its intended purpose. We are therefore in need of a stronger definition of refinement such that the empty process can no longer refine a system which performs actions.

2.3.3 Failures Refinement

Considering a *failures* semantics allows us to define a relation which preserves both safety and liveness properties[6] as per Definition 13. In the failures semantics, we record not only what processes can do but also what they *refuse* to do. This allows us to distinguish the systems represented in Figures 2.3 and 2.10 and to ensure that refinements of a given system allow at least the same functionality. That is, the empty process no longer refines a process which can carry out some action.

Definition 14. The pair $(\sigma, X) \in A^* \times \mathcal{P}(\Sigma)$ is a *failure* of a process p if

$$\exists q.\, p \xrightarrow{\sigma} q \wedge \textsc{next}(q) \cap X = \emptyset$$

where $\textsc{next}(q)$ represents the set of possible actions of process q. Let $\mathcal{F}(p)$ be the set of failures of p such that p is *failures refined* by q if $\mathcal{F}(q) \subseteq \mathcal{F}(p)$, i.e. if q fails less often than p.

[6]Since I discuss refinement in terms of LTSs as per Definition 5, which do not have τ actions, there is no possibility of divergence here.

Example 2.3.6. Consider the LTS in Figure 2.3. To calculate the *refusals* of the process, we ask "What can the system refuse to do initially?", and then for each event e ask "What can the system refuse to do after e?". This then progressively builds the set of failure pairs.

$$\begin{aligned}\mathcal{F}(\text{Figure 2.3}) = \{ & \\ & (\langle\rangle, \{\mathit{tea}, \mathit{coffee}\}), \\ & (\langle \mathit{coin}\rangle, \{\mathit{coin}\}), \\ & (\langle \mathit{coin}, \mathit{tea}\rangle, \{\mathit{coin}, \mathit{tea}, \mathit{coffee}\}), \\ & (\langle \mathit{coin}, \mathit{coffee}\rangle, \{\mathit{coin}, \mathit{tea}, \mathit{coffee}\}) \\ & \}\end{aligned}$$

The LTS in Figure 2.10 has the following set of failures.

$$\begin{aligned}\mathcal{F}(\text{Figure 2.10}) = \{ & \\ & (\langle\rangle, \{\mathit{tea}, \mathit{coffee}\}), \\ & (\langle \mathit{coin}\rangle, \{\mathit{coin}, \mathit{tea}, \mathit{coffee}\}), \\ & (\langle \mathit{coin}, \mathit{tea}\rangle, \{\mathit{coin}, \mathit{tea}, \mathit{coffee}\}), \\ & (\langle \mathit{coin}, \mathit{coffee}\rangle, \{\mathit{coin}, \mathit{tea}, \mathit{coffee}\}) \\ & \}\end{aligned}$$

We can now see that the refinement relation between Figures 2.3 and 2.10 is no longer bidirectional because of what the respective systems can refuse to do after the trace $\langle \mathit{coin}\rangle$. Figure 2.3 can only refuse to do a *coin* event; *tea* and *coffee* are both possibilities. The LTS in Figure 2.10, however, could refuse to do a *coffee* if the top *coin* transition was taken initially, or it could refuse to do a *tea* if the bottom *coin* transition was taken. Thus, failures refinement is strong enough to distinguish between the two systems.

Observing refusals in this way is sufficient to detect nondeterminism in models. Observe that both $(\langle \mathit{coin}\rangle, \{\mathit{coin}, \mathit{tea}, \mathit{coffee}\})$ and $(\langle \mathit{coin}, \mathit{tea}\rangle, \{\mathit{coin}, \mathit{tea}, \mathit{coffee}\})$ are in $\mathcal{F}$(Figure 2.10). Thus, after $\langle \mathit{coin}\rangle$, a *tea* event can either be accepted or refused. Therefore, there must be multiple *coin* transitions which take us to different states. A deterministic process does not behave like this, and refuses exactly those events which cannot extend a given trace.

Definition 15. A process p is *deterministic* iff

$$\forall \sigma \in \mathcal{T}(p).\, (\sigma, X) \in \mathcal{F}(p) \wedge a \in X \implies \sigma \cdot \langle a\rangle \notin \mathcal{T}(p)$$

2.3.4 Conformance and Extension

This section introduces two relations motivated by considerations from software testing. In software engineering, *conformance testing* refers to the activity of ensuring that the software meets (or *conforms to*) the specification [140]. If done properly, this is very similar to the notion of refinement. To define conformance, we need an auxiliary definition *refusals after a trace.*

Definition 16. For an LTS p, trace σ, and $X \subseteq \Sigma$, p **after** σ **ref** X if

$$\exists q.\, p \xrightarrow{\sigma} q \wedge \textsc{next}(q) \cap X = \emptyset$$

This allows us to redefine failures refinement as follows.

$$\forall \sigma \in \Sigma^*.\, \forall x \subseteq A.\, q \textbf{ after } \sigma \textbf{ ref } X \implies p \textbf{ after } \sigma \textbf{ ref } X$$

Note that this definition of failures refinement is too strong for conformance testing as it quantifies over all event sequences in Σ^* rather than just those of the original specification. To define conformance, we weaken the quantification as in Definition 17. Informally, for an environment restricted to traces of p, the system q will deadlock less often than p. The weakened quantification allows us to only test traces from the abstract specification, discounting vast swathes of the search space. It does mean, though, that q may implement functionality which is not in p. This is not necessarily a bad thing, just something to be aware of.

Definition 17. An implementation q *conforms* to a specification p iff

$$\forall \sigma \in \mathcal{T}(p), x \subseteq \Sigma.\, q \textbf{ after } \sigma \textbf{ ref } X \implies p \textbf{ after } \sigma \textbf{ ref } X$$

Example 2.3.7. Consider the LTS in Figure 2.12. This system conforms to the LTS in Figure 2.3 but has the additional functionality of being able to dispense *soup*. The conformance relation in Definition 17 has no way to detect this and, indeed, in environments restricted to traces of Figure 2.3, the two systems are indistinguishable as *soup* can never never selected.

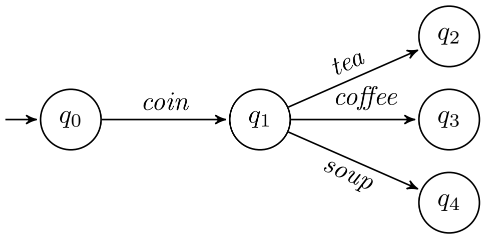


Figure 2.12: An LTS which implements additional functionality to Figure 2.3.

Conformance gives us a key characteristic we need later on for inference, namely the capacity to add functionality, but it cannot be used as an order relation as it is not transitive. That is, the property $x \sqsubseteq y \implies y \sqsubseteq z \implies x \sqsubseteq z$ does not hold. This is fine in real-world software engineering where, typically, one only needs to relate one implementation to one specification, but conformance is not suited to iterative refinement steps. This motivates the definition of *extension* which allows us to implement additional functionality but maintains transitivity, thus allowing multiple refinement steps.

Definition 18. LTS q is an *extension* of p if

$\mathcal{T}(p) \subseteq \mathcal{T}(q)$ and
$\forall \sigma \in \mathcal{T}(p), X \in A.(q \textbf{ after } \sigma \textbf{ ref } X) \implies (p \textbf{ after } \sigma \textbf{ ref } X)$

2.3.5 Simple Simulation

It is now time to introduce a simple form of *simulation*, which is another kind of refinement relation where we aim to construct a function which relates states in the concrete implementation to those in the abstract specification.

Definition 19. An implementation I *simulates* [49] a specification S if there exists a function $\mathcal{S}$ from the states of I to the states of S such that

1. If s is the initial state of I then $\mathcal{S}(s)$ is the initial state of S.
2. For all actions $a \in \Sigma$, if $s \xrightarrow{a}_I s'$ then $\mathcal{S}(s) \xrightarrow{a}_S \mathcal{S}(s')$.

Informally, the initial states are identical and every action in the concrete specification has a corresponding action in the abstract specification. Definition 19 has the implicit assumption that the two systems have the same sets of externally observable actions, but this is reasonable given that we require implementations to exhibit the behaviour of their specification. Simulation is important in FSM inference because we want the final model to simulate the observed traces. The existence of a simulation relation is sufficient to prove trace refinement, but there exist systems for which the reverse is not the case.

Example 2.3.8. Consider the two LTSs in Figure 2.13. These are *trace equivalent* (i.e. they are trace refinements of each other) but they do not simulate each other. I (on the right) simulates A (on the left) but A does not simulate I. This is because the simulation relation must be a *function*. If we look at the relation between I and A, we have $\mathcal{S} = \{(i_0, s_0), (i_1, s_0)\}$. Going the other way, we would need $\mathcal{S} = \{(s_0, i_0), (s_0, i_1)\}$ which we cannot have since s_0 evaluates to both i_0 and i_1.

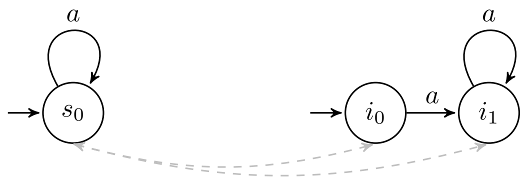


Figure 2.13: Two trace equivalent LTSs with the relation $\mathcal{S}$ between states illustrated with dashed lines.

Definition 19 is one of many simulation relations in the literature. Other definitions, such as [145], are more widely accepted and use a *relation* for $\mathcal{S}$ rather than a function. This would allow the two systems in Example 2.3.8 to simulate each other. The fact that Definition 19 uses a function is why it is described as *simple* in [49]. The idea here it to explicitly distinguish between the *implementation* and the *specification* such that, if an action can be carried out in the implementation, this must be explicitly allowed in the specification. The definition from [145] is concerned with the *equivalence* of two systems rather than the behaviour of one system being contained within that of another.

Concluding Remarks

This chapter explored the various finite-state models of computation which exist in the literature, as well as introducing the topic of refinement in the context of FSM models. This should provide the reader with sufficient background to understand the models which occur in this work. The next chapter introduces the concept of model inference, and provides the remainder of the background necessary to understand the remainder of this work and put it into context.

Chapter 3

Background II - Model Inference

The field of model inference arguably began with the pioneering work of Gold in 1967 [74]. This work presents the idea that we can learn a language from the strings within it. In a software context, this means that we can infer a model of a system from the traces it produces. The relationship between software traces, system tests, and models of execution was first highlighted by Weyuker in 1983 [153]. Most inference techniques can be thought of as either *active* or *passive.* Active techniques such as Angluin's famous L^* algorithm [10] make use of an oracle of some kind during the inference process, be that a human user, a model checker, or the system itself through some kind of *dynamic analysis.* By contrast, passive techniques such as Beirman's k-tails algorithm [16] rely solely on the data provided at the start of inference.

This chapter first gives an overview of how most existing passive inference techniques work, to serve as a foundation for my own passive inference technique presented in Chapter 6. I then outline the process of active FSM inference, before discussing some key inference algorithms from the literature, both for classical and extended finite state machines. I then talk about the related fields of Process Mining and Program Analysis, before discussing how the quality of inferred (E)FSM models can be evaluated, and identifying the limitations of current techniques and gaps within the literature.

3.1 The Inference Challenge

Given a set of traces, called the *training set,* the challenge is to infer a model which expresses this behaviour and is also able to predict how the system might behave when faced with new input sequences. Figure 3.1 illustrates the inference challenge as a Venn diagram.

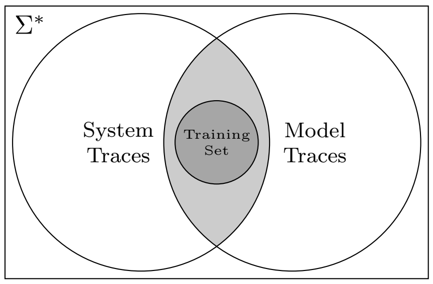


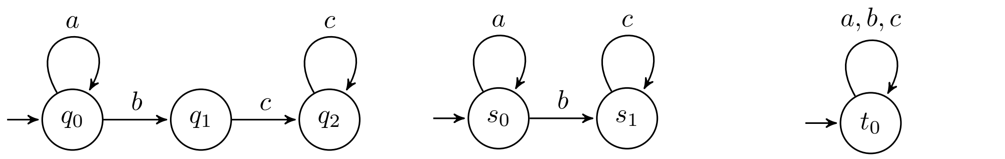
Figure 3.1: A Venn diagram illustrating the inference challenge in terms of traces.

The bounding box, Σ^*, represents all possible traces of a given alphabet. The leftmost circle represents those traces that the system is actually capable of producing. The central smaller circle represents the training set — the sample of system traces which is used to infer a model. The rightmost circle represents the traces of the inferred model. Clearly the model must accept all the traces in the training set, but we also want it to accept additional traces. The inference challenge is then to infer a model which maximises the intersection between the traces of the *system* and the traces of the *model*, while minimising those traces of the model which are not also traces of the system.

3.2 Inference vs. Minimisation

Before discussing model inference in further detail, it is important to differentiate it from the concept of *minimisation.* The aim of automaton minimisation is to create the smallest possible model which is trace equivalent to the original. By contrast, the aim of inference is to create a model which *conforms* to the original traces with the possibility of some additional behaviour.

Example 3.2.1. Consider the LTS in Figure 3.2a. This model is larger than it needs to be. The LTS in Figure 3.2b has only two states, but there is no trace which can distinguish between the two models. State s_1 in Figure 3.2b is *simulating* states q_1 and q_2 in Figure 3.2a.



(a) An unreduced LTS.

(b) A trace equivalent but smaller LTS.

(c) A model which conforms to Figure 3.2b but is not trace equivalent.

Figure 3.2: An LTS and the minimal trace equivalent model.

Inference takes this a step further. The aim here is to produce a model which exhibits all the original behaviour but may have additional functionality. That is to say that the traces of the inferred model can be a superset of those of the original, as is the case in Figure 3.2c. Every trace of Figure 3.2a is also a trace of Figure 3.2c, but $\langle c \rangle$ is not a trace of Figure 3.2a.

3.3 Basic State Merging

Most techniques for inferring models from traces involve some kind of *state merging.* That is, the initial trace set is transformed into a model which accepts exactly and only those traces. This model is iteratively condensed into a smaller model by merging states which are believed to represent the same state in the underlying program. To explain state merging algorithms in detail, let us establish a running example. Recall the drinks machine implementation from Figure 2.1. Here, customers first select their desired drink. Next, they insert coins to the value of one pound to pay for the drink. Finally, they dispense their drink by pressing *vend.* In Figure 2.1, if they have not inserted sufficient payment, *vend* returns `null`. To simplify the example, let us assume here that *vend* is never called unless sufficient payment has been inserted.

Given a set of black-box system traces, the aim is to infer a model which generalises the behaviour they exemplify. The most detailed black-box recording we can make lists the methods we call, along with any associated input arguments and return values. Some sample traces of the drinks machine are shown in Figure 3.3.

In the interest of simplifying this example as much as possible, in this section we will infer a classical FSM model of the traces in Figure 3.3. Unfortunately, these traces have inputs and outputs, which classical FSMs cannot handle, so we are left with two choices. Either we must

$\langle$*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"]$\rangle$

Figure 3.3: Some sample traces of the simple drinks machine.

abstract away the inputs and outputs to leave only the action names or we must transform the traces to include the inputs and outputs within the action labels. The input and output values are quite important here, so the latter approach is preferable. To transform the traces, we can simply join together all inputs and outputs with an underscore such that the event *coin*(50)/[50] (which represents event *coin* being called with input 50 and producing output 50) becomes *coin_50_50*. The transformed traces are shown in Figure 3.4.

$\langle$*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*$\rangle$

Figure 3.4: The traces in Figure 3.3, transformed to include inputs and outputs as part of the label.

Most modern state merging algorithms are variations on the procedure shown in Algorithm 1. There are four main stages here: prefix-tree construction (Line 2 – detailed in Subsection 3.3.1), scoring merges (Lines 5-7 – detailed in Subsection 3.3.2), merging states (Lines 9-11 – detailed in Subsection 3.3.3), and nondeterminism resolution (Lines 12-13 – detailed in Subsection 3.3.4).

Algorithm 1 Basic evidence-driven state merging algorithm.

```
1: function INFER(Traces)
2:     $(S, s_0, L, T) \leftarrow$ GENERATEPTA($traces$)
3:     for $s_1, s_2 \in$ CHOOSEPAIRS($S, T$) do
4:         $(S, T) \leftarrow$ MERGE($S, T, s_1, s_2$)
5: function CHOOSEPAIRS($S, T$)
6:     $pairScores \leftarrow [($SCOREMERGE$(s_1, s_2), (s_1, s_2))$ for $s_1, s_2$ in SELECTPAIRS$(S \times S)]$
7:     return SORTDESCENDING($pairScores$)
8: function MERGE($S, T, s_1, s_2$)
9:     $S \leftarrow (S \setminus \{s_1, s_2\}) \cup \{s_{1,2}\}$
10:    $T \leftarrow$ CHANGESOURCES($s_{1\ out}, s_{1,2}, T$)
11:    $T \leftarrow$ CHANGEDESTINATIONS($s_{1\ in}, s_{1,2}, T$)
12:    while $(s_3, s_4) \leftarrow$ FINDNONDETERMINISM($S, T$) do
13:        MERGE($S, T, s_3, s_4$)
```

3.3.1 Prefix Tree Acceptor

The first stage of Algorithm 1 is to transform the observed traces into an initial model called a *prefix tree acceptor* (PTA). Traces can be expressed as $p \cdot s$, where p is a *prefix* and s is a *suffix.* Thus, two traces which share a common prefix can be distinguished by their suffixes. In a PTA, traces with the same prefix share a common path through the model up until the point of divergence. For classical FSM models, this guarantees that the PTA is deterministic as there is a maximum of one outgoing transition for any given action from any given state. For models that support outputs, PTAs built from traces are not guaranteed to be deterministic since transitions have the potential to produce different outputs in response to the same action. This is discussed more in Chapter 6.

A PTA is built from a set of traces by starting with an initial state and iteratively attempting to walk each trace in the model. When a state is reached from which there is no outgoing transition for the current event, a new transition is added to a new destination state. From this point, new states and transitions are required for all subsequent events in the trace. The PTA built from the traces in Figure 3.4 is shown in Figure 3.5.

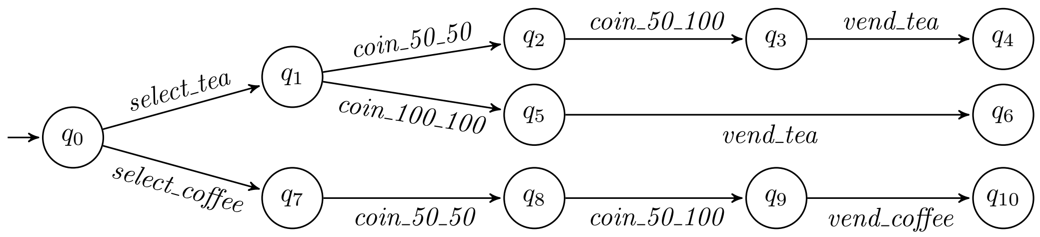


Figure 3.5: A PTA built from the traces traces in Figure 3.4.

It is here that the concept of prefix closure from Section 2.1 becomes important to inference. If we consider software systems to be prefix closed, we do not need to be concerned with explicit accept states. The rationale behind this is that accepting states are often looked upon as being “final” states, but software systems often do not have a concrete notion of a “complete” interaction. It is possible to stop interacting with a system at any time and have the trace up to that point be a valid trace of the system. Here, we are less concerned about whether a particular string is in the *language* of a model. Rather, we are interested in whether our inferred model can *respond* to every event in a given trace.

Additionally, many software systems are designed to operate for extremely long periods of time. This means that traces may also be extremely long. To apply inference successfully, it may be necessary to use trace *prefixes* of a reasonable length. This is especially true if we need to carry out offline learning of an online system which continues to produce trace data even as we are ready to begin inference. We can only do this, however, if we can assume prefix closure.

In our drinks machine example, it is quite possible for a customer to select a drink, insert coins to the value of a pound, and then walk away. The system has not behaved incorrectly here just because the customer has neglected to press *vend* to dispense their drink. Because of this, we treat all reachable states as implicitly accepting because a path to a state in the model represents a trace of the underlying system. Since the PTA is built only from positive traces, all paths through the model must represent feasible system behaviour.

A result of the property of prefix closure is that once a trace has revealed itself as negative, that is, it contains an event which does not represent feasible system behaviour, it cannot become positive again. That is not to say, however, that prefixes of negative traces are also negative. On the contrary, most negative traces have positive prefixes. For example, the trace ⟨*select_tea*, *coin_100_100*, *vend_coffee*⟩ is a negative trace of the vending machine since we receive a different drink to the one we selected, but the prefix ⟨*select_tea*, *coin_100_100*⟩ is perfectly valid behaviour.

3.3.2 Scoring Merges

Any PTA built from a sufficiently large number of traces will likely contain duplicate instances of the same program state. That is, if there are multiple ways to reach the same program state, that state will be represented in the PTA multiple times. The state merging challenge is to determine these duplicates and merge them into a single state in the model. Essentially this is a global optimisation problem of which states to merge in order to produce the best possible model.

To tackle the state merging challenge, we need some means of determining which states are compatible for merging. While k-tails [16], and many active techniques such as [142], divide states into equivalence classes, most passive inference algorithms merge states pairwise. Thus, we additionally require a way of ordering the state merges. In Algorithm 1, the CHOOSEPAIRS function determines and ranks the compatibility of state merges. The SELECTPAIRS function first determines all the state pairs which are compatible for merging. Each compatible state pair is then assigned a numeric score by the SCOREMERGE function, representing the compatibility of the merge. The potential merges are then sorted highest to lowest according to this score. It is these two functions which are the novel component of most of the state merging algorithms in the literature [98, 121, 53], but they are usually at least partially based on the commonality of outgoing transitions, the idea being that states from which we can perform the same actions are likely to be equivalent.

It is worth pointing out here that the task of determining transition equivalence is trivial for classical FSM models because actions only have a label, so transitions are equivalent iff their labels are equal. This is not so of EFSM models as their actions have additional inputs, which may be guarded, and may produce outputs by evaluating functions. This means that there is more than one way of expressing the same behaviour. Thus, the task of determining transition equivalence is much more difficult and is, to an extent, open to interpretation.

Returning to our running example, let us score state pair merges for the PTA in Figure 3.5. To make things as simple as possible, the score of each state pair will be the total number of outgoing transition pairs which share a label. Let the CHOOSEPAIRS function return those pairs of states which have at least one pair of outgoing transitions with the same label. Breaking ties based on proximity to the root, we get the following ordered list of state merges.

$$[(1,(q_1,q_7)),(1,(q_2,q_8)),(1,(q_3,q_5))]$$

States q_1 and q_7 both have an outgoing *coin_50_50* transition; states q_2 and q_8 both have an outgoing *coin_50_100* transition; and states q_3 and q_5 both have an outgoing *vend_tea* transition. Hence, the three pairs all have a merge score of one. The pair (q_1, q_7) is the first merge in the list as it is the closest to the root.

3.3.3 Merging States

Having determined and ordered the state pairs we wish to merge, we can then begin the merging itself. Merging states is a relatively simple process and is explained as follows. A pair of states q_x and q_y is merged into a single state $q_{x,y}$ by replacing all instances of q_x and q_y in the transition function with a new state, $q_{x,y}$, such that all transitions which previously left and arrived at q_x and q_y individually now respectively leave and arrive at $q_{x,y}$. The two states q_x and q_y can then safely be removed from the model. This is often implemented by replacing instances of q_x with q_y and then removing q_x from the model, or vice versa.

Figure 3.6 shows the PTA from Figure 3.5 after having merged states q_1 and q_7. Here, the transitions *select_tea* and *select_coffee,* which arrived at states q_1 and q_7 respectively in Figure 3.5, now both arrive at the merged state $q_{1,7}$. Similarly, the two *coin*(50)/[50] transitions, which left the separate states in Figure 3.6, now both leave the single merged state.

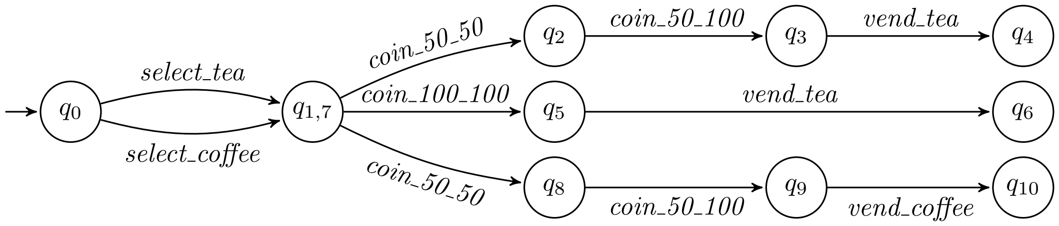


Figure 3.6: The PTA from Figure 3.5 after merging states q_1 and q_7.

3.3.4 Resolving Nondeterminism

As Figure 3.6 shows, merging states which have common outgoing transitions leads to nondeterministic models. This is not the intended consequence of state merging, though, as we merge states which we believe share the same behaviour. The "nondeterministic" model which results is then not nondeterministic at all. Rather, it has two representations of the same behaviour, one from each of the newly merged states.

A resolution to this is presented in [121] in the form of the dMerge operation. The idea here is that if we have two representations of the same behaviour from a given state, the respective destinations of those transitions are duplicate representations of the same program state. We can thus merge the two destinations into a single state without compromising the model. This is likely to introduce further nondeterminism to the model which can be resolved the same way, leading to a *zipping* effect occurring along branches of the PTA.

In classical FSM inference, merging the destination states is sufficient to resolve nondeterminism since two transitions with the same label, origin, and destination are indistinguishable. For models with more complex transitions, this is not the case as it may be possible for two non-identical transitions to express the same behaviour. This is a significant challenge in EFSM inference and is the subject of Chapter 5.

In our running example, state $q_{1,7}$ has two outgoing *coin_50_50* transitions which lead to states q_2 and q_8 respectively. If the two transitions represent the same behaviour, their destination states must represent the same program state, so should be merged. The resulting model then has a state $q_{2,8}$ with two outgoing *coin_50_100* transitions. Merging states q_3 and q_9 resolves this nondeterminism and results in the model shown in Figure 3.7.

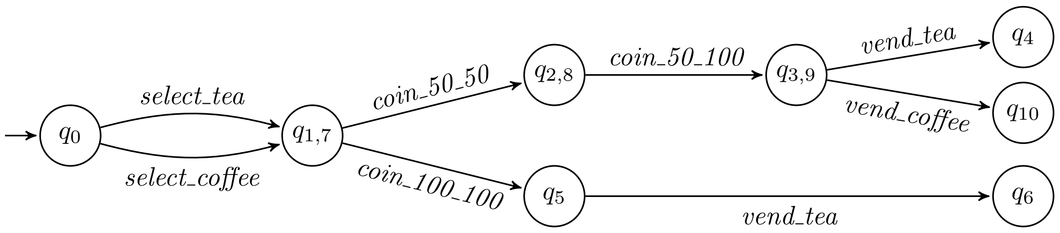


Figure 3.7: The PTA from Figure 3.5 after merging states q_1 and q_7 and resolving nondeterminism.

3.3.5 Iterative Merging

Once all nondeterminism has been resolved, another iteration of state merging begins and the next highest scoring pair of states is merged. In our example, this should be q_2 and q_8, but these were already merged as part of the nondeterminism resolution of the previous merge. We thus skip straight on to merging states $q_{3,9}$ and q_5, which both have an outgoing *vend_tea* transition. This then leaves us with two outgoing *vend_tea* transitions from $q_{3,5,9}$ with different destinations. Following the same process as before, we merge states q_4 and q_6 to resolve the nondeterminism, resulting in the model in Figure 3.8.

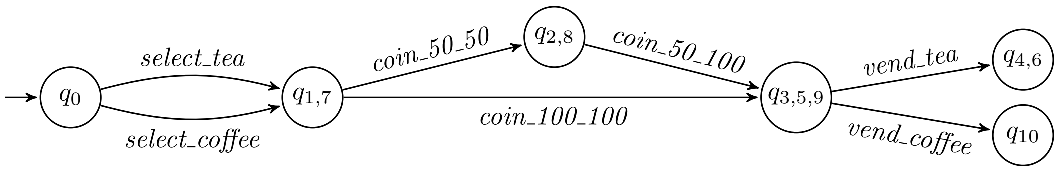


Figure 3.8: After merging states q_3 and q_5 and resolving nondeterminism.

There are now no more states which share a common outgoing transition, so the merging process ends here. Let us now pause to consider the model in Figure 3.8. Clearly it is smaller than the original PTA in Figure 3.5, which makes it slightly more readable. It is also now apparent that, no matter whether we insert a single one pound coin or two fifty pence pieces, we can still acquire either tea or coffee. This was not so in the original PTA as the trace in which a one pound coin was used to pay for coffee was not in the training set. Thus, we have inferred a model which correctly predicts the behaviour of the system for an unseen action sequence.

What is not so desirable in Figure 3.8 is that the trace $\langle$*select_tea, coin_100_100, vend_coffee*$\rangle$ is a valid trace of the model. Indeed, this is also a trace of Figure 3.7, and comes as a result of merging states q_1 and q_7. While there was no evidence to suggest that the inference process should not have merged these states, we know from basic contextual knowledge that we should not be able to receive a different drink to the one we selected. This is a typical example of *overgeneralisation,* which is an occupational hazard of passive inference. If, as mentioned in Subsection 2.1.4, we could give the above trace to the inference process as a *negative trace,* we could prevent the merge from going ahead and thus avoid the overgeneralisation. We could also prevent such overgeneralisation by using *active* inference such as [40] to query an oracle to find out whether introducing this trace to the inferred model is acceptable.

In fact, the only state pair it is safe to merge in this example is q_3 and q_5. This results in the model in Figure 3.9. While this model captures the property that we can only ever receive the drink we selected, it is an undergeneralisation in that it only allows us to pay for coffee with two 50p coins. Thus, model inference is a trade off between over- and under- generalisation.

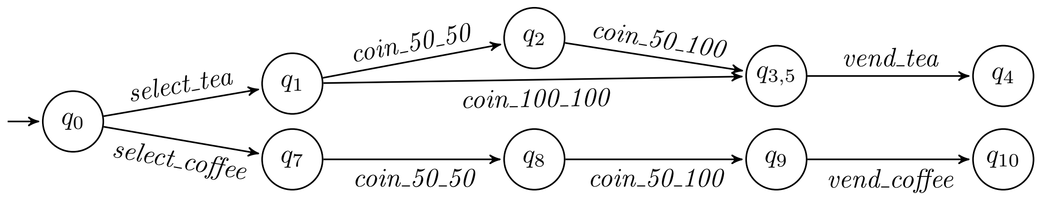


Figure 3.9: After merging states q_3 and q_5 and resolving nondeterminism.

3.4 Passive FSM Inference

This section provides an overview of the key state merging algorithms in the literature. As the field of EFSM inference is relatively new, most existing inference algorithms produce classical FSMs. The current state of the art EFSM inference algorithms is given in Section 3.6.

3.4.1 Bierman's k-tails Algorithm

The first state merging technique in the literature is the famous k-tails algorithm [16]. Like all subsequent state merging algorithms, it is based on the Nerode relation [118], which states that a language is *regular* (i.e. there exists a DFA which recognises it) if it has finitely many equivalence classes. The problem is that this relation requires access to *all* strings in the language, which is clearly impractical, especially if the language is infinite. To solve this, an alternative relation is proposed in [16] which only requires a finite subset of the language. The challenge is then to infer a model that not only expresses this behaviour, but can also *predict* how the system might behave when faced with input sequences that were not in the subset used to infer the model.

The k-tails algorithm is so named because of how it chooses which states to merge. Modern state merging algorithms tend to merge states pairwise, as in Algorithm 1, but k-tails is a little different. Here, states in the PTA are partitioned into *equivalence classes* according to their k-future, where a k-future is a path of length k from a given state. States which share the same set of k-futures are assumed to represent the same program state. The equivalence classes are then each merged into a single representative state. By merging equivalence classes rather than state pairs, k-tails is much closer to the original Nerode relation.

As we have seen in Subsection 3.3.4, merging states based on their k-future inevitably leads to nondeterministic models. States $q_1, \ldots, q_n$ which share the same k-future all have an outgoing transition with the same label. The resulting merged state $q_{1,\ldots,n}$ now has n outgoing transitions, one from each of its constituent states. The original technique described in [16] makes no explicit attempt to resolve this. Instead, it is proposed that the value of k be iteratively increased and the algorithm rerun until a deterministic model is achieved. Higher values of k lead to a higher number of smaller equivalence classes and, hence, less nondeterministic models. If k is greater than or equal to the length of the longest trace used to build the initial PTA, then no states are merged and the model is guaranteed to be deterministic.

3.4.2 Evidence-Driven State Merging

Competitions such as Abbadingo One [98] and StaMInA [151] have driven the field increasingly towards state merging algorithms. One class of algorithms which resulted from these competitions is the Evidence-Driven State Merging (EDSM) algorithm. The Blue-Fringe algorithm, which won the Abbadingo One competition [98], is the first algorithm to be described as such.

A major limitation of all state merging algorithms is that there is no way to be certain that the state merges we make are correct. As the inference process proceeds, incorrect merges tend to "snowball" into increasingly inappropriate models. The idea behind Blue-Fringe, and EDSM as a whole, is to mitigate this problem by merging states pairwise in order of the amount of *evidence* that the two states are the same. Here, evidence is provided by a *scoring function* which assigns a numeric value to every pair of states, representing the appropriateness of their merge. We see this as the scoreMerge function in Algorithm 1. We can then order our state merges by their appropriateness, enabling us to make good merges earlier on in the process.

As it was the first EDSM algorithm, let us examine the Blue-Fringe algorithm [98] in more detail. The algorithm begins by constructing a PTA from the initial traces. The root node is coloured red and its children coloured blue. The remaining states are left uncoloured. The choosePairs function returns all state pairs where the first state is red and the second state is blue. The score of each red-blue merge is decided based on the number of states removed by the complete merge (with resolution of nondeterminism). If a particular blue state cannot be merged with any red state, it too is coloured red and its uncoloured children coloured blue. When every state is coloured red, there are no more possible merges and the algorithm terminates. In this way, a "blue fringe" moves outwards from the root node as states are merged.

A fresh set of tests after Abbadingo One showed that Blue-Fringe is a fairly powerful technique. Indeed, the generalised version of EDSM shown in Algorithm 1 forms the basis for a number of (E)FSM inference algorithms in the literature [53, 121, 152] as well as my own inference algorithm, presented in Chapter 6.

3.4.3 Negative Traces

As we saw in Subsection 3.3.5, sometimes merging a particular pair of states introduces undesired behaviour to the model. A key contribution of the RPNI algorithm, presented in [121], is the idea of using negative traces as part of the inference process, as mentioned in Subsection 2.1.4. At each stage of inference, the model is checked against these traces to ensure that they are still rejected. If a state merge causes the model to accept any of the negative traces, the inference process backtracks and skips to the next potential state merge.

In our example, the inclusion of ⟨*select_tea*, *coin_100_100*, *vend_coffee*⟩ as a negative trace reveals that merging states q_1 and q_7 is a mistake. So too would have been the next state merge of q_2 and q_8. The only state pair that can be safely merged here is the final one, q_3 and q_5.

It is often the case that it is only the last event of a negative trace which exemplifies the negative behaviour. For example, in the trace ⟨*select_tea*, *coin_100_100*, *vend_coffee*⟩, it is only the dispensing of coffee which is illegal behaviour. The rest of the trace up to that point is perfectly valid. If this is known to be the case, then the positive prefixes of negative traces can also be included in the PTA.

3.5 Active FSM Inference

Another way to avoid overgeneralisation is to allow the inference process access to some kind of oracle to direct the inference process. Techniques which make use of an oracle are referred to as *active* learners. The first algorithm to do this was Angluin's famous L^* algorithm [10], upon which many subsequent inference algorithms are based [15, 28, 52].

Active inference is often presented in terms of the *minimally adequate teacher* model [10]. Here, the inference process is an interaction between a *learner* and a *teacher.* It is the learner's task to determine a suitable model of a language and the teacher's job to answer questions from the learner. The teacher answers two types of question: membership queries, and equivalence queries. Membership queries consist of the learner asking "Is this string in the language of the target model?", and the teacher replying either "yes" or "no". Equivalence queries consist of the learner presenting a candidate model to the teacher. If the presented model is equivalent to the target model, the teacher accepts it. If not, the teacher provides a *counterexample* of a string which is in the target language and is not accepted by the presented model.

As with all automata learning algorithms, active inference is based on the Nerode relation, the problem being to infer an infinite language classification from finite set of strings for which membership queries have been performed. Where passive inference must rely solely on the traces it is given, active inference can expand this set with queries, the general aim [30] being to construct two sets, U and V, where U is the set of *short prefixes* which represent each Nerode class (i.e. states in the model) and V is a set of *suffixes* which distinguish non-equivalent strings. Here, V represents an overapproximation of the Nerode equivalence relation.

L^* learns a DFA for a given alphabet Σ by beginning with the empty word and iteratively extending the prefixes in U with members of Σ, performing membership queries until U is *closed.* Formally, $\forall u \in U. \forall a \in \Sigma. \exists u' \in U. u \cdot a \textit{is Nerode equivalent to } u \textit{ WRT } V$. At this point, the hypothesis model is presented to the teacher for classification. If it is correct, the algorithm terminates. If not, the teacher provides a counterexample and the process continues. Because the teacher is able to classify correctness of the model, most active inference algorithms can be proven to be correct, at least to a degree.

The main problem with [10] is acquiring an oracle. Any realistic software system would require many hundreds of membership queries to be answered in order to construct a model. In practice, we are also often unable to specify exactly whether a candidate model is "correct". Nevertheless, the attraction of provably correct models is strong, and this has motivated a wealth of literature concerning the automation of the oracle role to infer both classical and extended finite state machines.

Query-Driven State Merging

Inspired by RPNI and the Blue Fringe algorithm, the Query-driven State Merging (QSM) algorithm [53] incorporates *membership queries* from the L^* algorithm into a state merging technique. The idea here is to allow the user to guide the inference process, but to use the evidence-driven approach to help reduce the number of queries to the user. Like RPNI, the algorithm takes in both a set of positive traces and a set of negative traces. The basic outline of the algorithm is as in Algorithm 1. A PTA is built from the positive traces and then compatible pairs of states are merged according to the implicit order on strings, as per the RPNI algorithm. The key contribution of QSM is what happens after this.

When an intermediate solution is compatible with both the positive and negative traces, new *scenarios* (i.e. traces) are generated and presented to the user for classification as positive or negative. Before merging a pair of states q and q', query strings are generated based on those strings which enter the language of the machine as a result. These strings take the form xvw, where x is the shortest string that gets us to state q', and vw is a suffix of q. By construction, xv is already an accepted behaviour of the model, so it is just the behaviour of w which is unknown. Queries can thus be presented in the form of a question: “After having done xv, can the system do w?”. The end user answers “yes” or “no” and the string is added to the correct set. Only if all the query strings are classed as positive can the merge go ahead. This process of querying and merging is iterated until no more state pairs are compatible.

Of course, with any active inference technique, a key evaluation factor is the number of queries generated. Experimental results in [53] show that the Blue-Fringe merging strategy generates far fewer queries than RPNI. The number of queries can be further reduced by providing the algorithm with *domain knowledge* if it is available. Even so, for non-trivial systems, the technique still generates enough queries to inhibit its applicability.

Automating Queries

Large numbers of queries become less of a problem if the classification step can be automated. This is the motivation behind the work presented in [149], which is a modification to the QSM algorithm that uses *dynamic analysis* so that the system under inference can act as its own oracle. The idea here is that the scenarios generated by QSM should correspond to real executions of the system. As such, the scenarios can be transformed into sequences of method invocations which can be executed on the real system. In the implementation presented in [149], the user is asked to provide an XML file corresponding to the execution of the provided scenario. Full automation of this step is listed as future work. Experimental results on two systems show that the technique is capable of producing accurate models with relatively sparse training data.

The main drawback of [149] is that the user must manually run the system to produce the XML files for the inference process. In [137], full automation of query evaluation was realised. The work presents a technique based on an improved implementation of QSM called StateChum[1] for automatically inferring models of the behaviour of Erlang modules. The key contribution is a wrapper system to dynamically execute the membership queries generated during inference. This wrapper system converts the scenarios generated by StateChum to sequences of executable Erlang methods. These traces are then executed and classed as either positive or negative without the need for human interaction. The major advantage of this is that the system can evaluate hundreds of queries per second and does not tire, become impatient, or make mistakes as would a human oracle. A major limitation of this technique is that some functions require additional input parameters which must be determined. It is suggested in [137] that such parameters can be captured by code instrumentation, but this requires the modification of the source code of the system, which may not always be possible.

Using Existing Models

Another way to reduce the burden on the user is to use existing models of the system, for example from previous versions, to help direct the inference progress. The idea, first proposed by [76], is

[1]`http://statechum.sourceforge.net` (Accessed 28/08/20)

to use inaccurate but not entirely irrelevant models to help with model learning and checking when using L^*-like algorithms. Various techniques have sprung up around this [41, 76, 89], with the field taking the name *adaptive learning.* The results of [76] show that existing models can be useful when changes are minor, however [41] shows that the utility of such models degrades over time as the underlying software system evolves.

The main limitation of adaptive learning is that we cannot use it when there are no existing system models. This inevitably leads us to question the origin of the first system model, since it must have been created using a different technique. With that in mind, it is not difficult to imagine using one of the many passive inference techniques in the literature to come up with a "first guess" model which could then be used as an oracle for an adaptive learner, although there does not appear to be any work in the literature investigating this.

Temporal Constraints

Perhaps the main limitation of the techniques proposed by [137] and [149] is that we cannot always run the system as part of the inference process. It may be computationally expensive to run traces on the real system or, if model inference is used during the *requirements capture* phase of development, the system may not exist yet. This motivated the work presented in [148], which is another modification of QSM that allows the user to specify *temporal constraints* specified in linear temporal logic (LTL), rather than having to classify individual traces.

The main idea of [148] is that temporal constraints discharge many trace queries at once by specifying general properties instead of specific instances of behaviour. For example, the user could specify "event y can never follow event x", which immediately classifies all traces of the form $\ldots x \ldots y \ldots$ as negative. Adherence to the provided constraints is verified automatically during inference by a model checker, with any generated counterexamples being added to the set of negative traces. This process can infer models fully automatically, however [148] also presents an active variant of the algorithm which generates additional queries in a similar manner to QSM.

Experimental evaluation on two case studies shows that not only does the provision of LTL constraints significantly reduce the number of queries to the user (when using the active variant of the algorithm) but also enables more accurate models to be produced, even with sparse training data. There is an obvious limitation to this technique, however, which is the ability of the user to generate reasonable LTL constraints.

3.6 Passive EFSM Inference

Thus far, all the inference techniques that I have discussed have been for classical FSM models with atomic transitions. The first real technique for inferring EFSM models from traces appears in [106]. This technique, although rather limited, remained the state of the art of passive EFSM inference for eight years until the publication of [152]. In this section, I review the state of the art of passive EFSM inference, before turning the focus to active inference in Section 3.7.

3.6.1 Automatic Generation of Software Behavioural Models

The work of [106] presents a technique called GK-tails, which extends Bierman's k-tails algorithm [16] to models with internal variables. Models are inferred from program execution traces by augmenting classical FSMs with guards inferred by Daikon [59].

GK-tails is designed to work with *white-box* traces, as shown in Figure 3.10, which reveal the values of internal system variables. Here, the anterior value of each variable is shown as part of each event, with the input to each action being a dummy variable called `input`. It is not really possible to show the output of transitions here as all variable values are taken prior to the transition executing. One possible workaround for this is to model the output of the previous transition as one of the inputs to the current transition, however this means that the last transition in each trace is left without an output.

$$\begin{array}{l}
\langle \\
\quad \textit{select}(\texttt{input} = \text{“tea”}), \\
\quad \textit{coin}(\texttt{input} = 50, \texttt{selected} = \text{“tea”}, \texttt{value} = 0), \\
\quad \textit{coin}(\texttt{input} = 50, \texttt{selected} = \text{“tea”}, \texttt{value} = 50), \\
\quad \textit{vend}(\texttt{selected} = \text{“tea”}, \texttt{value} = 100) \\
\rangle, \\
\langle \\
\quad \textit{select}(\texttt{input} = \text{“coffee”}), \\
\quad \textit{coin}(\texttt{input} = 50, \texttt{selected} = \text{“coffee”}, \texttt{value} = 0), \\
\quad \textit{coin}(\texttt{input} = 50, \texttt{selected} = \text{“coffee”}, \texttt{value} = 50), \\
\quad \textit{vend}(\texttt{selected} = \text{“coffee”}, \texttt{value} = 100) \\
\rangle, \\
\langle \\
\quad \textit{select}(\texttt{input} = \text{“coffee”}), \\
\quad \textit{coin}(\texttt{input} = 100, \texttt{selected} = \text{“coffee”}, \texttt{value} = 0), \\
\quad \textit{vend}(\texttt{selected} = \text{“coffee”}, \texttt{value} = 100) \\
\rangle
\end{array}$$

Figure 3.10: Traces of the simple drinks machine as would be input to GK-tails.

The inference algorithm is completely automated and has four main steps. The first step is a preprocessing step to merge input-equivalent traces. Two traces are said to be *input-equivalent* if they contain the same sequence of method calls which differ only in the input parameters. For example, the first two traces in Figure 3.10 are input equivalent because both involve a *select* action, two *coin* actions, and then a *vend* action.

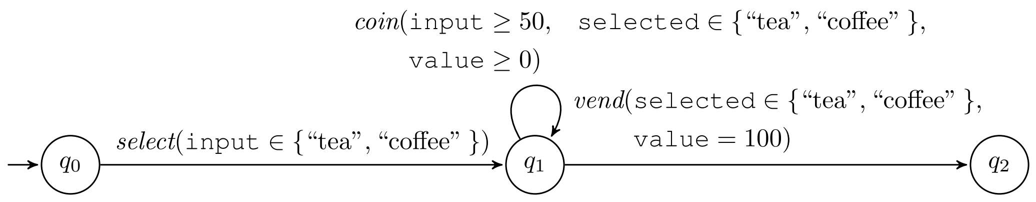
The second step involves using Daikon to derive guards from the traces. For each action, a predicate is generated which generalises the observed concrete data values. For example, the *select* action is called with the inputs “tea” and “coffee”. Daikon could then generate the guard $\texttt{input} \in \{\text{“tea”}, \text{“coffee”}\}$. The *coin* action is called with inputs 50 and 100. Daikon might generate the guard $\texttt{input} \in \{50, 100\}$, but the guard $\texttt{input} \geq 50$ also accounts for the observed values. Indeed, there are an infinite number of potential guard expressions here. Some are, of course, more sensible than others, but it is extremely difficult to evaluate their suitability without some prior knowledge of the system. This issue will be revisited in Chapter 7.

The next step is to create the initial EFSM. This is an automaton in which each trace in the set forms a separate path through the machine, with the only shared node being the root. Each method call forms a transition guarded by a predicate derived by Daikon in the previous step. This makes GK-tails one of the few inference algorithms which does not begin with a PTA.

The final stage is to merge compatible states. Compatibility is determined in a similar way to k-tails [16] — by comparing the k-futures of states — but, rather than dividing states into equivalence classes, they are merged pairwise like in classical EDSM algorithms. A pair of states s and s' are merged by removing state s and adding its transitions to s' if there is not already an existing transition for that action. For existing transitions, the guards of the transitions to and from s' are extended to include the predicates of the corresponding transitions to and from s. This process terminates when there are no more pairs of equivalent states.

Since the initial set of traces is likely to be incomplete, strict equivalence between them is unlikely. Hence, [106] defines some alternative criteria which can be used to determine the compatibility of state merges. A state *weakly subsumes* another state if the method calls of its k-tails are equal but the predicates of the transitions differ such that those of the subsuming state are more general. A state *strongly subsumes* another state if the k-tails of the subsumed state are contained within those of the subsuming state and the predicates differ such that those of the subsuming state are more general.

Figure 3.11 shows an EFSM that the GK-tails algorithm might infer from the traces in Figure 3.10. Although [106] calls the inferred models EFSMs, they are not as extended as they might be. Transitions can place guards on global data variables, but do not have the capacity to *mutate* these variables. While the inferred models are very useful for describing *what* values are normal and acceptable, they do not show *how* variable values evolve as the model executes.



q_0
$select(\texttt{input} \in \{\text{“tea”}, \text{“coffee”}\})$
q_1
$coin(\texttt{input} \geq 50, \texttt{selected} \in \{\text{“tea”}, \text{“coffee”}\}, \texttt{value} \geq 0)$
$vend(\texttt{selected} \in \{\text{“tea”}, \text{“coffee”}\}, \texttt{value} = 100)$
q_2

Figure 3.11: An EFSM of the simple drinks machine as might be inferred by GK-tails.

To evaluate their algorithm, the authors of [106] carried out two small studies to investigate firstly the *usefulness* of the inferred EFSMs over their classical counterparts, and secondly the relative *sizes* of the models produced using the different merging criteria. For the first experiment, traces were gathered from several applications, fed into GK-tails, and the “interplay” between control and data analysed. Results showed that EFSMs are more “useful” then classical FSMs when representing complex interactions. Only one out of the 59 EFSMs with fewer than five states included any interplay between method calls and data values not representable by classical FSMs. As complexity increases, there is more interplay and the EFSMs come into their own. Additionally, the inferred data constraints seemed to be mostly semantically relevant to the transitions, representing specific implementation details or particular uses of the systems.

To investigate how the merging criterion affects model generation, the authors of [106] implemented a simple shopping cart application and developed two test sets. The first of these was designed to exercise the whole set of paths through the model. Some data conditions were tested well and some conditions were tested sparsely. The second test set was designed to sample both paths and data conditions poorly.

The results confirm that exact equivalence is a poor merging criterion when the traces do not exercise a system fully. For the first test set, the EFSMs produced using this criterion were undergeneralised and failed to identify some loops. Weak and strong subsumption both successfully generalised the machines and produced suitably compact models.

The cart application was then extended to make it slightly more complex, and a test set designed to explore the space unevenly was produced. In this case, both equivalence and weak subsumption failed to adequately generalise the models, but strong subsumption produced a suitably compact machine. The reason for this is that weak subsumption requires that the outgoing sequences of method calls be exactly equal, which means that it is likely to fail to recognise similarities between states for sparse trace sets. Strong subsumption, however, recognises these similarities so is able to produce smaller models.

A proposed use for the EFSMs generated by GK-tails is as a basis for automated test case generation. Preliminary results from [106] show that test suites generated from EFSMs produce a higher statement coverage than those generated from classical FSMs with data constraints inferred separately. No further study of this was carried out, but it is easy to see how EFSMs might contribute towards solving the weakness identified in [137], namely that method calls have arguments which are not detailed in classical FSM models so may not be known.

There are two main limitations of the technique proposed in [106]. Firstly, transitions cannot mutate the data state, so the inferred models do not show *how* individual control flow events affect the values of system variables. Secondly, the EFSM models produced by GK-tails may be nondeterministic, meaning that there may be a choice of transitions from a given state for a given action. For the models inferred by GK-tails, transition guards play a descriptive role. They summarise the observed input values rather than attempting to causally link specific data configurations to control flow events. While nondeterminism can serve as a useful abstraction technique, nondeterministic models fail to capture the explicit logical relationship between data and control. As discussed in Subsection 3.4.2, the nondeterminism that arises from merging states is not real nondeterminism, just duplicate instances of the same behaviour. Consequently, we should merge these duplicates into a single transition to represent both instances. The GK-tails algorithm makes no attempt to do this.

In addition to the technical limitations, it is also worth mentioning that the evaluation in [106] is rather informal. While the *size* of the inferred models is discussed in detail, there is very little analysis of their *accuracy*. It is mentioned that the equivalence merging criterion led to undergeneralised models, but this is not discussed quantitatively in terms of their predictive power for unseen traces. There is also no formal definition of the so called *interplay* between control and data, nor any quantitative analysis of this.

3.6.2 Inferring EFSM Models from Software Executions

Another EFSM inference technique called MINT is presented in [152]. MINT works by using data-classifiers to infer guards on FSM transitions which explicitly link the data state of the system to subsequent event actuation. The input traces and models produced are almost identical in style to [106]. The main limitation of [106] which MINT solves is the resolution of nondeterminism which arises as a result of merging states.

The MINT algorithm follows the same basic structure as all EDSM algorithms. The novelty here is the use of classifiers both to identify and prioritise potential state merges, and to help resolve the nondeterminism which arises as a result. Another key contribution of the technique is that it is *modular*: arbitrary data classifiers can be used, which widens applicability since not all classifiers are equally suited to all systems. The implementation supports over fifty of the algorithms implemented in the Java WEKA library [78]. The MINT algorithm builds on Algorithm 1 by including an extra step in which a set of classifiers is inferred. Each classifier corresponds to a method signature such that, for each method call, the classifiers serve to predict the name of the subsequent method to be called.

Once the classifiers have been inferred from the training set, the PTA is produced from the same set of traces. The key addition here is that it is labelled with sets of variable values which correspond to each transition. Pairs of states only share the same prefix if the classifiers produce identical predictions for every observed data configuration in the prefixes of both states.

Having built the PTA, the next step is to merge state pairs. During state merging, the choosePairs function takes an additional parameter. Normally, this would just be the states and transitions in the model. Here, choosePairs also requires a minimum score which deems a state pair suitable for merging. As well as scoring suitably highly, the respective data values of a pair of states must lead to the same classifier predictions if they are to be considered equivalent.

The merge function is fairly similar to the one in Algorithm 1 except in the way that nondeterminism is dealt with. Instead of simply merging transitions with the same label, the data values must also be taken into account, since different values may be treated differently such that two outgoing transitions which share the same label may still be deterministic. When two transitions are merged, their data values are also merged by updating the map from transitions to variables such that the merged transition contains the variable sets of both individual transitions.

The final stage of the merging process is to check that the resulting model is consistent with the classifiers. For a given transition, its variable sets are presented to its classifier and the subsequent transition is predicted. If the destination state does not contain an outgoing transition with that label, the model is inconsistent. In this case, the merge is rejected and the next highest scoring merge is attempted.

The implementation also includes a decorator class which integrates with Daikon. This produces models which are quite complex but does provide some basic data constraints if the chosen classifier is unable to do this. The Daikon guards provide an alternative view of the data and can give insight into *why* the classifiers have inferred the rules they have.

MINT was tested with a collection of random traces from a number of systems. To assess the accuracy of the inferred models, negative traces were synthesised and fed to inferred models, the expectation being that they should be rejected. Results show that the choice of k for scoring state merges seems to be the main factor that affects the accuracy of the inferred models. As with [16], smaller values of k produce smaller models, but this is not necessarily better. In some cases, the less overgeneralised models resulting from larger values of k were more accurate overall. That said, there were specific instances where the choice of classifier was critical to the accuracy of the system. Additionally, there was no single optimal configuration. Instead, the best choice seems to depend on the characteristics of the system under inference.

The implementation is reported to run in reasonable time, however the choices of k and the classifier affect the runtime dramatically. Unsurprisingly, the runtime grows with the number of source traces. The choice of classifier here is instrumental as some models, such as naive Bayes, are particularly time-consuming.

The main limitation of [152], which is even highlighted in the work, is that the inferred models are still only declarative. Like in [106], they capture sequences of events and data configurations which are possible, but do not describe *how* individual variable values are changed. This means that the inferred models are useful only for monitoring systems and cannot be used to predict system behaviour for unseen traces.

3.6.3 Inferring Computational Models

The work presented in [152] is a key contribution to the field of EFSM inference but, as the paper itself admits, the models do not properly reflect the underlying system. The *update functions* — the functions which dictate *how* transitions mutate variables — are not inferred. This is an obvious requirement if we want to predict system behaviour rather than just monitor it.

An obvious place to look for a solution to this problem is the field of *machine learning*, which has spawned many techniques over the years (such as deep learning and neural networks) to solve the problem of using training data to predict outcomes for unseen inputs. While the methods developed here can achieve truly astonishing predictive accuracy, they rarely give much insight into the underlying data transformations. We would like human-readable expressions which provide this understanding. The space of possible expressions is clearly too vast to expect to find ready solutions, so we must find ways to explore it intelligently. *Evolutionary algorithms* (EAs) are one way of doing this, and have been used to great effect in [150] as a post-processing technique to infer variable update functions for the models produced by MINT [152].

This technique begins with an EFSM, as might be inferred by MINT [152] or GK-tails [106]. The first step is to build what are referred to as *training sets* for each variable of each transition. These are built by running the traces through the inferred model and identifying the anterior and posterior values of each variable for each step.

The second stage is the inference of functions which account for the behaviour exemplified in the training set. This is done using genetic programming (GP), a technique which uses the principals of EAs to infer expressions which generalise sets of input-output pairs. Here, a population of candidate solutions is evolved over successive generations, iteratively producing functions which better account for the behaviour of the training set until an optimum is reached.

A key aspect of GP is that it works with a predefined fixed set of operators. In [150], these are $+, -, \times, \div$ as well as powers, n^{th} roots and casting from a float to an integer (and vice versa) in order to allow variables of differing types to interact. The `cast` operation is necessary as the GP employed here is strongly typed, meaning that integer and float variables would not otherwise be able to interact with each other in binary operations such as addition.

In addition to the *non-terminal* arithmetic operations, there is also a set of *terminal* operators. These are either variable names or constant values. In the case of [150], the set of terminals for each type is made up of all variables of that type, the literal values seen in the traces and, in the case of numeric values, some additional constants. For doubles, these are $\{0.0, 0.5, 1.0, 2.0\}$. For integers, they are $\{0, 1, 2\}$. Boolean values get $\{\mathit{true}, \mathit{false}\}$. The current implementation focuses primarily on numerical functions. The inference of non-trivial functions over other types, such as strings and lists, is desirable future work.

The GP algorithm used in [150] produces an initial population of functions as random compositions of terminals and non-terminals. Each individual in the population is evaluated according to the difference between the expected and actual outputs. The next stage is to select good individuals which can be "bred" to form the next generation by combining characteristics from both *parents.* New characteristics are introduced to the population through *mutation.* This involves replacing part of an individual with a new randomly generated sub-expression. The generational loop terminates when either an individual has been found which scores correctly on all samples from the training set, or after a preset number of generations.

The technique was tested with two programs: cruiseControl [23] and liftDoors [136]. The accuracy of the technique was assessed using k-folds cross validation [95] with k set to 10. Over k iterations, data from $\mathrm{k}-1$ fields is used to train the system and the remaining portion is used to evaluate the model. To mitigate for the stochasticity of GP, the algorithm was repeated and evaluated 30 times with different random seeds. The accuracy was evaluated by comparing the values calculated by the model with the ones from the trace using the root mean squared error (RMSE) between two traces x_1 and x_2. Since this metric is scale dependent, it is important to *normalise* this value to ensure that different case studies are comparable. To calculate the normalised RMSE, the RMSE is divided by the difference between the maximum and minimum values. This gives a value between 0 and 1, where 0 is no error and 1 is consistent large error.

The results of these experiments led to the following conclusions. Firstly, increasing the number of state variables decreases the accuracy. This can be explained by the fact that each function which must be inferred has many more variables to account for, increasing the probability of error. Error is also increased by the breadth of possibility, i.e. the number of outgoing transitions from each state. If this is increased, there are many more possible paths through the system which means that much more training data is needed for an accurate result.

There are two clear limitations here. Firstly, this technique only works with variables which are present in the traces. There is no concept of "hidden" or "internal" variables here. This means that the technique is reliant on comprehensive white-box traces which record the values of all variables throughout system execution. It cannot work with black-box traces that only contain information available to an external observer of the system since these traces do not capture the values of internal system variables.

The second, somewhat less obvious limitation, is that this is a post-processing technique. It works with an already inferred model and simply augments it with functions. The functions inferred here are essentially decorations. They play no part in the inference process and cannot be used to help infer a better model.

3.7 Active EFSM Inference

Recall from Section 3.5 that active inference algorithms make use of an oracle. Learning is usually portrayed as an interaction between a learner and a teacher. There is a wealth of literature concerning the active inference of various kinds of EFSM models from Moore and Mealy machines to register automata as per Definition 9, including the inference of output and update functions. This section reviews the most relevant and prominent of these techniques.

3.7.1 Regular Inference for State Machines with Equality Tests

Work presented in [15] extends Angluin's L^* algorithm to infer models with data parameters. To do this, the paper first introduces *symbolic mealy machines*, in which input and output symbols are parametrised. Thus, traces look very much like my own traces in Figure 1.1. The algorithm has two stages. In the first phase of the technique, a modification of L^* [119] for learning Mealy machines is applied with a finite data domain. This works in exactly the same way as the

L^* algorithm, iteratively performing membership and equivalence queries until the hypothesis model is correct. This results in a Mealy machine with concrete inputs and outputs. It is the second stage of the algorithm which is the main contribution of [15]. Here, the authors present a *heuristic* based on storing and reusing values to abstract away concrete input and output values into symbolic functions. For example, in our simple drinks machine, this technique captures exactly the idea that the the input of *select* is later used as the output of *vend.* Indeed, I implement a similar heuristic in Chapter 6 to recognise the "store and reuse" pattern.

The transformation from concrete Mealy machines to symbolic ones has four phases. The first phase is to search for values which must be stored. In essence, this is done by observing concrete input values which are later used as part of transition output. The next step is to replace the concrete values with symbolic ones, and add register updates to those transitions which need them. For example, the transition *select*("tea") becomes $select(i_0)/[r_1 := i_0]$. The transition $vend/o_0 :=$ "tea" would then become $vend/o_0 := r_1$. This step reveals the fact that many transitions in the model exhibit the same behaviour with different data, so the third step is to merge states with identical symbolic behaviour. This is done using a variation of the standard partition refinement algorithm to divide state pairs of the form (q, d), where q is a state in the model and d is an ordering on the data values. In the final step of the algorithm, the form of the transitions is changed to use guards rather than pattern matching equality. Transitions which differ only in their guards are merged by using their disjunction. For example, the transitions $s_m \xrightarrow{\alpha(i_0)[i_0=r_1]/[i_0]} s_n$ and $s_m \xrightarrow{\alpha(i_0)[i_0\neq r_1]/[i_0]} s_n$ are merged to become $s_m \xrightarrow{\alpha(i_0)/[i_0]} s_n$.

The authors then go on to prove the correctness of their algorithm, but do not present an implementation. The algorithm was later implemented as part of a tool called LearnLib [125], which was later built on in [87, 88] to further extend L^* to a subset of register automata.

3.7.2 Counterexample-Guided Abstraction Refinement

Another tool which builds off LearnLib is Tomte [4, 3]. The idea here is that previous techniques [15] can only learn models with small alphabets. The authors of [4] define the dual operations of abstraction and concretisation and propose that a *mapper* module is placed between the learner and the teacher. This enables a large set of concrete actions to be mapped down to a smaller set of abstract ones, effectively reducing the alphabet size while still maintaining expressivity.

A technique for inferring *scalarset Mealy machines* (parametric guarded Mealy machines in which the only data operations are literal (in)equality and assignment) is then presented. This technique begins with the empty mapper and adds elements when necessary. First, a call is made to LearnLib to construct a hypothesis model. If this is correct, the algorithm terminates. If it is not, a counterexample trace is returned and a new abstraction is constructed. This is done by first assigning a colour to each parameter value in the trace. Values are coloured green if they are equal to a previous value already in the abstraction table, black if they are equal to a previous value *not* in the abstraction table, and red if they are fresh. At this point, the mapper constructs a new trace from the counterexample in which all black input parameters are converted to fresh (red) values. If, after abstraction, the two traces are the same, this indicates that the hypothesis model is incorrect, and the counterexample is forwarded to the learner.

If the two traces are different, a new entry in the abstraction table is required. This is found by iteratively replacing black parameters in the counterexample trace with fresh (red) values and performing experiments to see if the outputs change. If there is a change in the output, this indicates that the value is relevant, and an abstraction is added to the table. In situations where replacing a black value with a fresh value does not change the output, this results in a counterexample traces with fewer black values. Upon finding the new abstraction, the learner is restarted with the new abstraction table. The Tomte tool was evaluated on several realistic case studies, including a biometric passport system [5], for which it was able to infer small and correct models in less than a minute.

3.7.3 RALib

The active EFSM inference techniques discussed so far have only been able to infer models in which literal (in)equality and assignment are the only operations which are allowed on data values. In [28], the SL^* algorithm [28] (the S standing for *symbolic*) is presented to address this limitation by expanding the set of allowed guard operations to include greater and less than operations and simple sums of registers and pre-specified constants. Updates remain as literal assignments only.

Recall from Section 3.5 that the process of active inference involves constructing a set U of short prefixes representing Nerode equivalence classes, and a set V of suffixes which define an overapproximation of the Nerode equivalence relation. The authors of [28] highlight the fact that the processing of trace suffixes from each location is obviously dependent on the variable values. Thus, the set V, rather than being formed of trace suffixes, is instead formed of *symbolic decision trees.* A symbolic decision tree (SDT) is, effectively, a prefix tree automaton. For a language $\mathcal{L}$, a (u, V)-tree $\mathcal{T}$ is an SDT where u is a concrete sequence of input actions and V is a set of symbolic suffixes, i.e. sequences of actions with restrictions on symbolic inputs rather than concrete values. For all strings uv, where $v \in V$, uv is accepted if it is in $\mathcal{L}$ and rejected otherwise. Further, in any run of $\mathcal{T}$, the register x_i may only contain the i^{th} data value in uv.

The SL^* algorithm has three phases.

Hypothesis Construction This involves making *tree queries* to a tree oracle and adding the results to an observation table. For a given $\mathcal{L}$, u, and V as defined above, a tree oracle is a function $\mathcal{O}_{\mathcal{L}}$ which returns a (u, V)-tree satisfying the following constraints:

1. Adding more symbolic suffixes cannot make inequivalent trees equivalent;
2. Adding more symbolic suffixes only refines trees and does not merge transitions or remove registers;
3. Trees are constructed recursively using representative data values (i.e. values which meet the constraints imposed by guards).

The table is *closed* if all transitions in the automaton have a target location, i.e. each input event has a corresponding transition, and is *register consistent* if all registers are in scope at the point when they are used. The algorithm proceeds similarly to L^*, asking tree queries and processing the results accordingly until the observation table is closed and register consistent.

Hypothesis Validation When the observation table is closed and register consistent, the associated model is submitted to the teacher for validation. The "teacher" is, again, simulated with dynamic analysis rather than being a human person. If the model is correct, the algorithm terminates, else there exists a sequence of inputs which is accepted by the hypothesis but not by the target system or vice versa. If this is the case, the teacher returns one such sequence as a counterexample to the current hypothesis model. Even with dynamic analysis, equivalence queries cannot be implemented in a black-box situation. This is because it is infeasible to run the system for every possible input sequence to check equivalence with the model. Instead, this paper proposes approximating the answer by exhaustive exploration of the set of data words up to some fixed time limit.

Counterexample Processing The existence of a counterexample means that the current model either has a missing state or transition, or that data registers have not been used correctly. Any given counterexample of length m can be written as $u_i v_i$ where u_i is a prefix of length i where both the incorrect hypothesis model and the correct model behave identically and v_i is the suffix of length $m - i$, starting with the first event where the behaviour of the two models differs. To determine the value of i, and thus the problematic transition, the counterexample is analysed step-by-step by asking tree queries for each event. Once i has been determined, there are then two cases which can be distinguished for the tree $\mathcal{O}_{\mathcal{L}}(u_i, V_i)$.

1. The guard g_i on the i^{th} transition of $\mathcal{O}_{\mathcal{L}}(u_i, V_i)$ distinguishes cases that the current hypothesis model does not distinguish. In this case, the hypothesised model must split this transition into two transitions which refine the behaviour of the original.
2. The tree $\mathcal{O}_{\mathcal{L}}(u_i, V_i)$ is not isomorphic to $\mathcal{O}_{\mathcal{L}}(u_{i-1}, V_{i-1})$ under the renaming of registers used to construct the hypothesis model. In this case, the state u_{i-1} is split into two separate states.

A major problem to solve with active inference is the manifestation of the oracle. In [28], tree queries $\mathcal{O}_u(V)$ are implemented using ideas for constructing canonical constraint decision trees [31]. Essentially, the set of distinguishable classes of traces of the form uv can be represented as a decision tree with maximally refined guards. The SMT solver Z3 [45] is then used to generate test cases for all such guards in the trees which can then be executed on the system under inference. Equivalence queries are then based on tree queries by constructing $\mathcal{O}_{\mathcal{L}}(\epsilon, w)$ for all $w \in \Sigma^k$, starting with $k = 3$ and iteratively increasing k to a fixed time limit or until a counterexample is reached.

A subsequent work by the same authors [27] presents a number of practical improvements to the SL^* algorithm, including the capability for models to work with multiple different data types and the capability to infer models which support instantaneous input and output values as per Definition 21. These improvements are then implemented as an extension to LearnLib. This tool, referred to as RALib, is then evaluated on a number of XML model benchmarks. The results show that SL^* uses fewer tests than Tomte and the previous RA learning algorithms [87, 88] in LearnLib.

3.7.4 A Myhill-Nerode Theorem for Register Automata and Symbolic Trace Languages

The authors of [70] argue that existing active learning algorithms, including [27] do not scale very well, noting that black-box approaches such as running test cases on the system under inference is a costly process. As a solution to this, the propose a *grey-box* method. That is, integrating some aspects of white-box program analysis (discussed in in Section 3.8) into black-box techniques.

In [142], the authors argue that existing grey-box methods [21, 34, 71] are all somewhat ad hoc, and present an extension to the Myhill-Nerode theorem for symbolic trace languages (i.e. traces with data like those in Figure 1.1) which is intended to serve as a foundation for future grey-box approaches. The Myhill-Nerode equivalence defines two words w and w' in a language $\mathcal{L}$ to be equivalent if there does not exist a suffix v which can distinguish between them. That is, only one of $w \cdot v$ and $w' \cdot v$ is in $\mathcal{L}$. As discussed in Section 3.5, a language is regular if and only if this relation has a finite index, meaning that there is a finite number of states in the corresponding DFA.

The authors of [142] define their extension to the Myhill-Nerode theorem by defining three equivalence relations $\equiv_l$, $\equiv_t$ and $\equiv_r$, which refer to the locations (states), transitions, and registers respectively. Since register automata have finitely many of each, the three relations all have a finite index. Intuitively, two finite traces are *location equivalent* if they lead to the same location, *transition equivalent* if they share the same final transition, and symbols are *register equivalent* if they are stored in the same register after the execution of the trace. A symbolic language is defined to be regular if these relations can be defined over it with finite indices. It is then shown how to construct a register automaton for a regular symbolic language and proven that these relations can be defined for any register automaton. This is a major step forward in the field of active inference, but an inference algorithm based on this relation remains to be presented.

3.8 White-Box Program Analysis

Black-box inference techniques aim to infer models of programs without knowing any details of their structure. By contrast, *white-box* program analysis looks inside the program to analyse its internal functionality. Such techniques include symbolic execution [26], concolic execution [73], and tainting [90]. Of these, it is symbolic execution which is the most well established technique.

3.8.1 Symbolic Execution

Symbolic execution involves constructing a tree structure representing the possible paths through a program up to a given depth. Instead of using concrete values for variables, symbolic values are used instead. These variables then flow through the various operations of the program accumulating conditions on their values. For example, we might encounter an `if ... then ... else ...` statement which tests to see if a particular variable x has a value greater than ten. This splits the execution path into two: one where the test was true, and one where it was false. We can also obtain information from variable assignments. For example, if our `if` statement assigns the value 100 to x if it is less than 10, we know its posterior value.

The models constructed by symbolic execution have states and paths through the system, and look very similar to EFSM models. Indeed, I apply a very similar technique to symbolic execution in Chapter 5 when formulating my subsumption relation for the merging of EFSM transitions. In the literature, tool such as SED [81] can be used to construct symbolic models which can be used for debugging Java programs. In addition, the symbolic execution trees can also be used in conjunction with the KeY verification system [80] to verify that a program satisfies a given specification written in the Java Modelling Language. Not only that, but KeY can also highlight specific nodes in the execution tree where the specification is violated, thus helping the user to debug the problem.

Another feature of KeY [130] is that it is able to reduce the size of symbolic execution trees by merging states. In [130], an abstraction-based framework formalised in JavaDL [14] is presented to allow large numbers of states in symbolic execution trees to be merged, thus reducing the size of proofs over them and enabling more complex programs to be verified. The main idea here is to abstract away concrete variable values into sets of possible states which overapproximate the original program. For example, the symbolic execution of **`if`** `condition` **`then`** `x=1` **`else`** `x=2` from a state σ results in a state which is identical to σ except that `x` is updated to 1 if `condition` is true and 2 if `condition` is false. Thus, we need two different paths for the two different options. If we were instead to define an abstract domain for `x` which covers both branches of the `if` statement, we avoid the need for two separate paths in the symbolic execution tree and so reduce the size of any proofs over it.

A concrete execution state is a pair (σ, φ) consisting of a Kripke state σ and a formula φ called the *program counter.* A concrete execution state, for a given program counter, assigns each variable a concrete value from the universe. For example, the symbolic state $(x := x, \{c > 0\}, \varphi)$ could be concretised such that x holds any positive integer. Thus, each symbolic state σ has a *set* of possible concretisations, denoted $concr(\sigma)$. The authors of [130] define a *weakenning relation* such that one symbolic state s_2 is a weakenning of another state s_1 if $concr(s_1) \subseteq concr(s_2)$. This relation is then shown to be a partial order relation.

The merging of a pair of states in a symbolic execution tree, involves the computation of a sound abstraction of both states. That is, a state which is a weakenning of the two states to be merged. This introduces a family of join-semilattices of symbolic execution states. The authors show that this is sound, meaning that if the merged state is valid then the two separate unmerged states are valid.

This technique is implemented in the verification tool KeY by extending the Java Modelling Language such that the user can annotate the program to be verified with the state merging join operator in front of a Java block after which the join should be applied, for example an `if` statement. The authors evaluated their technique with four "micro benchmarks" and a well known bug [42] in the default Java sorting method. These experiments show that state merging can reduce the size of KeY proofs by up to 80%, and that the verification of methods previous out of reach [42] due to the path explosion problem are now possible.

3.8.2 Grey-Box Inference

White-box techniques such as symbolic execution can also be applied in the context of model inference, and there are several tools in the literature [70, 21, 34, 71] which make use of them. Sigma* [21] is one such technique and is an extension of the L^* algorithm which infers *symbolic transducers* [146] that store the k most recent data values. The basic idea here is to use symbolic execution to infer constraints on input and output values and build a symbolic alphabet from which L^* can be run. Instead of checking equivalence queries using the actual program, Sigma* constructs a (possibly nondeterministic) symbolic overapproximation of the program and uses that instead. This means that the deterministic hypothesis models built from membership queries can be algorithmically checked for equality against the overapproximation model rather than having to iteratively search for a counterexample by running traces through the program, as in [28] and other black-box inference algorithms.

In [70], the authors extend the SL^* algorithm in RALib to use constraints from Python programs using tainting [90]. The idea here is to *taint* each data value in the traces with a unique marker. This allows values to be traced around the system under test and predicates over them to be generated. In an untainted setting, the oracle answers membership queries with “yes” or “no”, depending on whether the proposed trace is in the language of the target model. In the tainted setting, constraints on the data variables are also included in the response. For example, for a simple stack, we might have the trace $\langle push(7),\ push(7),\ pop(7)\rangle$. In the tainted trace, each 7 is given a unique marker such that, in the white-box setting, we can tell that the 7 of $pop(7)$ is the same value as in the *first* $push(7)$.

A key part of the technique in [70] is the construction of *characteristic predicates* on the tainted data values in the traces. Such predicates are sufficient to construct the symbolic decision trees used in membership queries of SL^*. Characteristic predicates are also very useful when testing equivalence. The “random walk equivalence oracle” in RALib simply constructs random traces to find a counterexample. In [70], symbolic suffixes of a given length are generated, along with their characteristic predicates. For each concrete trace satisfying the predicate, it is confirmed whether the hypothesis model and target model respond in the same way. If not, the trace is a counterexample. The use of characteristic predicates here allows counterexamples to be constructed intelligently, thus they can be found much more often than if they are constructed randomly. The authors note that RALib struggles with what they call “combination lock” systems. That is, systems where a given behaviour is only “unlocked” after a specific sequence of inputs. The use of characteristic predicates here reveals the inputs, enabling reliable discovery of counterexamples for such systems.

The authors evaluated their system by stubbing various versions of the Python FIFO-Queue and Set modules as well as several hand-crafted “combination lock” models. The results show that tainting can improve the performance of RALib by up to two orders of magnitude, but note that there is still a need to consume fewer tree queries and infer models involving more complex operations.

While it is clear that white-box methods can improve the performance of inference algorithms hugely, they require access to the source code of the system under inference. There are many situations where this is perfectly acceptable, but other scenarios (such as legacy or proprietary components) necessitate black-box learning. Further, this thesis aims to work in the passive learning scenario, where the learner is not allowed any interaction with the system under inference (or other oracle) beyond the traces provided at the start of inference.

3.8.3 Call Graphs

The field of white-box program analysis also opens up another use-case for EFSM inference. A common starting point for many program analysis techniques is some sort of call graph [129]. With such a graph, we can analyse control and data flow through the program, verify correctness, and search for bugs. There are many algorithms to construct such graphs [9, 12, 48, 135], but these tend to be language specific and can struggle with some of the more advanced features of modern programming languages [9]. An alternative approach would be to view each line of code as an EFSM transition. Thus, we could treat a program as an abstract trace and try to infer an EFSM from it.

While this seems like an attractive proposal, it falls well outside the scope of the work presented in this thesis for several of reasons, the main reason being that the primary focus of this thesis is black-box inference from traces. Like most inference methods in the literature, the work in this thesis assumes traces to be linear in nature. By contrast, programs tend to be *hierachical* in nature, containing subroutines such as loops and object methods. Consequently, great care would have to be taken to transform programs into linear objects, for example by by "unrolling" loops to a fixed depth and making object methods "inline".

Another reason is that lines of code work with symbolic rather than concrete values, taking their inputs at runtime. The techniques I develop in this thesis are intended to work with traces which contain concrete input and output values. Given that the GP preprocessing technique I present in Chapter 7 essentially aims to convert concrete traces into symbolic ones, it should be relatively easy to adapt my inference tool from Chapter 6 to work with symbolic traces, but this does not contribute towards the main objectives of the thesis so will not be explored further.

3.9 Process Mining

Process mining is a relatively new field which sits between data mining and computational modelling, the idea being to "discover, monitor and improve real processes (i.e., not assumed processes) by extracting knowledge from event logs" [144]. Unlike the often more theoretical field of model inference, process mining is much more industry facing, and there are many practical tools which use process mining techniques such as Discovery Analyst (StereoLOGIC), Flow (Fourspark), and Interstage Automated Process Discovery (Fujitsu).

The field of process mining has three main aspects: discovery, conformance, and enhancement [144]. Of these, it is the discovery aspect which is most related to the work in this thesis. Like with passive inference, discovery is about inferring a model of a process from an event log, i.e. traces. These models may take the form of Petri-nets, BPMN, UML activity diagrams, or a number of other formalisms, but they are all (somewhat) comparable to (E)FSM models in that they show the flow of events in the process and key branching points.

A central idea in process mining is that traces can be recorded from five different *perspectives* [144]: control flow, data, time, resources, and function, with the control flow perspective being most comparable to the field of model inference. While most of the literature on process mining seems to focus on this perspective, research carried out in [109] considers problems in which multiple interacting process perspectives are considered together. That work presents two multi-perspective discovery tools.

The first tool uses recorded data values to infer DC-Net models [109, Definition 8.8] of data dependencies between different process activities, essentially learning models of the control flow and data perspectives at the same time. These models have variables, guard functions, and write operations, but they are *causal* models rather than *computational* ones. They show which variables are related to particular events, but do not show *how* they change throughout the execution of the model. Thus, the models cannot be used to predict system behaviour for new traces or to prove data flow properties. Indeed, there does not seem to be any work in the literature on process mining which infers predictive models, but prediction does not seem to be a major objective of process mining.

The second tool in [109] uses domain knowledge to abstract low level activities to high level ones to discover a more human-readable model. The idea here is that events in traces generally correspond to meaningful activities, but not necessarily high level functionality. When inferring system models for human consumption, we want models to be comprehensible. Part of this is ensuring that the activities (transitions) in the model are recognisable. For example, in our drinks machine example, we record the high level *select*, *coin* and *vend* actions without the lower level memory management activities which inevitably come about as a result of reading from and writing to variables. Such low level activities are not relevant to most analysts and do not correspond to meaningful activities, so it is desirable to group them together into a higher level abstraction. This is what tool in [109] is able to do.

Most (E)FSM inference papers do not consider the possibility that the traces used for inference may be from a lower level perspective than is desired for the model, nor do they consider the possibility of subroutines or other implementational features. Instead, traces are treated as linear *strings,* and a model is learned accordingly. While it would certainly be very interesting to apply ideas from [109] to allow us to infer EFSM models at an arbitrary level of abstraction, or even *hierarchical* models (such as in [136]) which separate out individual subroutines, this is very much outside of the scope of this thesis.

3.10 System Identification

System identification is the process of building mathematical models of dynamical systems based on data observations [103] and is mainly concerned with automated control systems. The field is a large one, much of which is not directly relevant to the work of this thesis. One area of interest, though, is the inference of models of hybrid systems, which can be modelled by *hybrid automata* [82]. While these are not directly comparable to EFSMs, both models use *transitions* to move between *states* and have internal data *variables.* Where EFSMs have their variables explicitly updated by transitions, with hybrid automata, data values change over time in the states according to some differential equation. For example, consider a simple heating controller. The heater may transition between the *on* and *off* states depending on the temperature. In the respective states, the temperature will go up and down over time.

In [113], a technique is presented to infer such models from timestamped sequences of variable values referred to as *signals.* These signals are first broken down into segments using some signal segmentation method [124] which detects abrupt changes in the signal. The segments are then clustered to identify the internal states of the system, and each cluster is assigned a unique symbol. This then allows *trace strings* to be formed. For example, the signal $[1, 1, 1, 5, 5, 5, 3, 3, 3]$ may be segmented as $[[1, 1, 1], [5, 5, 5], [3, 3, 3]]$. Here, there are three clusters, which may be assigned the symbols a, b, and c respectively. This would then form the trace string ⟨a, b, c⟩.

Input and output signals are both processed in this way, and can be combined to form I/O traces, which are defined to be a pair of trace strings $s_I = a_1, a_2, \ldots, a_i$ and $s_O = b_1, b_2, \ldots, b_i$ (representing input and output respectively) such that, for all n, segment a_n occurs concurrently with b_n and for any b_i and b_{i+1}, if $a_i = a_{i+1}$ and $a_i \neq \Delta$, $ai + 1 = \Delta$, where Δ represents a significant change in output without a change in input.

The authors of [124] use a modified version of LearnLib to transform the I/O traces into a Mealy machine. Here, membership queries are answered by searching through the traces for the longest possible match, and returning the corresponding output. Equivalence queries are answered by searching for a counterexample in the traces. The authors note that this does not guarantee correctness, but also that they do not deem this to be necessary.

The next step is to infer initial values and flow conditions. Initial values define the values of variables when the model enters a particular state. Flow conditions describe how a variable changes while the system is in that state. To infer these functions, the authors propose heuristics based on statistical methods such as linear regression. After this, *timing conditions* are added to transitions to account for the Δ points in the traces. Here, there was a marked change in output without there being a change in input. There may be any number of reasons for this, for example unobserved internal system values, but the authors attempt to resolve such inconsistencies with time-based transition guards inferred from the timestamps in the traces. The system from [124] is evaluated with respect to two case studies: an engine timing model and a fuel control system, and produces very promising results.

3.11 Evaluating Models

A clear objective of model inference is to obtain a model which *accurately* represents the underlying system. To effectively evaluate the quality of the models produced by an inference technique, we need a quantitative way to measure how closely the behaviour of an inferred model matches that of the underlying system. Unfortunately, there is no single metric which can give a complete view of this. Instead there are many different metrics, each of which gives a slightly different picture. This is especially true for EFSM models since both the models and the traces thereof are more feature rich.

While there are many potential metrics to assess the accuracy of a model, most involve *traces.* When evaluating the accuracy of an EFSM model in terms of a trace, we can divide the trace up into three parts, as in Figure 3.12. The first part of the trace (shown in green in Figure 3.12) is the *accepted prefix.* This is the part of the trace where the model matches the system perfectly. The portion of the trace shown in yellow in Figure 3.12 represents the part of the trace after there has been an observable difference between the system and the model (referred to as the *point of deviation*) but before the model has stopped recognising events altogether. The yellow and green sections together are referred to as the *recognised prefix.* It is worth noting that there may still be many correctly executed events after the point of deviation — that is, the output from the model perfectly matches that of the system — but, after the point of deviation, we know that the model is incorrect since we have observed a difference in behaviour between the model and the system. The red part of the trace represents the *unrecognised suffix.* This is the part of the trace that comes after the point at which the model has no outgoing transitions for the current event, referred to as the *point of non-recognition.* After this point, the model is unable to process the remainder of the trace.

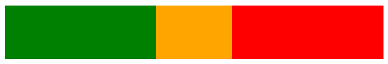


Figure 3.12: The three parts of a trace.

A key difference between the evaluation of EFSM models and their classical counterparts is that, for classical models, the point of deviation is the point of non-recognition. That is, the part of the trace shown in yellow in Figure 3.12 is non-existent. Since classical FSMs only have atomic actions, they either recognise an action or they do not. The only observable difference between a model and a system is that the model recognises an action where the system does not, or vice versa. By contrast, EFSM traces effectively have two parts: the *control* part and the *data* part. The control part is the sequence of actions which are called. The data part is the inputs with which each action is called and the outputs which are produced in response.

A popular technique in classical FSM inference [40, 98, 151] is behavioural classification. That is, the model's ability to classify traces as either valid or invalid. When given a system, a model, and a trace, there are four possible outcomes, as illustrated in Table 3.1.

TP The trace is a valid trace of the system and is accepted by the model.

FN The trace is a valid trace of the system and is not accepted by the model.

FP The trace is not a valid trace of the system and is accepted by the model.

TN The trace is not a valid trace of the system and is not accepted by the model.

		Model	
		Accept	Reject
System	Accept	TP	FN
	Reject	FP	TN

Table 3.1: Confusion matrix for trace classification.

The metrics of behavioural classification are built on top of these four situations and are defined as follows.

$$\text{sensitivity} = \frac{TP}{TP + FN}$$

$$\text{specificity} = \frac{TN}{TN + FP}$$

$$\text{precision} = \frac{TP}{TP + FP}$$

The metric of *sensitivity* (sometimes referred to as *recall*) measures a model's ability to accept the traces that it should, and is the proportion of positive system traces which were recognised by the model. *Specificity* is the dual of sensitivity, and measures a model's ability to correctly reject negative system traces. In other machine learning fields, it is the metric of *precision* that is normally paired with recall. This measures the proportion of the traces accepted by the model which were actually positive traces of the system. In the field of model inference, *specificity* is the more common pairing since a model's ability to correctly reject negative traces is more meaningful. The two metrics of sensitivity and specificity can be combined into a single metric, the *binary classification rate* (BCR) as follows.

$$\text{BCR} = \frac{\text{sensitivity} + \text{specificity}}{2}$$

While BCR is a popular accuracy metric for classical models, and has even been used to evaluate EFSM models [150], it has several drawbacks. Firstly, this method relies on the existence of negative traces. In [98] and [151], models are inferred from traces of existing reference models. Negative traces can be obtained relatively easily by constructing the complements of these. EFSM inference tools are generally evaluated from real programs rather than reference models, so it is much harder to obtain negative traces. By definition, the programs themselves can only produce positive traces. This means that negative traces often must be created by hand. A semi-automated approach is suggested in [152], however, this still involves a considerable amount of manual effort and contextual knowledge.

Because negative traces are so hard to obtain, many inference methods work only from positive traces. If the inference process is not given examples of negative behaviour, it seems a little unjust to use them as part of the evaluation. Rather than testing how the model behaves when faced with unseen traces, using negative traces to evaluate models inferred only from positive ones is asking the model to predict an entire class of unseen behaviour.

BCR is a good metric to use if the primary objective is to infer a model which can *monitor* behaviour, but it is not so good if we want to infer models which *predict* behaviour. Here, we are looking to measure how closely the outputs produced by the system correspond to those produced by the model. Consequently, the authors of [150] propose an alternative metric: the normalised root mean square error (NRMSE) between the different values. This is built on top of the RMSE metric, which measures the mean square error between the outputs produced by a model in response to a particular trace and the outputs produced by the real system in response to the same trace. In the context of [150], the updated values of the internal system variables were taken as the “outputs”.

For two sequences of values $X = x_1 \ldots x_n$ and $Y = y_1 \ldots y_n$, the RMSE is calculated as follows.

$$\text{RMSE} = \sqrt{\frac{\Sigma_{t=0}^{n}(x_t - y_t)^2}{n}}$$

Since the RMSE is scale dependent — its value depends on the literal values in the two sequences, it is *normalised* to produce a value between zero and one by dividing between the range of values within the traces. That is, the difference between the maximum and minimum observed values. This defines the NRMSE as the following.

$$\text{NRMSE} = \frac{\text{RMSE}}{\max X - \min X}$$

Note that we normalise solely with respect to the sequence X. Here, X represents the *reference sequence*, in this context the trace which comes from the system. Y represents the *subject sequence*, or the trace which has come from the model.

The main limitation of NRMSE is identified in [150]: we can only compare traces of the same length. This means that we are only able to work with the recognised prefix of each trace. There is no meaningful way to include the unrecognised suffixes of traces in this metric because the model cannot produce any outputs with which to compare those of the system. This means that a model which recognises nothing will score just as well as a perfect model. In this way, it is similar to simple trace refinement from Section 2.3. The accepted prefix contributes nothing to the model either, since there is no difference between the outputs produced by the model and those produced by the program. This means that the metric can make models with shorter recognised prefixes look worse than models with longer ones if the point of non-recognition comes closer to the point of deviation as there is less potential for contributing error.

The second problem with NRMSE as a metric of accuracy is more conceptual. Is it meaningful to compare the outputs of two different functions when we do not know what the correct function is? In contrast to other areas of machine learning, we are more interested in the model than we are its output. Here, we are arguably more interested that a particular output *is* wrong than we are in *how* wrong it is.

Example 3.11.1. Let us say that for an input of 1, the output of a transition in our inferred model for a particular input is 1000. The output from the system in response to the same event is 10. This is clearly a big difference and leads to a large NRMSE. Now let a different model produce the corresponding output of 101. This is much closer to the output of the system so leads to a smaller NRMSE meaning that this second model scores better.

In this context, though, the primary objective is not to evaluate the *output* of the model. What we really want to do is evaluate the model itself. We are just using output as a means of doing this. If we say that the real function used by the system to calculate this output in the above instance is $i_0 \times 10$ and that the functions used by the two models are $i_0 \times 1000$ and $i_0 + r_1 - r_2 + 4$ respectively, we are put in a difficult position. Here, it is just happenstance that the second model has produced an output closer to that of the real system. In fact, the first model has a function which is syntactically closer to the real system.

Unfortunately, we do not know the actual functions used by black-box systems, and have no meaningful way to compute the syntactic difference between functions even if we did. For a large enough test set, it is likely that poor functions will perform poorly, but the point remains that we are more interested in the model than its outputs here. We do not particularly wish to favour solutions which are off by a small amount for a large number of events over solutions which are off by a larger amount for a smaller number of events. Arguably, in this context, we prefer the latter situation because the proportion of correctly executed actions is higher.

Another popular metric to assess the quality of classical FSM models is the *complexity* of the model in terms of the number of states and transitions. This is quite sensible for classical FSMs since every model has a canonical *minimum.* This means that smaller models are necessarily better than bigger ones. This is not so for EFSMs since we can arbitrarily move information between the control flow and data states. Indeed, for every EFSM there exists a trace equivalent model with only one control flow state and many reflexive transitions. Such models do not intuitively represent the control flow of an application, so may not always be more useful than a multi-state model.

3.12 Industrial Applications

This section briefly discusses how (E)FSM inference and trace analysis are used in industry. We have already seen how tools such as `strace` allow us to trace the kernel calls made during program execution. The software giant SAP provides tools for trace analysis [1] with all of its systems, and third party tools such as Percepio's Tracealyzer [2] can construct graphs showing the communications between different system objects. While these tools are widely used, they work very much at the level of traces.

Despite the utility of system models and the apparent benefits of automated inference, there appears to be relatively little of this going on in practice. Attempts have recently been made to infer FSM models of the Red Hat kernel [47] but this is, as of yet, a very manual process. Instances like this beg the question "Why is automated inference not more prevalent?". There are many reasons for this, possibly the most significant being that most passive inference techniques are simply not up to the job. Even the models inferred by MINT [152] (the current state of the art of EFSM inference) are not sufficiently accurate to be commercially applicable. Another factor, which I will revisit in Section 3.13 is that traces of systems are actually quite hard to come by. Most end users do not have comprehensive logging enabled by default on their software as it reduces performance and is simply not necessary for most applications. If we have to exercise a program for the express purpose of obtaining traces for inference, we may as well put the same effort into manually inferring a model, as is likely to be more accurate than anything inferred by current tools.

While there is not enough data to make the use of *passive* inference tools feasible, this does not explain why *active* techniques like [10, 28] are also neglected by industrial practitioners. One reason for this could be that such techniques require some sort of oracle. This is often the system itself, with membership queries being answered by running test cases. As mentioned in [70], this does not scale particularly well to the sort of large scale industrial applications for which a model might be required. While grey-box approaches such as [70] are a possible solution to this, most of the techniques are not sufficiently mature to have been picked up by industry.

As discussed in Section 3.9, the field of process mining is much more industrially applicable than model inference, with many industrial tools — such as Discovery Analyst (StereoLOGIC), Flow (Fourspark), and Interstage Automated Process Discovery (Fujitsu) — using process mining techniques. Here, models are used to analyse and improve business processes, as well as for conformance testing however, like classical FSM inference, the main focus is on the control flow of systems. While techniques such as [109] have been developed to include multiple perspectives, the models inferred here show *which* variables are updated by particular actions without showing *how*. Thus, we cannot use them to prove data flow properties of systems, to predict system behaviour for unseen executions.

In summary, there are two main barriers which must be overcome before automated inference tools can be used widely in industrial situations. The first is the *accuracy* of automatically inferred models. Most automated tools are simply not able to infer models which are sufficiently accurate to make it worth running them. The work presented in this thesis aims to push the boundaries of this to enable us to infer more accurate and comprehensive models. Secondly, there is insufficient *availability of traces* to be able to run most inference techniques. People often do not have execution traces to hand from which to infer models. This is more an implementation decision on the part of developers and lies outside the scope of this thesis.

3.13 Limitations and Gaps

In this section, I discuss the limitations of current techniques in more detail and identify gaps in the literature to motivate the novel work I present in subsequent chapters.

The models inferred by both [106] and [152] were originally presented as *Finite State Automata with Parameters* (FSAPs) in [105]. These are defined as in Definition 20. These models can be thought of as guarded FSMs with simple memory. Input events are simple atomic labels and transitions may place restrictions on data variables in the form of guards (called *enabling functions* in [105]) such that they can only be taken under particular circumstances.

Definition 20. An FSAP is an eight tuple $(Q, \Sigma, D, F, \delta, \varphi, q_0, Q_E)$ where

Q is a finite non-empty set of states.

$q_0 \in Q$ is the initial state.

$Q_E \subseteq Q \setminus \{\emptyset\}$ is the set of final states.

Σ is a finite non-empty set of input symbols.

D is an n-dimensional space $D_1 \times \cdots \times D_N \cup \{\emptyset\}$.

F is a set of *enabling functions* of the form $f_i : D \to \{0, 1\}$.

δ is the *transition function* $\delta : Q \times \Sigma \times D \to \mathcal{P}(Q)$.

φ is the *selecting function* $\varphi : Q \times \Sigma \to f$ with

$\forall q \in Q, Q' \subseteq Q \setminus \{\emptyset\}, m \in \Sigma, d \in D.\, \delta(q, m, d) = Q' \implies \varphi(q, m)(d) = 1.$

$\forall q \in Q, m \in \Sigma, d \in D\, \varphi(q, m)(d) = 1 \implies \exists Q' \subseteq Q \setminus \{\emptyset\}.\, \delta(q, m, d) = Q'.$

These models are good for showing *what* data values are admissible, but transitions cannot update variable values, so the models cannot show *how* values change during model execution — they are not *computational.* This means that their applicability is limited to monitoring systems. Given a sequence of method invocations with associated data values, a model can be used to determine whether the system is capable of producing that sequence. They cannot, however, be used to answer predictive questions such as “What is the value of variable v after executing transition t?”. To do this, the inference of update functions — the actual functions which compute individual data values from inputs — is essential.

The technique presented in [150] is a means of adding transition updates to existing models, but requires detailed knowledge about all variable values throughout the execution of the system, i.e. white-box traces. If we do not have access to the source code of the system, it can be very difficult to obtain such traces. Black-box traces only contain input and output values, which means that [150] is only applicable if there are no internal variables, i.e. if the output of a transition depends only on its input. While existing GP techniques like the one employed by [150] are very effective at inferring functions where all the inputs are known, there is no literature currently available on inferring functions where the output is dependent on hidden variables.

An alternative to [150] is [28]. Rather than postprocessing existing models, this technique can be used to infer RAs, complete with data updates, from black-box traces. The problem here is that this technique requires the ability to run the system under inference. For live systems, it may not be possible to run arbitrary method calls to observe the outcome.

From these limitations, three main contributions can be identified:

1. It is clear that there is a need for a passive inference technique, like in [152] which infers fully computational EFSM models from black-box traces that only contain inputs and outputs. This is presented in Chapter 6.
2. Most passive inference techniques are based on *state merging.* To apply this to EFSMs, we will need some way to compare and merge transitions with data updates. While [106] presents the idea of *subsumption* as a means of comparing transitions with guards, no consideration is given to transitions which update data variables. Chapter 5 extends the idea of subsumption presented in [106] to transitions with data updates.
3. Our inference technique also needs a means of inferring the existence and use of variables which are internal to the system but do not occur in the traces. This is presented in Chapter 7.

In addition to these contributions, we also need a good method of objectively evaluating the models which come out of any inference we perform. Essentially, we need to answer the question "Is the model we have inferred any good?". It turns out that this is a surprisingly difficult question to answer, and there are many metrics which can be used. Even in classical FSM inference, there is no single standard metric which is used throughout the literature, and the problem is even worse for EFSM inference.

Many classical FSM inference papers, including [98, 151] evaluate models in terms of their ability to *classify* legal and illegal behaviour. That is, do they accept valid traces of the system and reject invalid ones? Another popular metric is the size of the model in terms of the number of states and transitions, with smaller models generally being seen as better. This is often the case for classical FSM models, but does not extend to EFSMs since we can arbitrarily move information between the control state and the data state, meaning that we can convert every EFSM into an equivalent single-state model.

To test inference tools, we obviously need traces from lots of different subject systems. For classical FSM inference, it is quite easy to generate pseudo-random FSM models. We can then generate random walks of these models to obtain traces. This is the approach taken in [98, 151] but there is no work in the literature about whether tools which perform well on synthesised systems perform similarly well on real-world systems. This is desirable future work.

For EFSM inference, much of the work in the literature falls a little short when it comes to evaluating inference techniques. A likely explanation for this is that evaluating the quality of EFSM models is much more difficult than for their classical counterparts. While, in other fields, it is standard practice to compare new techniques against a common baseline approach, as well as the current state of the art, the multitude of different model definitions means that models inferred by different tools often cannot be meaningfully compared. As a consequence, tools are often considered in isolation.

Another contributing factor is that it is much harder to obtain the traces needed to infer EFSM models. This is because, as well as sequences of action labels, we also need the inputs and outputs associated with each action. Unlike classical FSM models, which can be randomly generated reasonably easily, to generate an EFSM model we effectively have to generate an entire computer program. To do this in a meaningful way is extremely difficult and there does not appear to be any work in the literature which does this.

It is even difficult to automate the task of obtaining traces of existing EFSM models as we cannot simply pick a random action from the current state like we can for classical FSMs. Indeed, we cannot even generate the control part of the trace this way since the values of registers affect whether or not a given path is feasible [91, 147]. Not only that, but we also require data inputs for each action. The task of obtaining a suitable set of traces which fully covers the behaviour of an EFSM model is an open problem and effectively reduces to *test-case generation,* which remains an active research area [7, 92, 102].

Because of the difficulty of generating random EFSM models, techniques in the literature are often evaluated with respect to a small number of real-world or handcrafted case studies which have been modified to produce suitable traces. While the use of real-world systems is obviously preferable to randomly generated examples, the fact that only two or three systems can feasibly be used is a real threat to the validity of most results. Further, there is no standard library of traces or systems, meaning that most tools are evaluated using different systems.

In summary, there is clearly a need for some standardisation in the metrics used when evaluating EFSM models. It is also desirable to compare new inference techniques to a baseline approach and to the current state of the art, so we know whether the field is improving. Ideally, it would also be nice to have access to a bank of subject systems and traces from these systems with which to evaluate inference techniques so that it is not necessary to first find (or write) programs from which to obtain traces. In Chapter 8, I present a comprehensive evaluation of my inference tool. I compare the inferred models to both the original PTA and to the models produced by MINT [152] using the postprocessing technique presented in [150].

Concluding Remarks

This chapter has introduced the concept of model inference from traces and has laid the foundations for my own EFSM inference technique, which will be presented in Chapter 6. I have reviewed the state of the art of EFSM inference, both active and passive, and have given details of how the quality of the models we infer can be evaluated. Finally, I discussed the limitations of current techniques and identified the main gaps in the literature which the remainder of this thesis aims to fill.

Chapter 4

Extended Finite State Machines

Classical FSM models are well-established and widely used, but there is no single generally accepted EFSM definition. Chapter 2 mentions various definitions from the literature [101, 105] as well as similar ideas under different names [20, 56]. As discussed in Section 3.13, none of them are particularly well suited to our purposes. This chapter reviews the limitations of current models and formulates a new EFSM definition, which was part of the contribution of my work published in [64], that meets these requirements.

4.1 Introduction

The EFSM models discussed in Chapter 2 extend classical FSM models in various different ways, with the three main additional features being the following.

1. Parametrised *inputs* upon which we can place conditions.
2. The ability to produce *outputs* in response to inputs.
3. State variables which may be *updated* by evaluating functions in terms of inputs and anterior variable values.

Most FSM models which claim to be "extended" have at least one of these features, but no existing model in the literature has all three. Figure 4.1 summarises the features of the various models discussed in Chapter 2.

Mealy machines [112] are the first real attempt to extend classical FSM models. As well as responding to actions, transitions also produce an observable output. These models are not sufficient for our needs, though, as they have neither input parameters nor internal variables.

Example 4.1.1. Consider the Mealy machine transition $q_m \xrightarrow{coin50/100} q_n$ which represents the event $coin(50)/[100]$. While the output is a distinct part of the event, the input is not. Instead, inputs must be encoded within the transition label so are effectively lost. This means that we cannot *compute* output from input. There are also no internal variables here, meaning that we cannot explicitly store information about the current state for later reuse. Like with classical models, we must encode data within the control flow of the model.

The PFSMs of [101] have both input parameters and observable output values. There is no data state, though, so we are still forced to encode all information about the current state within the control flow of the system, just like with Mealy machines and classical FSM models.

Example 4.1.2. Consider the PFSM transition $q_m \xrightarrow{coin(i)[i=50]/100} q_n$ which represents the event $coin(50)/[100]$ from our simple vending machine. While the *observable* behaviour of the event is represented perfectly, the transition does not capture the full picture. In the real system, when a user inserts a coin, an internal variable is updated to keep a running total of how much money has been inserted so far. Without this internal variable, the running total must be encoded within the control state of the model, just like with classical FSMs.

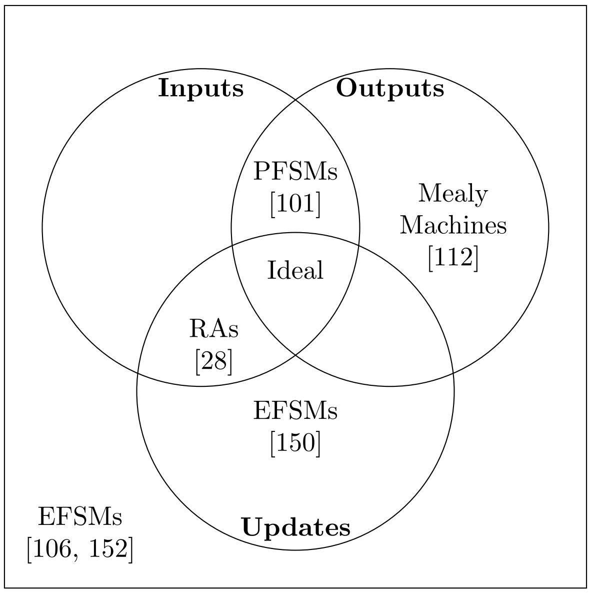


Figure 4.1: The various features of different EFSM models in the literature.

The current state of the art of passive EFSM inference is MINT [152], which produces models as in Definition 20. As can be seen from Figure 4.1, these models lack all three of our desirable features. They are classed as *extended* because they make use of global data variables, but these cannot be updated by transitions. Thus, they appear to "magically" change during execution rather than being explicitly computed at each stage. This means that we cannot predict how systems might behave for previously unseen input sequences. Additionally, these models have no real notion of input and output. While it is possible to use variables to model these, it can be helpful to explicitly distinguish between hidden state variables and observable behaviour.

Example 4.1.3. Consider the transitions $q_m \xrightarrow{coin[i_0=50, r_1=0]} q_n \xrightarrow{coin[i_0=50, r_1=50]} q_p$ which correspond to the event sequence $\langle coin(50)/[50], coin(50)/[100]\rangle$. These are the kind of transitions which appear in the models inferred by [152]. Here, we have one *coin* transition with a guard $r_1 = 0$ and a second transition with the guard $r_1 = 50$. We can see here that between the first and second transitions, r_1 must be set to 50 but, because the transitions lack update functions, we do not know *how* internal variable values are updated during execution.

The work of [150] attempts to extend the models inferred by MINT by augmenting the transitions with data update functions, providing one of our desirable features, but the models still lack explicit inputs and outputs. It is possible to use global variables to model inputs and outputs, but this is neither intuitive nor particularly accurate as it fails to capture the notion that inputs and outputs can be observed, while internal data values cannot.

The state of the art of *active* EFSM inference [28] produces models which are closer to our needs. The register automata used here (defined in Definition 9) separate instantaneous transition input parameters from persistent global variables, which transitions may update. Thus, these models exhibit two of our desired features, but lack explicitly observable outputs.

Example 4.1.4. Consider the transition $q_m \xrightarrow{coin(i_0)|i_0=50\, r_1:=50} q_n$ which is a register automaton transition for the event *coin*(50)/[50]. Here, the transition takes a single input i_0 (which is guarded to equal 50) and assigns that value to register r_1. Here, we have update functions and parametrised inputs, but we do not have outputs. While we can model outputs by assigning further internal variables, this adds an extra layer of complication and doesn't really capture the idea of *observable* behaviour.

What we would really like is the ability to capture the externally observable behaviour of a system, and its internal data transformations, as separate aspects of the same model. To do this, we need input arguments which are passed as parameters to the transition. These must be distinct from the data registers and should not persist after the firing of the transition. We also need to separate observable outputs (which correspond to method return values), from internal variables. This allows us to distinguish behaviour that is visible to an external observer from that which is internal to the system.

The remainder of this chapter is structured as follows. Section 4.2 lays out the formalities of my proposed EFSM definition. Next, Section 4.3 discusses what events and traces look like in this setting and how they are handled. Section 4.4 defines what it means for an EFSM to be nondeterministic, and why nondeterminism is problematic when analysing models. Section 4.5 gives definitions for the terms *acceptance* and *recognition*, and discusses the differences between the two. In Section 4.6, I provide definitions of behavioural equivalence and simulation between models. Finally, in Section 4.7, I present my formalisation of EFSMs in Isabelle/HOL, and provide the proofs of various key properties.

4.2 Formal Definition

This section presents my EFSM definition, which formed part of the contribution of [64] and will be used throughout the rest of this work. One of the main objectives of this thesis is to infer functions to relate *inputs* and *outputs*. Consequently, the model definition explicitly separates the two. As discussed in Subsection 2.2.5, we also need to store explicitly information about the current state for later use as we do not want to encode this within the control state of the model. Consequently, I make use of a set of data *registers* which persist throughout the execution of a model and can be updated by transitions. The most similar definition to mine in the literature is RAs [28], except that my models produce observable outputs and are able to perform arithmetic as part of data update functions.

It is helpful to discuss the definition in terms of an example. Recall the EFSM in Figure 1.5 (reproduced below for convenience), which represents the simple drinks machine from Section 1.1. There are four transitions here. The *select* transition allows customers to choose their drink by passing its name in as an input parameter. This is stored in register r_1, and a second register, r_2, is initialised to zero. The *coin* transition also takes one input, which represents the value of the inserted coin. The running total is displayed to the customer as an output, and is maintained by updating the value of r_2. When this total reaches a value greater than or equal to 100, the $q_1 \xrightarrow{vend} q_2$ transition can be taken to dispense the selected drink to the user. If the user tries to trigger *vend* before this point, the $q_1 \xrightarrow{vend} q_1$ transition is taken, which produces no observable output and leaves all registers unchanged. Having established our running example, let us now move on to the formal definition. This is shown in Definition 21.

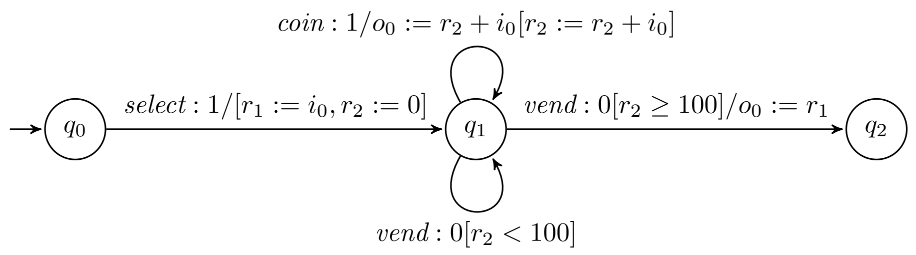

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_0
$select : 1/[r_1 := i_0, r_2 := 0]$
q_1
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_2
$vend : 0[r_2 < 100]$

Figure 1.5: An EFSM model of the drinks machine.

Like all (E)FSMs, my models are made up of *states* and *transitions.* Unlike other definitions [112], there is no restriction that the model outputs (or indeed the inputs) must come from a finite enumeration. The main reason for this comes from the intended application. We want to use traces to infer EFSM models that are able to *predict* the behaviour of systems when faced with previously unseen action sequences. As the traces used to infer a model are almost always a subset of the full language accepted by the system, if we were forced to define an input alphabet for the inferred model from these traces, we would almost inevitably end up with an overly restricted alphabet that limits the predictive power of the model. For example, the traces in Figure 1.1 only show two of the possible inputs to the *select* transition. If we were forced to define an input alphabet for the inferred model from these traces, the *select* transition would only be able to accept inputs "tea" and "coffee". We would then not be able to predict what would happen if we were to try selecting "soup".

Definition 21. 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.
T is a finite transition matrix $T : (S \times S) \to \mathcal{P}(L \times \mathbb{N} \times G \times F \times U)$ 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 $G : (I \times R) \to \mathbb{B}$.
F is a set of *output functions* $F : (I \times R) \to O$.
U is a set of *update functions* $U : (I \times R) \to R$.

In G, F, and U

I is a list $[i_0, i_1, \ldots, i_{m-1}]$ of values representing the inputs of a transition, which is empty if the arity is zero.
R is a mapping from variables $[r_0, r_1, \ldots]$ to their values.
O is a list $[o_0, o_1, \ldots, o_{n-1}]$ of values, which may be empty, representing the outputs of a transition.

As in Figure 1.5, a little syntactic sugar allows an EFSM transition from anterior state q_m to posterior state q_n to take the following general form.

$$q_m \xrightarrow{label:arity[g_1,\dots,g_g]/f_1,\dots,f_f[u_1,\dots,u_u]} q_n$$

Further to this, in the interest of brevity, transitions will be referenced in this text simply as $q_m \xrightarrow{label} q_n$ wherever this does not cause ambiguity.

The first part of each transition is an atomic *label* which is the name of the event. In Figure 1.5, the labels are *select*, *coin*, and *vend.* My definition also ascribes an *arity* to each transition. This is the number of inputs the transition expects to receive. A transition can only be taken if it is presented with the correct number of inputs. This is common practice in most programming languages, where method definitions declare the variables they expect to receive. Indeed, the input parameters of RAs are exactly this. Here, inputs are indexed by natural numbers instead of being explicitly named, but the principle is the same. In Figure 1.5, the *select* and *coin* transitions each take one input and the *vend* transitions do not take any.

Guard expressions $[g_1, \dots, g_g]$ place conditions on transition inputs and the anterior data state. While Definition 21 only allows transitions to have one guard, it is often more aesthetically pleasing to express guards as a list which is implicitly conjoined. A transition can only be taken if its guards are satisfied. If, for a given (*label*, *input*) pair, there is a transition from the current state where the guards are satisfied, the EFSM is said to have *recognised* the input. In Figure 1.5, guard expressions can be seen on both *vend* transitions. The $q_1 \xrightarrow{vend} q_1$ transition has the guard $r_2 < 100$ and the $q_1 \xrightarrow{vend} q_2$ transition has the guard $r_2 \geq 100$.

After the guards comes a slash, after which expressions $f_1, \dots, f_f$ define the outputs in terms of the inputs and current data state. In Figure 1.5, the *coin* transition has a single output: r_2+i_0. The $q_1 \xrightarrow{vend} q_2$ transition also has a single output: r_1.

Finally, update expressions $u_1, \dots, u_u$, enclosed in square brackets, define the posterior data state. There should be at most one update function per register per transition in order to maintain consistency. Both outputs and updates are computed by evaluating expressions over literals and variables (inputs and registers) from the *anterior data state.* Assignment syntax $\_ := \_$ is used to identify the value being computed. Registers not explicitly updated by a transition remain unchanged. For transitions without outputs or updates, the corresponding components are omitted. In Figure 1.5, the *select* transition has two updates. Firstly, the register r_1 is assigned the value of i_0 to record what drink the user selected. Secondly, the register r_2 is initialised to zero. The *coin* transition updates r_2 to $r_2 + i_0$ to record the total amount of money inserted so far.

Data states are written as comma-separated lists of assignments enclosed in angle brackets. For example, the register state where r_1 holds value “tea” and r_2 holds value 0 is written as $\langle r_1 := \text{“tea”}, r_2 := 0\rangle$. Registers are globally accessible throughout the execution of the model, but inputs are instantaneous and can only be accessed by their respective transition. Each register is initially undefined, and cannot be accessed before a value has been assigned to it. A consequence of this is that it is not possible to check whether a particular register is defined. This is very important for the soundness of my transition merging criterion discussed in Chapter 5. While the number of registers is technically unbounded, since the number of transitions and the number of register updates on any transition is finite, it follows that the number of registers used by any given EFSM will also be finite.

Example 4.2.1. Consider again the lift door controller from Example 2.2.5. Its EFSM representation is shown in Figure 4.2. This looks quite similar to Example 2.2.5, the only significant difference being a numerical input arity instead of named parameters.

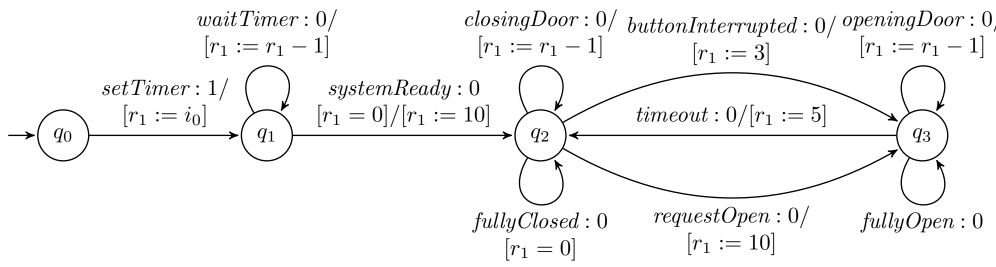


Figure 4.2: An EFSM representation of the lift doors controller.

Example 4.2.2. Consider the EFSM in Figure 4.3 which represents a simple game of Space Invaders. There are three states here. The user first enters the game by pressing *start*. This initialises the variables which store the state of the game. Register r_1 stores the x coordinate of the gun turret and is initialised to 200px, which represents the middle of the screen. Register r_2 stores how many aliens are left on the board. This is taken from the input to *start*. Finally, r_3 represents how many lives the player has left, and is initialised to three.

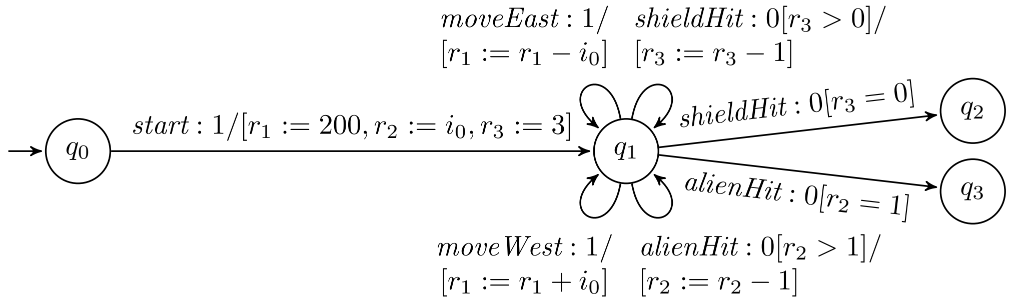


Figure 4.3: An EFSM representation of a game of Space Invaders.

From q_1, there are four reflexive transitions. The two movement transitions, *moveWest* and *moveEast*, both take one input parameter which is the step size. This is added or subtracted from the current x value depending on the direction. The *shieldHit* action represents the user being hit by an alien and subtracts one from the value of r_3. This represents the player losing a life. There is also a guard here that $r_3 > 0$. If the player has no lives left and is hit again, the $q_1 \xrightarrow{shieldHit} q_2$ transition is taken instead. This represents the player losing the game.

Similarly, the *alienHit* transition subtracts one from r_2, representing an alien being "killed". Here, the guard requires r_2 to be greater than one, meaning that there is at least one alien left to kill. When the final alien is hit, the $q_1 \xrightarrow{alienHit} q_3$ transition is taken, which represents the player winning the game. We will revisit this case study in Chapter 8.

4.3 Events and Traces

Section 2.1 discussed events and traces generally. This section sets out how they work in the context of Definition 21. Traces of classical FSM models are simply sequences of atomic actions $\langle$a, b, c, ...$\rangle$. EFSMs have actions which take additional input parameters that can be guarded. Consequently, our definition of actions must reflect this.

Definition 22. An *action* is a pair consisting of the event name and a list of input parameters. An *execution* is a sequence of actions.

Actions represent the commands input to the system by the user. They correspond to methods and functions defined in conventional programming languages. Following the syntax of languages like C and Java, we express action pairs as $label(i_0, \ldots, i_n)$ rather than $(label, [i_0, \ldots, i_n])$.

When we *observe* an execution of the model, we see the outputs which are produced in response to the inputs. If we zip the two sequences together, we get a *trace* of the system. Input-output traces of this form have already been used in this work, for example in Figure 1.1. While transitions may also update the data state, in our model data registers represent the inner workings of a system. As such, their values are not visible to an outside observer. Only the label, inputs, and outputs are visible in traces.

Definition 23. An *event* is a (*label*, *input*, *output*) triple made up of the event name, the input parameters, and the outputs produced by the model. A *trace* is a sequence of *events*. While traces are usually assumed to be finite, there is no formal restriction on this.

Example 4.3.1. Some executions of the simple drinks machine are as follows.

$$\langle \textit{select}(\text{“tea”}), \textit{coin}(50), \textit{coin}(50), \textit{vend}() \rangle$$
$$\langle \textit{select}(\text{“tea”}), \textit{coin}(100), \textit{vend}() \rangle$$
$$\langle \textit{select}(\text{“coffee”}), \textit{coin}(50), \textit{coin}(50), \textit{vend}() \rangle$$
$$\langle \textit{select}(\text{“soup”}), \textit{vend}(), \textit{coin}(100), \textit{vend}() \rangle$$

When we observe these traces, we get the following sequences of outputs.

$$\langle [], [50], [100], [tea] \rangle$$
$$\langle [], [100], [tea] \rangle$$
$$\langle [], [50], [100], [coffee] \rangle$$
$$\langle [], \textit{vend}(), [100], [soup] \rangle$$

Combining the executions and observations, we obtain the following traces, also shown in Figure 1.1. For aesthetic reasons, we write event triples as $label(i_0, \ldots, i_n)/[o_0, \ldots, o_n]$ rather than $(\textit{label},[i_0, \ldots, i_n], [o_0, \ldots, o_n])$.

$$\langle \textit{select}(\text{“tea”}), \textit{coin}(50)/[50], \textit{coin}(50)/[100], \textit{vend}()/[\text{“tea”}] \rangle$$
$$\langle \textit{select}(\text{“tea”}), \textit{coin}(100)/[100], \textit{vend}()/[\text{“tea”}] \rangle$$
$$\langle \textit{select}(\text{“coffee”}), \textit{coin}(50)/[50], \textit{coin}(50)/[100], \textit{vend}()/[\text{“coffee”}] \rangle$$
$$\langle \textit{select}(\text{“soup”}), \textit{vend}(), \textit{coin}(100)/[100], \textit{vend}()/[\text{“soup”}] \rangle$$

4.3.1 Executing Traces

Now that I have defined events and traces in the context of Definition 21, it is time to discuss how executions are processed to produce observations. Classical FSM models simply react to each action in sequence and follow the corresponding transition to the next control state. EFSMs are a little more complex since transitions can also produce outputs and update the data state.

Like with classical FSM models, an EFSM begins in its initial state. Unlike classical models, EFSMs have both a control and a data state. This data state also begins in its initial state. That is, all registers are undefined. As discussed in Section 4.2, this is not a fixed value which can be tested for, but true undefinedness which cannot be evaluated or compared.

Upon receipt of an action, the EFSM constructs the set of transitions which can be taken. This set is made up of those transitions which leave the current state, have a label and arity which matches the current action, and have guards which are satisfied by the inputs and current data state. From these transitions, one is chosen at random to be the transition which "fires" and the output and update functions are evaluated. These are evaluated in the *anterior context.* That is, register values in expressions evaluate to those held before updates were applied. If there are no viable transitions, the system deadlocks and execution terminates. If there is more than one possible step for a given input, the model is *nondeterministic.* The implications of this are discussed in more detail in Section 4.4.

Example 4.3.2. Consider the execution $\langle \mathit{select}(\text{"tea"}),\ \mathit{coin}(100)/[100],\ \mathit{vend}()/[\text{"tea"}]\rangle$, which is executed in the EFSM shown in Figure 1.5 as follows.

1. The EFSM is in its initial state q_0 and all registers are undefined.
2. The *select*("tea") action comes in and the set of viable transitions is evaluated. Here, this is $\{q_0 \xrightarrow{\mathit{select}:1/[r_1:=i_0, r_2:=0]} q_1\}$.
3. Since there is only one viable transition, this is taken and the model moves into state q_1. There are no outputs here, but there are two updates. r_1 is assigned the value of i_0, which here is "tea", and r_2 is assigned the value 0.
4. The next action is *coin*(50). The only viable transition is $q_1 \xrightarrow{\mathit{coin}:1/o_0:=r_2+i_0[r_2:=r_2+i_0]} q_1$. This is taken, and the control flow state does not change. Here, there is an output function which is evaluated with $r_2 = 0$ and $i_0 = 100$ such that $o_0 = 100$. Finally, r_2 is updated to 100. The value of r_1 remains unchanged since it is not updated.
5. The final action is *vend*(). From q_1, there are two *vend* transitions with arity 0, but only one for which $r_2 = 100$ satisfies the guard. That is $q_1 \xrightarrow{\mathit{vend}:0[r_2 \geq 100]/o_0:=r_1} q_2$. This transition is taken, moving the model into state q_2. The output function is evaluated with $r_1 = \text{"tea"}$ so $o_0 = \text{"tea"}$. Here, there are no update functions so the registers remain unchanged.

Example 4.3.3. Consider the transition $f\colon 0/[r_1 := 5, r_2 := r_1 + 7]$. As stated above, updates are evaluated with the *anterior* register values. If r_1 holds the value 3 before this transition fires, the posterior register values will be 5 and 10 respectively. Register r_2 will not hold the value 12, as would be the case if updates were evaluated sequentially. This means that the order in which update functions are executed does not affect the semantics of transitions.

It is important to note here that the notion of *output* is not the same as that of process algebras like CSP [85]. In CSP, outputs must *synchronise* with the environment. That is, the environment must be willing to receive whatever value the model produces. If not, the output is *blocked* and the system deadlocks, unable to continue further.

Here, outputs are not required to synchronise with the environment. They happen as a consequence of events so cannot be blocked. For example, in our simple vending machine example, if the customer selects tea and pays a suitable fee, but there is a problem with the machine such that it dispenses coffee, this will happen regardless of the fact that this is not what the customer expects to receive. In CSP, the machine would not be able to produce the output "coffee" if the environment (the customer) only expects to receive "tea".

4.4 Nondeterminism

Nondeterminism is an important concept for all FSM models, but is particularly noteworthy here as it is much more subtle than with classical FSMs. Here, we must not only consider the transition labels, but also their arities and *guards*. This is because it is possible for two transitions with the same label and arity to be deterministic if their guards are mutually exclusive.

Definition 24. An EFSM is *nondeterministic* if there exists a state with two or more outgoing transitions with the same label and arity and guards which are not mutually exclusive.

Example 4.4.1. Consider the EFSM in Figure 4.4. Here, the model is nondeterministic since, if $r_1 = 100$, either of the two *vend* transitions may be taken. This illustrates the subtlety of nondeterminism in EFSMs. Here, there is only one scenario which satisfies both guards. For most values for r_1, the model will behave as expected.

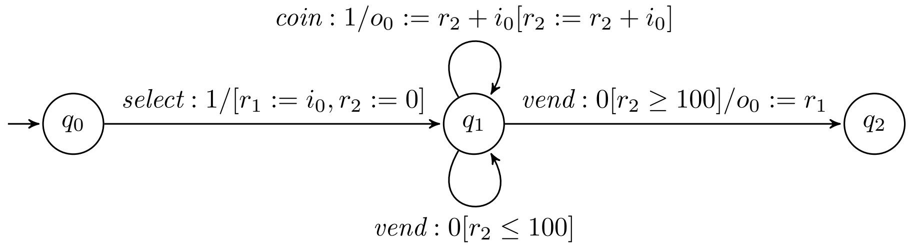

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select : 1/[r_1 := i_0, r_2 := 0]$
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_0
q_1
q_2
$vend : 0[r_2 \leq 100]$

Figure 4.4: A nondeterministic EFSM model of the drinks machine.

Nondeterministic models pose several problems from an analysis point of view. It is argued in [152] that nondeterministic models fail to capture the logical relationship between control and data that EFSMs are used to show. If there are multiple paths for the same data state, then there is no *causal* relationship between the data state and the control flow of the model.

This is a major problem if the model is used to classify correct and incorrect behaviour. Since, for each action, a transition is chosen at random from the set of viable transitions, it is quite possible to choose the "wrong" transition such that the model reaches a state from which there are no possible steps for a subsequent input. This means that nondeterministic models may have traces which are simultaneously accepted and rejected, thus making them useless for the task of behaviour classification.

Another application of EFSM models is automated test generation. Since most software is (or at least is intended to be) deterministic, it makes sense to generate tests from a deterministic model. While approaches that use nondeterministic models exist [83], it is clearly better if we can infer a deterministic model to start with. Additionally, nondeterministic models tend to produce many more test cases [84], which is undesirable as larger test suites take longer to run.

4.5 Acceptance and Recognition

A common application of (E)FSMs is the classification of behaviour. Here, traces which are accepted by the model are deemed "correct" and those which are rejected are deemed "incorrect". To classify behaviour on that basis, we must have a solid idea of what it means for an EFSM to *accept* a trace. Essentially, an EFSM accepts a trace if that trace is a trace of the model. An alternative definition is shown in Section 4.7. All traces which are not accepted are rejected.

Definition 25. Let $\mathcal{T}(e)$ represent the set of traces of EFSM e. An EFSM, e, *accepts* a trace t if $t \in \mathcal{T}(e)$.

As well as acceptance, it is helpful to discuss the idea of *recognition.* For classical FSMs, this is the same as acceptance since traces are made up of atomic actions, but the two are slightly different for EFSMs as these models produce outputs. Where acceptance is defined in terms of *traces,* recognition is defined in terms of *executions.*

Definition 26. An EFSM *recognises* an execution if it is able to respond to all the actions in sequence, even if it does not produce the correct output.

Example 4.5.1. Consider again the EFSM in Figure 1.5. It accepts all the traces in Figure 1.1. The trace $\langle$*select*("tea"), *coin*(100)/[100], *vend*()/["coffee"]$\rangle$ is not accepted, however, as there is no way for *vend* to produce the output "coffee" if this was not the input to *select.* While this not accepted by our simple drinks machine, the corresponding execution is *recognised* as the EFSM is able to respond to every action.

The notion of recognition is important in Chapter 5 when we wish to use one transition in place of another. Here, we are interested in executions which get the model to the origin state of a particular transition. We are not interested in the outputs which are produced along the way, simply the control and data states at a particular point during execution.

4.6 Model Equivalence

There are numerous relations which can be used to determine model equivalence. These are discussed in detail for classical FSMs in [145], and are arranged into a hierarchy in terms of their distinguishing power. Let us attempt to lift some of these metrics to EFSMs.

Since we intend to infer models from traces, we are most interested in those metrics defined in terms of trace semantics. Two classical FSM models a and b are deemed *trace equivalent* if the set of traces of a is equal to that of b. Since classical FSMs do not have inputs or outputs, their actions are atomic and, thus, are not distinct from events. This is not the case for EFSMs, so we need to be more careful with our definition of trace equivalence.

Definition 27. Two EFSMs e_1 and e_2 are *trace equivalent* if $\mathcal{T}(e_1) = \mathcal{T}(e_2)$.

Trace equivalence is proven inductively on traces. Two models are clearly trace equivalent on the empty trace. The inductive step is to prove that trace equivalence for traces of length k implies trace equivalence for traces of length $k + 1$. This is usually done by prepending an arbitrary event. Proofs begin by starting off both EFSMs in the initial state and running them in parallel, proving equivalence from each subsequent state. This amounts to forming a bidirectional *simulation relation* between the two models.

Definition 28. For two EFSMs e_1 and e_2, e_1 *simulates* e_2 if there exists a function $\mathcal{S}$ from the states of e_1 to those of e_2 such that

1. If s is the initial state of e_1 then $\mathcal{S}(s)$ is the initial state of e_2.
2. For all actions (l, i), if $s \xrightarrow{l(i)/o}_{e_1} s'$ then $\mathcal{S}(s) \xrightarrow{l(i)/o}_{e_2} \mathcal{S}(s')$ and the outputs are equivalent.

Example 4.6.1. Consider the EFSM in Figure 4.5. This machine is trace equivalent to the one in Figure 1.5. It may have an extra state, but this is simply an "unrolling" of the reflexive *coin* transition. While it is the case that Figure 4.5 simulates Figure 1.5, the inverse is not true since state q_1 in Figure 1.5 is simulated by states q_1 and q_2 in Figure 4.5 meaning that we cannot set up a *function* between the states.

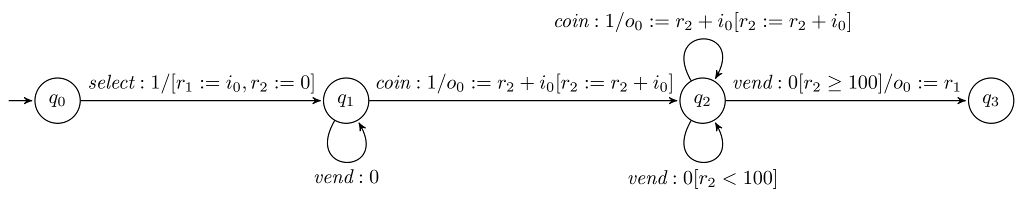

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_0
$select : 1/[r_1 := i_0, r_2 := 0]$
q_1
$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_2
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_3
$vend : 0$
$vend : 0[r_2 < 100]$

Figure 4.5: An EFSM which is trace equivalent to the one in Figure 1.5.

If we are inferring EFSMs by merging states in a PTA built from a trace set, equivalence between the initial PTA and the final model is too strong a condition. If we assert that $\mathcal{T}(\text{PTA}) = \mathcal{T}(\text{inferred})$, then we are minimising the PTA rather than inferring a model which is able to respond to new traces. Thus, we cannot predict the behaviour of the system. Instead, we would like the inferred model to *simulate* the PTA such that $\mathcal{T}(\text{PTA}) \subseteq \mathcal{T}(\text{inferred})$.

For EFSMs, the distinction between executions and traces is vital. If we define equivalence based on executions, we can end up with EFSMs with completely different output behaviour being classed as equivalent.

Example 4.6.2. Consider the EFSM in Figure 4.6. This machine is an errant version of the one in Figure 1.5 which only outputs tea. It is able to respond to all the same actions as Figure 4.6 but there is an observable difference in the behaviour of the $q_1 \xrightarrow{vend} q_2$ transition. In Figure 1.5, the output is the value of r_2. Here, it is the value "tea". Thus, if the customer selects "coffee", the difference in behaviour is revealed. Defining *execution equivalence* in a similar way to trace equivalence but with executions instead of traces, we can say that Figure 4.6 is *executionally equivalent* to Figure 1.5 but the two are not *trace equivalent.*

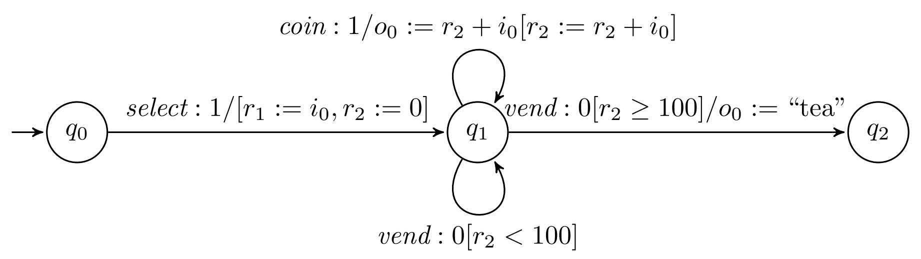

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select : 1/[r_1 := i_0, r_2 := 0]$
$vend : 0[r_2 \geq 100]/o_0 :=$ "tea"
$vend : 0[r_2 < 100]$
q_0
q_1
q_2

Figure 4.6: An errant vending machine which only dispenses tea, regardless of the customer's drink choice.

4.7 Formalisation in Isabelle

Isabelle [120] is an interactive theorem prover which allows formulae to be expressed in a formal language and provides tools to automatically prove properties of those formulae in a logical calculus. Isabelle has support for several different logics, the most common being Higher Order Logic (HOL). Here, we can define datatypes and functions using a Haskell-like syntax, and prove properties over these definitions with a high degree of automation. This makes Isabelle the perfect tool with which to properly formalise Definition 21 and the various equivalence and simulation relations from Section 2.3. This section details my Isabelle formalisation of EFSMs, which is available at [65], and also provides proofs that our formalisation satisfies some key intuitive properties such as prefix closure and finite register usage.

While similar tools such as Coq[1] and Agda[2] exist, Isabelle is more well established and has a wealth of search tools and automated proof tactics such as `sledgehammer` which improve the user experience. It is also equipped with a *code generator* [77] which enables functions and datatypes to be exported to runnable code in several languages. This allowed me to use my EFSM formalisation as the basis of my inference tool in Chapter 6 without having to manually code up an EFSM data structure.

4.7.1 Transitions

As described in Section 4.2, EFSM transitions have five components: label, arity, guards, outputs, and updates. To implement this in Isabelle, we make use of the built-in `record` type such that each component can be easily accessed by its name. The type definitions for these components are shown in Figure 4.7.

Transition labels are strings, and the arities natural numbers. Guards have a defined expression type `gexp` (detailed in Subsection 4.7.5) and the output and update functions are defined using another datatype `aexp` (detailed in Subsection 4.7.4). Outputs are simply a list of expressions to be evaluated. Updates are a list of pairs, the first element being the index of the register to be updated, and the second element being an arithmetic expression to be evaluated.

[1]`https://coq.inria.fr` (Accessed 24/03/20)
[2]`https://github.com/agda/agda` (Accessed 24/03/20)

```
record transition =
  Label :: String.literal
  Arity :: nat
  Guards :: vname gexp list
  Outputs :: vname aexp list
  Updates :: (nat × vname aexp) list
```

Figure 4.7: Isabelle/HOL type definitions for transitions.

Example 4.7.1. The definition below shows how the transition $coin : 1/o_0 := r_2 + i_1[r_2 := r_2 + i_0]$ is represented in Isabelle.

```
definition coin :: transition where
coin ≡ (|
        Label = STR ''coin'',
        Arity = 1,
        Guards = [],
        Outputs = [Plus (V (R 2)) (V (I 0))],
        Updates = [(2, Plus (V (R 2)) (V (I 0)))]
      |)
```

4.7.2 EFSMs as Finite Sets

So far in this thesis, states have been indexed numerically in the form q_n, where $n \in \mathbb{N}$. For any given EFSM, we thus have $S \subset \mathbb{N}$, where S is finite. It therefore makes sense to use natural numbers to index states in the Isabelle formalisation. Definition 21 only allows us to have finitely many states, however it is a necessity to use an infinite datatype here if we wish to formalise the inference process presented in Chapter 6 because we need to be able to arbitrarily add and remove states. Since the vast majority of "states" are unused — they have no outgoing or incoming transitions — they can be safely ignored. If we enforce the fact that any EFSM has finitely many transitions between states, we can be safe in the knowledge that only finitely many states will be used.

Since Definition 21 states that the transition matrix T is finite, it must be the case that all EFSM models have only finitely many transitions between states. We can thus represent any EFSM transition matrix as a finite set of triples of the form $((s, s'), t)$, in which s is the origin state, s' is the destination state, and t is a transition between the two. In Isabelle, I make use of the theory of finite sets (FSet) from the HOL library. Using *finite* sets to represent transition matrices is vital in ensuring that the formalisation is consistent with Definition 21. Without this restriction, there would be the potential for EFSMs to contain infinitely many transitions and thus have infinite states.

Introducing the convention that state q_0 is always the initial state removes the need to explicitly specify one for each EFSM. This means that any EFSM $e = (S, s_0, T)$ can then be characterised solely by its transition matrix, T as follows.

$$s_0 = 0$$
$$S = \{s | \exists s' t.((s, s'), t) \in dom(T) \vee ((s', s), t) \in dom(T)\}$$

While there is no explicit need to index states numerically (hence Definition 21 does not require this), I made the decision to index states this way in Isabelle as it allows the arbitrary addition and removal of states, as well as an implicit initial state. An alternative (and possibly more desirable) to this would be to allow the user to explicitly pass in a datatype representing the state space, and then have the initial state be the infimum of this type. To represent EFSMs purely as sets of transitions, it is essential that the initial state be implicit in this way since there is no way to explicitly specify it as such without defining a more complex datatype.

4.7.3 Input and Data Values

Definition 21 deliberately does not restrict the types of inputs, outputs, and register values, however any concrete implementation of EFSMs clearly requires these types to be specified. Looking at the traces we have seen so far, for example in Figure 1.1, we at least need the capability to handle integers and strings. Other basic types include booleans and floating-point numbers, but we will stick to integers and strings for now. Boolean values can be modelled with the strings "true" and "false", and floating-point numbers are not particularly easy to work with in Isabelle.

Our definition of EFSMs is not strongly typed, but Isabelle very much is, so we must define a datatype `value` which aggregates our supported types into a single datatype. This way, we can define things in terms of the `value` type and dynamically handle the different cases. The `value` datatype, is a *sum type* of integers and strings that tags its members as either a number (`Num`) or a string (`Str`). In Isabelle, it is defined as follows.

datatype *value* = *Num int* | *Str String.literal*

Since register values are strictly undefined until they are first assigned, the data state is formalised as a function from the register index (a natural number) to a `value option`. The Isabelle `option` type, equivalent to the Haskell `Maybe` type, is used to make partial functions total. It takes a type argument and is defined as being either `None` (the bottom element used to represent an undefined value) or `Some x`, where `x` is an element from the specified type, in this case a `value`. Since the number of registers used by any EFSM is known to be finite[3], we make use of the theory of finite functions (FinFun [104]) from the HOL library. Here, a FinFun is a function which is constant except for finitely many points. This corresponds to a *map* (or *dictionary*) in conventional programming languages.

4.7.4 Arithmetic and Outputs

Transition outputs and updates take the form of arithmetic expressions. For the inference technique presented in Chapter 6, we will need to recognise and potentially transform the different functions. Consequently, I use a *deep embedding* for arithmetic expressions rather than simply defining them as functions. A deep embedding defines operators as a datatype over which we can define functions. The converse of this, a *shallow embedding*, uses Isabelle's existing syntax to define purely semantic operations. The deep embedding allows us to recognise and transform different expressions where the shallow embedding does not. The `aexp` datatype is defined as follows.

datatype *'a aexp* = *L value* | *V 'a* | *Plus 'a aexp 'a aexp* | *Minus 'a aexp 'a aexp* | *Times 'a aexp 'a aexp*

[3]The proof of this appears later in this section as one of our "key properties" in Subsection 4.7.11.

Here, there are two base cases and three recursive operations. The two base cases are literal constants (tagged `L`) and variable values (tagged `V`). The recursive cases are addition, subtraction, and multiplication. Division is not supported since the `value` datatype does not support floats. There are also no non-terminal operations over strings, such as concatenation, although their addition to the datatype would be relatively straightforward.

Next, we must define a function `aval` to evaluate arithmetic expressions. Because our definition of EFSMs is not strongly typed, we cannot use conventional arithmetic to evaluate such expressions. The reason for this is best illustrated with an example.

Example 4.7.2. Consider the output expression $r_2 + i_0$ from the *coin* transition in Figure 4.5. Clearly this is intended to be a numerical operation but, because we are using the `value` type to represent registers and inputs, we must be prepared for the possibility of a malformed input. What would happen if, instead of a numeric value, the *coin* transition received a string value input? Clearly we cannot add a string to an integer in any meaningful way, so the result of evaluating this expression is undefined.

Since Isabelle functions must be total, I define an arithmetic for the `value` datatype in terms of `options` to allow the bottom element, `None`, to represent undefined values. Well formed expressions evaluate as normal. In general, a binary function $f : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ is lifted to the optional arithmetic to become $f' : \mathbb{Z}$ `option` $\to \mathbb{Z}$ `option` $\to \mathbb{Z}$ `option`. To do this, I define a function `maybeIntArith` as follows.

```
fun maybe-arith-int :: (int ⇒ int ⇒ int) ⇒ value option ⇒ value option ⇒ value option where
  maybe-arith-int f (Some (Num x)) (Some (Num y)) = Some (Num (f x y)) |
  maybe-arith-int - - - = None
```

Because of our optional arithmetic and the possibility for undefinedness, outputs are of type `value option` rather than simply `value`. Having said that, for well-typed inputs, the outputs will all evaluate to `Some` value such that the output is well-defined.

4.7.5 Guards

Transition guards are expressed with guard expressions that are evaluated to test whether the current register values and supplied inputs meet the specified conditions. Guards are defined as a datatype `gexp` in terms of arithmetic expressions as follows.

```
datatype 'a gexp = Bc bool | Eq 'a aexp 'a aexp | Gt 'a aexp 'a aexp | In 'a value list | Nor 'a gexp 'a gexp
```

Here, the only terminal cases are boolean constants *true* and *false*. We can compare the results of two arithmetic expressions either for equality (`Eq`) or inequality, in the form of the “greater than” operator (`Gt`). Finally, we have the logical connective NOR, which is equivalent to “not or”. The reason for using NOR logic rather than the more conventional logical operations of conjunction, disjunction, and negation is that NOR is able to capture the functionality of all three in a single case. This reduces the number of elements in the datatype and, consequently, the number of subgoals in the various proofs involving guard expressions. For ease of expression, I defined functions for AND, OR, and NOT in terms of NOR, as well as functions for the $<$, $\leq$, $\geq$, and $\neq$ operators.

definition $gNot :: {}'a\ gexp \Rightarrow {}'a\ gexp$ **where**
$gNot\ g \equiv Nor\ g\ g$

definition $gOr :: {}'a\ gexp \Rightarrow {}'a\ gexp \Rightarrow {}'a\ gexp$ **where**
$gOr\ v\ va \equiv Nor\ (Nor\ v\ va)\ (Nor\ v\ va)$

definition $gAnd :: {}'a\ gexp \Rightarrow {}'a\ gexp \Rightarrow {}'a\ gexp$ **where**
$gAnd\ v\ va \equiv Nor\ (Nor\ v\ v)\ (Nor\ va\ va)$

definition $gImplies :: {}'a\ gexp \Rightarrow {}'a\ gexp \Rightarrow {}'a\ gexp$ **where**
$gImplies\ p\ q \equiv gOr\ (gNot\ p)\ q$

definition $Lt :: {}'a\ aexp \Rightarrow {}'a\ aexp \Rightarrow {}'a\ gexp$ **where**
$Lt\ a\ b \equiv Gt\ b\ a$

definition $Le :: {}'a\ aexp \Rightarrow {}'a\ aexp \Rightarrow {}'a\ gexp$ **where**
$Le\ v\ va \equiv gNot\ (Gt\ v\ va)$

definition $Ge :: {}'a\ aexp \Rightarrow {}'a\ aexp \Rightarrow {}'a\ gexp$ **where**
$Ge\ v\ va \equiv gNot\ (Lt\ v\ va)$

definition $Ne :: {}'a\ aexp \Rightarrow {}'a\ aexp \Rightarrow {}'a\ gexp$ **where**
$Ne\ v\ va \equiv gNot\ (Eq\ v\ va)$

Again, the untyped nature of EFSMs, introduces additional complexity when we attempt to evaluate guard expressions, meaning that we cannot rely on conventional Boolean logic. The reason for this is, again, best illustrated with an example.

Example 4.7.3. Consider the guard expression $r_1 > i_0$. Now let $r_1 = 5$ and $i_0 =$ "hello" . The "greater than" relation is not well-defined here. Thus, r_1 is not greater than i_0 and the guard expression must evaluate to *false*. Now consider the guard $5 \geq$ "hello" , which is the negation of $r_1 > i_0$. Under the same variable valuation, this expression must also evaluate to *false* since 5 is neither greater than nor equal to "hello" . Then problem is then how to handle $\neg(r_1 > i_0)$. If $r_1 > i_0$ evaluates to *false*, then its negation must evaluate to *true*, but this is inconsistent with the evaluation of $5 \geq$ "hello" . We need to separate this notion of inconsistency from simple logical falsity.

To fix this, we must use a three-valued logic instead of the conventional binary one. For this, I use Bochvar logic [19]. Here, there are three operators *true*, *false*, and *invalid*. Here, *true* and *false* behave like their boolean counterparts. The *invalid* truth value signifies that something has gone wrong, and allows us to propagate this fact over negation. We can then only take a transition if its guards evaluate to *true*.

The conventional operators of conjunction, disjunction, and negation are defined as follows. In these operations, any expression involving *invalid* evaluates to *invalid*. This corresponds to the type errors which might be thrown by a conventional dynamically typed programming language such as Python when asked to evaluate a badly typed expression.

$\wedge_?$	T	F	I
T	T	F	I
F	F	F	I
I	I	I	I

$\vee_?$	T	F	I
T	T	T	I
F	T	F	I
I	I	I	I

$\neg_?$	
T	F
F	T
I	I

4.7.6 Expression Orderings

It is helpful to define a lexicographical ordering over expressions such that we can say that one is "less than" another.[4] This is mainly useful for implementational reasons as it allows us to transform the set representation of EFSMs to lists for efficient processing. This transformation requires an ordering since lists are inherently ordered structures and sets are not.

If we define our expression ordering correctly, it also provides a way of "breaking ties" between semantically equivalent but syntactically different expressions, as we can use the "smallest" one. This makes for more easily understood models and becomes quite important in Chapter 7 where we try to infer output and update functions automatically from trace data.

In order to transform sets into sorted lists, we require a *linear ordering.* A linear order on a set S is a (binary) relation $<$ with the following properties.

Irreflexivity $x \not< x$

Asymmetry $x < y \implies y \not< x$

Transitivity $x < y < z \implies x < z$

Comparison $x < z \implies x < y \lor y < z$

Connectedness $x \not< y \implies y \not< x \implies x = y$

This enables us to compare any two expressions and have one be less than the other, unless they are equivalent. As we have encoded expressions with a deep embedding, our order must be over the *syntax* of the expressions, without reference to the *semantics.* This has the somewhat frustrating consequence that commutative operators cannot be deemed equivalent. For example, the expressions $i_1 + r_1$ and $r_1 + i_1$ are semantically equivalent but syntactically different, so one must be less than the other. While this is annoying, it is simply a feature of deep embeddings.

Since our expressions are defined on top of each other (guard expressions being made up of arithmetic expressions, which are themselves made up of literals and variables), let us first define orderings for values and variable names. These are fairly intuitive as neither data type is recursive. Firstly, the `value` data type has two cases: integers and strings, both of which have existing natural orderings. The order for `values` is then defined on top of this as follows.

```
fun less-value :: value ⇒ value ⇒ bool where
  (Num n) < (Str s) = True |
  (Str s) < (Num n) = False |
  (Str s1) < (Str s2) = (s1 < s2) |
  (Num n1) < (Num n2) = (n1 < n2)
```

Here, the `Str` case has arbitrarily been chosen to be greater than the `Num` case, with the natural orders over the respective datatypes being used in cases where we wish to compare two elements of the same type. The decision to place strings higher in the ordering is an arbitrary one as there is no meaningful way to compare integers and strings.

The ordering on variable names is similar. Here, we also have two cases: inputs and registers. Since there is no obvious way to compare the two, we define inputs to be less than registers. When we want to compare two inputs or registers, we use the natural ordering on their indices.

[4]This is used purely over the syntax of expressions. It is not intended or used for guard evaluation like in Example 4.7.3 as it does not solve the problem of the result of an arithmetic expression being undefined.

```
fun less-vname :: vname ⇒ vname ⇒ bool where
  (I n1) < (R n2) = True |
  (R n1) < (I n2) = False |
  (I n1) < (I n2) = (n1 < n2) |
  (R n1) < (R n2) = (n1 < n2)
```

Next we must define an ordering on arithmetic expressions. This is a little more complicated as there are more cases, some of which are recursive. Unlike values and variables, there is an intuitive way to order expressions. The expression i_1+r_1 is intuitively smaller than $(i_1+r_1)-5$, for example. We can formalise this intuition by defining a function `height` which measures the height of the expression's *parse tree*.

```
fun height :: 'a aexp ⇒ nat  where
  height (L l2) = 1 |
  height (V v2) = 1 |
  height (Plus e1 e2) = 1 + max (height e1) (height e2) |
  height (Minus e1 e2) = 1 + max (height e1) (height e2) |
  height (Times e1 e2) = 1 + max (height e1) (height e2)
```

Unfortunately, we cannot order purely by the height of the parse tree as this only forms a preorder, so only satisfies one of the conditions necessary for a linear ordering. Let us then attempt to incorporate height into a stricter order relation. Firstly, we say that if the height of expression a_1 is less than that of a_2, then $a_1 < a_2$. If it is greater, then $a_1 > a_2$. If the two formulae have equal heights, then we must consider the syntax tree itself. Let us order the syntactic cases as follows.

$$\text{literal constants} < \text{variables} < (a_1 + a_2) < (a_1 - a_2) < (a_1 \times a_2)$$

While this ordering is somewhat arbitrary, it makes intuitive sense to place the base cases below the recursive ones. We have already defined an ordering on values and variable names, so it just remains to define what happens when we wish to compare two instances of the same binary operation.

There is a natural ordering on pairs of elements as follows.

$$(a_1, a_2) < (a_1', a_2') \iff ((a_1 < a_1') \vee (a_1 = a_1' \wedge a_2 < a_2'))$$

We can apply this ordering to the arguments of the two binary functions such that

$$(a_1 + a_2) < (a_1' + a_2') \iff ((a_1 < a_1') \vee (a_1 = a_1' \wedge a_2 < a_2'))$$

and similar for the other cases.

Finally, I define a similar ordering on guard expressions. In the data type, we have five cases: boolean constant, equivalence, set membership, and the NOR of two guards. Let us arbitrarily place the cases in this order such that boolean constants are the smallest case and NOR is the largest. When we wish to compare two equivalent binary operators, we use the ordering on their respective pairs of arguments as defined above.

Having defined orderings for all components of transitions, we can now define an order on transitions themselves. This is simply a recursive application of the pair ordering such that

$$\begin{aligned} t_1 < t_2 \iff & (t_1.\mathit{label}, (t_1.\mathit{arity}, (t_1.\mathit{guards}, (t_1.\mathit{outputs}, t_1.\mathit{updates})))) < \\ & (t_2.\mathit{label}, (t_2.\mathit{arity}, (t_2.\mathit{guards}, (t_2.\mathit{outputs}, t_2.\mathit{updates})))) \end{aligned}$$

EFSMs are represented by finite sets of pairs of the form $((s_1, s_2), t)$, where s_1 and s_2 are states indexed by natural numbers, and t is a transition. Now that we have linear orders for pairs and transitions, we can arbitrarily convert between a set representation and a sorted list representation such that we can make use of efficient list processing. It also eases certain proofs as it enables us to induct over EFSM transition matrices.

4.7.7 Processing Traces

Definition 22 defines an action as a label-input pair. This is exactly the same in Isabelle. Like transition labels, action labels are represented by strings. Inputs are represented by lists of `values`. Executions are lists of actions. The function `observe_execution` proceeds as described in Subsection 4.3.1 to produce the corresponding observation of an execution.

```
fun observe-execution :: transition-matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ outputs list where
  observe-execution - - - [] = [] |
  observe-execution e s r ((l, i)#as)  = (
    let viable = possible-steps e s r l i in
    if viable = {||} then
      []
    else
      let (s', t) = Eps (λx. x |∈| viable) in
      (evaluate-outputs t i r)#(observe-execution e s' (evaluate-updates t i r) as)
  )
```

To observe an action, we first calculate the set of viable transitions. If this is empty, there are no possible steps the model can make so we halt execution here. If there are possible steps, we pick one at random (with the `Eps` operator), evaluate the outputs, and continue to observe the execution with the updated control flow and data states.

We can see from the `observe_execution` function that, in order to observe a trace, we must know the register values at each step in execution. While they are not directly observable, the values of registers are very much connected to how an EFSM behaves. This means that the semantics of a given EFSM are *context-dependent* such that its behaviour is not entirely dependent on the control flow state but is also affected by the data state. This turns out to be critical to the inference process, and is discussed in detail in Section 5.2.

4.7.8 Acceptance and Recognition

As discussed in Section 4.5, EFSM models are characterised by the traces they accept and the executions they recognise. In Isabelle, trace acceptance and execution recognition are defined inductively as follows.

```
inductive accepts-trace :: transition-matrix ⇒ cfstate ⇒ registers ⇒ trace ⇒ bool where
  base [simp]: accepts-trace e s r [] |
  step: ∃(s', T) |∈| possible-steps e s r l i.
         evaluate-outputs T i r = map Some p ∧ accepts-trace e s' (evaluate-updates T i r) t ⟹
         accepts-trace e s r ((l, i, p)#t)

inductive recognises-execution :: transition-matrix ⇒ nat ⇒ registers ⇒ execution ⇒ bool where
  base [simp]: recognises-execution e s r [] |
  step: ∃(s', T) |∈| possible-steps e s r l i.
         recognises-execution e s' (evaluate-updates T i r) t ⟹
         recognises-execution e s r ((l, i)#t)
```

For the base case, any EFSM trivially accepts the empty trace since there is nothing for it to respond to. For the step case, there must be a viable transition for the first event that produces the correct output and updates the data state such that the rest of the trace is accepted. Recognition is defined similarly but without the restriction on outputs.

4.7.9 Equivalence and Simulation

Section 4.6 defines trace equivalence and simulation. These are defined in Isabelle as follows.

```
definition T :: transition-matrix ⇒ trace set where
  T e = {t. accepts-trace e 0 <> t}
definition trace-equivalent e1 e2 = (T e1 = T e2)
inductive trace-simulation :: (cfstate ⇒ cfstate) ⇒ transition-matrix ⇒ cfstate ⇒ registers ⇒
transition-matrix ⇒ cfstate ⇒ registers ⇒ trace ⇒ bool where
  base: s2 = f s1 ⟹ trace-simulation f e1 s1 r1 e2 s2 r2 [] |
  step: s2 = f s1 ⟹
        ∀(s1', t1) |∈| ffilter (λ(s1', t1). evaluate-outputs t1 i r1 = map Some o) (possible-steps e1 s1 r1 l i).
          ∃(s2', t2) |∈| possible-steps e2 s2 r2 l i. evaluate-outputs t2 i r2 = map Some o ∧
            trace-simulation f e1 s1' (evaluate-updates t1 i r1) e2 s2' (evaluate-updates t2 i r2) es ⟹
        trace-simulation f e1 s1 r1 e2 s2 r2 ((l, i, o)#es)
```

The `trace_equivalent` function is defined exactly as in Definition 27. The `simulates` predicate is defined inductively on traces for a given simulation function $f : \mathbb{N} \to \mathbb{N}$. For an EFSM e_1 in state s_1 to simulate e_2 in state s_2, the first condition that must hold is $s_2 = f(s_1)$. For the empty trace, this is the only condition that must hold. For the step case, e_2 must have a corresponding step for every step that e_1 can make which produces the expected output. To make the relation type check, we must `Some` the list of expected outputs (which are `values`) to turn them into `value options`, since this is the returned by `evluate_outputs`.

It is also possible to define equivalence and simulation in terms of *executions.* These definitions are similar to the ones above but without the expected output that comes from traces. Here, the outputs produced by the two EFSMs must simply be equivalent to each other rather than some external value. This actually means that executional equivalence and simulation are actually stronger predicates than their trace counterparts.

```
inductive executionally-equivalent :: transition-matrix ⇒ cfstate ⇒ registers ⇒
transition-matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ bool where
  base [simp]: executionally-equivalent e1 s1 r1 e2 s2 r2 [] |
  step: ∀(s1', t1) |∈| possible-steps e1 s1 r1 l i.
          ∃(s2', t2) |∈| possible-steps e2 s2 r2 l i.
            evaluate-outputs t1 i r1 = evaluate-outputs t2 i r2 ∧
            executionally-equivalent e1 s1' (evaluate-updates t1 i r1) e2 s2' (evaluate-updates t2 i r2) es ⟹
        ∀(s2', t2) |∈| possible-steps e2 s2 r2 l i.
          ∃(s1', t1) |∈| possible-steps e1 s1 r1 l i.
            evaluate-outputs t1 i r1 = evaluate-outputs t2 i r2 ∧
            executionally-equivalent e1 s1' (evaluate-updates t1 i r1) e2 s2' (evaluate-updates t2 i r2) es ⟹
        executionally-equivalent e1 s1 r1 e2 s2 r2 ((l, i)#es)

inductive execution-simulation :: (cfstate ⇒ cfstate) ⇒ transition-matrix ⇒ cfstate ⇒
registers ⇒ transition-matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ bool where
  base: s2 = f s1 ⟹ execution-simulation f e1 s1 r1 e2 s2 r2 [] |
  step: s2 = f s1 ⟹
        ∀(s1', t1) |∈| (possible-steps e1 s1 r1 l i).
          ∃(s2', t2) |∈| possible-steps e2 s2 r2 l i.
            evaluate-outputs t1 i r1 = evaluate-outputs t2 i r2 ∧
            execution-simulation f e1 s1' (evaluate-updates t1 i r1) e2 s2' (evaluate-updates t2 i r2) es ⟹
        execution-simulation f e1 s1 r1 e2 s2 r2 ((l, i)#es)
```

Here, rather than being defined simply in terms of recognition, executional equivalence is defined inductively. Essentially two EFSMs are executionally equivalent if, for every step of every execution, they produce equivalent outputs. This means that the two EFSMs not only accept the same traces, but also behave the same on traces that are rejected. This means that, in terms of observable behaviour, there is no difference between the two models. It therefore makes sense to call executional equivalence *observational equivalence*.

4.7.10 Reachability

Reachability is an important part of any (E)FSM formalism. The basic idea is that states are reachable if there exists a trace which visits that state. The `visits` predicate is defined inductively on traces as follows.

```
inductive visits :: cfstate ⇒ transition-matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ bool where
  base [simp]: visits s e s r [] |
  step: ∃ (s', T) |∈| possible-steps e s r l i. visits target e s' (evaluate-updates T i r) t ⟹
        visits target e s r ((l, i)#t)

definition reachable s e = (∃ t. visits s e 0 <> t)
```

In essence, the empty trace visits a state s' if the model is already in that state. If the model is not already in the target state, there must be a transition we can take which will allow us to visit the target state. Using this predicate, it is then trivial to define reachability.

A key feature of EFSMs is that they have a separate *data state* in addition to their control flow state. It therefore makes sense to define a reachability criterion for this too. This is a very similar pattern to the `visits` and `reachable` functions above except that they include a target data state as well as a control flow state.

```
inductive obtains :: cfstate ⇒ registers ⇒ transition-matrix ⇒ cfstate ⇒ registers ⇒ execution ⇒ bool where
  base [simp]: obtains s r e s r [] |
  step: ∃ (s'', T) |∈| possible-steps e s' r' l i. obtains s r e s'' (evaluate-updates T i r') t ⟹
        obtains s r e s' r' ((l, i)#t)

definition obtainable s r e = (∃ t. obtains s r e 0 <> t)
```

4.7.11 Key Properties

Having formalised the various aspects of EFSMs in Isabelle, we can now prove certain key properties which show that our formalisation is consistent with our intuitive understanding. Some of these, for example the fact that there are finitely many states in S hold by construction. Others require more substantial proof. While the proofs here are not particularly earth-shattering, they show that the formalisation is consistent with our intuitive understanding of how EFSMs should behave and also lay the foundations for more interesting proofs in Chapter 5.

Finite Registers

Because registers are indexed by natural numbers, all EFSMs have access to infinitely many storage locations. In fact, though, since any EFSM has finitely many transitions and each transition only has finitely many data updates, each EFSM only uses finitely many registers.

```
lemma finite-all-regs: finite (all-regs e)
```

Prefix Closure

The idea of *prefix closure*, discussed in Subsection 2.1.1, is critical to most (E)FSM inference techniques. In short, prefix closure means that if a trace is accepted then all of its prefixes are also accepted. This allows us to begin our inference with a PTA.

lemma *prefix-closure*: *accepts-trace e s r* $(t@t') \Longrightarrow$ *accepts-trace e s r t*

Simulation and Equivalence

As discussed in Section 4.6, when we test for trace equivalence, we are effectively building a simulation relation between two models. Thus, if there exists a simulation relation between the two models, they should be trace equivalent.

lemma *simulation-implies-trace-equivalent*:
trace-simulates e1 e2 $\Longrightarrow$ *trace-simulates e2 e1* $\Longrightarrow$ *trace-equivalent e1 e2*

The implication does not hold the other way, however. Example 4.6.1 serves as a counterexample. Essentially, the problem lies in the fact that one state can simulate multiple states. The fact that the simulation relation must be a *function* means that we cannot have one state being simulated by multiple states, so we cannot always set up a function in the other direction.

It was stated in Subsection 4.7.9 that executional simulation is a stronger relation than trace simulation. This means that if one model executionally simulates another, then there is also a trace simulation between the two models.

lemma *execution-simulation-trace-simulation*:
execution-simulation f e1 s1 r1 e2 s2 r2 (*map* $(\lambda(l, i, o). (l, i))$ *t*) $\Longrightarrow$ *trace-simulation f e1 s1 r1 e2 s2 r2 t*

4.7.12 Removing Unreachable States

A key property of unreachable states is that we should be able to remove them without any effect on behaviour. In terms of our equivalence metrics, an EFSM should be observationally equivalent to itself with its unreachable state removed. The proof of this is by induction on traces using an arbitrary starting state, but is interesting as it requires the starting state to be *obtainable*, rather than just being *reachable*. This is because the possible steps we can take from a given state depend not just on the control flow state, but also on the values of the registers.

lemma *executionally-equivalent-remove-unreachable-state*:
$\neg$ *reachable* s' *e* $\Longrightarrow$ *executionally-equivalent e 0* <> (*remove-state* s' *e*) *0* <> *x*

lemma *executionally-equivalent-remove-unreachable-state-arbitrary*:
obtainable s c e $\Longrightarrow$ $\neg$ *reachable* s' *e* $\Longrightarrow$ *executionally-equivalent e s c* (*remove-state* s' *e*) *s c x*

The first lemma states that if a state s' is unreachable in EFSM e, the original model is executionally equivalent to the model with that state removed for an arbitrary execution x when both models are started in the initial state q_0 with the initial context $\langle\rangle$. The proof of this makes reference to the second lemma which states that if a context c is obtainable in state s of EFSM e and state s' is unreachable, the two models are executionally equivalent if run from state s with context c. This is proven by induction on executions. Then, since the initial context is trivially reachable in the initial context, the first lemma can be trivially proved.

4.8 Conclusion

This chapter presented the definition of EFSMs which will be used for the rest of this work. Here, transitions have parametrised inputs which are used, along with the persistent data state, to produce observable outputs. Transitions also have the ability to update the data state by evaluating functions in terms of inputs and registers. This means that it is possible to compute the corresponding outputs of any execution.

I have also formalised Definition 21 in Isabelle and proven various key properties of EFSMs. Not only does this help to show that the definition is consistent with our intuitive understanding of how EFSMs should behave, but it also paves the way for the work presented in subsequent chapters. Chapter 5 presents a criterion for determining exactly when a pair of transitions can be merged. Again, this is formalised in Isabelle [66], with various properties proven. Chapter 6 incorporates this criterion into a technique to infer EFSM models, complete with output and update functions, from system traces. The implementation of this technique also relies upon the EFSM Isabelle formalisation presented here, and makes use of Isabelle's *code generator* to transform the formalisation into an executable tool.

Chapter 5

Formalising EFSM Transition Merging

Having now formed my definition of EFSMs and provided the necessary background on FSM inference, I now begin to extend classical FSM inference methods to EFSM models. Chapter 6 realises this technique in full, but there is still a problem which needs to be solved before we can do this. Inference by state merging often requires the merging of *transitions* in order to resolve the duplication of behaviour that arises when states are merged. For classical FSMs, this is relatively straightforward, but it is much more complicated for EFSMs. Before we can infer any models, we first need a technique for determining whether transitions can be safely merged.

This chapter is the first substantial contribution of the thesis and is based on my original work published in [64] and [63]. It lays the foundations for the EFSM inference technique, detailed in Chapter 6, which involves the merging of transitions that update the internal data state. The primary contributions of this chapter are the following:

- A scheme by which constraints on data values may be traced through EFSMs.
- The definition of the *subsumption in context* relation based on this scheme, which can be used to determine under what circumstances one transition is able to account for the behaviour of another.
- The definition of the *directly subsumes* relation, which works at EFSM level to determine when it is safe to merge a given pair of transitions in a model.

5.1 Introduction

Let us begin by reminding ourselves of the problem at hand, and of our running drinks machine example from Section 1.1. We have an operational software system, in this case a simple drinks machine controller, for which we do not have access to the source code. We can, however, observe the behaviour of the system in the form of input-output traces as exemplified below.

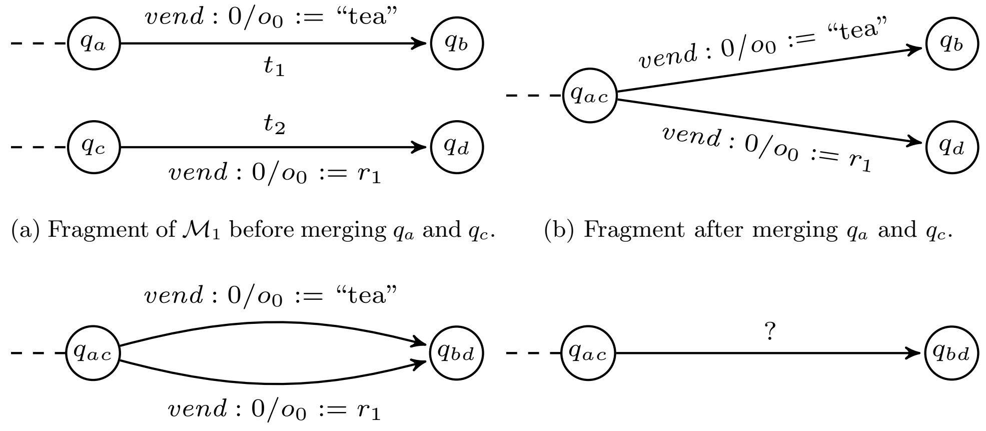
$$\langle \mathit{select}(\text{“tea”}), \mathit{coin}(50)/[50], \mathit{coin}(50)/[100], \mathit{vend}()/[\text{“tea”}]\rangle$$
$$\langle \mathit{select}(\text{“tea”}), \mathit{coin}(100)/[100], \mathit{vend}()/[\text{“tea”}]\rangle$$
$$\langle \mathit{select}(\text{“coffee”}), \mathit{coin}(50)/[50], \mathit{vend}(), \mathit{coin}(50)/[100], \mathit{vend}()/[\text{“coffee”}]\rangle$$

Given these traces, we would like to infer an EFSM model which captures the high-level behaviour of the system by *generalising* from the observed behaviour. One way to go about this is to begin by transforming the traces into a PTA, and then iteratively merge states which we believe represent the same state in the underlying program.

The decision to merge a pair of states is often made based on the commonality of outgoing transitions. As discussed in Subsection 3.3.4, this often results in nondeterminism between the outgoing transitions of newly merged states. This is not true nondeterminism, however, merely a manifestation of the fact that our model contains duplicate representations of the same behaviour. We should therefore be able to resolve the nondeterminism by merging duplicated behaviours into single representative transitions such that each behaviour only appears once.

For classical FSMs, transitions only represent the same behaviour if their labels are exactly equal. This means that, in classical FSM inference, we can resolve nondeterminism by simply merging the destination states of the offending transitions since two transitions with the same label, origin, and destination are not distinct. With EFSMs, transitions that express the same behaviour may not be identical, so the merging of transitions needs more careful consideration.

Example 5.1.1. Consider the EFSM fragments in Figure 5.1. Let us refer to the transition $\mathit{vend}: 0/o_0 :=$ "tea" as t_1 and $\mathit{vend}: 0/o_0 := r_1$ as t_2. Say that the inference process merges states q_a and q_c under the belief that the two *vend* transitions represent the same behaviour. This results in the model in Figure 5.1b, in which state q_{ac} has two outgoing *vend* transitions. Even though the transitions are not identical, we merged q_a and q_c because we believed that these transitions represent the same behaviour. To reflect this belief, we need to merge them into one transition which represents both behaviours.



(a) Fragment of $\mathcal{M}_1$ before merging q_a and q_c. (b) Fragment after merging q_a and q_c.

(c) Fragment after merging the destination states q_b and q_d to form $\mathcal{M}_2$. (d) We are now stuck, as we do not know how to merge the two behaviours.

Figure 5.1: The evolution of an EFSM fragment during the merging process.

To begin with, we merge the destination states of the two transitions (q_b and q_d) to form $\mathcal{M}_2$, shown in Figure 5.1c. We must then attempt to merge the two transitions themselves. This would be trivial if they were exactly identical, but they are not. One transition outputs the literal string "tea" and the other outputs the content of register r_1.

At this point, there are four possible scenarios. Firstly, t_1 may account for the behaviour of t_2. If this is the case, then it can be used as the behavioural representative and t_2 can be deleted without affecting the observable behaviour of the model. Secondly, t_2 might account for the behaviour of t_1, meaning that t_1 can be deleted and t_2 can be used as the representative. Thirdly, it may be that neither transition accounts for the behaviour of the other, but there exists a third transition which accounts for the behaviour of both. In this case, we can safely delete t_1 and t_2 and use this third transition instead. Finally, it may be that none of these situations hold, and the two transitions cannot be merged into a single behaviour. If this is the case, we were not correct to merge q_a and q_c and they must remain distinct.

The ability to merge transitions like those in Example 5.1.1 with confidence requires us to be able to check whether one transition is able to *account for* the behaviour of another. This property is called *subsumption* and was first introduced in [106] for transitions with guards, and a very similar idea is was used in [130] to merge states in symbolic execution trees. In both of these works, subsumption is about weakening the conditions on variable values, so it is sufficient to define subsumption as logical implication [106], or equivalently as a subset relation [130]. In this way, one transition is able to react to more inputs than the other, thus enabling it to *account for* the behaviour, allowing the transitions to be merged.

In the context of EFSMs in Definition 21, however, we are looking to merge transitions with output and update functions. Because of this, logical implication is not enough. We are in need of something more like the refinement relations discussed in Section 2.3. In addition to having a weaker guard, the subsuming transition must also produce identical outputs in situations where both transitions may be taken. It must also result in a posterior data state which is *consistent* with that of the subsumed transition. To extend the subsumption relation to take these factors into account, Section 5.2 introduces *contexts*, a scheme for recording constraints on the data state of EFSMs.

Example 5.1.2. Consider the EFSM in Figure 5.2, in which there are two nondeterministic *vend* transitions. Here, the bottom transition subsumes the top one since, if register r_2 is greater than 75 then it is also greater than 50. Thus, the guard of the bottom transition has a broader precondition. The two transitions are equal in every other way.

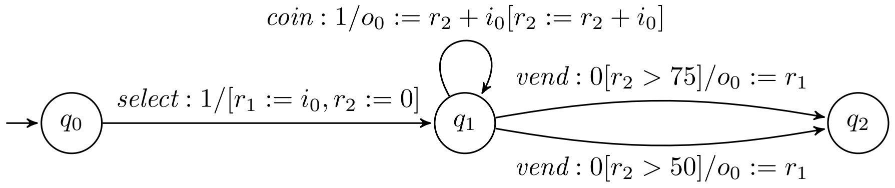

$\mathit{coin} : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$\mathit{select} : 1/[r_1 := i_0, r_2 := 0]$
$\mathit{vend} : 0[r_2 > 75]/o_0 := r_1$
$\mathit{vend} : 0[r_2 > 50]/o_0 := r_1$
q_0
q_1
q_2

Figure 5.2: An EFSM with two nondeterministic *vend* transitions.

In addition to broadening the precondition, it is crucial that the data updates performed by the two transitions are consistent with each other such that the output of subsequent transitions is not affected by the merge. It is this part of the definition which is not considered in [106, 130].

Example 5.1.3. Consider the EFSM in Figure 5.3. Here, there are two nondeterministic *coin* transitions but neither subsumes the other. While the top transition is able to account for the *observable* behaviour of the bottom one, if $i_0 = 50$ and $r_2 = 0$ then the updates of the two transitions are not *consistent.* The top transition updates r_2 to 50 while the bottom transition leaves it unchanged. This is likely to disturb future outputs that make use of r_2, so we cannot use the top one in place of the bottom one.

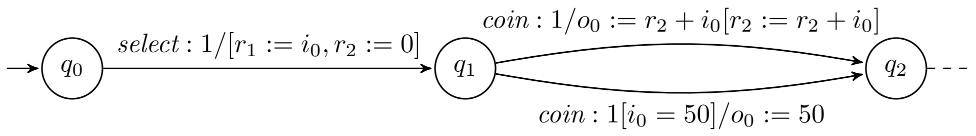

$\mathit{select} : 1/[r_1 := i_0, r_2 := 0]$
$\mathit{coin} : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$\mathit{coin} : 1[i_0 = 50]/o_0 := 50$
q_0
q_1
q_2

Figure 5.3: A fragment of an EFSM with two nondeterministic *coin* transitions.

The remainder of this chapter is structured as follows. Section 5.2 introduces contexts and shows how we can use the guards and updates of transitions to infer constraints on the values of variables. In Section 5.3, I use contexts to extend the concept of subsumption to transitions with output and update functions. In Section 5.4, I use subsumption in context as the foundation for the *direct subsumption* relation, which can be used to determine whether it is safe to merge a given pair of transitions. Finally, in Section 5.5, I show how contexts can be used to analyse properties of models once they have been inferred.

5.2 Contexts

To extend subsumption to transitions with output and update functions, we need to relate the internal data state of the system to the potential output values of transitions. This section proposes the use of *contexts* to help with this, but first let us examine why they are necessary.

As discussed in Chapter 3, the models produced by [106] have transitions which place guards on the data state of the model. Those transitions do not have observable outputs, nor do they update the data state. It is therefore sufficient to define subsumption in terms of logical implication on the guards of transitions. If the guard of one transition is logically implied by the other, then it *accounts for* that transition's behaviour.

Example 5.2.1. Consider the two *vend* transitions from Example 5.1.2. In [106], they might be represented as $vend(r_1 > 75)$ and $vend(r_1 > 50)$. Here, since $r_1 > 75 \implies r_1 > 50$, the guard of the second transition is more permissive than that of the first. This means that all the behaviour of the first transition is implemented by the second. Consequently, the first transition *subsumes* the second.

The transitions in my EFSM models not only have guards, but also have observable outputs and functions that update the data state. Since the internal data state of the system is not directly observable, it might be tempting to try to use *observational equivalence* as a behavioural equality metric. We have already seen observational equivalence for EFSMs in Subsection 4.7.9, but we really want subsumption to work at the *transition* level. We could try to adapt the relation to work for transitions by saying that a pair of transitions is observationally equivalent if they always produce identical outputs, but this is not adequate for our needs.

As mentioned in Chapter 4, the semantics of an EFSM are *context-dependent.* Knowing the control flow state alone is not enough to determine the output of an EFSM when faced with a given execution. We also need to know the current *data state.* This is because, while the registers themselves are not directly observable, they have an observable effect on the behaviour of an EFSM. The problem with trying to define observational equivalence on transitions is that data updates cannot be taken into account. Registers could be updated differently by the two transitions and then used as part of subsequent output functions, thus allowing the behavioural disparity to manifest itself at a later point in model execution. Thus, the use of observational equivalence on a per-transition basis is not a strong enough criterion for transition merging.

Example 5.2.2. Consider the EFSM from Figure 5.3. Register r_2 always holds the value 0 in state q_1 as the only way to get there from the initial state is to take the *select* transition, which sets the value of r_2 to 0. This means that, when $i_0 = 50$, there is no observable difference between the two *coin* transitions. There is, however, a difference in how the transitions update the internal data state.

Figure 5.4 shows another segment of the model from Example 5.1.3. Here, depending on which $q_1 \xrightarrow{coin} q_2$ transition is taken, there is an observable difference in the output of the $q_2 \xrightarrow{coin} q_3$ transition. This is clearly undesirable when merging transitions.

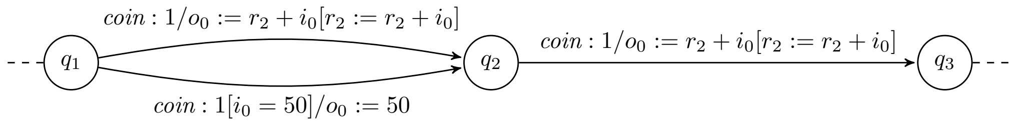

$coin: 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_1
q_2
$coin: 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_3
$coin: 1[i_0 = 50]/o_0 := 50$

Figure 5.4: Another segment of the EFSM from Figure 5.3.

Looking at the examples we have seen so far in this chapter, attempts to determine behavioural equivalence often end up being dependent on the values of particular registers, and are often qualified with restrictions such as "register r must hold value x" or "the value of register r' must be greater than y". This leads us to define *contextual equivalence* which relates possible register values to observable output.

When transitions evaluate guards, outputs, and updates, they do so in a *context.* That is, it is necessary to have a mapping between the variables which occur in an expression and their values. This is not the same as the *data state* of EFSMs, since this only maps *registers* to their values. A *context* also includes the values of the *inputs* that the transition has received. This means that all data states are contexts, but not all contexts are data states.

Definition 29. For an EFSM which works with data of type V, a *context* is a mapping from variables (inputs and registers) to values of type V.

In the Isabelle formalisation of EFSMs from Chapter 4, EFSM inputs are of type `value`. Variables are implemented with the `vname` datatype, so contexts are maps from type `vname` to `value`. In Isabelle, maps are implemented as functions from type `a` to `b option`. Definition 29 is a little more abstract than this as it is designed to work with arbitrary EFSMs as per Definition 21, not just those implemented in Isabelle. Thus, an EFSM could accept different kinds of inputs such as lists, floating-point numbers, or IP addresses in addition to (or instead of) integers and strings. A more general EFSM implementation in Isabelle which is parametrised on the types of input the EFSM expects to receive is desirable future work.

Example 5.2.3. Consider the expression $(3 \times i_0) + r_1$. To evaluate this, we need to know the values of i_0 and r_1. We need a *context.* If we evaluate the expression in the context where i_0 holds value 11 and r_1 holds the value 9, the expression evaluates to $(3 \times 11) + 9 = 42$. Thus, in this context, the expression $(3 \times i_0) + r_1$ is equivalent to the expression 42.

5.2.1 Contextual Constraints

Working at a concrete level is clearly not feasible if we have an infinite input space as, to prove that one expression accounts for another, we may end up with an infinite number of proof obligations. It is therefore preferable to make more general statements such as "in contexts where $r_1 > 50$, transition x is able to account for the behaviour of transition y". Rather than just recording the concrete values of registers as an EFSM executes a trace, we can use the guards and updates of successive transitions to infer sets of *constraints* on their values.

Definition 30. *Contextual constraints* relate variables within contexts and place restrictions on their values. The empty context, in which all inputs and registers are undefined, is expressed as $\{\}$. Variables which do not appear in the constraint set are not in scope and cannot be accessed. If a particular variable v is known to be defined but not restricted in any way, we declare this with an existential quantifier, i.e. $\exists v$.

The reason Definition 30 requires all the variables that are in scope to explicitly appear in the constraint set is to allow us to distinguish between variables that are *unrestricted* and those which are *undefined.* Contextual constraints are closely related to the SMT problem, where a set of constraints are specified, the conjunction of which is either satisfiable or unsatisfiable. A common way of specifying these constraints is using the SMT language [13]. Here, variables must first be declared before constraints over them can be specified. Variables which are not declared are unrecognised and cannot be used in expressions. In our sets of contextual constraints, the existential quantifier serves as an aesthetically acceptable way to declare variables.

Example 5.2.4. If we wish to represent those contexts in which r_1 is greater than 5, we would write $\{r_1 > 5\}$. Here, the only variable that is defined is r_1, which is known to hold a value greater than 5. We do not explicitly need the constraint $\exists r_1$ here since, if r_1 holds a value greater than 5, it must have a defined value.

Example 5.2.5. Consider the *select* transition in Figure 5.5. The transition has no guard, but does have an arity of one. If we take the transition, there must be exactly one input, i_0, in scope. We can express this as $\{\exists i_0\}$. Here, i_0 is the only variable which is defined.

When an EFSM executes a trace, the data state is updated by successive transitions. This means that contexts flow through the EFSM as it processes an execution such that the posterior context of one transition is the anterior context of the next. Applying a specific form of a general technique known as *symbolic execution* [94], we can carry constraints on input and register values around an EFSM without having to execute it with concrete traces. The basic idea here is that, in the initial state, all registers are undefined and there has been no input. If a transition is taken, we know that its guard must have been satisfied, so we can add this information to the context. Transitions may make changes to the data state, which we must also take account of. Each transition will have three contexts during its evaluation:

- The *anterior context* is the valuation of the registers before the transition fires. This context contains only registers since the transition has not yet received any input.
- The *medial context* is the valuation used to evaluate the guard expressions. This includes input values as these are currently in scope.
- The *posterior context* is the valuation after the updates have executed. Like the anterior context, this does not include inputs as they do not persist after the transition has fired.

Example 5.2.6. Consider the EFSM in Figure 5.5. In the initial state, q_0, no input has been received and all registers are undefined. The initial context is empty. The only transition which leaves q_0 is *select.* The *anterior context* is the initial empty context. The *select* transition has no guard but does have an arity of one. This means that there must be exactly one defined input, i_0, in the *medial context* for this transition to fire. We do not know anything about the value of i_0, but we know that it is in scope.

The *posterior context* is calculated from the medial context by applying the update functions of the transition. Here, r_1 is assigned the value of i_0. We do not know the value of i_0 but we know that it is defined, so its value can be assigned to r_1. The *select* transition additionally brings a second register, r_2, into scope by assigning it the literal value zero.

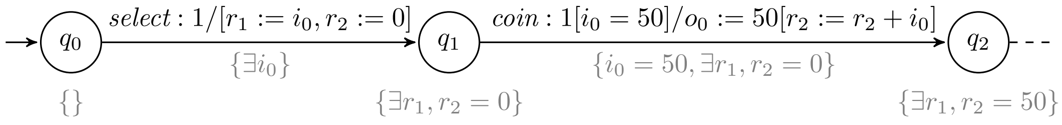

q_0
$select: 1/[r_1 := i_0, r_2 := 0]$
$\{\exists i_0\}$
$\{\}$
q_1
$coin: 1[i_0 = 50]/o_0 := 50[r_2 := r_2 + i_0]$
$\{i_0 = 50, \exists r_1, r_2 = 0\}$
$\{\exists r_1, r_2 = 0\}$
q_2
$\{\exists r_1, r_2 = 50\}$

Figure 5.5: A fragment of an EFSM representing the simple vending machine with contextual constraints annotated in grey.

Let us now consider the *coin* transition. The *anterior* context here is the *posterior* context of the *select* transition we have just taken. The arity of the *coin* transition is one, so we know that exactly one input, i_0, is defined in the medial context. Furthermore, the guard of *coin* ($i_0 = 50$) gives us information about its value which can be added to the medial context. The posterior context is then calculated by applying the transition update functions. Register r_2 is assigned the sum of its anterior value (which we know is zero) and the value of i_0, which we know must be 50. Thus the posterior value of r_2 is 50. The value of r_1 remains unchanged.

The contextual restrictions published in [64] used a slightly different syntax and did not allow pairs of arbitrary expressions to be related, only expressions and literal values. For example the constraint $r_1 > 5$ would have been allowed, but the constraint $r_1 > r_2$ would not. Here, we remove this restriction such that arbitrary expressions can be related. The need for this generalisation is best illustrated with an example.

Example 5.2.7. Consider the EFSM in Figure 5.6. Tracing the contexts to state q_1 leaves transition g with the anterior constraints $\{\exists r_1, \exists r_2\}$. The guard of g is $r_1 > r_2$ so this is added to the constraint set, as it must hold if g is taken. Since g makes no change to the data state, the posterior context also has $r_1 > r_2$.

Transition h is where the problem arises. Here, there is a guard which asserts that $r_1 < r_2$, but we know from the anterior context that this is not true. This leaves us with an *unsatisfiable* medial context for h meaning that it cannot be taken.

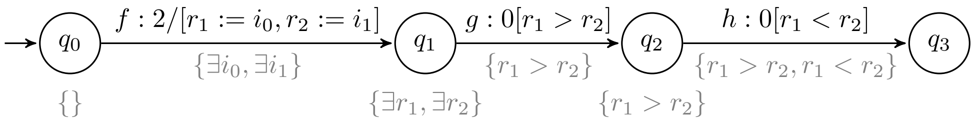

q_0
$f: 2/[r_1 := i_0, r_2 := i_1]$
$\{\exists i_0, \exists i_1\}$
$\{\}$
q_1
$g: 0[r_1 > r_2]$
$\{r_1 > r_2\}$
$\{\exists r_1, \exists r_2\}$
q_2
$h: 0[r_1 < r_2]$
$\{r_1 > r_2, r_1 < r_2\}$
$\{r_1 > r_2\}$
q_3

Figure 5.6: An EFSM in which it is necessary for contexts to relate variables.

Although this example is somewhat Byzantine, it illustrates the need for the ability to relate variables. If we were not able to relate r_1 and r_2 directly, we would not be able to carry the information necessary to infer that transition h can never be taken. Instead, we would only know that r_1 and r_2 existed and were defined.

Having shown several examples in detail, I now present the general procedure for computing the posterior constraint set from a given anterior one. Algorithm 2 describes this process in detail. Line 2 adds the guards of the transition to the set to form the medial context. If this is *consistent*, the update functions can be applied, looking up constraints from the medial context as necessary. Any constraints involving input values are then removed since, as discussed in Chapter 4, inputs to events are instantaneous and do not persist after the firing of transitions.

Algorithm 2 Computing the posterior constraints.

```
1: function POSTERIOR(Transition $t$, AnteriorConstraints $c$)
2:     $medial \leftarrow c \cup t.guards$
3:     if CONSISTENT($medial$) then
4:         return APPLYUPDATES($medial, t.updates$)
5:     else
6:         return FALSE
```

If a constraint set is *consistent,* this means that all the constraints are satisfiable simultaneously. An inconsistent set of constraints on the medial context means that the transition guard is not satisfied and the transition cannot be taken with the given anterior context. For example, the constraint set $\{r_1 > 7, i_0 = 2\}$ is consistent because there exist satisfying assignments of i_0 and r_1. The constraint set $\{r_1 > r_2, r_1 < r_2\}$ from Example 5.2.7 is not consistent, since there is no assignment of r_1 such that it is simultaneously greater than and less than r_2.

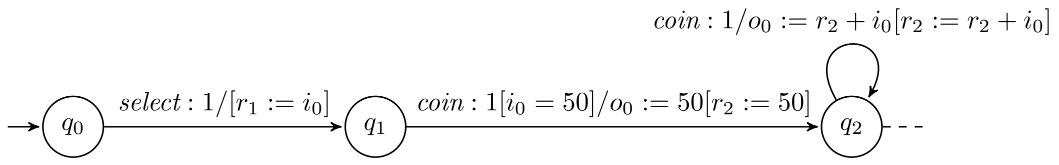
While the examples we have seen so far only include constraints which relate literals and variables, we can relate arbitrarily complicated expressions. For example, if a transition had a guard $[r_1 + i_0 > r_2 + 7]$, this expression would be carried into the medial constraint set as-is. It would not make it into the posterior context, however, as it constrains the input variable i_0 which does not persist after the firing of the transition. The only time information about input variables can be carried into the posterior constraint set is if the medial context contains sufficient information about the value of an input that it can be substituted for.

Example 5.2.8. Consider a transition with guards $[r_1 > i_0, i_0 > 6]$ and no update expressions. Both of these expressions are carried into the medial context but, since both constrain i_0, neither would make it into the posterior context. By transitivity of the "greater than" relation, however, we know that r_1 must be greater than 6. This relation constrains only register r_1, which persists throughout the execution of the EFSM so, in the posterior context, it is known that $r_1 > 6$. This rearrangement of expressions is very important, but means that correctly following the contextual flow through a model without losing any information can be quite difficult if the transitions have complex guard expressions.

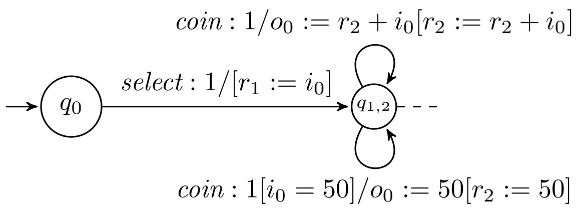
5.3 Subsumption and Generalisation

We want to merge those transitions which we believe to represent the same behaviour into a single representative transition. To merge transitions, we require some notion of behavioural equality and generalisation. The idea of *subsumption*, introduced by [106], provides such a relation for guarded transitions, but that work does not attempt to handle transitions with data update functions. With the help of a running example, this section uses contexts to extend the idea of subsumption to transitions with output and update functions.

Observe the EFSM fragment in Figure 5.7a and note transitions $q_1 \xrightarrow{coin} q_2$ and $q_2 \xrightarrow{coin} q_2$, which will be referred to as c_1 and c_2 respectively. States q_1 and q_2 are now merged into a new state, $q_{1,2}$, which results in a nondeterministic model since the transitions c_1 and c_2 now both leave $q_{1,2}$. When $i_0 = 50$, either of the transitions may be taken, but this nondeterminism is not what we intended when we merged q_1 and q_2. We did this because we believed that c_1 and c_2 represent the same behaviour. To complete the merging process, we need to condense the two representations of the behaviour into a single transition.


$coin: 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select: 1/[r_1 := i_0]$
$coin: 1[i_0 = 50]/o_0 := 50[r_2 := 50]$
q_0
q_1
q_2

(a) A fragment of an EFSM before merging states q_1 and q_2.


$coin: 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select: 1/[r_1 := i_0]$
q_0
$q_{1,2}$
$coin: 1[i_0 = 50]/o_0 := 50[r_2 := 50]$

(b) The same fragment after merging states q_1 and q_2.

Figure 5.7: A fragment of an EFSM before and after merging states q_1 and q_2.

Transitions may not always be compatible for merging, so we need a test to determine this. To represent the same behaviour, transitions must at least have the same label and arity, and produce the same number of outputs. If this is not the case, there is a trivially observable difference in their behaviour. We are then left to consider guards, outputs, and updates.

In [106], one transition is said to subsume another if its guard is *more general.* Applying this principle to the EFSM in Figure 5.7b has c_2 subsuming c_1, as c_2 has no guard where c_1 requires its input to equal 50. This looks promising but the outputs and updates need to be considered too. The principle of subsumption must therefore be extended to take these into account.

5.3.1 Extending Subsumption

Before I present the formal definition of my extended version of subsumption, it is helpful to first establish an intuition of how the relation should behave. Definition 31 gives a breakdown of the three conditions that intuitively must hold for a transition t_2 to subsume another transition t_1. The general idea of subsumption is similar to conformance and extension, introduced in Subsection 2.3.4, the aim being to determine whether one transition implements the behaviour of another, with the possibility of some additional behaviour. I now explore the three conditions in Definition 31 in detail, and will formalise them in the next section to arrive at Definition 32.

Definition 31. Informally, a transition t_2 can be said to *subsume* transition t_1 if

1. When we can take t_1, we can also take t_2.
2. t_1 and t_2 produce equivalent output in cases where it is possible to take either transition.
3. The posterior data state of t_2 is *consistent* with that of t_1.

Condition 1: Weakening the Guard

Recall that the main objective of inference from traces is to take observations of behaviour and create a generalised model that is able to predict how the system might behave when presented with unseen input sequences. To this end, we want to weaken the guards of transitions as much as possible to allow them to respond to more input values. As in [106], the relation which determines the relative strength of guards is logical implication. If a guard g_1 is implied by the guard g_2, then g_1 is *weaker* than g_2.

Example 5.3.1. Consider the guards $i_0 = 3$ and $i_0 < 6$. If the guard $i_0 = 3$ is satisfied, then we know that $i_0 < 6$. Here, the guard $i_0 < 6$ is implied by $i_0 = 3$ so is the weaker of the two guards and would be chosen in preference to $i_0 = 3$ as it is satisfied by more input values.

Condition 2: Maintaining Observational Equivalence

While it is desirable to weaken guards such that they are satisfied by more inputs, it is vital that we do not introduce inconsistencies in the observable behaviour of the model. If there is an observable difference between the behaviour of the two transitions under circumstances where the more restrictive transition may be taken, they cannot be merged.

Example 5.3.2. Consider the transitions $t_1 = t : 1[i_0 < 6]/o_0 := 3$ and $t_2 = t : 1[i_0 = 3]/o_0 := 4$. Transition t_1 has a weaker guard than the transition t_2, but in the case where both transitions can be taken (i.e. when i_0 holds the value 3), the output of the two transitions is not identical, so neither accounts for the behaviour of the other.

Condition 3: Reducing Unspecified Behaviour

Even though data registers are not directly observable — it is not possible to ask "What is the value of register r?" — they may be used as part of output functions so have the potential to affect the observable behaviour of future transitions. It is therefore important that any register updates performed by the subsuming transition are consistent with those performed by the one being subsumed, but what does it mean for update functions to be *consistent*? To solve this, we again borrow ideas from the field of refinement.

One of the main aims of refinement is to reduce unspecified behaviour. When one program *refines* another, its posterior state satisfies stronger conditions. That is, we know more about the posterior state of the subsuming program than the one being subsumed. Using this as part of the criterion for merging transitions allows us to be confident that we haven't introduced a problem with the data state that only manifests later on in the execution of the model.

Example 5.3.3. Consider the transitions $t : 1/[r_1 := 5]$ and $t : 1/[r_1 := i_0]$. These two transitions are equivalent except for how they update register r_1. While we certainly know more about the posterior data state of the first transition than the second, the two update functions are not consistent. When i_0 is not 5, there is a disparity between the two posterior data states which cannot be resolved. This creates the potential for different observable behaviour later on in the execution of the model if r_1 is used as part of an output function.

The problem here is that the guards are too broad. While we know more about the posterior state of the function $r_1 := 5$ than of $r_1 := i_0$, there is the potential for both transitions to accept inputs other than 5. If we instead had $t : 1[i_0 = 5]/[r_1 := 5]$, this would be an *instance* of the second behaviour since, when i_0 holds value 5, r_1 is updated to that value.

Example 5.3.3 shows that we cannot consider update functions in isolation. It is not sufficient to ask "Do we know more about the data state of t_1 than t_2?". The question we need to ask is "*When we can take both transitions,* do we know more about the data state of t_1 than t_2?". This way, we take into account the fact that the subsuming transition needs to account for all of the behaviour that the transition it is subsuming can perform, but also that it *only* needs to account for that behaviour.

5.3.2 Formalising the Definition

We have now established the intuition of how we want the subsumption relation to behave. For transition t_2 to subsume t_1, it must respond to more (or at least the same) inputs. Additionally, in situations where it is possible to take both transitions, their outputs must be identical and t_2 must produce a more tightly constrained posterior context than t_1. It is now time to formalise the definition of *subsumption in context* into a mathematical predicate.

Let us first tackle the weakening of the guard. It is reasonably easy to formalise the statement "when we can take t_1, we can also take t_2" into a logical predicate. To do this, I first define the function canTakeTransition as follows. This returns *true* if a transition t can be taken with the given inputs, i, and register valuation, r. If not, it returns *false*.

$$\textsc{canTakeTransition}(t, i, r) = (\textsc{length}(i) = \textsc{arity}(t) \wedge \textsc{applyGuards}(t, i, r))$$

The first conjunct of canTakeTransition checks that the number of inputs matches the arity of the transition. If we have not received the correct number of inputs, we cannot take the transition. The second conjunct checks that the transition guards are satisfied by the inputs and current data state. The canTakeTransition function is always evaluated in the *anterior context.* That is, the state of the model before a transition has been taken. As discussed in Section 5.2, the anterior context only contains information about the values of registers, so is also a data state. Then, for a given anterior context r, we formalise the first condition in Definition 31 as the following, where i is a list of inputs and r is the current data state.

$$\forall i.\textsc{canTakeTransition}(t_1, i, r) \implies \textsc{canTakeTransition}(t_2, i, r) \tag{5.1}$$

The formalisation of the second condition of Definition 31 also makes use of the canTakeTransition function and is relatively straightforward.

$$\forall i.\textsc{canTakeTransition}(t_1, i, r) \implies \textsc{applyOutputs}(t_1, i, r) = \textsc{applyOutputs}(t_2, i, r) \tag{5.2}$$

The final statement of Definition 31 is somewhat more interesting. In situations where we want to check the subsumption of two transitions, both of which update the same registers, we want the posterior context of the transition with the weaker guard to be more (or at least equally) constrained than that of the transition with the stronger guard in situations where we can take both transitions. We formalise this statement as follows.

$$\forall i.\, \forall P.\, \textsc{canTakeTransition}(t_1, i, r) \implies (\text{P}(\textsc{applyUpdates}(t_2, i, r)) \implies \text{P}(\textsc{applyUpdates}(t_1, i, r))) \quad (5.3)$$

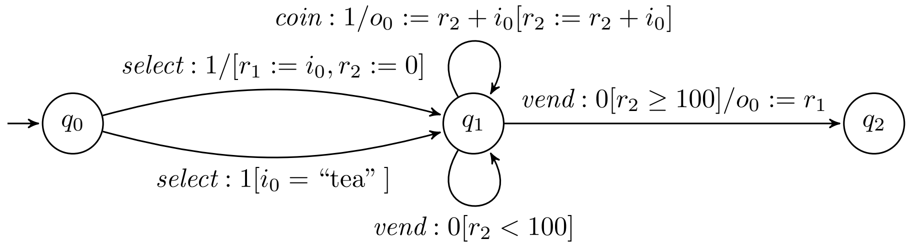
where

P is a predicate which takes a context and returns boolean *true* or *false*.

applyUpdates is a function which takes a transition t, a list of input arguments i, and a data state r, and returns the updated data state according to the updates of t.

In Equation 5.3, P is a predicate which takes a context and returns boolean *true* or *false*, for example, we might define P to be the predicate which returns true if $r_2 = 7$. Thus, in contexts where this holds, P returns *true*. In other contexts, P returns *false*. Because we quantify over all inputs i and all predicated P, this ensures that, for all inputs, every possible posterior context of the transition t_2 is more constrained than that of t_1. For transitions which update exactly the same register(s), the above predicate is an adequate description of the intuition of subsumption, but there is a case where Equation 5.3 does not behave how we want it to.

Example 5.3.4. Consider the EFSM in Figure 5.8, in which we have two nondeterministic *select* transitions. The top transition is a general description of the behaviour of the drinks machine. It accepts any input, assigns it to r_1, and initialises r_2 to zero. The bottom transition is, presumably, a survivor from the original PTA which is yet to be merged. Obviously, we would like the top transition to subsume it, but Equation 5.3 does not allow for this since there exist assignments of P and r where Equation 5.3 does not hold. One such value of P is the function which, for a given context, checks that $r_2 = 0$. For the top transition (t_2), this will be the case, but for the bottom transition t_1, r_2 remains unchanged from the anterior context since it is not explicitly updated. Thus, t_2 cannot subsume t_1 in any context where r_2 is not already zero. Since our anterior context here is the initial context, in which r_2 is undefined, the subsumption cannot go ahead.



$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select : 1/[r_1 := i_0, r_2 := 0]$
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_0
q_1
q_2
$select : 1[i_0 =$ “tea” $]$
$vend : 0[r_2 < 100]$

Figure 5.8: An EFSM with two nondeterministic *select* transitions.

The argument here for why we want the top transition to subsume the bottom one comes again from refinement. Both *select* transitions originate from the initial state, so the anterior context will be empty. The posterior context of the bottom transition is also empty as it makes no changes to the data state. Thus, the values of r_1 and r_2 are undefined. The posterior constraints of the top transition, however, are $\{\exists r_1, r_2 = 0\}$. Here, r_1 and r_2 are both defined. Thus, the top transition has less unspecified behaviour.

To account for the fact that we would like posterior contexts in which a particular register is undefined to be subsumed by posterior contexts in which that register is defined, we modify Equation 5.3 to make it the following.

$$\begin{aligned}&\forall i.\, \forall P.\, \forall r'.\, \textsc{canTakeTransition}(t_1, i, r) \implies \\ &\quad ((\mathrm{P}(\textsc{applyUpdates}(t_2, i, r)(r')) \implies \mathrm{P}(\textsc{applyUpdates}(t_1, i, r)(r'))) \vee \\ &\qquad \textsc{applyUpdates}(t_1, i, r)(r') = \textsc{undef})\end{aligned} \tag{5.4}$$

where

P is a predicate which takes a value option and returns boolean *true* or *false*.

r' is a register index.

UNDEF represents register r' being undefined.

Equation 5.4 captures precisely the intuitive notion expressed in condition 3 of Definition 31, namely that the posterior data state of t_2 must be *consistent* with that of t_1. That means that for all registers r' and predicates P, if the value of r' in the posterior data state of t_2 satisfies P then the value of r' in the posterior data state of t_1 must also satisfy P. The second clause of the disjunction captures the intuition from 5.3.4 that we also want to reduce unspecified behaviour by allowing previously undefined registers to be updated by the subsuming transition.

If we think of contexts as sets of (register, value) pairs, we can think of 5.4 as saying that the posterior context of t_1 is a subset of the posterior context of t_2. Indeed, thinking of 5.4 in this way makes it very apparent why 5.3 is not what we want: this only holds if the two posterior contexts are equivalent. The phrasing in 5.4, however, is much more compatible with my Isabelle formalisation of EFSMs from Chapter 4 which is why I use it here. Since I formalised contexts and data states as finite functions, there is no easy way to *subset* over them. It is possible to explicitly convert finite functions to lists of pairs with the `finfun_to_list` function, but proofs involving this function are extremely arduous.

Combining Equations 5.1, 5.2, and 5.4, as well as the fact that transitions clearly must have the same label and arity to account for each other's behaviour, we arrive at the formal definition of subsumption shown in Definition 32.

Definition 32. Transition t_2 *subsumes* t_1 in anterior context r if

$$\begin{aligned}&\textsc{label}(t_1) = \textsc{label}(t_2) \wedge \textsc{arity}(t_1) = \textsc{arity}(t_2) \wedge \\ &\forall i.\, \textsc{canTakeTransition}(t_1, i, r) \implies \textsc{canTakeTransition}(t_2, i, r) \wedge \\ &\forall i.\, \textsc{canTakeTransition}(t_1, i, r) \implies \\ &\quad \textsc{applyOutputs}(t_1, i, r) = \textsc{applyOutputs}(t_2, i, r) \wedge \\ &\forall i.\, \forall P.\, \forall r'.\, \textsc{canTakeTransition}(t_1, i, r) \implies \\ &\quad ((\mathrm{P}(\textsc{applyUpdates}(t_2, i, r)(r')) \implies \mathrm{P}(\textsc{applyUpdates}(t_1, i, r)(r'))) \vee \\ &\qquad \textsc{applyUpdates}(t_1, i, r)(r') = \textsc{undef})\end{aligned}$$

The value UNDEF, used in Equation 5.4, represents a register being undefined. For Equation 5.4 to work in Definition 32, it is critical that UNDEF operates only at the meta level and that we cannot check undefinedness as part of a transition guard. That is to say that we cannot have a guard $\textsc{null}(r)$ which returns *true* if register r is undefined and *false* if it is defined.

While many languages (such as Java and Python) do have such a construct, we cannot allow it here as it would change the semantics of UNDEF from truly undefined unspecified behaviour to being just another concrete value we can check for.

Example 5.3.5. Consider the EFSM in Figure 5.9. There are two nondeterministic $q_0 \xrightarrow{f} q_1$ transitions. According to Definition 32, the top transition should subsume the bottom one, however, we do not want this to be the case here as the two transitions produce different posterior data states, only one of which allows the $q_1 \xrightarrow{g} q_2$ transition to be taken. The top transition assigns the value 0 to r_1 such that it is no longer undefined. This means that the guard of g is not satisfied by the resulting context. Definition 32 does not take proper account of this, however, since it gives UNDEF a special status.

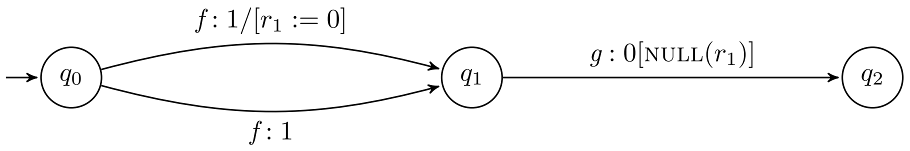


Figure 5.9: An example of an EFSM in which having a NULL(r) guard breaks subsumption.

The NULL(r) guard effectively brings the value UNDEF down to the object level, thus making it a value in and of itself. This breaks the law from Section 4.2 that a register's value cannot be accessed before it has been defined and renders undefinedness as a concept somewhat moot. Example 5.3.5 is an illustration of this. If it is necessary to check a register's undefinedness, for example to ensure that it can only be written to if it does not already have a value, this can be done by carefully setting up the control flow states such that this holds without the need for an explicit NULL guard.

5.3.3 Subsumption as a Preorder

Since subsumption captures the idea of generality, it makes sense for it to be some kind of *order relation.* Ideally, we would like subsumption to be a partial order, satisfying the properties of reflexivity, transitivity, and antisymmetry. Using the existing infrastructure from Section 4.7 as a basis, it is relatively easy to formalise the subsumption relation in Isabelle and prove that it is a *preorder* relation.

```
lemma subsumes-reflexive: subsumes t c t

lemma subsumes-transitive:
  assumes p1: subsumes t1 c t2
     and p2: subsumes t2 c t3
  shows subsumes t1 c t3
```

This means that, in any context, a transition always subsumes itself and, if a transition t_1 subsumes another transition t_2 which in turn subsumes a third transition t_3, then t_1 also subsumes t_3. Thinking about subsumption as *accounting for behaviour,* this is intuitively what we want. Clearly a transition must be able to account for its own behaviour, and it would be rather strange if the relation was not transitive. This would mean that the bottom transition t_3 could exhibit some behaviour which the top transition t_1 could not.

For subsumption to be a partial order relation, we also need antisymmetry, but alas this property is not true so subsumption is only a preorder. For subsumption to be antisymmetric, the property $\textsc{subsumes}(t_1, c, t_2) \implies \textsc{subsumes}(t_2, c, t_1) \implies t_1 = t_2$ must hold. This is untrue for two reasons. The first of these is implementational; the second is more fundamental.

For a pair of transitions to be equal, their labels, arities, guards, outputs, and updates must all be identical. Because transitions in Isabelle make use of deeply embedded guard and update expressions, semantically identical expressions can be syntactically different. This means that they are not equal. For example, the expressions $i_1 + 1$ and $1 + i_1$ are semantically identical, but their representations in Isabelle are not. The term `Plus (V (I 1)) (L (Num 1))` is clearly different to the term `Plus (L (Num 1)) (V (I 1))`. This means that, although transitions might be *semantically* identical, they are not equal in the eyes of Isabelle.

This problem could be solved by defining a semantic equality operator on transitions, but there is a second reason why transitivity does not hold which is more deeply rooted in the definition of subsumption. The latter conjuncts of Definition 32 are all predicated on the fact that either t_1 or t_2 can be taken in the given context. For example, the third conjunct asserts that *if we can take t_1, then the outputs of t_1 and t_2 are identical.* This is necessary for the subsumption relation to behave as expected — we only require outputs to be identical when both transitions can be taken — but means that antisymmetry is also dependent on the fact that we can take t_1 in the current context, so cannot hold in its own right.

5.4 Direct Subsumption

When merging EFSM transitions, one must *account for* the behaviour of the other. The *subsumption in context* relation formalises the intuition that, in certain contexts, a transition t_2 reproduces the behaviour of, and updates the data state in a manner consistent with, another transition t_1, meaning that t_2 can be used in place of t_1 with no observable difference in behaviour. The subsumption in context relation requires us to supply a context in which to test subsumption, but there is a problem when we try to apply this to inference: Which context should we use? This section answers this question and incorporates the subsumption in context relation into a new relation *direct subsumption* which serves as the transition merging criterion we are looking to formulate in this chapter.

Example 5.4.1. Recall the EFSM fragments from Example 5.1.1. Since we believe that the two *vend* transitions represent the same behaviour, we would like to merge them into a single transition. To do this, we ask if one transition *accounts for* the behaviour of the other such that it can be deleted. This means that in every situation where we could have taken t_1 in $\mathcal{M}_1$, we should now be able to take t_2 in $\mathcal{M}_2$ with no observable difference in behaviour.

Here, we know that t_2 accounts for the behaviour of t_1 if r_1 holds the value “tea”, but this glosses over the detail somewhat as it is unlikely that r_1 will *always* hold the value “tea” in state q_{ac}. Surely it could cause a problem if we are in state q_{ac} and r_1 does not hold the value “tea” ? Analysing the situation in more detail, we can see that t_1 can only be taken in $\mathcal{M}_1$ from state q_a. We therefore only need t_2 to account for the behaviour of t_1 *in situations where it could be taken in $\mathcal{M}_1$*. That is, we only need t_2 to subsume t_1 in contexts which are obtainable in state q_a. This means that traces which got us to q_a in $\mathcal{M}_1$ must, when run in $\mathcal{M}_2$, produce contexts in which t_2 subsumes t_1, i.e. contexts in which $r_1 =$ “tea” . If this is the case, we say that t_2 *directly subsumes* t_1.

5.4.1 Formal Definition

In Example 5.4.1, we want to use the direct subsumption relation as a means of determining whether it is safe to merge a pair of nondeterministic transitions. Since the nondeterminism has arisen as a consequence of merging states, if we are to use t_2 in place of t_1, it must account for all of the behaviour which t_1 could exhibit before the merge. This means that t_2 must subsume t_1 all contexts which could be obtained in its origin state.

Definition 33. To define obtainability, we require an additional function, OBTAINS, which returns *true* if a given trace obtains context r' in state s' when executed in EFSM e.

$$\text{base}: \; t = [] \implies \textsc{obtains}(s', r', e, s', r', t)$$

$$\text{step}: \; \exists(s'', tr) \in \textsc{possibleSteps}(e, s', r', l, i).$$
$$\textsc{obtains}(s', r', e, s'', \textsc{evaluateUpdates}(tr, i, r'), t) \implies$$
$$\textsc{obtains}(s', r', e, s, r, (l, i)\#t)$$

Thus, a context c is *obtainable* in state s in EFSM $e = (Q, q_0, T)$ if $\exists t.\textsc{obtains}(s, c, e, q_0, \langle\rangle, t)$, i.e. if there exists a trace t which, when run in EFSM e starting from the initial state q_0 and the empty context $\langle\rangle$, leaves the model in state s with context c.

Having defined obtainability, we can then define direct subsumption on top of this. Definition 34 incorporates subsumption into a relation which can be used to determine if it is safe to merge a pair of transitions. It is this which allows us to take the subsumption relation from Definition 32 and use it in the actual inference process. The *directly subsumes* relation was one of the main contributions of my original work published in [63], although Definition 32 is a much more elegant phrasing.

Definition 34. Transition t_2, originating from state s_1 in an EFSM $e_1 = (P, p_0, T)$, *directly subsumes* transition t_2 which originates from state s_2 in EFSM $e_2 = (Q, q_0, T')$ if $\forall c_1\, c_2\, t.\, (\textsc{obtains}(s_1, c_1, e_1, p_0, \langle\rangle, t) \wedge \textsc{obtains}(s_2, c_2, e_2, q_0, \langle\rangle, t)) \implies \textsc{subsumes}(t_1, c_2, t_2)$.

Example 5.4.2. Recall the EFSM shown in Figure 4.5 (reproduced below for convenience), which is observationally equivalent to the model in Figure 1.5. Here, states q_1 and q_2 both have outgoing *coin* and *vend* transitions, so it would make sense for the inference process to attempt to merge these states. Since the two *coin* transitions are identical and end up with the same origin and destination after the merge, they are not distinct so one can be trivially deleted. The merge does, however, introduce nondeterminism between the *vend* transitions. Indeed the transition $q_1 \xrightarrow{vend:0} q_1$ is nondeterministic with both of the outgoing *vend* transitions from q_2. To resolve this, we must use the direct subsumption relation to investigate which pair should be merged.

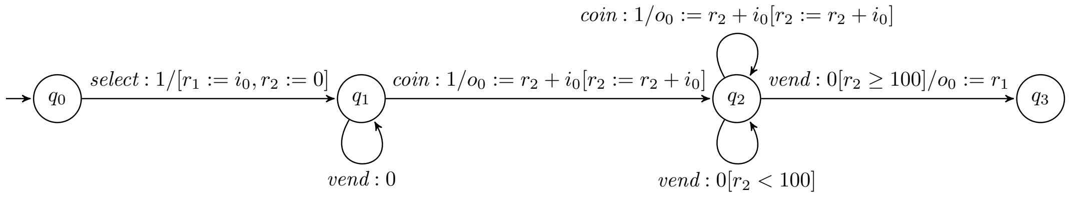

q_0
$select : 1/[r_1 := i_0, r_2 := 0]$
q_1
$vend : 0$
$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_2
$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$vend : 0[r_2 < 100]$
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_3

Figure 4.5: An EFSM which is trace equivalent to the one in Figure 1.5.

Looking at the transitions, we can see that there is clearly no way to merge *vend* : 0 with *vend* : $0[r_2 \geq 100]/o_0 := r_1$ since they produce different numbers of outputs. By our definition of subsumption in Definition 32, neither can subsume the other in any context, so we clearly cannot merge this pair. Let us then consider the other outgoing *vend* transition, *vend* : $0[r_2 < 100]$. This transition produces the correct output (nothing), but does it directly subsume *vend* : 0?

Definition 34 requires that every trace which reaches q_1 in Figure 4.5, when run in the EFSM after the merge, produces a context c_2 in which *vend* : $0[r_2 < 100]$ subsumes *vend* : 0. Looking at Figure 4.5, we can see that this is the case since the only traces which get the model to q_1 leave r_2 holding the value zero. We first *select* a drink, which initialises r_2 to zero, and then can press *vend* any number of times, which does not change the value. The only other action we can take from q_1 is *coin*, which does change the value of r_2 but moves the model into state q_2, from which we cannot return to q_1. Armed with this information, it is then sufficient to prove that *vend* : $0[r_2 < 100]$ subsumes *vend* : 0 in all contexts where r_2 holds the value zero. According to Definition 32, the subsumption can go through since they transitions have the same label and arity, both have guards which are satisfied when $r_2 = 0$, and neither produces any outputs or has any update functions. We can therefore use *vend* : $0[r_2 < 100]$ in place of *vend* : 0 without affecting the behaviour of the model.

The story is not yet complete, though, since direct subsumption also holds in the other direction. That is, *vend* : 0 directly subsumes *vend* : $0[r_2 < 100]$. This should mean that we could use it in place of *vend* : $0[r_2 < 100]$, but this is not the case. If we do this, there is still nondeterminism between *vend* : 0 and the other *vend* transition, *vend* : $0[r_2 \geq 100]/o_0 := r_1$, but these transitions cannot be merged since they produce different numbers of outputs. Thus, we must take *vend* : $0[r_2 < 100]$ over *vend* : 0. We are therefore still in need of a means of determining which transition to use in circumstances such as this, when both transitions subsume each other. This is resolved in the next chapter.

5.4.2 Key Properties

Since we have already formalised EFSMs and *subsumption in context* in Isabelle, it makes sense to formalise *direct subsumption* as well. The formalisation of both obtainability in Definition 33 and direct subsumption in Definition 34 is pretty straightforward, with the definitions appearing more or less as above. We can then prove some key properties of the *directly subsumes* relation.

Direct Subsumption as a Preorder

Having already proved this for subsumption in context, it makes sense to prove it for direct subsumption. As before, it would be nice to also have antisymmetry, but we cannot have this for the same reasons as before.

```
lemma directly-subsumes-reflexive: directly-subsumes e1 e2 s1 s2 t t
```

```
lemma directly-subsumes-transitive:
  assumes p1: directly-subsumes e1 e2 s1 s2 t1 t2
    and p2: directly-subsumes e1 e2 s1 s2 t2 t3
  shows directly-subsumes e1 e2 s1 s2 t1 t3
```

Direct Subsumption in Certain Contexts

In Example 5.4.2, the technique I used to prove direct subsumption was to first prove that all traces which reached state q_1 in Figure 4.5 obtained a context in which r_2 held the value zero when run in the EFSM after the merge. This can be generalised into the below lemma, which uses Isabelle's meta-implication construct $\Longrightarrow$ to break direct subsumption proofs into two subgoals. Meta-implication behaves similarly to logical implication and, in most cases, the two can be used equivalently, but the real meaning is closer to logical "semantically entails" $(x \models y)$, i.e. "*given that* we have x, we also have y" rather than the "if x then y" semantics of implication.

lemma *direct-subsumption*:
$(\bigwedge t\ c1\ c2.\ obtains\ s1\ c1\ e1\ 0\ <>\ t \Longrightarrow obtains\ s2\ c2\ e2\ 0\ <>\ t \Longrightarrow f\ c2) \Longrightarrow$
$(\bigwedge c.\ f\ c \Longrightarrow subsumes\ t1\ c\ t2) \Longrightarrow$
$directly\text{-}subsumes\ e1\ e2\ s1\ s2\ t1\ t2$

The first subgoal is to prove that all traces which reach a given state in the pre-merge model produce contexts which satisfy a certain condition, f, when run in the post-merge model. This must usually be proved by induction on traces over the two models. The predicate f must be determined on a case by case basis and supplied in the application of the rule. The second subgoal is to prove that, for all contexts which satisfy f, transition t_1 subsumes t_2. This is usually relatively straightforward if f is specific enough.

Subsumption in All Contexts

If we can prove that one transition subsumes the other in all contexts, this is sufficient to prove direct subsumption. This is a useful lemma as it allows us to skip the first subgoal of the above rule, meaning that we do not have to do induction on traces to prove direct subsumption.

lemma *subsumes-in-all-contexts-directly-subsumes*:
$(\bigwedge c.\ subsumes\ t2\ c\ t1) \Longrightarrow directly\text{-}subsumes\ e1\ e2\ s\ s'\ t2\ t1$

No Direct Subsumption

Equally important to proving that one transition directly subsumes another is proving that a transition does *not* directly subsume another. For this, we must show that there is a trace which gets the pre-merge model to the right state which, when run in the post-merge model, obtains a context in which subsumption cannot occur. This is usually proven by induction on traces.

lemma *visits-and-not-subsumes*:
$(\exists\, c1\ c2\ t.\ obtains\ s1\ c1\ e1\ 0\ <>\ t \wedge obtains\ s2\ c2\ e2\ 0\ <>\ t \wedge \neg\ subsumes\ t1\ c2\ t2) \Longrightarrow$
$\neg\ directly\text{-}subsumes\ e1\ e2\ s1\ s2\ t1\ t2$

Somewhat frustratingly, we cannot bypass an inductive proof here by proving that the relevant transition does not subsume its other in any context. While this would be very useful, it is only relevant if the origin state of our transition is reachable. If the origin state is not reachable, the transition can never execute, meaning that we can replace it with any other transition (or even remove it entirely) without affecting the observable behaviour of the model. The definition of direct subsumption I published in [63] got around this by imposing an additional condition that there must exist a context in which subsumption could occur, but this complicated certain proofs so I have removed it here. In inference, all states are reachable by construction, so the removal of this condition does not affect the models we infer.

5.5 Analysing System Properties

Having created an EFSM model of a system, it is possible (and indeed *necessary*) to use contexts to analyse data flow and help prove certain properties. For example, with the drinks machine, we want to guarantee that customers will always receive the drink they originally selected. Another desirable property, for the proprietors at least, is that customers only receive their drinks if they have inserted enough money. The analysis and verification of properties of models is the subject of Chapter 9, but it is helpful to give an informal example here to illustrate the use of contexts.

Example 5.5.1. Recall our simple drinks machine from Figure 1.5 (reproduced below for convenience). Looking only at the labels, as would be provided by a classical FSM, it appears that we can go straight from q_1 to q_2 without inserting any coins. The trace $\langle \mathit{select}(\text{“tea”}), \mathit{vend}()/[\text{“tea”}]\rangle$ seems like it might be valid, meaning that a user could get their drink for free. Contexts help to show that this is not a valid trace of the model.

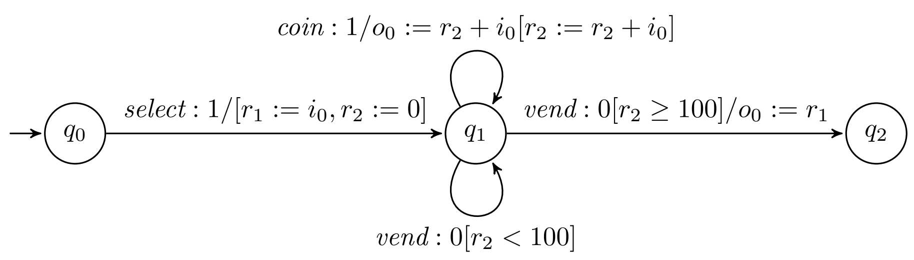

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_0
$select : 1/[r_1 := i_0, r_2 := 0]$
q_1
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_2
$vend : 0[r_2 < 100]$

Figure 1.5: An EFSM model of the drinks machine.

The only way to obtain a drink from the machine is to fire the $q_1 \xrightarrow{vend} q_2$ transition. The only way to reach q_1 from the initial state is via the *select* transition. This transition produces posterior constraints $\{\exists r_1, r_2 = 0\}$. Triggering *vend* with an anterior context in which $r_2 = 0$ only allows $q_1 \xrightarrow{vend} q_1$ to fire, since r_2 holds value zero which is less than 100. This transition does not produce any output so the user does not obtain their drink, meaning that the above trace is not accepted by the model.

Transition $q_1 \xrightarrow{vend} q_2$ can only be taken when r_2 holds a value greater than or equal to 100. The only transition from q_1 with an update function with the potential to increase the value of r_2 is the *coin* transition. This means that the customer must insert at least one coin to receive their drink.

The exact proof strategy varies depending on the property being proven, but the main point here is that we cannot rely solely on control flow information. If we wish to analyse data-flow properties, we must also take into account the values of registers. In other words, the validity of many EFSM properties is *context-dependent.* In the case of Example 5.5.1, the guard of the *vend* transition with the desired output cannot be satisfied with an anterior context in which the value of r_2 is less than 100.

As part of such proofs, we many need to analyse transition update functions to see if any have the potential to affect variables of interest in desired (or undesired) ways. In Example 5.5.1, if we wish to take the *vend* transition, the variable of interest is r_2 and needs to be increased. The *coin* transition has no guard, so may be taken with any anterior context, and produces a posterior context with r_2 incremented by the value of the input. Assuming that coins have a positive value, this increases the value of r_2. The destination state is equal to the origin, so the transition may be taken again if the input value was insufficient.

We also need contexts to prove observational equivalence between EFSM models. We have seen in Subsection 4.7.9 that this is defined inductively on executions, so proofs often proceed by induction. The general strategy is to start both models off in their initial states and explore the various possible paths through the models. The proof is complete when all possible paths have been explored. Because the definition of observational equivalence in Subsection 4.7.9 effectively executes the two models in parallel, it must take their respective data states into account. To prove equivalence for arbitrary traces, it is necessary to generalise beyond concrete contexts and, instead, examine the restrictions on the possible values of important registers. Note that inductive proofs only allow us to prove properties involving finite traces. To prove properties over infinite traces, we need to use *coinduction*, which is introduced in Chapter 9.

Example 5.5.2. Recall the EFSM from Figure 4.5, which is observationally equivalent to the model in Figure 1.5. The use of contexts is vital to prove equivalence of the two models. Let us refer to the EFSM in Figure 1.5 as M_1 and the one in Figure 4.5 as M_2. The proof proceeds by induction on executions, starting both models off in their initial states. From our definition in Subsection 4.7.9, we can see that any two models are executionally equivalent on the empty execution. We therefore only have the step case to prove. That is, for an arbitrary execution x, if the two models are observationally equivalent for that execution then they are observationally equivalent for an arbitrary action a_1 prefixed onto x.

From the initial states, both models have an identical outgoing *select* transition. We therefore have two cases of interest: either a_1 is a *select* action, or it is not. If a_1 is *not* a select action, both models deadlock so are equivalent henceforth. If a_1 is a *select* action, both models move into their respective q_1 states and produce posterior constraints $\{\exists r_1, r_2 = 0\}$.

The next stage is inductively prove observational equivalence from this point. Again, we only need to prove the step case. There are three cases to consider: either our prefixed action a_2 is *coin*, *vend*, or neither. In the latter case, both models deadlock so are equivalent. If we have a *vend* action, both models process this in a similar way. Recall that our anterior context is the posterior context of the *select* action we have just performed. That is $\{\exists r_1, r_2 = 0\}$. In M_1, we must take the $q_1 \xrightarrow{vend} q_1$ transition, since $r_2 = 0$, which is less than 100. This leaves the control flow and data states unchanged. In M_2, we also take the $q_1 \xrightarrow{vend} q_1$ transition, which has no guard and leaves both control flow and data state unchanged.

Let us now imagine that our prefixed action a_2 is a *coin* action. From q_1, both machines can do an unguarded *coin* transition to produce the constraints $\{\exists r_1, \exists r_2\}$. This happens because there are no guards to restrict the value of the input. It could therefore be anything. All we know is that r_2 must exist in the posterior context. After this, M_1 is in state q_2 and M_2 is in state q_1. We must now prove executional equivalence from here. It is at this point that contexts begin to show their worth.

Both models have three outgoing transitions: one *coin* and two *vend,* so we have the same three top-level cases. As before, if our prefixed action, which we shall call a_3, is neither a *coin* action nor a *vend* action, both models deadlock so are equivalent. If a_3 is a *coin* action, this is handled similarly to the previous step, with the difference that neither model changes its control state. While both models update their *data state,* there is no change to the posterior *constraints.* This means that this case is effectively discharged.

Finally, let us consider the *vend* action. Here, there are two subgoals we must consider, which come from the guards on the two outgoing *vend* transitions. Let us first consider contexts in which $r_2 < 100$. In this case, the *vend* transition in both cases leaves the state unchanged but produces posterior constraints $\{\exists r_1, r_2 < 100\}$. Since our inductive hypothesis involves the constraints $\{\exists r_1, \exists r_2\}$, this subgoal is discharged as our posterior context is more restricted than the one in the inductive hypothesis. In the other case, where $r_2 \geq 100$, the *vend* transition which outputs the selected drink may be taken. This takes both models into a state from which there are no outgoing transitions. Thus, both models deadlock for any non-empty trace so are equivalent.

The proofs for examples 5.5.1 and 5.5.2 have both been formalised in Isabelle [66] and are just some of the ways contexts can be used to prove properties of systems. A more detailed explanation of how properties of EFSMs can be verified is presented in Chapter 9.

5.6 Conclusion

The main contributions of this chapter were the *subsumption in context* and *direct subsumption* relations. I began by introducing contexts, the mappings from variables to values under which guards, outputs, and updates are evaluated. I then introduced a scheme to record constraints on possible data values at different points during the execution of an EFSM model, and used this to extend the concept of subsumption, originally presented in [106] for guarded transitions, to EFSM transitions which include data update functions. I used this as a foundation for the *direct subsumption* relation, which is used in the next chapter to fully realise a technique to infer EFSM models with output and update functions from black-box traces. Finally, I briefly showed how contexts are used to prove certain properties of EFSM models. This is an important part of Chapter 9, which discusses how *temporal* properties of EFSMs can be analysed and proven.

Chapter 6

EFSM Inference from Traces

The previous chapter introduced contexts and presented the *subsumption in context* and *direct subsumption* relations as a means of determining whether one transition is able to account for the behaviour of another. This relation is essential to EFSM inference, but there is more to the inference challenge. In this chapter, I present a complete approach for inferring EFSM models, including functions to compute outputs and modify the internal data state, from black-box traces which only contain information visible to an external observer of the system. This chapter is based on my original work presented in [63], the primary contributions being the following.

- A technique which uses black-box traces to infer EFSM models which explicitly relate input and output values.
- A prototype tool which implements this technique.

In addition to these technical contributions, this chapter has the following key discussion points.

- The scoring of EFSM state pairs to determine their compatibility for merging.
- The resolution of nondeterminism caused by merging states with similar transitions.
- The practical compromises which must be made in order to implement the state merging algorithm as a usable tool.

6.1 Introduction

Recall that our simple vending machine produces traces like those in Figure 6.1. The aim is to use traces like these to infer a model which is able to predict how the system might behave when faced with new traces. Previous work on EFSM inference [106, 152] focusses on establishing transition guards that aggregate the observed data values. While this is a valuable contribution, the models inferred by these techniques fail to capture the fact that input *determines* subsequent output. The ability to model this is particularly desirable because it allows us to *predict* system behaviour, rather than simply *monitor* it.

$$\langle \mathit{select}(\text{“tea”}), \mathit{coin}(50)/[50], \mathit{coin}(50)/[100], \mathit{vend}()/[\text{“tea”}]\rangle$$
$$\langle \mathit{select}(\text{“tea”}), \mathit{coin}(100)/[100], \mathit{vend}()/[\text{“tea”}]\rangle$$
$$\langle \mathit{select}(\text{“coffee”}), \mathit{coin}(50)/[50], \mathit{coin}(50)/[100], \mathit{vend}()/[\text{“coffee”}]\rangle$$

Figure 6.1: Some sample traces of the simple vending machine.

What we would really like to do is use the traces in Figure 6.1 to infer a model like the one in Figure 6.2. Here, a register r_1 records the selected drink, and a second register r_2 keeps track of the amount of money inserted so far. Transitions have *update functions* which specify how each register is modified, such that their values at each stage of execution can be computed.

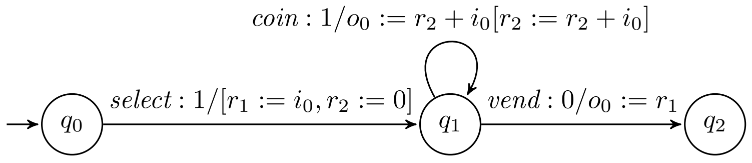

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select : 1/[r_1 := i_0, r_2 := 0]$
$vend : 0/o_0 := r_1$
q_0
q_1
q_2

Figure 6.2: The ideal EFSM representing the traces in Figure 6.1.

The remainder of this chapter is structured as follows. Section 6.2 presents my state merging technique for EFSMs. Section 6.3 shows how we can make use of *heuristics* to generalise transition behaviour at merge time and discusses the limitations of this approach. Finally, in Section 6.4, I discuss how I transformed my technique into an executable tool, and discuss the compromises and optimisations I had to make to achieve this.

6.2 Extending the Inference Process

In this section, I present my technique to infer EFSM models from traces. The process follows the same basic structure as Algorithm 1, the classical FSM inference algorithm presented in Section 3.3. First, a PTA is built from the observed traces, and then states are iteratively merged to form a smaller model. As with classical inference, we want to merge states in the *model* which we believe represent the same state in the underlying *program.* Algorithm 3 shows an outline of the technique.

Algorithm 3 The top level inference process.

```
function LEARN($l$, $scoringMetric$)
    return INFER(MAKEPTA($l$), $scoringMetric$)
function INFER($efsm$, $scoringMetric$)
    switch INFERENCESTEP($efsm$, SCOREMERGES($efsm$, $scoringMetric$)) do
        case None
            return $efsm$
        case Some $new$
            return INFER($new$, $scoringMetric$)
function INFERENCESTEP($e$, $merges$)
    switch $merges$ do
        case []
            return None
        case $((s_1, s_2)\#t)$
            $e' =$ MERGESTATES($s_1, s_2, e$)
            switch RESOLVENONDETERMINISM(NONDETPAIRS($e'$), $e, e'$) do
                case Some $new$
                    return Some $new$
                case None
                    return INFERENCESTEP($e, t$)
```

There are two main challenges to address here. Firstly, because EFSM transitions are not simply atomic actions, duplicated behaviours cannot be resolved into a single transition by simply merging destination states, as it can in classical FSM inference. I address this in Subsection 6.2.3. Secondly, as a consequence of this, it is possible for attempts to resolve the nondeterminism introduced by merging states to fail, meaning that two states which initially seemed compatible cannot actually be merged. This is not the case in classical FSM inference. I tackle this in Subsection 6.2.4.

6.2.1 PTA Construction

The first step is to construct a PTA from the observed traces in the same way as for classical FSM inference. Beginning with the empty EFSM, we iteratively attempt to walk each observed trace in the model. When we reach a point where there is no available transition, one is added. For classical FSMs, this is simply an atomic label. EFSMs deal with data, so we need to add guards which test for the observed input values and outputs which produce the observed values. For example, the event *coin*(50)/[100] would cause the transition $coin : 1[i_0 = 50]/o_0 := 100$ to be added to the model. The event label is *coin*. It takes one input, which must be equal to the observed input value of 50, and produces the literal output 100. The PTA representing the traces in Figure 6.1 is shown in Figure 6.3.

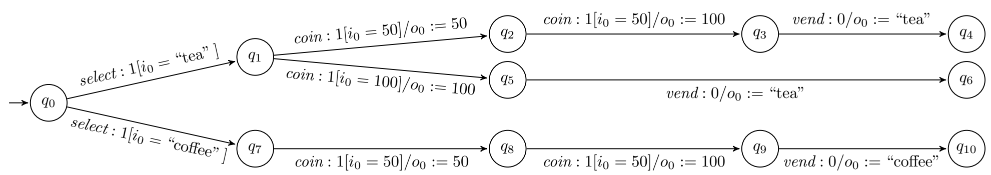

q_0
$select : 1[i_0 = \text{“tea”}]$
q_1
$coin : 1[i_0 = 50]/o_0 := 50$
q_2
$coin : 1[i_0 = 50]/o_0 := 100$
q_3
$vend : 0/o_0 := \text{“tea”}$
q_4
$coin : 1[i_0 = 100]/o_0 := 100$
q_5
$vend : 0/o_0 := \text{“tea”}$
q_6
$select : 1[i_0 = \text{“coffee”}]$
q_7
$coin : 1[i_0 = 50]/o_0 := 50$
q_8
$coin : 1[i_0 = 50]/o_0 := 100$
q_9
$vend : 0/o_0 := \text{“coffee”}$
q_{10}

Figure 6.3: The PTA representing the traces in Figure 6.1.

A key aspect that we must be aware of here is that, because our traces have outputs, our PTA is not guaranteed to be deterministic. For example, consider the traces ⟨a()/[1], b()/[1]⟩ and ⟨a()/[1], b()/[2]⟩. Both traces share the prefix ⟨a()/[1]⟩, after which we may do a *b* action to receive an output of either 1 or 2. A PTA built from these traces would be nondeterministic. This is not to be confused with the situation in Figure 6.3, in which calling the *coin* action with 50 as an input can produce either 50 or 100 as an output. Here, the PTA is deterministic since the respective actions occur after different prefixes.

In general, we do not end up with a nondeterministic PTA unless the underlying system is also nondeterministic. This work makes the assumption that the systems we are trying to infer models of are deterministic, so no explicit attempt to handle a nondeterministic PTA is made.

6.2.2 Scoring State Merges

As discussed in Subsection 3.4.2, a *scoring function* is used to determine and order potential state merges. The SCOREMERGES function in Algorithm 3 takes an EFSM and a scoring metric and, for each pair of states, applies the scoring metric to produce a numeric value representing how likely it is that two states represent the same program state. This results in a sorted list of triples of the form $(score, s_1, s_2)$, where a high score represents a strong merge candidate.

A scoring metric takes two states and returns a numerical value representing how likely it is that they represent the same program state. For classical FSMs, this is often based on how many outgoing transitions the two states have in common. For EFSMs, determining common behaviour is more complicated. In addition to an atomic label, transitions also have an arity, guards, outputs, and updates, all of which may be taken into account when determining how similar a pair of transitions are. Additionally, as explained in the previous chapter, transitions may be able to account for each other's behaviour without being exactly identical. We want to be able to take this into account when scoring potential state merges.

The most naive way to score EFSM transitions would be to do the same as for classical FSMs and assign a pair of states one point for each shared outgoing transition. The problem with this is that it does not take into account the fact that EFSM transitions do not need to be exactly identical to express the same behaviour. We thus need a more permissive scoring function which allows non-identical transitions to contribute to the merge score of a pair of states.

The scoring approach I take here and used in [63] is inspired by [152], where the merge score of each pair of states is equal to the number of outgoing transitions with a shared label. Because my transitions are slightly more complex than those in [152], I can have a slightly more fine-grained metric. Here, as well as sharing a label, transitions must have the same arity and produce the same number of outputs to contribute to the merge score of their respective origin states. This is because two transitions with different input or output arities cannot represent the same behaviour since they are respectively either mutually exclusive or observably different.

As well as contributing one point if their labels and arities are the same, transition pairs get an additional "bonus point" if they are exactly equal. That is, if they have identical guards, outputs, and update functions. States with lots of identical outgoing transitions more obviously represent the same program state than those with outgoing transitions which only share a label and input and output arities. It therefore makes sense to try to merge these earlier on in the inference process not only because it is more likely that they will actually represent the same state, but also because the nondeterminism associated with the merge is likely to reveal more of such states, enabling us to merge many states in a single iteration.

It is easy to conceive more sophisticated scoring approaches. We could, for example, try to use the subsumption relation from the previous section. We might also attempt to lift either RPNI or Blue-Fringe to EFSMs, or simply apply these techniques as they are using just the labels of transitions, although this is unlikely to be generally applicable since it neglects the data state of the model as well as the inputs and outputs of individual transitions. We could even attempt to use classifiers on the different data values like MINT [152].

Since most of the novelty between different classical FSM inference techniques in the literature concerns the way potential states merges are determined and ranked, this suggests that the scoring function is a crucial part of the inference process so warrants attention in its own right. A comprehensive investigation into the effects of different merging criteria in terms of both accuracy and runtime is desirable future work but, since my models are so different to those which occur in existing literature on passive inference, I chose to to keep things simple here to maintain focus on the main objective, which is to establish a state merging algorithm for inferring computational EFSM models from black-box traces. As we will see in Chapter 8, though, my simple scoring metric can still lead to models which are both small and accurate.

In my implementation, detailed in Section 6.4, the scoring function is a parameter to the inference process so can easily be swapped for something more sophisticated, although this is currently likely to have a significant impact on the runtime as the current implementation scores every pair of states per iteration. For an EFSM with n states, we score ${}^{n}C_{2}$ state pairs each iteration. Even for moderately sized EFSMs, this is a very large number. Thus, more complex scoring metrics are likely to take a prohibitively long time to run. This is partially why algorithms such as Blue-Fringe do not examine every state pair per merge.

It is also worth mentioning here that, like in [152], I only look at the immediate outgoing transitions from states. I do not examine longer paths as in k-tails [16]. The reason behind this is that, unless we are going to consider the update functions within the scoring metric, it does not make sense to look further ahead. Since we can arbitrarily move information between the control flow state and the data state by utilising registers, if we wish to apply the principles from k-tails to EFSMs in a meaningful way, we need to consider the data state as well. Again, a more detailed exploration of this is desirable future work.

6.2.3 Resolving Nondeterminism by Merging Transitions

When we merge a pair of states, the resulting state then has both sets of outgoing transitions. This means that the same behaviour may be represented more than once, since we merge states with outgoing transitions that we believe represent the same behaviour. Duplicated behaviour often manifests as nondeterminism between outgoing transitions from newly merged states. We would like to resolve this by merging the offending transitions and their respective destinations, which must represent the same program state if the transitions represent the same behaviour.

In classical inference, there is no need to explicitly merge transitions. Classical FSM transitions only represent the same behaviour if their labels are equal, so this happens "for free" when their destination states are merged because two transitions with the same label, origin, and destination are not distinct. With EFSMs, transitions that express the same behaviour may not be exactly identical. Thus, the merging of transitions becomes an explicit step in the algorithm in addition to merging their destination states.

There is also the possibility that two nondeterministic transitions may not be able to be merged, which does not occur in classical FSM inference. For example, consider the transitions $\mathit{vend} : 0/[o_0 := \text{“tea”}]$ and $\mathit{vend} : 0$. If we have merged their respective origin states under the belief that they represent the same behaviour, we would also want to merge the transitions to reflect this belief. Unfortunately, they produce different numbers of outputs so cannot possibly represent the same behaviour.

The way we go about resolving nondeterminism is similar to how it is done in classical FSM inference. We simply resolve one nondeterministic transition pair at a time until the model becomes deterministic again. The two main additional problems to solve here are that, firstly, EFSM transitions allow multiple expressions of the same behaviour, meaning that we may need to merge non-identical transitions to resolve nondeterminism; and secondly, we must have a way to handle what happens if an attempt to resolve nondeterminism fails.

The way we handle the first problem is with *subsumption.* After merging the destination states of a nondeterministic pair of transitions, the resolveNondeterminism function calls mergeTransitions to merge the transitions themselves. When merging EFSM transitions, one must *account for* the behaviour of the other under all circumstances where it could have been taken. The idea of *subsumption in context* was introduced in Chapter 5 and formalises the intuition that, in certain contexts, a transition t_2 reproduces the behaviour of t_1 and updates the data state in a manner consistent with t_1, meaning that t_2 can be used in place of t_1 with no observable difference in behaviour.

Algorithm 4 Resolving nondeterminism.

```
function resolveNondeterminism($[]$, $\_$, $new$)
    if deterministic($new$) then
        return Some $new$
    else
        return None
function resolveNondeterminism($((from, (d_1, d_2), (t_1, t_2))\#ss)$, $old$, $new$)
    $destMerge \leftarrow$ mergeStates($d_1, d_2, new$)
    switch mergeTransitions($old, destMerge, t_1, t_2$) do
        case None
            resolveNondeterminism($ss, old, new$)
        case Some $merged$
            $newPairs \leftarrow$ nondetPairs($merged$)
            switch resolveNondeterminism($newPairs, old, merged$) do
                case Some $new'$
                    return Some $new'$
                case None
                    resolveNondeterminism($ss, old, new$)
function mergeTransitions($old, destMerge, t_1, t_2$)
    if directlySubsumes($old, destMerge,$ origin($t_1, old$)$, t_2, t_1$) then
        return Some replaceTransition($destMerge, t_1, t_2$)
    else if directlySubsumes($old, destMerge,$ origin($t_2, old$)$, t_1, t_2$) then
        return Some replaceTransition($destMerge, t_2, t_1$)
    else
        return None
```

To use the subsumption relation, we must have a *context* in which to test subsumption. When merging transitions, one must subsume the other in all obtainable contexts of its origin before the merge. This was introduced as *direct subsumption* in Chapter 5. The mergeTransitions function in Algorithm 4 tests if one transition directly subsumes the other, meaning that it can be safely deleted without affecting the behaviour of the model. If this is not the case, neither transition can be used in place of the other without risking an observable difference in the behaviour of the model. In this case, mergeTransitions fails, returning **None**.

This leads us on to the second problem that must be solved. What happens when neither transition directly subsumes the other? If mergeTransitions fails, nondeterminism might be resolved by merging a different transition pair. Successive attempts are made until either one is successful or there are no more potential merges. In the latter case, resolveNondeterminism fails, indicating that the original state pair should not have been merged.

In Example 5.4.2, it was identified that we sometimes need to be careful which transition we use in situations where both transitions subsume each other. There, we had it such that we needed to merge transitions $vend : 0$ and $vend : 0[r_2 < 100]$. The correct strategy was to use $vend : 0[r_2 < 100]$ to prevent nondeterminism with a third transition, $vend : 0[r_2 \geq 100]/o_0 := r_1$, which cannot be resolved by merging. In that case, if the resolveNondeterminism function chose to take $vend : 0$ in the first instance, it would need to make a recursive call to resolve the nondeterminism between this transition and $vend : 0[r_2 \geq 100]/o_0 := r_1$. Since this nondeterminism cannot be resolved by merging, the recursive call would return **None**, indicating that the nondeterminism could not be resolved. Because NONDETPAIRS returns all configurations of nondeterministic transitions, RESOLVENONDETERMINISM would then be able to have another go at the merge with the $vend : 0[r_2 < 100]$ being the dominant transition.

An alternative approach to resolving nondeterminism pairwise would be to form *equivalence classes* of nondeterministic transitions as in [142] and merge each class in one go. When merging states pairwise, it is quite rare for nondeterminism to be introduced between more than two transitions at once, meaning that, if we were to form equivalence classes, most would contain only two transitions. Since forming equivalence classes requires a pairwise comparison anyway, I here decided to take the implementationally simpler decision to merge transitions pairwise.

6.2.4 Merging States

The INFERENCESTEP function in Algorithm 3 merges the first (highest scoring) state pair in the list of potential merges and calls RESOLVENONDETERMINISM (detailed in the Subsection 6.2.3) to resolve any resulting nondeterminism. If this succeeds, the merging process begins again with a new list of potential merges, continuing until no more states can be merged. If RESOLVENONDETERMINISM fails, this indicates that our belief of the two states representing the same program state was false, as we were unable to merge their respective behaviours. INFERENCESTEP then successively attempts to merge lower scoring state pairs until either one is successful or it runs out of possible merges, at which point inference terminates.

6.3 Introducing Registers

The technique in Section 6.2 allows us to infer deterministic EFSM models from traces by merging transitions where one directly subsumes the other, but we cannot yet fully capture the causal relationship between input and output. Recall from Subsection 2.2.5 that most real systems make use of *internal variables* that store information about the current state for later use. To accurately represent such systems, we need models that make use of data variables. We therefore need a way to infer their use within a model. This section proposes a way do this.

Example 6.3.1. The EFSM in Figure 6.4 is the best model of the traces in Figure 6.1 that we can infer so far. The model contains two pairs of identical *coin* transitions which we could merge by *zipping* the path $q_1 \to q_3 \to q_4 \to q_7$ with $q_2 \to q_5 \to q_6 \to q_7$. We cannot do this, though, as it would result in nondeterminism between transitions $vend : 0/o_0 :=$ "tea" and $vend : 0/o_0 :=$ "coffee" , which suggests they are instances of the same behaviour and should be merged. Since they have different literal outputs, neither can directly subsume the other so they cannot be merged. This means we cannot condense Figure 6.4 any further.

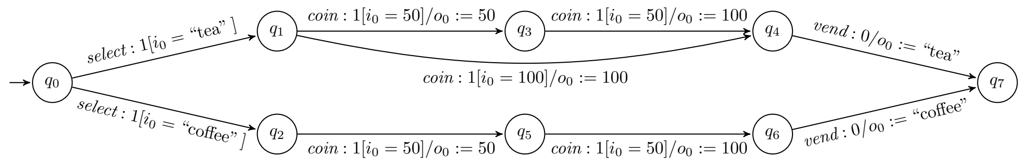

$select : 1[i_0 = \text{"tea"}]$
$coin : 1[i_0 = 50]/o_0 := 50$
$coin : 1[i_0 = 50]/o_0 := 100$
$vend : 0/o_0 := \text{"tea"}$
$coin : 1[i_0 = 100]/o_0 := 100$
$select : 1[i_0 = \text{"coffee"}]$
$coin : 1[i_0 = 50]/o_0 := 50$
$coin : 1[i_0 = 50]/o_0 := 100$
$vend : 0/o_0 := \text{"coffee"}$
q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7

Figure 6.4: An EFSM model inferred from the traces in Figure 6.1.

Looking at the bigger picture, the two *vend* transitions do actually exhibit the same behaviour. Both produce, as output, the input of the initial *select* transition. If we could abstract away the concrete values, we could infer a smaller and more general model of the system.

The approach I proposed in [63] was to provide MERGETRANSITIONS with a number of small *heuristics* to be used if neither transition directly subsumes the other. The aim here is to come up with arithmetic functions which generalise from the concrete values in the traces, with each heuristic focussing on a particular *data usage pattern.* This makes transitions which express the same behaviour with different data identical, allowing them to be merged. As mentioned in Chapter 3, traditional machine learning approaches such as deep learning are extremely good at this sort of thing, however, such methods are black-box predictors. Here, we are interested in *how* data are transformed, so require human-readable functions. If neither transition directly subsumes the other and none of the heuristics are successful, the transition merge fails. While this approach can be helpful, it has several limitations which I discuss at the end of this section and improve upon in Chapter 7.

Before continuing, it is worth noting that similar principles are applied in the field of active inference. As discussed in Chapter 3, work presented in [15] applies a technique which searches for the reuse of data input values as subsequent outputs, and abstracts these values away by storing them in a register. This is a very similar principle to the “store and reuse” heuristic presented later in this section. Additionally, in [28], states are merged when their symbolic suffixes are isomorphic up to the renaming of registers. In my work published in [63] I discuss a similar heuristic for recognising when multiple variables have been introduced to a model to serve the same purpose. In Chapter 7, I employ genetic programming as a *hyperheuristic* which generalises these ideas to discover more complex functions.

In Algorithm 5, the TRYHEURISTICS function attempts to apply the heuristics in the order in which they were supplied until either one of them is successful or there are no more left to apply. This approach makes the tool extensible and gives the user a degree of control over the characteristics of the final model as they can choose to provide or withhold particular heuristics. It also means that the order in which the heuristics are supplied has the potential to affect the model we infer.

Algorithm 5 The redefined MERGETRANSITIONS function which can now use *heuristics*.

```
function TRYHEURISTICS($old$, $destMerge$, $t_1$, $t_2$, $heuristics$)
    switch $heuristics$ do
        case []
            return None
        case H#$hs$
            switch H($old$, $destMerge$, $t_1$, $t_2$) do
                case None
                    return TRYHEURISTICS($old$, $destMerge$, $t_1$, $t_2$, $hs$)
                case Some $e$
                    return Some $e$
function MERGETRANSITIONS($old$, $destMerge$, $t_1$, $t_2$, $heuristics$)
    if DIRECTLYSUBSUMES($old$, $destMerge$, ORIGIN($t_1$, $old$), $t_2$, $t_1$) then
        return Some REPLACETRANSITION($destMerge$, $t_1$, $t_2$)
    else if DIRECTLYSUBSUMES($old$, $destMerge$, ORIGIN($t_2$, $old$), $t_1$, $t_2$) then
        return Some REPLACETRANSITION($destMerge$, $t_2$, $t_1$)
    else
        return TRYHEURISTICS($heuristics$)
```

We must be careful with heuristics, though. Since they are external, essentially arbitrary code, they cannot be deemed inherently trustworthy. That is, there is no explicit obligation on heuristics to produce models which reflect the traces. For example, we could supply a heuristic which always returns the empty EFSM. This will always successfully return a model, but it is clearly unacceptable for the inference process to attempt to proceed with this as it will lead to a model which does not conform to the original PTA. We must therefore be suspicious of solutions offered by heuristics if we want our inference process, as a whole, to maintain trace inclusion between the PTA and the final model.

Since the original set of traces is finite, trace inclusion is very easy to verify. We can simply run each one through the model and compare the output to the original.[1] I run this sanity check at two points during the inference process. Firstly, the TRYHEURISTICS function applies it when a heuristic claims to have been successful. Secondly, I apply it after each iteration of INFER to be absolutely certain that the model still reflects the observed behaviour. If this is not the case, the model is discarded as if the state merge had failed. This leads to a trivial trace inclusion proof of the inference process — the property is explicitly checked each iteration — but proving this in Isabelle turns out to be extremely difficult as it requires custom induction rules for the INFER function. Since the intuition is so trivial, the automated proof is left for future work.

6.3.1 Example Heuristics

This section details some heuristics which are relevant to our running drinks machine example.

The Store and Reuse Heuristic

An obvious candidate for generalisation is the "store and reuse" pattern. This occurs in Example 6.3.1 when the input of *select* is later used as the output of *vend*. Recognising this pattern allows us to introduce a *storage register* to abstract away concrete data values and replace two transitions whose outputs differ with a single transition that outputs the content of the register.

The first step is to find *intratrace* matches — instances of data reuse *within* traces. We walk each trace in the current EFSM, recording when the output of a transition matches the input of an earlier transition to obtain a set of matches for each trace of the following form.

$$\{((\mathit{transition}, \mathit{inputIndex}), (\mathit{transition}, \mathit{outputIndex}))\}$$

We then look to see if any of the matches concern the transitions we are trying to merge. If so, we attempt to *generalise* these transitions. This consists of introducing a fresh register to act as storage, adding an update to this register, and dropping the restriction on the relevant input value. The value of this register then becomes the output of the second transition. For example, we would generalise the pair $((\mathit{select} : 1[i_0 = \text{“tea”}], 1), (\mathit{vend} : 0/o_0 := \text{“tea”}, 1))$ to $((\mathit{select} : 1/[r_1 := i_0], 1), (\mathit{vend} : 0/o_0 := r_1, 1))$, where r_1 does not already occur in the EFSM.

When multiple transition pairs generalise in the same way *between* multiple traces, we call this an *intertrace* match. Finding intertrace matches indicates that the same kind of behaviour occurs across multiple traces, potentially with different data values. This provides evidence in favour of generalising and merging transitions in the model.

[1]In an ideal world, it would be good to properly verify simulation as per Definition 19 from one iteration of inference to the next. While it is possible to use a model checker as a (partial) oracle to this effect, this is simply too slow to be practical for large models.

Example 6.3.2. Consider the EFSM in Figure 6.5. To resolve the nondeterminism between the *vend* transitions, we need to merge them into a single transition. Neither directly subsumes the other since there is always a difference in behaviour. Consequently, we need to try to find a transition which accounts for both of their behaviours.

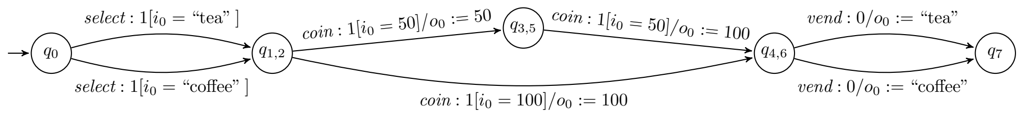

$select : 1[i_0 = \text{“tea”}]$
$select : 1[i_0 = \text{“coffee”}]$
q_0
$q_{1,2}$
$coin : 1[i_0 = 50]/o_0 := 50$
$q_{3,5}$
$coin : 1[i_0 = 50]/o_0 := 100$
$coin : 1[i_0 = 100]/o_0 := 100$
$q_{4,6}$
$vend : 0/o_0 := \text{“tea”}$
$vend : 0/o_0 := \text{“coffee”}$
q_7

Figure 6.5: The EFSM from Figure 6.4 after *zipping* the path $q_1 \to q_3 \to q_4 \to q_7$ with $q_2 \to q_5 \to q_6 \to q_7$.

Assuming that Figure 6.5 is inferred from the PTA in Figure 6.3, the store and reuse heuristic has three traces to consider. The first step is to find intratrace matches by looking for transitions where an output is equal to an earlier input. We then filter these to remove those which are not relevant. That is, we only keep those intratrace matches which concern the transitions we are currently trying to merge, in this case the *vend* transitions. This leaves us with the following list of matches.

$$\begin{array}{l}
[\\
\{((\mathit{select} : 1[i_0 = \text{“tea”}], i_0), (\mathit{vend} : 0/[o_0 := \text{“tea”}], o_0))\},\\
\{((\mathit{select} : 1[i_0 = \text{“tea”}], i_0), (\mathit{vend} : 0/[o_0 := \text{“tea”}], o_0))\},\\
\{((\mathit{select} : 1[i_0 = \text{“coffee”}], i_0), (\mathit{vend} : 0/[o_0 := \text{“coffee”}], o_0))\}\\
]
\end{array}$$

The next stage is to *generalise* the transition pairs as described above. To do this, we introduce a fresh register called r_1. For each transition pair, the input of interest is derestricted and an update is added to assign the input value to the r_1. The literal outputs are replaced with the value of r_1. This leaves the following.

$$\begin{array}{l}
[\\
\{((\mathit{select} : 1/[r_1 := i_0], i_0), (\mathit{vend} : 0/[o_0 := r_1], o_0))\},\\
\{((\mathit{select} : 1/[r_1 := i_0], i_0), (\mathit{vend} : 0/[o_0 := r_1], o_0))\},\\
\{((\mathit{select} : 1/[r_1 := i_0], i_0), (\mathit{vend} : 0/[o_0 := r_1], o_0))\}\\
]
\end{array}$$

We now look for intertrace matches. That is, do multiple transition pairs generalise in the same way? If so, this indicates that different data is used in the same way, and serves as evidence that we can carry the generalisation into the model. Here, all three generalised pairs are equal, so the generalisation goes ahead. This leaves us with the model in Figure 6.6.

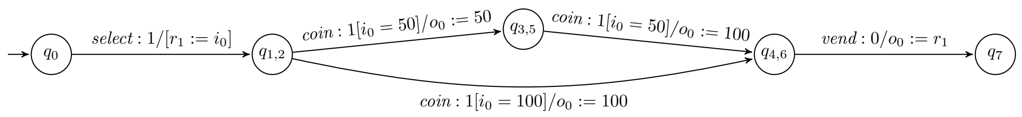

q_0
$select : 1/[r_1 := i_0]$
$q_{1,2}$
$coin : 1[i_0 = 50]/o_0 := 50$
$q_{3,5}$
$coin : 1[i_0 = 50]/o_0 := 100$
$coin : 1[i_0 = 100]/o_0 := 100$
$q_{4,6}$
$vend : 0/o_0 := r_1$
q_7

Figure 6.6: The EFSM from Figure 6.5 after generalising the model.

The Increment and Reset Heuristic

Another usage pattern is "increment and reset". In our drinks machine example, the *coin* action outputs the sum of the previous *coin* inputs. This allows customers to use multiple coins to pay for their drink and to observe a running total of the payment they have inserted so far. Correctly identifying this usage pattern is not an easy problem to solve, and a general approach to this is the subject of Chapter 7, but a naive heuristic is not difficult to implement.

The idea is that, if we want to merge two transitions with identical input values and different outputs, for example $coin : 1[i_0 = 50]/o_0 := 50$ and $coin : 1[i_0 = 50]/o_0 := 100$, then the behaviour must depend on the value of an internal variable. We implement a heuristic which, when faced with such a merge, drops the input guard and adds an update to a fresh register, in this case summing the current register value with the input. For this to work, we must ensure that the register is initialised before our modified transitions are taken. To do this, we augment transitions incident to the origin state with an update function which sets the relevant register to zero. This is the "reset" part of the heuristic which ensures that the register is defined before it is used. A similar principle can be applied to other numeric functions such as subtraction.

Example 6.3.3. Let us begin with the EFSM in Figure 6.6. States $q_{1,2}$ and $q_{3,5}$ both have outgoing *coin* transitions, so let us attempt to merge them. The resulting state has two nondeterministic *coin* transitions, which take an input of 50. We need to merge these into a single transition, so we merge their respective destination states,

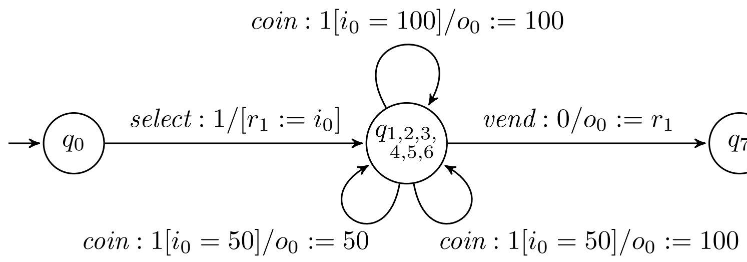

$coin : 1[i_0 = 100]/o_0 := 100$
$select : 1/[r_1 := i_0]$
q_0
$q_{1,2,3,4,5,6}$
$vend : 0/o_0 := r_1$
q_7
$coin : 1[i_0 = 50]/o_0 := 50$
$coin : 1[i_0 = 50]/o_0 := 100$

Figure 6.7: The EFSM from Figure 6.5 merging states $q_{1,2}$, $q_{3,5}$, and $q_{4,6}$.

We now need to merge the two $coin : 1[i_0 = 50]$ transitions into a single representation of their behaviour. Since they have different literal outputs, neither subsumes the other directly. It is now that the increment and reset heuristic comes into play. When we have two transitions with the same literal input but different literal output, we need to introduce a register to account for the behaviour. Like with the store and reuse heuristic, this register must be fresh. Let us call this register r_2. The naive approach of the increment and reset heuristic is to simply replace the literal output with $r_2 + i_0$, add this as an update, and drop the guard. This leaves both $coin : 1[i_0 = 50]$ transitions as $coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$.

In this case, dropping the guard causes a problem. The other *coin* transition, with input guarded to be 100, is now nondeterministic with our newly merged behaviour. If this happens, the increment and reset heuristic simply removes the offending transition, working under the assumption that the generalisation we have just made accounts for this behaviour too. This leaves us with the EFSM in Figure 6.8.

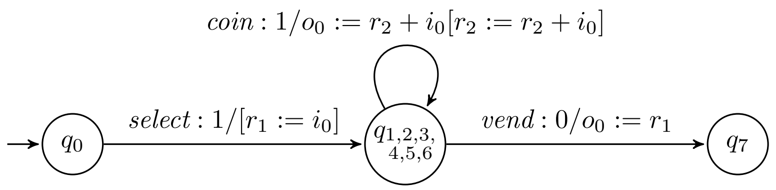

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_0
$select : 1/[r_1 := i_0]$
$q_{1,2,3,4,5,6}$
$vend : 0/o_0 := r_1$
q_7

Figure 6.8: The EFSM from Figure 6.7 merging the *coin* transitions.

We then need the "reset" part of the heuristic to ensure that our fresh register is initialised before it is used. To do this, we add an update $r_2 := 0$ to all transitions incident to the origin state of *coin*. Having done this, it is vital to check that what we have done still reproduces the behaviour observed in the traces. If it does, the heuristic has been successful. If not, it has failed to generalise correctly and the heuristic must fail gracefully. This is an important step since the naivety of the heuristic makes it likely to produce models which do not behave as expected. This is a major limitation of the approach and is improved upon in Chapter 7.

6.3.2 The Limitations of Heuristics

It is worth mentioning that heuristics in their current form are a serious limitation to the inference process. While certain heuristics like "store and reuse" have some general applicability, the scope of individual heuristics is likely to be very limited. The challenge then, for each system, is to determine and implement heuristics are likely to be applicable. In our simple drinks machine example, the traces make it clear what the underlying system does, so it is not difficult to work out where we need to introduce abstractions. For more complex systems, the underlying behaviour may not be so apparent, meaning that it is much harder to know exactly what the heuristics need to do. We are thus in a situation where the user effectively needs to have a "right answer" in mind before they apply inference.

Ideally, we want a more general approach which gives the inference process a way of figuring out the correct output and update functions on its own, without the help of specially coded heuristics. In other words, the inference process needs a way to automatically build the heuristics it needs "on the fly" using only the data from the observed traces. This is an extremely difficult problem to solve, which is why I have not tackled it here. Instead, this is the subject of Chapter 7.

Currently, heuristics are applied at merge time. This means that we only generalise outputs and introduce registers as a last resort and that heuristics must work with nondeterministic models in the midst of resolution. It also means that heuristics only get to see two transitions at a time. This clearly does not scale since large models built from many traces are likely to contain lots of duplicated behaviour which will lead us to apply the same heuristic many times during inference. Unless we are very careful about how we implement heuristics, we may end up introducing lots of different registers to do the same job.

Another problem with applying heuristics at merge time is that, as well as resolving nondeterminism, they can also *introduce* it. We see this in Example 6.3.3 where the "increment and reset" heuristic needs to also modify the $coin : 1[i_0 = 100]/o_0 := 100$ transition to avoid introducing new nondeterminism. Again, this is manageable in small models but does not scale to larger ones. Indeed, preliminary experiments with more than the three traces of the simple drinks machine in Figure 6.1 revealed that the "store and reuse" heuristic can introduce more nondeterminism than it solves, effectively rendering it useless.

Instead of trying to generalise transitions as a last ditch attempt to merge behaviour, it makes sense to try to do this *before* we begin to merge states. Doing this makes achieving our goal of finding and merging duplicated behaviour much easier since generalised transitions are exactly identical. This makes scoring merges and resolving the resulting nondeterminism much easier, so reduces the work we have to do during merging.

In summary, there are two main problems with my current approach: the scope of applicability and the time of application. We need a more general way to abstract concrete values into functions, and we need to apply this at the correct time so that we can use it effectively. In Chapter 7, I present a preprocessing technique based on genetic programming which delivers on both of these fronts. For now, as in [63], the heuristics serve as a placeholder to bridge the gap between pure transition merging and the general solution presented in the next chapter.

While it is clear that using merge time heuristics to generalise behaviour is far from ideal, there are some very useful heuristics which need to be applied during the merging process. In Subsection 6.4.3, I show how we can use a heuristic to make the directlySubsumes function executable. In Section 7.6, I present a heuristic to identify *value-dependent behaviour* and infer guards which distinguish between transitions.

6.4 Implementation

Having now formulated the inference algorithm abstractly, the next task is to code this up into an executable tool which can be practically evaluated. This tool should take a set of traces as input and produce an EFSM model which represents those traces. Effectively, we are trying to obtain an executable version of the learn function from Algorithm 3. The user should also be able to supply heuristics to use during merging process. Unfortunately, some parts of my technique, most notably the directlySubsumes function, cannot be effectively computed. This section details how I tackled this to produce a prototype inference tool. An example of its use is shown in Appendix A.

In Chapters 4 and 5, I presented an Isabelle formalisation of both EFSMs and direct subsumption. As well as being a very effective proof assistant, Isabelle also includes a *code generator* [77] which can be used to automatically transform Isabelle functions and data types to executable code in various conventional programming languages. Since the generated code satisfies the same properties as the original definitions, it makes sense to develop my tool in Isabelle as I can use my existing EFSM formalisation as a basis for this rather than having to start from scratch in a more conventional language and risk there being inconsistencies between my formalisation and implementation. This also paves the way for various correctness proofs of the implementation, such as trace inclusion between the PTA and the final model. Although such proofs are intuitively very simple and mainly follow "by construction", they actually turn out to be extremely difficult to phrase in Isabelle, requiring custom induction rules for the various functions, so have consequently been left for future work.

Another advantage of Isabelle is that functions can be expressed at a very high level of abstraction, meaning that Algorithms 3 and 4 can be expressed in Isabelle almost identically to how they appear in Section 6.2. The fact that it is a purely functional language like Haskell also means that we have intuitive symbolic equality and full referential transparency, so there is no risk of accidentally overwriting variables. This all makes Isabelle a very pleasant language in which to program.

Having implemented my technique in Isabelle, I then used the code generator to convert these functions and data types to runnable code. The language I chose was Scala,[2] a language which runs on the Java Virtual Machine and, consequently, has access to the various Java libraries. This makes getting traces into the system, and inferred models out again, quite straightforward. It also affords access to libraries such as Z3, which I use to check satisfiability of guards as detailed in Subsection 6.4.2 and allows integration with other tools implemented in Java such as MINT, which I use as the basis for my GP technique to infer output and update functions in Chapter 7.

6.4.1 Proving Termination

Isabelle's code generator will only export code for *total functions.* That is, functions which always have a defined return value. While it has ways of dealing with what the documentation calls "explicit partiality", for example missing patterns in `case` matches, it refuses to generate code for functions which cannot be proven to terminate. Isabelle can often do this automatically, so explicit termination proofs are rarely required. Unfortunately, our inference process contains two functions for which Isabelle could not automatically prove termination. These functions are INFER in Algorithm 3 and RESOLVENONDETERMINISM in Algorithm 4.

When we do need to prove termination manually, what we generally need to do is define a function that measures the arguments and returns a natural number. We then prove that this strictly decreases every time the function recursively calls itself. Eventually the size of the arguments will reach 0, at which point the function should hit its base case and terminate. Defining such measurement functions is notoriously tricky. Indeed, it is not currently possible to define one for either of the two functions as they currently appear in Algorithms 3 and 4. They need to be modified a little.

The INFER function performs the outer loop of the inference process, recursively calling INFERENCESTEP until no more state merges can be made. The measurement function here, then, is obvious. If INFERENCESTEP merges states, the size of the model should be smaller for each recursive call to INFER. The problem here is that this assumed property of INFERENCESTEP is not explicitly proven. Indeed, because we can make use of arbitrary heuristics, as discussed in Section 6.3, it is not even necessarily true.

There is nothing to stop our heuristics from maintaining or even increasing the number of states in an EFSM, so it is not guaranteed that the size of the model returned by INFERENCESTEP is necessarily smaller than the original. We do, however, want to enforce that every iteration of inference reduces the number of states in the model. While it is certainly possible to write heuristics which increase the size of an EFSM, this is considered bad behaviour. I therefore explicitly check for this in INFER such that the number of states in the model gets provably smaller with each recursive call to INFER. If the model is not smaller, INFER immediately terminates. This does not penalise heuristics which maintain the number of states in the model since heuristics are only called by RESOLVENONDETERMINISM once INFERENCESTEP has merged a pair of states, thus making the model smaller to begin with.

Defining a measurement function for RESOLVENONDETERMINISM turned out to be much more challenging. The function is called after merging a pair of states introduces nondeterminism to the model. The destination states of nondeterministic transitions are merged, and the transitions merged into single representatives of their behaviour such that the EFSM becomes deterministic

[2]`https://www.scala-lang.org` (Accessed 13/05/20)

again. The first argument to RESOLVENONDETERMINISM is a list of "nondeterministic pairs". These are tuples consisting of a pair of nondeterministic transitions, their common origin, and their respective destinations. Each iteration should then decrease the size of this list to make the model less nondeterministic with each recursive call.

Resolving a nondeterministic pair of transitions involves first merging their respective destination states, and then merging the transitions themselves. The first step of merging the destination states may, however, introduce further nondeterminism to the model, meaning that the problem gets worse before it gets better. This does not necessarily spiral out of control and, if allowed to continue, RESOLVENONDETERMINISM may be able to make the model deterministic. The fact that we do not want to enforce that the number of nondeterministic pairs monotonically decreases with each iteration means that we cannot use this alone as our termination measure. We must take other arguments into account.

Considering the fact that, like INFERENCESTEP, RESOLVENONDETERMINISM merges states and transitions, perhaps measuring the size of the model like with INFER might be a better tactic. It may be the case, though, that two nondeterministic transitions represent genuinely different behaviours and that, rather than merging the transitions, the correct course of action would be for a heuristic to infer mutually exclusive guards to distinguish the two transitions. Such a heuristic is presented at the end of Chapter 7. This resolves the nondeterminism but maintains the number of states and transitions.

The only sensible approach to take is to use all three features (the number of nondeterministic pairs and both the number of states and the number of transitions in the model) in the measurement function. Like with INFER, we need an explicit test to ensure that our measurement function is decreasing with each recursive call. Instead of using a single value, I use the tuple ($numStates$, $numTransitions$, $numPairs$). First, I compare the number of states in the current model to the number in the model we would like to make the recursive call with. If it is less, this is fine and the recursive call can proceed. If it is more, this is bad and the function terminates with an error. If it is equal, I perform a similar check with the transitions. If the number of transitions is also equal, I look at the number of nondeterministic pairs.

While the order we compare states and transitions is arbitrary, we must compare the number of nondeterministic pairs last, since this is the only metric we would like to be allowed to *increase* and still have the recursive call go ahead. Of course, there is the chance that RESOLVENONDETERMINISM may still terminate even with the removal of this restriction, but this is the most liberal we can reasonably be while still guaranteeing termination.

6.4.2 Non-Computable Functions

Having proven termination of the various functions, the vast majority of the inference process can be exported automatically to executable Scala code using the command `export_code learn in Scala`. This generates executable Scala code for the LEARN function and all of its associated dependencies. The code generator still presents us with an error, though.

Looking at Algorithms 3 and 4, there are two functions which cannot be effectively computed: NONDETPAIRS and DIRECTLYSUBSUMES. The code generator cannot generate code for these functions. This leaves us with gaps in the implementation that must be implemented manually. For these, the `code_printing` statement provides the ability to replace functions with custom implementations in the target language.

Detecting Nondeterminism

The nondetPairs function takes a model and produces a list of nondeterministic transition pairs. For each state, it checks if there is a choice between any pair of outgoing transitions. This involves checking if the conjunction of their guards is satisfiable. In Isabelle, we simply write $\exists i\, r.\textsc{applyGuards}(t_1, i, r) \wedge \textsc{applyGuards}t_2, i, r$. We cannot effectively compute this though because, to know that there exists a satisfying assignment, we must find a witness. Because the search space is infinite, the code generator does not know how to handle this.

In the Scala implementation, I leverage an existing SMT solver, Z3 [45], by converting transition guards to the appropriate format at runtime. This was not as straightforward as it initially sounds, though, because of the way EFSMs handle arithmetic. As discussed in Chapter 4, EFSMs are dynamically typed. Because of this we need to use an "optional" evaluation semantics for arithmetic expressions and a three-valued logic for guards. For Z3 to provide an accurate answer to the question of satisfiability, I must provide it with the same optional arithmetic and three-valued logic from Isabelle.

To achieve this, I first defined the necessary datatypes of `Value`, `Option`, and `Trilean` as in my Isabelle formalisation of EFSMs. The `Option` datatype takes a type parameter `X` as an argument and is either `None` or `Some (x::X)`. The `Value` datatype is either `Num (n::Int)` or `Str (s::String)`, and the `Trilean` datatype is either `true`, `false`, or `invalid`.

```
(declare-datatype Option (par (X) ((None) (Some (val X)))))
(declare-datatype Value ((Num (num Int)) (Str (str String))))
(declare-datatype Trilean ((true) (false) (invalid)))
```

Next, I defined the arithmetic operators. My arithmetic consists of literal values, input and register variables, addition, subtraction, and multiplication. My Isabelle formalisation of EFSM arithmetic works in terms of `Value Options` rather than `Values`, and returns `None` for badly typed expressions and those involving uninitialised registers. This means that we cannot simply use Z3's native operators and, instead, must define our own. The semantics of the `Plus` function are that, if its two arguments are `Some (Num n1)` and `Some (Num n2)`, then it will return `Some (Num (n1 + n2))`. Otherwise, it will return `None`. This is defined in Z3 as follows, and the other binary operators are defined similarly.

```
(define-fun Plus ((x (Option Value)) (y (Option Value))) (Option Value)
  (match x (
    ((Some v1)
      (match y (
        ((Some v2)
          (match v1 (
            ((Num n1)
              (match v2 (
                ((Num n2) (Some (Num (+ n1 n2))))
                (_ None))
              ))
            (_ None))
          ))
        (_ None))
      ))
    (_ None))
  )
)
```

My Isabelle EFSM formalism supports a minimal but functional set of guard operators: equality, greater than, and set membership. To reduce the number of operators, logical NOR is the only connective, with the more conventional operators being defined in terms of this. Lifting this to our three valued logic gives the following definition in Z3.

```
(define-fun Nor ((x Trilean) (y Trilean)) Trilean
  (ite (and (= x true) (= y true)) false
  (ite (and (= x true) (= y false)) false
  (ite (and (= x false) (= y true)) false
  (ite (and (= x false) (= y false)) true
  invalid))))
)
```

The guard operation of *greater than* is very similar to `Plus` in that the function is only defined over numeric arguments. Since register variables (and the result of evaluating arithmetic expressions) may be undefined, it must also work over `Value Options` rather than simple `Values`. If both of its arguments are `Some` number and the first argument is greater than the second, it returns *true*. If the second argument is less than or equal to the first, it returns *false*. Otherwise, one or both of the arguments is a string or `None` so it returns *invalid*. This is defined in Z3 as follows. The equality test is lifted to return trilean *true* if its arguments are equal and trilean *false* otherwise.

```
(define-fun Gt ((x (Option Value)) (y (Option Value))) Trilean
  (ite (exists ((x1 Int)) (exists ((y1 Int))
    (and (= x (Some (Num x1)))
    (and (= y (Some (Num y1))) (> x1 y1))))) true
  (ite (exists ((x1 Int)) (exists ((y1 Int))
    (and (= x (Some (Num x1)))
    (and (= y (Some (Num y1))) (not (> x1 y1)))))) false
  invalid))
)
```

Translating set membership to Z3 is non-trivial since it has no native support for sets. While it is possible to model sets in Z3 with arrays or boolean predicates, this is overkill here. Instead, I exploit the fact that $s \in S \equiv \bigvee_{t \in S} s = t$ and translate membership guards as a disjunction of equality tests. This requires the logical OR operation to be defined in Z3 in terms of `Trileans`, but this is just the negation of the `Nor` operator defined above.

With the necessary operators defined in Z3, translating guards at runtime can be done with a couple of simple recursive functions, one for arithmetic expressions and one for guard expressions. We can then assert that the translated guard expression is equal to trilean *true* and check satisfiability. Although I made no special effort to construct the above operations in such a way as to feed into the various existing optimisations of Z3, they do not appear to have a significant effect on the runtime to check the kinds of simple guards which are likely to occur during inference. Indeed, informal experiments revealed that there is no noticeable difference between the native operations and my custom ones, even for relatively complex guards involving non-linear arithmetic such as $((((i_1 \times i_1) \leq 3)) \wedge ((r_1 \neq 5)))$. Both representations can be checked in a few milliseconds. It may be the case that very complex guards involving tens or hundreds of variables result in a noticeable difference in checking time, but these did not occur during the evaluation of my tool.

Merging Transitions

The nondeterminism that arises when we merge states is resolved by merging first the destination states of the offending transitions and then the transitions themselves. Transitions can only be merged if one *directly subsumes* the other, otherwise there is a risk that the behaviour of the resulting model will be observably different to that of the original. Testing whether one transition directly subsumes another is therefore a critical part of the inference process for EFSMs, however the definition of DIRECTLYSUBSUMES in Algorithm 4 cannot be effectively computed in the general case as it involves checking subsumption for all traces which bring the model to a particular state. Reflexive transitions and an infinite input domain means that the number of traces which must be checked is not necessarily finite.

The ideal solution would be to somehow hook Isabelle into the implementation such that the system could obtain the relevant proofs during inference. While Isabelle's `sledgehammer` tool provides a high degree of automation, it is not sufficiently advanced to be able to find subsumption proofs on its own. If we were to make the inference tool interactive, we could allow the user to supply proofs. Unfortunately, many hundreds of such proofs may be necessary during inference. While they are not usually intellectually challenging, for larger models they are still extremely laborious, so it is simply not reasonable to ask this of the user.

The solution to this lies in the fact that the inference process only encounters transitions from the original PTA and those introduced by the heuristics provided. This means that, for a particular set of heuristics, the general form of every transition the inference process will come across is known. It is therefore possible to prove direct subsumption for various pattern combinations offline such that, at runtime, the direct subsumption check is reduced to a pattern matching exercise. For example, every transition trivially directly subsumes itself. A proof of the statement $\forall e_1.\, \forall e_2.\, \forall s_1.\, \forall s_2.\, \textsc{directlySubsumes}(e_1, e_2, s_1, s_2, t, t)$ allows us to check at runtime whether two transitions are equal and, if they are, return *true* without further investigation.

Sometimes it may not be possible to fulfil all the proof obligations for direct subsumption offline. This is especially true of transitions involving registers, where observable behaviour depends on the values that are held. Here, subsumption often relies on the anterior context satisfying certain properties, for example that a particular register must hold a certain value. We can then reduce the rather abstract notion of subsumption down to this more concrete check.

Even though this approach can significantly reduce the complexity of the properties that need to be validated at runtime, it does not change the fact that it may still be necessary to check properties for an infinite number of traces. To solve this problem, I use SAL,[3] a model checker with a similar representation to my EFSM models. The intricacies of both the translation of EFSMs to SAL and the verification process are detailed in Chapter 9. While it is not feasible to encode the DIRECTLYSUBSUMES predicate in SAL, simple properties like "register r is always undefined in state s" can be verified (or refuted) in milliseconds.

The catch is that SAL (and model checkers in general) only work with finite datatypes and finite subsets of infinite types, such as the integers. This means that we sacrifice some of the safety of an inductive proof, but the payoff is complete automation. If we check traces over a suitable subset of inputs, then we can be reasonably confident that transition merges made as a result are safe. Somewhat frustratingly, the definition of direct subsumption requires the existence of an input which satisfies the guard of the subsumed transition for a particular context. This means that we often need SAL even when one transition does *not* subsume another.

[3]`http://SAL.csl.sri.com` (Accessed 13/05/19)

Applying this approach to the patterns that occur when using the heuristics detailed in Section 6.3 allows our executable DIRECTLYSUBSUMES function to simply step through the cases until one matches. If none of the cases match, the "catch all" case is to ask the user but, for the heuristics detailed in Section 6.3, this is not required. Let us now examine the various transitions which arise from these heuristics.

Different Literal Outputs. If two transitions have outputs which always differ, for example $\mathit{vend} : 0/[o_0 := \text{"tea"}]$ and $\mathit{vend} : 0/[o_0 := \text{"coffee"}]$, then there is always an observable difference in behaviour. Along similar lines, transitions which produce different numbers of outputs are always distinguishable. In both of these cases neither transition directly subsumes the other as long as, for each transition, there exists a context which can be obtained in the origin state for which there exists an input which can satisfy the guard.

Drop Guard Add Update. The "store and reuse" heuristic exchanges a concrete-value guard on an input for an assignment to a fresh storage register. For a pair of transitions, in which one has been generalised and the other has not, for example $\mathit{select} : 1/[r_1 := i_0]$ and $\mathit{select} : 1[i_0 := \text{"tea"}]$, if we can ascertain that the relevant register (in this case r_1) is undefined in the origin state, then the general transition directly subsumes the specific one.

Register Output. The "store and reuse" heuristic also replaces a literal output with the value of a register. For a generalised transition to subsume an ungeneralised one, we need to show that the register holds the original output value in all relevant contexts which can be obtained in the origin state. This is more difficult than it sounds due to the fact that it is not necessary for the register to hold the relevant value in *all* contexts, just in all *relevant* contexts.

Example 6.4.1. Recall Example 5.1.1 in which we try to merge $\mathit{vend} : 0/[o_0 := \text{"tea"}]$ and $\mathit{vend} : 0/[o_0 := r_1]$. To prove direct subsumption, it is sufficient to prove that for all traces which were recognised by the model in Figure 5.1a and brought it to state q_a, that are also recognised by the model in Figure 5.1c and bring it to state q_{ac} leave r_1 with value "tea". This can be checked in SAL by *composing* the two models.

Increment and Reset. The pattern introduced by "increment and reset" is more subtle. This heuristic drops a guard and introduces an update which *mutates* the data state. We end up testing whether a transition of the form $\mathit{coin} : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$ subsumes one of the form $\mathit{coin} : 1[i_0 = n]/o_0 := m$. Neither can account for the behaviour of the other here, as the updates are not *consistent* with each other. This means that the increment and reset heuristic only tends to be successful towards the end of the inference process when it is able to replace many transitions of the form $\mathit{coin} : 1[i_0 = n]/o_0 := m$ at once.

Further Heuristics. If additional heuristics were used that introduced new kinds of transitions to the model, further case proofs would be required to avoid queries to the user. Depending on the difficulty of the proofs, this may not be particularly arduous, but is still a weakness of heuristics in their current form.

6.4.3 A Practical Compromise

While the approach of using a model checker to check properties which imply direct subsumption initially seems attractive, it makes the inference process prohibitively slow for all but the smallest of examples. This is due to the problem of state space explosion experienced by all model checkers. To allow the inference tool to scale to the realistic examples needed to properly evaluate it in Chapter 8, a different approach is needed.

To solve this problem, I reduce the direct subsumption down to a heuristic which deletes each transition in turn and runs the original traces used to build the PTA are still accepted. The justification for this is that this check will intuitively go through in all situations where we have full direct subsumption (by “forall elimination”). In situations where we do not have full direct subsumption but this weaker check still passes, an overgeneralisation is made but we still maintain trace inclusion on the resulting model. This approach is clearly not as formal as the use of a model checker, and only considers individual concrete transitions, but the compromise is necessary to allow models to be inferred automatically in reasonable time.

6.4.4 Code Generator Optimisations

As mentioned above, the code produced by Isabelle’s code generator is not particularly fast or well optimised. This is compounded by the fact that the Scala compiler has very little optimisation for recursive functions, upon which this technique relies heavily. To handle larger case studies more efficiently, certain optimisations had to be applied. These are detailed below. While no formal experiments were run to quantify the speed-up gained from these changes, they had a noticeable impact on the runtime of the software on all but the smallest of examples.

Numbers and Strings

The most obvious optimisation is to, wherever possible, use the target language’s built-in constructs. Isabelle does this to a certain extent by default, already making use of Scala’s built-in `List` and `Option` types. It does, however, like to define its own numeric datatypes to maximise safety. Most numeric datatypes (`int`, `float`, `double`, etc.) are subject to arithmetic overflow which can lead to unexpected differences between the Isabelle formalisation and its executable implementation. Consequently, the default behaviour of the code generator is to define natural numbers (the `nat` datatype) and then redefine integers in terms of this. This does not make for efficient computation or readable code. Fortunately, the `Code_Target_Nat` and `Code_Target_Integer` theories exist in the HOL library.[4] Importing these changes the behaviour of the code generator to instead use the native integer type (in Scala’s case the `BigInt` type is used instead of `int` as these are not subject to overflow) with just a thin wrapper.

The `Code_Numeral` theory file goes a step further and provides additional data types `integer` and `natural` which can be used in place of `int` and `nat` respectively. This reportedly causes the code generator to use the native integer representation without any wrapping at all. This, however, is somewhat fiddly to set up retrospectively, and complicates proofs somewhat, so was not used here.

The default behaviour for strings is to treat them as lists of characters, where characters are represented by 8-tuples of bits. This is not only inefficient but makes code very verbose and totally unreadable. The `String` theory file provides the datatype `String.literal` which is implemented by the native string datatype in the target language. Like the `Code_Numeral` approach, this is somewhat fiddly to retrospectively incorporate into existing theory files, however the benefits in efficiency and readability were deemed worth the effort.

[4]`https://isabelle.in.tum.de/doc/codegen.pdf` (Accessed 13/05/19)

Lists

While the code generator uses the Scala's native `List` type, it does not make use of any natively implemented functions over lists, instead defining its own recursive versions of basic list utilities like `map` and `filter`. When these are used extensively, the lack of recursive optimisation on the part of the Scala compiler becomes painfully apparent. To alleviate this issue, the `code_printing` statement can be used to map calls to certain functions down to their native implementations which are, of course, much faster. I did this with the following functions.

`Cons` was mapped to the native infix “`_::_`” operator, mainly to improve readability.

`rev` was mapped to the native function “`_.reverse`” to reverse a list.

`member` was mapped to the native function “`_ contains _`” to check list membership.

`remdups` was mapped to the native function “`_.distinct`” which removes duplicate items from a list.

`length` was mapped to the native “`_.length`” function which returns list length.

`zip` was mapped to the native infix “`_ zip _`” operator which zips two lists together.

`map` was mapped to the native `map` function which applies a given function to each element of the given list.

`maps` was mapped to the native `flatMap` function which flattens a list of lists to a single list and applies `map` to it.

`null` was mapped to the native `_.isEmpty` function to check list emptiness.

`filter` was mapped to the native `filter` function which returns all elements of a list which pass a given test.

`all` was mapped to the native `_.forAll` function which returns true if all elements of a list pass a given test.

`ex` was mapped to the native `_.exists` function which returns true if any element of a list passes a given test.

`nth` was mapped to the native list indexing function `_[n]` which, when given integer n returns the $(n+1)^{th}$ element of a list.

The final function which I converted to its native implementation was `fold`, which reduces a list down to a single value (e.g. its sum or its maximum). This was a little more complicated as Isabelle implements three different fold functions: `fold`, `foldl`, and `foldr` which are all expressible in terms of the others. Scala, on the other hand, only has direct conversions of `foldl` and `foldr`. Since all three fold functions are expressible in terms of `foldl`, the relevant lemmas were added to the code generator telling it to do this, and the `code_printing` statement was used to replace calls to `foldl` by calls to Scala's native `foldLeft` method.

Further speed-ups can be gained by handling list operations in parallel. Scala has out of the box support for this, so functions such as “map” and “for all” can be replaced by their parallel equivalents for maximum efficiency when using processors with multiple cores. There is a little additional overhead here, as sequential lists must be converted to their parallel counterpart before executing parallel functions, but there is a noticeable speed-up for larger examples.

Another area where efficiency can be gained is sorting. Clearly it makes sense to use Scala's native `sort` method rather than defining a custom one, however this is more complicated than a simple replacement. The reason for this is that, while both Isabelle and Scala have a similar notion of an *ordering* which datatypes can be made a member of by providing “less than” and “less than or equal to” functions which (in Isabelle’s case) satisfy the conditions of a partial (or total) order, the code generator does not make use of this. Consequently, Scala lacks the evidence it needs to be able to natively sort a list of elements. Fortunately, Scala provides the `sortWith` command which allows an arbitrary “less than” function to be supplied. Isabelle defines all such functions in a Scala object called `Orderings`. The `sortWith` function can then be called with the `Orderings.less` function, and the relevant type can be inferred by the Scala compiler.

Sets

The default behaviour of the code generator is to implement sets as lists with a thin wrapper to the native `List` datatype. While the Java Collections Framework has several implementations of sets, my attempts to get the code generator to use them were futile. There are a number of speed-ups and code clarifications to be gained, however.

Firstly, the default code generator setup covers much more behaviour than was necessary for this implementation. True mathematical sets can, of course, be infinite, however this obviously cannot hold if we want to handle them on a real computer. The code generator thus implements sets as lists of elements and infinite sets as *cosets* (lists of non-elements). Since none of the sets used in my implementation are infinite, the code can be simplified to not have to handle cosets. This does little to improve efficiency but does improve code readability.

Finite Sets

Much of my Isabelle formalisation is in terms of necessarily finite sets. The Isabelle `fset` datatype is a subtype of `set` where its members are finite. This is implemented by the code generator as a thin wrapper to `Set` which is itself a thin wrapper to `List`. In order to improve readability and reduce unnecessary function calls, various equivalences can be proven and added to the code generator setup to remove the `Set` wrapper such that both `Set` and `FSet` are thin wrappers to the native `List` datatype. Again, this does little to improve efficiency but does improve readability.

6.4.5 Memoisation

Another way to speed up computation is *memoisation*, where the results of certain computations are stored in a lookup table for later use. I make use of memoisation to store those state merges which have previously failed, such that they need not be attempted again. State pairs which could not be merged remain distinct in the model, meaning that they will score the same on the every subsequent iteration. This means that, without memoising failed state pairs, the inference process builds up a buffer of failed state merges which it must repeatedly try and fail to merge each iteration before it gets to a state pair which has not yet been tried. For larger models, this buffer can grow quite large very quickly, which makes the inference process very slow.

This memoisation is used in both the INFERENCESTEP function, where the highest scoring pair of states is merged, and the RESOLVENONDETERMINISM function, where the destination states of nondeterministic transitions are merged. In both of these functions, if the pair of states to be merged is in the set of failed merges, it is skipped.

The set of failed merges is built by the same functions which use it. When the resolveNondeterminism function is unable to merge a pair of nondeterministic transitions after merging their destination states, that state pair is added to the set of failed merges since, if the transitions cannot be merged now, they will not be able to merged subsequently either. Thus, the merge of their origin states can never be successful. Similar to the inferenceStep function, when the resolveNondeterminism function returns **None**, the pair of states which were to be merged is added to the set of failed attempts.

6.5 Conclusion

Building on the direct subsumption relation from Chapter 5, this chapter presented a technique to infer EFSM models from black-box system traces. I also showed how heuristics which recognise data usage patterns can be used to abstract away concrete values by introducing registers to the model. This enables transitions to be merged when neither directly subsumes the other.

The implementation of my technique is based on my Isabelle formalisation of EFSMs, with the code generator being used to create an executable implementation. For the functions which could not be made executable, I found other solutions, in the end having to compromise the formality of the *directly subsumes* relation in order to allow the tool to run in reasonable time.

Before we can comprehensively evaluate my inference tool, there still remains a hole in the approach which needs filling: the heuristics. The problem here is that the performance of the inference technique is almost entirely dependent on the quality and applicability of the heuristics provided to it. Producing high-quality heuristics often requires some inside-knowledge of the system under inference. If the user has this knowledge already, they are unlikely to require automated inference. Ideally, we would like something more generally applicable. This is the subject of Chapter 7. Once we have this, it is then possible to empirically evaluate the inference technique on some larger case studies. I do this in Chapter 8.

Chapter 7

Using Genetic Programming to Infer Computation

In Chapter 6, I presented an EFSM inference technique based on state merging which uses simple *heuristics* to identify and generalise certain data usage patterns to enable more transitions to be merged. The effectiveness of this technique depends on the user's ability to provide appropriate heuristics for the traces at hand. This means that the user must either have a vast collection of heuristics or have some inside knowledge of how the system works. This is a serious limitation to the applicability of the inference technique from Chapter 6.

Another limitation is the fact that the heuristics are applied at merge time, meaning they can only really work with two transitions at once. If there are lots of transitions in a model which represent the same behaviour, it makes sense to try to generalise them all at the same time. Ideally, it would be good to perform this generalisation as a separate step before we begin merging states, since this allows us to work with the entire model. Additionally, if we can make transitions which perform the same function identical by abstracting away concrete values, this effectively reduces the inference problem back down to classical inference, as we should only need to merge identical transitions.

The work presented in this chapter aims to solve the limitations of the heuristics from Chapter 6 by presenting a more general approach for inferring the output and update functions of transitions, the main contributions being the following.

- An approach based on genetic programming to infer functions which account for sets of input-output pairs where certain inputs are elided.
- A preprocessing technique for the inference process which makes use of this technique to infer output and update functions on transitions in the PTA.
- A heuristic for use during inference which uses GP to infer guard functions to distinguish transitions that exhibit value-dependent behaviour when they cannot be merged.

7.1 Introduction

The task here is to come up with a general way of inferring the functions on transitions which compute output from input. As part of this, it may be necessary to make use of state variables, or *registers*, hence it is also necessary to infer update functions for these as well. Examples 6.3.2 and 6.3.3 in Chapter 6 give a good idea of what we want to achieve here. In situations where we have transitions that exhibit the same kind of behaviour with different literal values, we hypothesise that these transitions may be *instances* of a more general behaviour. If we can identify this behaviour, we can generalise the model such that more states and transitions can be merged. This not only results in a smaller model, but also in one which is better able to predict the behaviour of the system when faced with new executions.

The simple heuristics we saw in Chapter 6 can be thought of as a set of oracles, each providing a suggestion to the inference process, for example "Have you tried storing the input in a register for later use?" or "Have you tried adding this input to that register?". The fact that we must implement a heuristic for each data usage pattern we wish to generalise is a serious limitation of the current approach. Implementing good heuristics requires some understanding of the system at hand, so we really need to know how the system works before we apply inference. To extend the applicability of EFSM inference beyond simple examples, we need a general purpose way to infer output and update functions that does not require us to already have an answer in mind.

As we saw in Subsection 3.6.3, *genetic programming* has already been successfully used in the literature [150] to infer update functions for EFSM transitions. Where the heuristics in Chapter 6 each suggest a single possible solution which may or may not be appropriate, the idea here is to use evolutionary computation to come up with a bespoke solution "on the fly" which works for the data at hand. The advantage of this is that users of the inference tool do not need to have any knowledge about the system under inference to infer a model.

In [150], the GP required comprehensive white-box traces which laid bare the internal variables of the system. What we would ideally like is to work with black-box traces, which only contain information visible to an external observer of the system. This makes the task of inferring functions much harder, as we are effectively eliding some of the inputs to each function.

Example 7.1.1. Table 7.1 shows some input-output pairs for the function $i_0 + r_1$. If the values of both input variables (i_0 and r_1) are known, it is quite easy to work out what the function is. Indeed, existing GP implementations such as [150] are reliably able to infer such functions in a few milliseconds.

i_0	r_1	result
50	0	50
50	50	100
10	0	10
20	10	30
50	10	60

Table 7.1: A set of input-output pairs for the function $i_0 + r_1$.

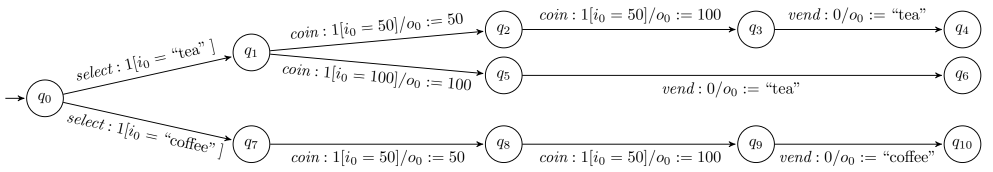
Now consider the case where variable r_1 is elided and we only have access to the values of i_0. We know that r_1 exists and *might* be used in calculating the result, but we do not know for sure either way, and certainly do not have access to its values. How might we go about inferring a suitable function? The problem is now twofold. Firstly, the original "guess the function" problem remains, but is made harder by the fact that we do not have values for r_1. Secondly, because GP relies on evaluating candidate functions to assess their suitability, we must infer suitable values of r_1 such that candidate functions involving it can be effectively evaluated. Of the many successful applications of GP in the literature, for example [96, 25, 107, 150], there do not appear to be any which tackle the problem of elided inputs.

Another noteworthy aspect of the technique in [150] is that it was a *postprocessing* technique. This means that the GP is stuck with whatever model the inference process produces and the inferred functions play no part in the inference process. Here, I propose a *preprocessing* technique to be performed on the proginal PTA before merging. This means that the inferred functions can influence the merging process and is particularly important here since, unlike in [150], my transitions produce outputs meaning that they cannot be merged with other transitions unless the behaviours are consistent. Inferring generalised output functions before attempting merging helps with this because we can generalise different concrete values to the same symbolic function, allowing us to merge more transitions.

The remainder of this chapter is laid out as follows. Section 7.2 first presents a high level overview of what we would like to achieve here. Section 7.3 provides some necessary background needed to understand the details my GP approach for inferring functions from incomplete input sets presented in Section 7.4. Section 7.5 details how this is applied to the inference process. Section 7.6 presents a means of using GP to detect value-dependent behaviour. Finally, since GP is quite computationally expensive, Section 7.7 presents some implementational shortcuts which can be used to reduce the cost.

7.2 Motivating Example

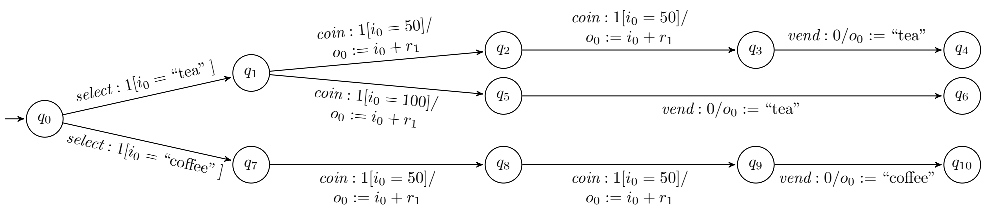
Before delving into the technical details of how GP is applied to the inference process, I will first present an outline of what we want to achieve in terms of the running drinks machine example. Recall that the traces in Figure 6.1 can be transformed into the PTA in Figure 6.3, shown below for convenience. If we wish to generalise the various behaviours in this PTA, we must first divide the transitions into groups that we think might be instances of the same general behaviour. Here, there are three such groups: *select*, *coin*, and *vend*. We now need to obtain output and update functions for each group.



q_0
$select : 1[i_0 = \text{“tea”}]$
$select : 1[i_0 = \text{“coffee”}]$
q_1
$coin : 1[i_0 = 50]/o_0 := 50$
$coin : 1[i_0 = 100]/o_0 := 100$
q_2
$coin : 1[i_0 = 50]/o_0 := 100$
q_3
$vend : 0/o_0 := \text{“tea”}$
q_4
q_5
$vend : 0/o_0 := \text{“tea”}$
q_6
q_7
$coin : 1[i_0 = 50]/o_0 := 50$
q_8
$coin : 1[i_0 = 50]/o_0 := 100$
q_9
$vend : 0/o_0 := \text{“coffee”}$
q_{10}

Figure 6.3: The PTA representing the traces in Figure 6.1.

The transitions in the *select* group do not produce outputs, so no functions need to be inferred. For the *coin* group, there are three distinct transitions: *coin*(50)/[50], *coin*(50)/[100], and *coin*(100)/[100]. To generalise this behaviour, we need a function which accounts for the set of input-output pairs $\{([50],[50]),([50],[100]),([100],[100])\}$. Here, there are two possible output values for $i_0 = 50$, so we cannot express the behaviour purely in terms of inputs. The output must depend on the value of a *register*. If we call this register r_1 and assume that it holds the correct anterior value when each *coin* transition fires, we can use GP to evolve the output function $i_0 + r_1$. This results in the PTA in Figure 7.1.



q_0
$select : 1[i_0 = \text{“tea”}]$
$select : 1[i_0 = \text{“coffee”}]$
q_1
$coin : 1[i_0 = 50]/$ $o_0 := i_0 + r_1$
$coin : 1[i_0 = 100]/$ $o_0 := i_0 + r_1$
q_2
$coin : 1[i_0 = 50]/$ $o_0 := i_0 + r_1$
q_3
$vend : 0/o_0 := \text{“tea”}$
q_4
q_5
$vend : 0/o_0 := \text{“tea”}$
q_6
q_7
$coin : 1[i_0 = 50]/$ $o_0 := i_0 + r_1$
q_8
$coin : 1[i_0 = 50]/$ $o_0 := i_0 + r_1$
q_9
$vend : 0/o_0 := \text{“coffee”}$
q_{10}

Figure 7.1: The PTA from Figure 6.3 with the output behaviour of the *coin* transitions generalised to $i_0 + r_1$.

The function $i_0 + r_1$ only accounts for the input-output behaviours of the *coin* transitions if r_1 holds the correct value at the point that a transition fires. To use this output function, we need to ensure that this is the case. The way we do this is to use GP again to infer *update functions* for the different transition groups. Looking at Figure 7.1, we can see that the *select* transitions need to initialise r_1 to zero. This handles two out of the three cases in our set of *coin* input-output pairs. For the pair $([50], 100)$, which corresponds to the $q_2 \to q_3$ and $q_8 \to q_9$ transitions, we need to update the value of r_1. For both of these transitions, r_1 needs to hold the value 50. Since both transitions immediately follow a *coin* transition, the output from which was 50, we could use the same function, $i_0 + r_1$, as an update.

Finally, the *vend* group contains two distinct transitions. These transitions do not take any input, so the set of input-output pairs looks like $\{([], [\text{“tea”}]), ([], [\text{“coffee”}])\}$. Since there are no inputs at all here, we clearly need to introduce another register to account for the different output values. Let us call this register r_2. As with the *coin* transitions, we must perform a second round of GP to evolve update functions such that our newly introduced register holds the correct values when the *vend* transitions fire. Here, it makes sense to assign the input of *select* to r_2. This results in the PTA in Figure 7.2.

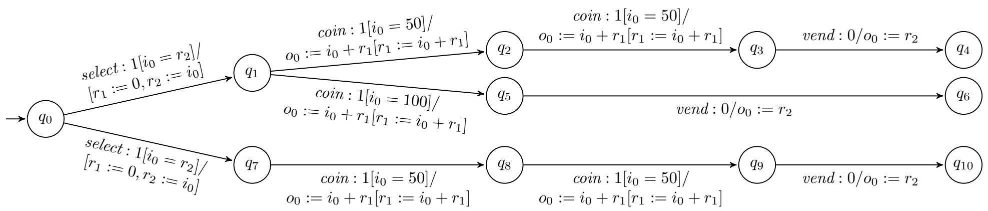

$select : 1[i_0 = r_2]/ [r_1 := 0, r_2 := i_0]$
$coin : 1[i_0 = 50]/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$coin : 1[i_0 = 100]/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$vend : 0/o_0 := r_2$
q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10}

Figure 7.2: The PTA after replacing literal outputs with functions and adding updates where necessary.

Currently, each transition in Figure 7.2 still has its original guard. Ideally, we would like to remove these guards so that the model can respond to events with different input values to those observed in the original traces. Dropping the guards introduces nondeterminism but, like in Subsection 3.3.4, this is not real nondeterminism. It is just a manifestation of the fact that the model contains duplicated behaviour. Thus, like in Subsection 3.3.4, we can resolve this nondeterminism by calling the resolveNondeterminism function to merge states and transitions such that the model becomes deterministic again.

After resolving the nondeterminism, we end up with the PTA in Figure 7.3. This is much transformed from the original PTA in Figure 6.3, but it is still tree-shaped such that traces with a common prefix share a common path through the model so still warrants the name PTA. We are now ready to begin scoring and merging states as detailed in Chapter 6.

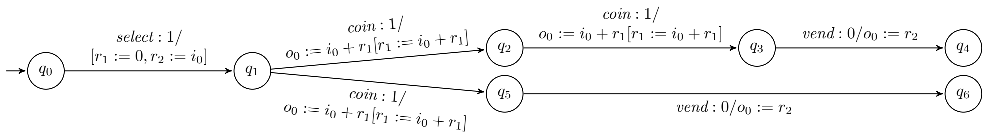

$select : 1/ [r_1 := 0, r_2 := i_0]$
$coin : 1/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$vend : 0/o_0 := r_2$
q_0 q_1 q_2 q_3 q_4 q_5 q_6

Figure 7.3: The PTA after resolving nondeterminism.

We can see from this example that GP effectively performs the role of both the "store and reuse" and "increment and reset" heuristics from Chapter 6. Rather than handing the inference technique a solution "off the shelf" which may or may not be appropriate, GP looks at the sets of relevant input-output pairs and tries to come up with an expression which fits. This means that we could apply GP to any set of inputs and outputs and have it infer a function. Although the functions we see here and in my evaluative case studies in Chapter 8 are relatively simple such that they could reasonably be found through exhaustive search, the use of GP as a search heuristic means that we can find arbitrarily complex functions from a very large search space of potential operators, variables, and constants. Of course, the larger the search space, the more generations the GP will need to find a suitable function.

There are several implementations of GP in the literature, including [150], which can evolve functions for sets of input-output pairs where all the input values are known. We need more than this here, though, since our outputs may depend on the values of *registers* which we cannot see. The task now is to implement a version of GP that can work with incomplete sets of inputs.

7.3 Background

Having shown the role that GP is to play in the inference process, I now provide some necessary background material on the underlying principles.

7.3.1 Evolutionary Algorithms

Genetic programming is a specific application of evolutionary algorithms (EAs), a general purpose problem solving technique inspired by the biological process of evolution. The idea here is that natural resources are limited, so a given population of individuals will become better suited to their environment over many generations. This process is commonly referred to as *natural selection,* and the idea has been successfully used solve many computational problems for which no exact algorithm is known [8, 33, 55].

The class of problems for which EAs are most useful is *combinatorial optimisation* problems, examples of which include route finding, scheduling, and bin packing. Such problems involve rearranging a finite set of entities or operations into an optimal configuration, and often have a very large number of possible solutions. Since these problems are either NP-complete or NP-hard, there is no known way of efficiently computing an optimal solution directly. We are therefore forced to search for one in the vast space of possible solutions. In the context of GP, we are looking to arrange various computational operations into a program that generalises the behaviour exemplified in a given set of input-output pairs.

Evolutionary algorithms help us to systematically explore the search space by directing exploration towards individuals which seem more promising. A *population* of candidate solutions is maintained and evaluated with a *fitness function,* which takes an individual and returns a numerical value representing the suitability of the candidate solution. The fitness function can be viewed as environmental pressure which mimics biological factors such as the availability of food and prevalence of predators. Individuals with a higher fitness are better suited to their environment and are more likely to survive to the next generation.

While EAs have many highly technical and specialised variants, the basic procedure is quite simple. Algorithm 6 shows the high level structure which is common to many EAs. The first step is to generate an initial population of μ individuals. Then, over successive *generations*, this population is evolved by *mating* and *mutating* individuals, keeping only those with the best *fitness* and discarding the rest. In this way, the average fitness of the population is likely to improve over successive generations. This means that if we run the GP for long enough, we are likely to discover an optimal solution if one exists.

Algorithm 6 The general structure of a $\mu + \lambda$ genetic algorithm.

```
INITIALISE population with $\mu$ random individuals
EVALUATE each individual
repeat
    CROSSOVER pairs of parents to create $\lambda$ offspring
    MUTATE the resulting offspring
    EVALUATE the offspring
    SELECT $\mu$ individuals for the next generation
until Optimal solution or stopping condition
```

In the main loop, pairs of parents are *crossed over*[1] to form λ new offspring. These offspring are then *mutated* slightly. This simulates the random genetic mutations that occur during natural reproduction. The mutated offspring are then evaluated and added to the population. The best μ individuals are then chosen to go on to the next generation. This kind of algorithm is called a $\mu + \lambda$ algorithm since, when the λ offspring are added to the existing population of μ individuals, the new size is $\mu + \lambda$.

The $\mu + \lambda$ evolution strategy is one of many in the literature. It is referred to as a *steady state* strategy as individuals remain in the population for many generations if their fitness is sufficiently high. New individuals only enter the main population if their fitness value is higher than the current worst individual, so the average fitness increases monotonically over time.

The counterpart to steady state algorithms is *generational* algorithms. Here, the old population is replaced by the new one, regardless of fitness. Thus, the average fitness of the population can go down between generations. More complex evolutionary strategies such as the $1 + (\lambda, \lambda)$ EA [50] exist in the literature, as well as parameterless strategies such as in [75], but these are not well-established. The intricacies of the various evolutionary strategies are outside of the scope of this work, so these will not be discussed further here.

Aside from basic evolutionary strategy, the main two areas of variation between EAs are *crossover*, which simulates the biological process of sexual reproduction, and *mutation* which simulates genetic mutation that occurs as part of natural reproduction. These operations affect how the search space is explored so have a dramatic impact on the performance of an algorithm.

7.3.2 Genetic Programming

We have already seen in Subsection 3.6.3 that *genetic programming* can be used to infer functions that account for sets of input-output pairs. GP was introduced by Koza in [96] as a way of applying evolutionary algorithms to discover computer programs that produce the expected outputs when presented with the given inputs. Program fitness is evaluated using a *training set* of inputs mapped to their corresponding output. The result of a successful GP run is a function which is not only correct with respect to the training set, but also generalises to other inputs. Unlike conventional machine learning techniques, the solutions produced here are readable such that we can understand *how* inputs are transformed to produce the output.

[1]EAs which use crossover are commonly referred to as *genetic algorithms*, although this is not universal.

7.3.3 Individual Representation

When applying evolutionary algorithms to a new problem, the first thing we must consider is how individuals are represented. In standard GP, individuals are represented as *syntax trees* with *terminal* and *non-terminal* nodes rather than lines of code. Representing individuals like this allows us to manipulate them with crossover and mutation much more easily.

Terminal nodes are leaf nodes in the parse tree. These are either literal constants or variables to be evaluated in a given *context.* Non-terminals are branch nodes in the tree and correspond to functions. These can be arithmetic and boolean operations, string concatenation, conditional branches, etc. As an example, consider the expression $5 + (i_0 \times r_1)$, the syntax tree for which is shown in Figure 7.4. Here, the terminals are 5, i_0, and r_1. The non-terminal are $+$ and $\times$.

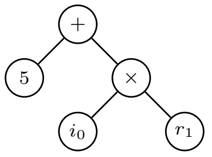


Figure 7.4: The parse tree for the expression $5 + (i_0 \times r_1)$.

It is important to note that syntax trees have no fixed size and can grow arbitrarily large during evolution, unless an artificial bound is imposed. This can make for functions which are difficult to read and understand, but does mean that we do not need to know the size of the function in advance. Several variants of GP exist that do not use a tree representation. Examples include Linear GP [22], Cartesian GP [115], and Geometric Semantic GP [117]. Since it is the most established technique, tree-based GP will be the focus here.

7.3.4 Fitness Evaluation

Perhaps the most important thing to consider when trying to solve any problem is how to evaluate solutions. This is particularly important when using an evolutionary algorithm since we need to be able to compare the suitability of different individuals. This is done through the use of a *fitness function* which takes in an individual and returns a numeric value representing the suitability of that individual. The biological analogy here is that the fitness function represents how suited an individual is to the environment. It can also be thought of as a measure of the acceptability of a given solution. Because of this, it is vital to ensure that the fitness function is an accurate evaluation of the desired outcome. If not, evolutionary algorithms often find “creative” ways of circumventing the intended objectives to produce individuals which optimise the fitness function without actually solving the problem [99].

We can also think of the fitness function as an *error function,* or the distance of a given candidate from the optimal solution. For tree-based genetic programming, the general approach is to have an input-output table like Table 7.1 called a *training set.* For each row (or *target*) in the table, the candidate solution is evaluated under the prescribed input valuations and the distance between the expected value and the actual value is measured.[2] This produces a list of distances, one for each target, which can then be aggregated to produce a single numeric value such that the best individual in the population has the lowest value.

[2]This assumes the existence of distance function between the expected and actual values. The definition of this will vary depending on the their datatype. For numeric values, we can use their arithmetic difference. For strings, we can use metrics such as Levenshtein distance. More complex data types are not considered here.

There are various ways to aggregate the list of distances, the most basic being to simply sum them. This approach is not particularly “fair”, however, as it may give certain targets a disproportionate effect on the fitness value. If the expected evaluation of one particular target is very different to the rest, this will lead towards solutions which satisfy that target at the expense of the others. For example, in Example 7.1.1, the second row has an expected value of 100. Since this is somewhat larger than the other expected values, the GP will yield functions which produce values closer to 100 than to any other expected value. This is mitigated in [150] by using the *root mean square error* (RMSE) to spread the error more evenly across the elements of the training set so certain elements are not unduly optimised for. The RMSE is calculated as follows, where n is the number of entries in the training set.

$$RMSE = \sqrt{\frac{\Sigma_{t=0}^{n}(expected_t - actual_t)^2}{n}}$$

7.3.5 Crossover

The crossover operator determines how individuals from the current population are *recombined* to form new individuals. This is meant to simulate the biological process of sexual reproduction. As in biology, the most common number of parents is two, but this need not be a hard limitation here if a sensible strategy can be found for recombining more than two parents. This, of course, is highly dependent on how individuals are represented.

While [96] did not use crossover, studies since then [133] have shown that crossover is a beneficial operation and it is used in numerous implementations in the literature including [150], upon which I base my own implementation. For tree-based GP, crossover consists of exchanging subtrees between individuals. Two parent individuals are chosen and a random subtree is swapped between parents. An example is shown in Figure 7.5, in which the expressions $5 + (i_0 \times r_1)$ and $(3 - r_1) \times i_1$ are the parents. The nodes r_1 and $3 - r_1$ have been selected as the crossover points, making the offspring $5 + (i_0 \times (3 - r_1))$ and $r_1 \times i_1$ respectively.

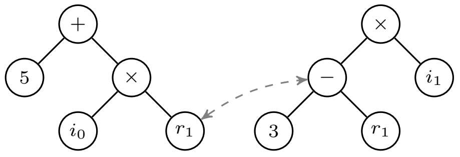


Figure 7.5: Tree-based crossover.

7.3.6 Parent Selection

Before the crossover operator can be applied, the parent individuals must be selected from the population. One way to select parents is to simply pick them arbitrarily, with or without replacement, from the population. This is not intuitively sensible, though, as it is likely that crossing over better individuals will lead to better offspring. Another way of obtaining such parent individuals is to take the best n individuals from the population and recombine them in various ways until λ children have been created. Most algorithms do not do this, though, because it tends to lead to *premature convergence.*

A common compromise between these two methods of selection is *tournament selection.* Here, n pools of m individuals are created by selecting from the population at random. Next, the individual from each pool with the highest fitness is selected to be a parent. A popular configuration is $m = n = 2$, which has been shown to perform well on a number of problems [114]. This configuration is commonly referred to as “binary tournament selection”.

7.3.7 Mutation

Mutation operators are meant to simulate the small changes in DNA that occur during natural reproduction. This allows new characteristics to enter the population. Where synthetic mutation differs from its biological counterpart is that natural mutation is completely random (and often not beneficial) where most synthetic mutation operators are built intelligently, with a specific purpose in mind.

As with crossover, mutation is highly dependent on the problem at hand and how individuals are represented. There are three main mutations which can be applied to individuals represented as trees. These are illustrated in Figure 7.6 and are defined as follows.

Insertion (INS) replaces a leaf node with a new random subtree, effectively *inserting* a new node into the tree.

Substitution (SUB) replaces a branch node with another branch node of the same type, e.g. substituting a + for a −.

Deletion (DEL) replaces the individual with one of its subtrees, effectively promoting a random branch node to the root.

The three operators are brought together in the HLV-Prime mutation operator introduced in [54] which selects uniformly at random one of the three operators to apply.

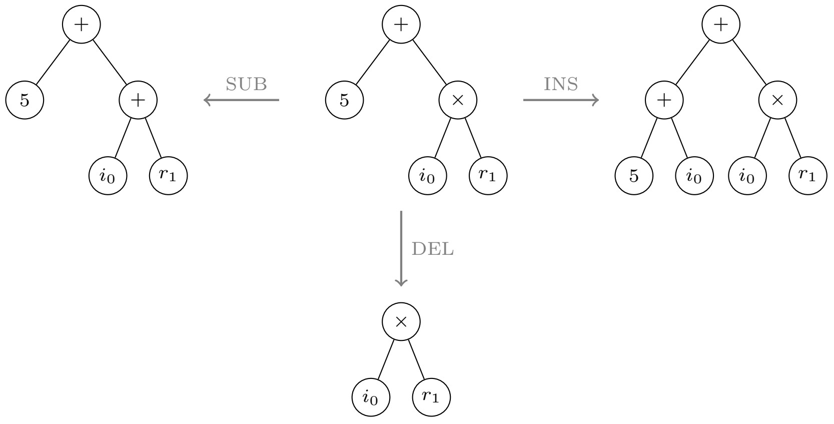


Figure 7.6: The three mutation operators of HLV-Prime.

7.3.8 Bloat Control

The candidate solutions produced by GP tend to increase in size as the algorithm proceeds, often without this growth providing much benefit. The same effect can be observed in natural evolution too, with some species having body parts which serve no useful function, for example the appendix in humans. Such biological features are referred to as *spandrels.* In GP this phenomenon is known as *bloat,* and affects most EAs where individuals do not have a fixed maximum size. A number of different techniques exist in the literature to control this.

Steady-State Iterations - While bloat management is often not the primary reason for using a steady-state algorithm, it does assist in this up to a point. With a steady-state algorithm, children are added to the population such that it becomes oversized, and the weakest are killed. This means that individuals with high fitness stay in the population over multiple generations. Since candidate solutions tend to grow over time, older individuals will be smaller. Thus, keeping them in the population will result in slower growth of the average solution size. With generational algorithms, where the new population is made up entirely of child individuals, there are no older individuals in the population so the average size of individuals will grow much faster.

Maximum Depth Restriction - Perhaps the simplest bloat management technique is an artificially imposed maximum size for individuals. Here, children with a tree depth larger than a pre-specified size are rejected from the population. This guarantees that the final solution will be small but requires some idea of the size of the optimal solution, which may not always be known even approximately.

Parsimony Pressure - Given that there are potentially infinitely many ways of expressing any given function, it is extremely likely that at some point there will be multiple individuals in the population with the same fitness. Here, there is no hard limit on the size of individuals, but size is used to break ties in fitness such that smaller individuals are preferred. While certain GP approaches use candidate size as part of the fitness function [96, 25], this approach can lead to a function's size overwhelming its correctness. That is to say that smaller functions which produce less correct outputs score better than larger functions which are better able to explain the behaviour. This is especially true during the later stages of the algorithm when a population's fitness values are converging.

An alternative is to make tree size a secondary objective alongside correctness. This technique has mixed results in the literature [18, 43, 57]. The problem again here is that we first and foremost want *correct* solutions. It is not acceptable to compromise a candidate solution's correctness such that it might become smaller in the same way that this kind of trade-off is acceptable in truly multi-objective situations.

Another approach, first proposed in [107], is called *lexicographic parsimony pressure.* Here, correctness is the only metric used in the fitness function, with size only coming into play only if it is necessary to decide between two individuals with equal fitness.

Expression Simplification - This approach involves exploiting mathematical identities to remove redundant nodes from an individual's parse tree. For example, the expression $x+0$ is equivalent to $x+y-y$, both of which are equivalent to just x. Simplification is used in [132], which reports strong results, but [79] warns that it might lead to premature convergence. The reason for this is that the population may be filled with semantically equivalent but syntactically different individuals with the same fitness which, when simplified, become identical. This then leads to a high number of duplicates in the population, causing the algorithm to get stuck in a *local optimum* from which it is difficult to escape.

7.4 Genetic Programming with Latent Variables

Having covered the basics of how GP can be used to infer functions which relate sets of input-output pairs, we can now move on to the problem at hand. We would like to automatically infer output functions for our EFSM transitions from black-box traces. We cannot apply existing GP techniques to do this as the outputs of certain actions may depend on the values of internal system variables which are not visible. They are *latent*. Existing techniques are not designed to cope with this. We therefore need a new technique to infer functions to relate sets of input-output pairs where not all the inputs are known. This section presents such a technique.

Algorithm 7 shows the outer loop of my algorithm. The approach is very similar to the standard $(\mu + \lambda)$GA, detailed in Subsection 3.6.3, and existing GP methods. Indeed, my implementation builds on the one from [150].[3] An outline of the top-level behaviour is as follows.

Algorithm 7 Outer loop of my GA.

```
function EVOLVEEXPRESSION($trainingSet$, $size$, $depth$, $generations$)
    $population \leftarrow$ GENERATEPOPULATION($size$, $depth$, $type$)
    EVALUATEPOPULATION($newIndividuals, trainingSet$)
    $bestIndividual \leftarrow$ CHOOSEBEST($population$)
    for $i < generations$ do
        for $i < numCrossovers$ do
            $(p_1, p_2) \leftarrow$ SELECTPARENTS($population$)
            $(c_1, c_2) \leftarrow$ CROSSOVER($p_1, p_2$)
            $population \leftarrow population \cup \{c_1\}$
        for $i < numMutations$ do
            $population \leftarrow population \cup \{$MUTATE($population[random]$)$\}$
        SIMPLIFYPOPULATION($population$)
        EVALUATEPOPULATION($population$)
        REMOVEDUPLICATES($population$)
        if $population.size > size$ then
            $population \leftarrow$ REMOVEWEAKEST($population$)
        if $population.size < size$ then
            $newIndividuals \leftarrow$ GENERATEPOPULATION($size - population.size$)
            EVALUATEPOPULATION($newIndividuals, trainingSet$)
            $population \leftarrow population \cup newIndividuals$
        $bestIndividual \leftarrow$ CHOOSEBEST($population$)
        if $bestIndividual.fitness = 0$ then
            return $bestIndividual$
    return $bestIndividual$
```

The first stage of any genetic algorithm is to generate an initial population of individuals. I do this by randomly combining terminal and non-terminal operators. The next stage of the process is to evaluate these individuals according to the fitness function. Here, each candidate expression is evaluated according to the input-output pairs in the training set. A key challenge here is determining how to evaluate functions which involve *latent variables*, since their values do not appear in the training set. The solution to this problem is presented in Subsection 7.4.2.

[3]Openly available at `https://github.com/neilwalkinshaw/mintframework`. (Accessed 13/02/20)

The main loop of the algorithm iterates for a fixed maximum number of *generations.* Here, new individuals are generated through crossover and mutation, added to the population, and the weakest individuals are discarded to keep the size of the population constant. After generating new individuals, I *simplify* the new population to remove redundant nodes from the candidate expressions as discussed above.

The next step is to evaluate the simplified population with the fitness function and discard those individuals with the lowest fitness so that the size of the population remains constant. Before doing this, however, we must take account of the fact that simplification often results in a population which contains duplicate individuals. This inhibits the exploration of the search space, so it is much better if we can keep the population diverse. To this end, I remove duplicate individuals from the population before discarding the individuals with the lowest fitness. If the population contained many duplicates, this step may make the population too small. In this case, new random individuals are generated to fill it back up to the correct size before continuing on to the next generation.

7.4.1 Initial Population Generation

Before the GP is run, we first need to obtain sets of *terminals* and *non-terminals.* Terminals are either constants such as 5 and "tea", or variables such as i_0 and r_1. In the context of EFSM inference, these values can be taken from the traces as detailed in Section 7.5. Non-terminals are functions such as $\_+\_$, $\_-\_$, and $\_\times\_$. In the case of this implementation, these are the only functions currently supported by the `aexp` datatype from Chapter 4 which is used to express outputs and updates. That is not to say that we could not include more complex functions such as exponentials, logs, or even hashing functions, although additional features such as password *salting* mean that we are unlikely to be able to use GP to crack any major security protocols.

Since the aim here is to infer functions that can use latent variables (i.e. EFSM *registers*), variables are tagged with whether they are latent. If a variable is latent, this means that its value does not appear in the traces or the set of input-output pairs. When evaluating expressions, we must find a suitable value for it to take. This is explained in more detail in Subsection 7.4.2.

Algorithm 8 shows how the initial population is generated. The standard process is to randomly combine terminals and non-terminals of a given type until a maximum tree depth is reached. If the set of non-terminals which return elements of the specified type is empty or the maximum depth is less than two, a random terminal is chosen. Otherwise, either a random terminal or a random non-terminal is returned. In Algorithm 8, the probabilities of choosing a non-terminal or a terminal are 0.7 and 0.3 respectively. These probabilities were chosen arbitrarily such that it is more probable that a non-terminal is chosen and the tree will be extended further. Whether and how much these probabilities affect the final output of the algorithm is useful future work, but is somewhat outside of the scope of the project.

For non-terminal nodes, the CREATEINSTANCE function in Line 10 makes recursive calls to GENERATERANDOMEXPRESSION to generate appropriate child nodes of height $maxD - 1$ in the same way. This needs to be specific to each operator such that the correct number of children with the correct respective return types can be generated.

The interesting part of Algorithm 8 is the `while` loop on line Line 17 that tries to ensure that the initial population is made up of distinct individuals. This is not featured in [150], although it is reasonably common practice in the field of evolutionary computing in general to stop the initial population becoming clogged with duplicate individuals, which can cause premature convergence. If the number of terminals and non-terminals is sufficiently large, this occurs naturally, but if there aren't many, we need to give the algorithm as much help as possible to generate a diverse initial population. It is necessary, though, to provide a fixed *TIMEOUT* for this as, without it, population generation will not terminate if the specified population size exceeds the number of possible individuals.

Algorithm 8 Initial population generation.

```
function GENERATERANDOMEXPRESSION($depth$, $type$)
    $nonTerms \leftarrow$ NONTERMINALS($type$)
    $terms \leftarrow$ TERMINALS($type$)
    if $nonTerms = \emptyset \vee maxD < 2$ then
        return SELECTRANDOMTERMINAL($terms$)
    else
        if $random() > 0.7$ then
            return SELECTRANDOMTERMINAL($terms$)
        $selected \leftarrow$ SELECTRANDOMNONTERMINAL($nonTerms$)
        return $selected$.CREATEINSTANCE($maxD - 1$)
function GENERATEPOPULATION($size$, $depth$, $type$)
    $population \leftarrow \{\}$
    for $i \in [0 \ldots size]$ do
        $instance =$ GENERATERANDOMEXPRESSION($maxD + 1$, $type$)
        if NONTERMINALS($type$) $\neq \emptyset$ then
            $iteration = 0$
            while $instance \in population \wedge iteration < TIMEOUT)$ do
                $iteration$++
                $instance =$ GENERATERANDOMEXPRESSION($maxD + 1$, $type$)
        $population \leftarrow population \cup \{instance\}$
    return population
```

Example 7.4.1. Consider the situation with integer terminals 0, 1, and 2, and non-terminals $+$ and $-$. For a maximum tree depth of 2, there are only 21 individuals which exist. If the specified population size is greater than 21, without a *TIMEOUT,* the `while` loop on line Line 17 would never terminate since the population will contain every feasible individual. After this point, only individuals which are already in the population could possibly be generated.

It is important to note that individuals in the initial population are left as they are generated without any effort to control bloat. Bloat control is only employed after crossover and mutation have been applied. This is to further help to avoid premature convergence as per [79].

7.4.2 Fitness Evaluation

Having generated the initial population, the next thing we must consider is how to evaluate individuals. In tree-based GP, the fitness of an individual is determined by evaluating the expression with the different inputs in the training set and comparing the result with the expected output. This yields a set of values which can be aggregated together to produce a single numerical value that represents the candidate solution's suitability, with better individuals having lower values.

Example 7.4.2. Consider again the input-output pairs from Table 7.1 and the expression $i_0 - r_1$. Clearly this candidate is not correct, but we need some way of quantifying *how* incorrect it is. Table 7.2 shows the evaluation of the function compared to the expected values, and the difference between these values. Aggregating the differences by summing them together gives an error (or fitness value) of 160. This is a quantification of how well $i_0 - r_1$ accounts for the observed behaviour, and allows us to compare it to other candidate expressions.

i_0	r_1	expected	actual	distance
50	0	50	50	0
50	50	100	0	100
10	0	10	10	0
20	10	30	10	20
50	10	60	20	40

Table 7.2: Evaluating the fitness of the expression $i_0 - r_1$ for the input-output pairs in Table 7.1.

The problem here is that not all input values are known. In software systems, the output of a function may depend on the value of an internal variable which does not appear in the traces. These correspond to *registers* in our EFSM models, the values of which are also not directly observable. If, as in Example 7.1.1, r_1 corresponds to such a variable, evaluating an expression which uses it suddenly becomes extremely difficult as we do not know what value it holds.

A naive way of solving the problem of unknown variable values would be to simply use a default constant value. The problem with this approach is that latent variables then effectively act as constants and there is no benefit to using them. The main application of latent variables is to allow the inference of expressions which relate input-output pairs where the inputs alone are insufficient to do so. For example, there does not exist an expression purely in terms of i_0 and constants which explains the behaviour described by Table 7.1 because there are three different possible outputs when i_0 holds the value 50. Unless we have a truly nondeterministic system, there must be at least one other variable involved.

We want to allow latent variables to act as a "fiddle factor" to allow expressions to evaluate correctly. It is therefore important to allow them to change their value on a per-target basis, just as any other input can. If we only allow one latent variable per expression, we could, for each target, solve the expression for the latent variable and use that value in its evaluation. For example, for the candidate expression $i_1 - r_1$ from Example 7.4.2, if we know that the expected result is 50 and i_0 holds value 50, then we know that r_1 must hold the value zero. The problem with this approach is that it allows every expression to evaluate correctly every time. Thus, every expression involving a latent variable has perfect fitness. It also limits us to only being able to use one latent variable per expression, which we may not wish to be constrained to.

We need something in between a static default value and allowing latent variables to magically make every expression evaluate correctly. The approach I take here is to have a set of predefined values for latent variables from which the best value is used. These values come from the set of terminals with which the GP was initially provided. In the context of EFSM inference, these values come from the inputs and outputs recorded in the traces. Working under the assumption that registers get their values from inputs, and can manifest as part of outputs, the set of constant terminals provides a reasonable place to look for potential values of latent variables. This is the premise behind the CALCULATEDISTANCE function in Algorithm 9.

The CALCULATEDISTANCE function takes an individual and a target. The minimum distance is first initialised to infinity. If there are no latent variables, the expression is simply evaluated for the current target, and the distance between the expected and actual values is calculated. If the expression makes use of latent variables, we wish to find the valuation which gives us the least distance between the expected and actual values. For each latent variable, we iterate through the set of possible values, searching for the valuation with the smallest distance between the expected and actual values. The possible values of latent variables are the literal constants of the correct type from the set of terminals.

Algorithm 9 Fitness evaluation of expressions involving latent variables.

```
 1: function CALCULATEDISTANCE(individual, target)
 2:     $minDistance \leftarrow \infty$
 3:     $latent \leftarrow$ LATENTVARS($individual$)
 4:     if $latent = \emptyset$ then
 5:         $minDistance \leftarrow$ DISTANCE(EVALUATE($individual, target$), EXPECTED($target$))
 6:     for $var \in latent$ do
 7:         for $value \in var.type.values$ do
 8:             $distance \leftarrow$ DISTANCE(EVALUATE($var, value, individual, target$), EXPECTED($target$))
 9:             if $distance < minDistance$ then
10:                 $minDistance \leftarrow distance$
11: function EVALUATE(individual)
12:     $mistakes \leftarrow 0$
13:     $distances \leftarrow []$
14:     $latent \leftarrow$ LATENTVARS($individual$)
15:     $totalUnusedVars \leftarrow$ TOTALUSEDVARS($trainingSet$) $\setminus$ VARSINTREE($individual$)
16:     for $target \in trainingSet$ do
17:         $minDistance \leftarrow$ CALCULATEDISTANCE($individual, current$)
18:         $distances \leftarrow minDistance \# distances$
19:         if $minDistance > 0$ then
20:             $mistakes{+}{+}$
21:     $fitness \leftarrow mistakes +$ RMSE($distances$)
22:     if $totalUnusedVars = \emptyset$ then
23:         return $fitness$
24:     return $fitness + latent.size()$
```

The EVALUATE function in Algorithm 9 shows how CALCULATEDISTANCE is used to evaluate candidate expressions. The **for** loop spanning lines 16 - 20 calls the CALCULATEDISTANCE function for each target and records these distances. If the distance is greater than zero, this indicates that the expression is not correct for that particular target and the variable *mistakes* is incremented. The fitness of the individual is then the number of mistakes plus the aggregated distance values. This is calculated using their RMSE, as detailed in Subsection 7.3.3.

Most tree-based GP simply has the fitness value being the aggregated distances. Here, I additionally incorporate the number of incorrect targets. This gives a bonus to functions which evaluate correctly for a large proportion of candidates in the training set, and helps to stop the GP from favouring functions with a small error for every target in the training set over those which evaluate correctly for most targets, but produce a larger error for only a few.

Line 22 tests to see whether the individual uses all the available non-latent variables. If so, the fitness is simply returned as is. If the candidate uses only some of the available non-latent variables, the number of latent variables it uses is added to the fitness value. For example, if there are two input variables and the register r_1 is a latent variable, the expression $i_1 + r_1$ incurs a penalty of +1 to the fitness value because it makes use of the latent variable without using the non-latent variable i_0. The function $i_0 + i_1 + r_1$ does not incur a latent variable penalty, however, because all the available inputs are used.

The justification for this is that it is expected that transitions will use their inputs in some way, otherwise they need not be there. We want to penalise the use of latent variables when they are not necessary such that they are used as a last resort only. If the GP is able to use them without penalty, it often discovers that simply outputting the content of a latent variable always has the potential to yield the correct answer. This is an example of the problem mentioned in Section 7.3, where EAs find ways to maximise fitness without solving the problem, and is not a particularly useful or desirable solution in most circumstances. Even when the algorithm has discovered that outputting the content of a latent variable always has the potential to be correct, we want to keep searching for a solution which takes the inputs into account. This does mean that, in situations where this loophole is the only potentially correct answer, the algorithm can never achieve perfect fitness but, since the algorithm runs for a fixed maximum number of generations, this will not stop it from terminating.

7.4.3 Mutation

The original mutation operator in [150] is simply the SUB operator of HLV-Prime shown in Figure 7.6 with the ability to generate new random constants. While [150] reports good results with this, preliminary experiments revealed that something more complex is needed here. Consequently, I implemented my own mutation operators, taking inspiration from HLV-Prime.

HLV-Prime is a popular general purpose mutation operator in tree-based evolutionary algorithms, but it only effectively operates over non-trivial trees. If the individual to be mutated consists solely of a single terminal root node, HLV-Prime can only insert or substitute. In the context of such trivial trees, the substitution operation effectively generates an entirely new individual. This is not really what mutation is about. In evolutionary algorithms, mutation is meant to simulate the small changes in DNA which occur during biological reproduction. Note the phrase "*small* changes". Substituting subtrees in a larger individual is in keeping with the idea that mutations should be small, but creating an entirely new individual is simply not appropriate. Thus, I made the decision to treat trivial trees separately from non-trivial ones.

Trivial Trees

For trivial trees made up of a single terminal root node, the INS and SUB operations of HLV-Prime are disproportionate to the size of the individual. They are not in keeping with the idea that mutations represent "small" changes in DNA that occur during natural reproduction. If the node is a constant literal value (for example the number 5 or the string "potato"), a more appropriate operation is to *fuzz* according to its datatype as follows.

Double A random value between -1 and 1 is added to the current value.
Integer The value is incremented or decremented by 1.
String Either the first letter is removed or a single character is added to the end.
Boolean The value is negated.

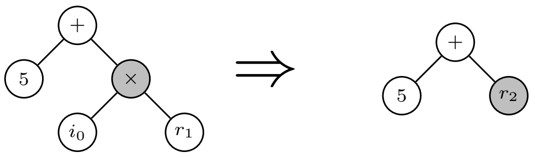
In addition to fuzzing the values by small amounts, preliminary experiments revealed that it is extremely helpful to be able to replace a terminal with another value of the same type from the set of terminals used to generate the initial population. The reason for this is that it helps to direct the search towards values which are known to be significant. This is obviously not applicable to boolean constants, though, as there are only two values to pick from. The implementation of [150] also allows new constant values to be generated at random. While this does not appear to be particularly harmful, because the field of possible values is so large, it is also extremely unlikely to be useful. Consequently, I consider it to be a wasted operation, and did not include this functionality in my own implementation.

If the value is a variable (for example i_0, or r_2) rather than a constant, fuzzing cannot be applied so it is mutated by the INS strategy from HLV-Prime, which is the only strategy detailed here which is appropriate for both terminal and non-terminal root nodes.

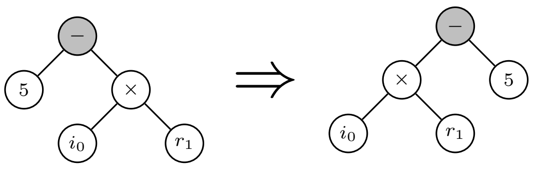
Non-trivial Trees

In addition to the three operations from HLV-Prime, I defined three more operations for non-terminals to bring the total to six possible mutation operators which are selected from uniformly at random. These are defined as follows.

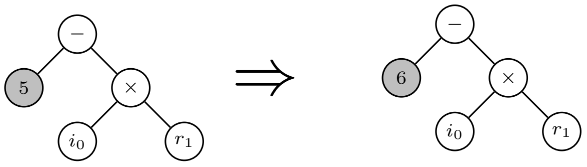
Random Change of Node to Terminal Here, a random node in the tree is changed to a random terminal. The idea here is to act as a means of randomised bloat control and help to remove spurious subtrees added by the INS and SUB operators.



Reversing Child Order Here, the order of the children is reversed. This obviously has no semantic effect on commutative operators like plus but can dramatically change the output of non-commutative functions like minus.



Fuzz a terminal Here, a random leaf node is chosen and *fuzzed* as discussed above.



In contrast to most genetic algorithms, which only apply one mutation operation per individual per generation, I apply up to three in sequence with a probability of 0.5 (after the first operation), so an individual could, for example, have a node substituted, a node added, and a terminal fuzzed. Since the probability of a node having three mutation operators applied is quite small ($1 \times 0.5 \times 0.5 = 0.25$), this acts a little like the fast mutation operators in [51], the idea being that mutations are mostly quite small but are occasionally very big. This helps to escape local optima and avoid premature convergence.

There are, of course, any number of potential mutation operations for candidate expressions, with the precise details of this being somewhat outside the scope of this project. Here, I simply aim to minimally extend the work of [150] to allow it to infer functions involving latent variables such that I can use it as part of the preprocessing technique I present in the next section. Since the original mutation operator used in [150] did not lead the algorithm to converge on a suitable expression acceptably quickly, I sought to extend the range of operations and capture the intuition that mutation operations should only make small changes to individuals. The operators presented here are either obvious modifications to HLV-Prime or ad-hoc additions which improved the outcome of preliminary experiments. A detailed investigation into the effect of individual mutation operations falls somewhat outside the scope of this thesis and remains desirable future work.

7.4.4 Types

The issue of types has thus far been somewhat glossed over. In fact, building expressions which are type correct, and maintaining this through crossover and mutation, is a challenge in itself. To do this, I give each operator a type signature, similar to Haskell functions. For example, the plus operator has the type signature [INT, INT, INT]. It takes in two integers and returns an integer. Thus, the two subtrees of the operator must also return integers.

Because each operator has a well-defined type signature, it then becomes relatively straightforward to maintain type correctness. When generating random individuals, we know what return type each node needs to be. We can also choose crossover points which maintain type correctness, and only apply mutations which are type correct. For example, we cannot swap the children of a node if they have different return types because this would not type check. This becomes very important when we attempt to infer guards in Section 7.6.

7.4.5 Bloat Control

The candidate solutions produced by genetic programming tend to increase rapidly in size as the algorithm proceeds, without this growth providing much benefit. As mentioned in Subsection 7.3.8, the use of a steady-state algorithm has the added benefit of helping to reduce the rate of solution growth, but this alone is not enough. I employ a number of different techniques from the literature to help reduce bloat as much as possible.

Maximum Depth Restriction

Much of the work in GP follows the technique proposed in [96]. Here, children with a tree depth larger than a pre-specified size are rejected from the population. I employ this technique during the generation of the initial population, where individuals are given a hard maximum size. As discussed in Subsection 7.4.1, individuals are generated randomly such that they may be smaller than this maximum size but they are guaranteed not to be larger. Since the size of individuals in the population tends to grow over time, imposing an initial hard size limit helps to ensure that smaller expressions are explored. It also stops the system from continually generating non-terminal nodes as it could if the size of individuals was unbounded, thus guaranteeing termination of individual generation.

No maximum depth restriction is applied during crossover and mutation, however, as there is no reliable way to determine in advance what this should be, and it is obviously undesirable to apply arbitrary restrictions too aggressively. In general, it seems that the other bloat control strategies that I use are enough to ensure that expressions do not grow unacceptably large during evolution.

Lexicographic Parsimony Pressure

Given that there are potentially infinitely many ways of expressing any given function, it is extremely likely that at some point there will be multiple individuals in the population with the same fitness. Under such circumstances, we can break ties in fitness using *parsimony pressure* as discussed in Subsection 7.3.8. Here, the correctness of a candidate solution is the only metric used in the fitness function. Size only comes into play when we need to decide between two individuals with equal fitness.

We cannot apply a similar technique to penalise unnecessary register usage, however, as this would leave the only fitness metric being the aggregated distance between expected and actual values. Since the "cheats'" solution of simply returning the value of a fresh register is always an optimal solution, removing the fitness penalty for introducing registers would allow the GP to terminate as soon as this individual enters the population. Thus, the penalty for using registers must be part of the fitness function so that the algorithm keeps searching for expressions which can explain the outputs in terms of inputs.

Expression Simplification

This approach involves exploiting mathematical identities to edit an individual's parse tree and remove redundant nodes which, if allowed to persist over many generations, lead to very bloated final solutions. In my implementation, I make use of Z3 [45] to simplify expressions after crossover and mutation. The Z3 simplifier works by applying a set of bottom-up rewriting rules to a given expression and is quite simple to use. Expressions are converted from my representation to Z3's representation, simplified, and converted back to my representation.

The simplifier does not always return the smallest representation of a given expression (for example, it chooses to represent the expression $r_1 > 51$ as $\neg(r_1 \leq 51)$), but it is deterministic in its simplification (up to commutativity) such that equivalent expressions simplify to the same thing. Since it does not impose an order on arguments, expressions like $r_1 + i_0$ and $i_0 + r_1$ remain distinct even though they are equivalent.

Simplification is used in [132] and, to a lesser extent, in the original implementation of [150], both of which report strong results, but [79] warns that it might lead to premature convergence. To mitigate this, as discussed at the start of this section, duplicate individuals are removed from the population after simplification and new random individuals are generated to replace those lost. This has the effect of maintaining a diverse population to help escape local optima.

7.5 Application to EFSM Inference

The simple heuristics in Chapter 6 are used to generalise particular data usage patterns and abstract away concrete values. The main problem with this is that the heuristics are too specific to be of much use. We have now seen how GP can be used to produce expressions which account for sets of input-output pairs, and how *latent variables* can be introduced when no expression purely in terms of inputs and constants can be found. What remains is to apply this to the EFSM inference process to infer output and update functions for transitions.

The heuristics in Chapter 6 are only brought into play when we want to merge a pair of transitions, neither of which directly subsumes the other. This works for simple rule-based heuristics which look for a single pattern, but is not suitable for GP. Like every machine learning technique, GP only tends to perform well when it has a lot of *training data*. The set of input-output pairs of two transitions is simply not enough information to reliably infer functions which generalise across a whole EFSM. Since the whole purpose of EFSM inference by merging is to generalise and merge instances of the same behaviour, it makes sense to try to do as much of this as possible before the merging begins. Hence, in this section, I propose that GP be used as part of a preprocessing technique which is applied to the PTA before merging states.

7.5.1 General Approach

We have already seen a basic outline of what we want to do in Section 7.2. For each group of transitions with the same *structure* — that is, those transitions with the same label and arity which produce the same number and types of literal outputs — we want to use GP to infer output functions. The enables us to abstract away concrete literal values.

The preprocessing technique I propose here has five main operations: transition grouping, output inference, update inference, standardisation, and generalisation. Before delving into the details of each stage, I will first present the general approach of my technique in terms of the running drinks machine example. Recall that the traces in Figure 6.1 can be transformed into the PTA in Figure 6.3. To fully illustrate my preprocessing technique, we require an extra trace, $\langle$*select*("coffee"), *vend*(), *coin*(100)/[100], *vend*()/["coffee"]$\rangle$. Here, the user has selected coffee and then pressed *vend* before inserting a coin, so did not receive their drink. They subsequently paid and then successfully dispensed their drink. The four traces can then be transformed into the PTA shown in Figure 7.7.

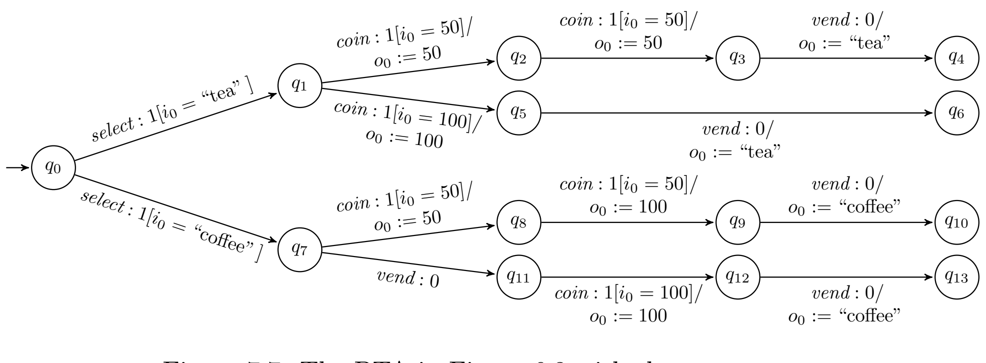

$select : 1[i_0 = \text{"tea"}]$
$coin : 1[i_0 = 50]/$ $o_0 := 50$
$coin : 1[i_0 = 50]/$ $o_0 := 50$
$vend : 0/$ $o_0 := \text{"tea"}$
$coin : 1[i_0 = 100]/$ $o_0 := 100$
$vend : 0/$ $o_0 := \text{"tea"}$
$select : 1[i_0 = \text{"coffee"}]$
$coin : 1[i_0 = 50]/$ $o_0 := 50$
$coin : 1[i_0 = 50]/$ $o_0 := 100$
$vend : 0/$ $o_0 := \text{"coffee"}$
$vend : 0$
$coin : 1[i_0 = 100]/$ $o_0 := 100$
$vend : 0/$ $o_0 := \text{"coffee"}$
q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10} q_{11} q_{12} q_{13}

Figure 7.7: The PTA in Figure 6.3 with the extra trace.

The first step in the preprocessing technique is to divide transitions into groups according to their structure. This process is detailed in Subsection 7.5.2 and yields four groups of transitions: *select*, *coin*, *vend* with one output, and *vend* with no output. As a shorthand, I will distinguish the two structural *vend* groups by writing *vend/1* and *vend/0* to denote the fact that they have one and zero outputs, respectively.

For reasons explained in Subsection 7.5.2, structural groups are split according to the last transition which was taken that did not have the same structure. This leads to the *coin* structural group being split into two historical subgroups, one containing transitions $q_1 \rightarrow q_2$, $q_2 \rightarrow q_3$, $q_1 \rightarrow q_5$, $q_7 \rightarrow q_8$, and $q_8 \rightarrow q_9$, all of which follow *select,* the other containing only the transition $q_{11} \rightarrow q_{12}$, which follows *vend.*

We must now obtain output and update functions for each group. The transitions in the *select* group do not produce outputs so no functions need to be inferred. The first *coin* group contains three distinct transitions: *coin*(50)/[50], *coin*(50)/[100], and *coin*(100)/[100]. This produces a GP training set which looks like $\{[i_0 = 50] = [o_0 = 50], [i_0 = 50] = [o_0 = 100], [i_0 = 100] = [o = 100]\}$. Since there are two possible output values for $i_0 = 50$, we need an extra variable to relate inputs and outputs. The GP infers that the output function could be $i_0 + r_1$, if r_1 holds the correct value at the point when each respective *coin* transition is called.

Introducing a register to the model means that we need to infer update functions to ensure that it holds the correct value whenever it is evaluated. The details of this are explained in Subsection 7.5.4, but the general idea is to work out target values for each state in the model and use these as "outputs" for another round of GP (this time without any latent variables) to infer update functions for each transition group. This results in the *select* transitions initialising the value of r_1 to zero, and each *coin* transition updating it to $i_0 + r_1$. The *vend* transitions do not need to change its value.

The next stage in my preprocessing technique is standardisation. As we have seen with the two *coin* groups, transitions with the same structure are sometimes divided into subgroups by their history. This means that they may end up with different output and update functions. Standardisation is about trying to make these functions the same for each structural group. Subsection 7.5.5 describes the details of this, but the process essentially involves searching for a configuration of output and update functions from the various subgroups that works for the whole structural group. Here, the output $i_0 + r_1$ and update $r_1 := i_0 + r_1$ work for both *coin* subgroups, so we do not need to bother trying to infer output and update functions for the second *coin* historical subgroup.

The *vend/1* group contains two distinct transitions: *vend*()/["tea"] and *vend*()/["coffee"]. These transitions do not take any input, so the training set looks like $\{[] = [o_0 = \text{“coffee”}, o_0 = \text{“tea”}]\}$. We clearly need to introduce another register to account for the different output values. For training sets like this which have multiple possible outputs with no inputs, we can take a shortcut and simply say that the output must be the value of a register without the need to even run GP. Let us call this register r_2. The justification for this is that we do not have sufficient information to try to evolve a more complex function. Again, we must evolve update functions such that our newly introduced register holds the correct values when necessary. Here, we end up assigning the input of *select* to r_2. The *coin* and *vend/0* transitions can then leave it unchanged. This results in the PTA in Figure 7.8.

The generalisation step is about making the model able to respond to as many traces as possible. Currently, each transition in Figure 7.8 still has its original guard. Ideally, we would like to remove these so that the model can respond to different input values. The problem

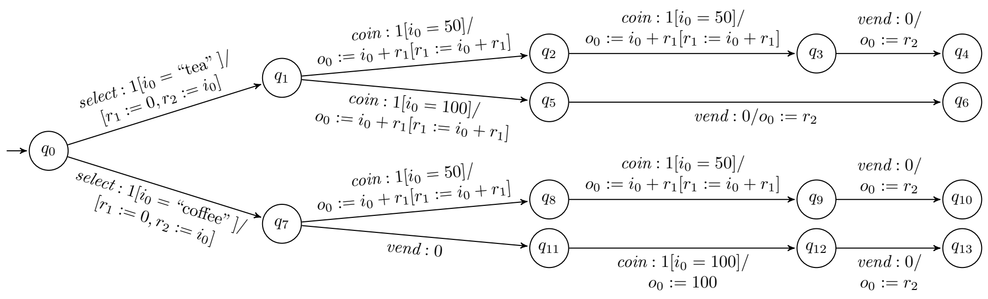

q_0
q_1
q_2
q_3
q_4
q_5
q_6
q_7
q_8
q_9
q_{10}
q_{11}
q_{12}
q_{13}
$select : 1[i_0 = \text{“tea”}]/ [r_1 := 0, r_2 := i_0]$
$select : 1[i_0 = \text{“coffee”}]/ [r_1 := 0, r_2 := i_0]$
$coin : 1[i_0 = 50]/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$coin : 1[i_0 = 100]/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$vend : 0/ o_0 := r_2$
$vend : 0/o_0 := r_2$
$vend : 0$
$coin : 1[i_0 = 100]/ o_0 := 100$

Figure 7.8: The PTA after replacing literal outputs with functions and adding updates where necessary.

then is nondeterminism but, like in Subsection 3.3.4, this is really just duplication of behaviour. We can resolve this nondeterminism by calling the RESOLVENONDETERMINISM function from Algorithm 4 to merge states and transitions such that the model becomes deterministic again. After resolving all the nondeterminism, we end up with the PTA in Figure 7.9.

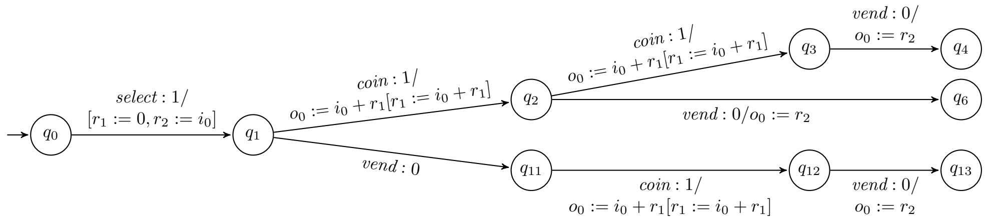


Figure 7.9: The PTA after resolving nondeterminism.

This completes the preprocessing step, and we can now begin scoring and merging states exactly as detailed in Chapter 6. The fact that preprocessed models contain transitions with output and update functions from the outset of inference proper further motivates the need for the direct subsumption relation from Chapter 5 since it is highly likely that such transitions will need to be merged.

7.5.2 Transition Grouping

The first step in my preprocessing technique is to form groups of transitions which represent the same behaviour. This is decided based on their *structure.* That is, their label and the number and types of inputs and outputs. The justification for this is that transitions with the same structure are likely to represent instances of the same abstract behaviour with different data values. There are three structural transition groups in Figure 6.3: *select, coin,* and *vend.*

In more complex systems, different transitions may read from and write to the same variable. This means that transitions with the same structure may be subject to the side effects of different transitions depending on where in a trace they occur. This means that they may need their anterior context “setting up” differently if they are to evaluate correctly. If we group only by structure, we ignore this fact and make the GP training data “unclean” such that it is nearly impossible to infer suitable functions. To mitigate for this, we divide structural groups by their *history.* That is, the last transition which was not of the same structure. This means that transitions in the same group are not subject to interference by other transitions and the training data is clean. Thus, it is much easier for the GP to infer output and update functions.

Example 7.5.1. Consider the case where the drinks machine from Figure 1.5 has an extra transition, $q_1 \xrightarrow{refund} q_1$, which returns the customer's coins and resets the running total held in r_1, allowing them to restart the payment process. Here, grouping the *coin* transitions all together ignores the effect that *refund* has on the data state. The training data for the GP is thus "unclean", making it much harder to infer output and update functions. Instead, we must divide the *coin* transitions into those which follow *select* and those which follow *reset.*

The disadvantage of this is that the more groups we have, the more registers we end up introducing. In Example 7.5.1, the two *coin* groups end up using two separate registers. This is not ideal, as we know that the transitions actually implement the same behaviour. I solve this issue with the *standardisation* step, explained in Subsection 7.5.5.

Another observation which can be made in retrospect of Chapter 8 is that, in highly reactive systems, subdividing structural groups can lead to very small (or even singleton) training sets. This obviously hinders the GP hugely and often leads to the original literal output. An alternative approach would be to first try to infer a function for the entire structural group and only split by history if this fails. This has the advantage that, in situations where the variable used by a particular structural group is not subject to interference from other transitions, there is a much larger training set for the GP to work with. Unfortunately, I only made this observation after performing the evaluation in Chapter 8 so an investigation into this is left as future work.

7.5.3 Output Function Inference

Output function inference is a call to the GP system from Section 7.4. For each group identified in the previous step, the GP is asked to provide a function for each output. The GP is first called without access to a latent variable. If it is unable to come up with a function purely in terms of inputs and constants, it is called again, this time with access to a latent variable. The reasoning behind this is that we don't want to introduce registers unless there is no other solution. If we can explain output purely in terms of input, it does not make sense to introduce additional variables for which we must then infer update functions.

Example 7.5.2. Consider again the PTA in Figure 7.7. When we want to infer an output function for the *coin* transitions, we have a training set which looks like $\{[i_0 = 50] = [o_0 = 50], [i_0 = 50] = [o_0 = 100], [i_0 = 100] = [o = 100]\}$. We first call the GP without a latent variable. Obviously, it fails since no function exists which can explain the behaviour here. Calling the GP a second time with a latent variable fixes this problem though, since this variable can change its value for each target to allow the correct output to be produced.

Consider now, a different set of traces which yields the training set $\{[i_0 = 50] = [o_0 = 60], [i_0 = 60] = [o_0 = 70], [i_0 = 70] = [o = 80]\}$. Here, we can see that the output is the input plus ten. Ideally, we would like this to be the function returned by the GP. If we give access to a latent variable r_n straight away, however, we are quite likely to get a function which includes this, for example $i_0 + r_n$, $i_0 - r_n$, and just r_n all produce the expected output if we assume that r_n holds the correct value. The problem with this is that we do not need the register here. The behaviour can be explained by the function $i_0 + 10$. The GP is much more likely to discover this if we first call it without access to the register.

For each output function that introduces a new register, we must attempt to infer update functions such that the register holds the correct value when the output function is called. Details of this are given in Subsection 7.5.4. Once this is done, we check to ensure that the new PTA still satisfies the original traces used to build it. If we cannot infer a set of updates such that the PTA still accepts the original traces, we cannot use the output function and must fall back to the original literal outputs.

It is worth discussing here how the sets of terminals and non-terminals required by the GP are obtained. The set of non-terminals is simply the supported operations from my EFSM implementation, $+$, $-$, and $\times$. Terminals come from the traces. The set of constants is formed of all inputs and outputs observed in the traces used to build the PTA. As per [150], I also include the values 0, 1, and 2, regardless of whether they appear in the original traces. For a transition with arity n, the non-latent variables are $i_0, \ldots, i_{n-1}$, as well as any registers which are currently defined. Since the main aim of this exercise is to infer functions which use latent variables (i.e. EFSM *registers*), we can also provide the GP with one fresh register which is tagged as being latent, meaning that its value can change as necessary.

We could provide any number of latent registers but, to simplify things a little, I only include one for each run of GP. This introduces the limitation that the technique will miss cases where the output of a transition depends on the values held by multiple state variables, but there is no way of knowing in advance how many variables are used by the actual system. An investigation into how the ability to use multiple latent variables affects the accuracy of the resulting EFSMs is desirable future work.

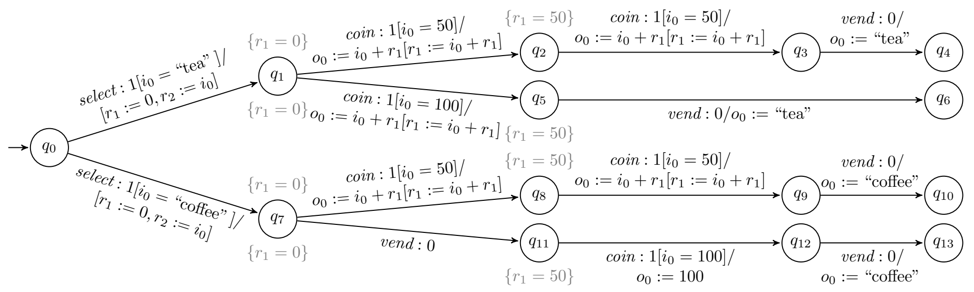
Example 7.5.3. Consider the traces used to build the PTA in Figure 7.7. Here, the observed constants are "tea", "coffee", 50, and 100. Along with 0, 1, and 2, these are the constants which are given to the GP. The *coin* transitions have arity one, so we provide one input variable i_0. Since this is the first set of transitions for which we infer an output, there are no existing registers, but we give access to the latent variable r_1 the second time we call the GP.

When we infer an output function for the *vend/1* transitions, which produce a single output, these have input arity zero. There are no inputs here but, if we can infer the correct update function for r_1, this will hold the value 100 when we call each *vend/1* transition. This has no bearing on the output of the transitions, since all recorded outputs are strings, but it is included anyway.

7.5.4 Update Function Inference

Having inferred a function which accounts for the outputs of a particular transition group, if that function makes use of a new register, we need to infer a set of update functions which ensures that the register holds the correct value each time it is required. In short, this process involves walking the traces in the PTA, solving for the register such that we have a target value, and running the GP for each transition group.

Example 7.5.4. Consider the PTA in Figure 7.10. We have inferred that the output of the *coin* transitions is $r_1 + i_0$, but r_1 is never initialised or updated. We need to make sure that r_1 holds the correct value every time a *coin* transition needs it. To do this, we walk each trace in the model and record the expected output for each coin transition. We then solve the output function for r_1 and this becomes the target value. These are shown in grey in Figure 7.10.



$\{r_1 = 0\}$
$select : 1[i_0 = \text{“tea”}]/ [r_1 := 0, r_2 := i_0]$
$coin : 1[i_0 = 50]/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$\{r_1 = 50\}$
$coin : 1[i_0 = 50]/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$vend : 0/ o_0 := \text{“tea”}$
$coin : 1[i_0 = 100]/ o_0 := i_0 + r_1[r_1 := i_0 + r_1]$
$vend : 0/o_0 := \text{“tea”}$
$select : 1[i_0 = \text{“coffee”}]/ [r_1 := 0, r_2 := i_0]$
$vend : 0$
$coin : 1[i_0 = 100]/ o_0 := 100$
$vend : 0/ o_0 := \text{“coffee”}$
q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7 q_8 q_9 q_{10} q_{11} q_{12} q_{13}

Figure 7.10: The PTA from Figure 7.8 before the update functions were added with the necessary anterior contexts annotated in grey.

With each target value now known, we can then call the GP again on each group of transitions. For the *select* group, for example, the training set looks like $\{[i_0 = \text{“tea”}] = [r_1 = 0], [i_0 = \text{“coffee”}] = [r_1 = 0]\}$. As there is only a single target value and no input parameters, the update function for the group is trivial: $r_1 := 0$.

With this in place, we can then infer an update function for the *coin* group. This is slightly more complicated. The training set looks like $\{[r_1 = 0, i_0 = 50] = [r_1 = 50]\}$. Here, there are several possible outcomes. As there is a single target value of 50, we could simply have the *coin* transitions set $r_1 := 50$. This fulfils the requirements of the training set and of the traces used to build the PTA, but is somewhat underwhelming.

The update functions $r_1 := i_0$ and $r_1 := r_1 + i_0$ also satisfy the requirements of the training set and of the traces. If given the right random seed, the GP is able to come up with any of these functions. Whatever function the GP returns is added to the updates of the *coin* transitions. There is no need to give the *vend/1* transitions updates to r_1 because it is not used after this point.

The *vend/0* transition could additionally set r_1 to zero before $q_{11} \xrightarrow{coin} q_{12}$ however, since r_1 is already zero at this point after having just been initialised by $q_0 \xrightarrow{select} q_7$, there is no benefit in this. Indeed, redundant initialisations like this can cause problems when we get to state merging, so updates are only inserted when they are necessary.

The first step to inferring update functions is to walk each trace in the model and establish what values the register we are interested in needs to hold at each point in time. For each trace, this produces a list of tuples containing the current model state, the current register values, the register values we need to have for the computed output to match the expected output,[4] the input values, and the transitions which was taken.

The required register values are propagated back through the model such that every state has a target value, even if they do not have an outgoing transition which reads from a register. This means that registers can be initialised and modified at any point before they are used, not just by the transition immediately before. For example, when we infer the updates for the *vend/1* output function in Example 7.5.4, r_2 is only required to have a value in states q_3, q_5, q_9, and q_{12}, but it is the *select* transitions which intuitively need to initialise it.

[4]This is established by solving for the newly introduced register, for example if we have the expression $o_0 := i_0 + r_1$ for input $i_0 = 50$ and the expected value of o_0 is 100, then we know that r_1 needs to hold the value 50. In the implementation, this is done using Z3.

Having established the values that the register needs to hold at each point in the execution, the transitions are then divided into the same groups as for output function inference. Each group then has an update function inferred for it. This is done by treating the target register value as the "output" and calling the GP exactly as for output function inference but without providing a latent variable. If it can come up with a function which satisfies the training set, this is added to the transition group. If not, the group remains unchanged.

The main drawback of inferring updates this way is that an attempt will be made to infer an update function for every historical group. For more complex systems, this means that we often end up trying to infer updates for transitions which, to a human observer, clearly have no relevance whatsoever to the register at hand. This is very inefficient, but there is no way to generally code this human intuition into the inference process. If the transition group genuinely has nothing to do with the current register, the GP will fail to come up with an update function which satisfies the training set, so no update will be added.

In the training sets for update functions, existing registers which are not the subject of the update function are removed from the set of inputs. This is because the more input variables the GP has access to, the more data it needs to accurately infer a function. Giving the GP access to too many variables can cause it to infer very strange update functions which are technically correct with respect to the training set but do not generalise. The effect of stripping extraneous registers means that each update function can only read from its own register, however, this is not an unreasonable assumption to make since each output function only gets access to one new register. Thus, registers are introduced sequentially, one at time. Under these conditions, it does not make much sense for an update function to read from multiple registers as they are most likely unrelated to the one at hand.

It is worth noting at this point that this it not the ideal way of inferring output and update functions. Ideally, we would have the ability to infer suitable updates feed into the inference of output functions, most likely by including the ability to infer suitable updates as part of the fitness evaluation of each output function. This is not really feasible in practice, though, since GP is both computationally and temporally expensive to run. It can take several minutes to infer update functions for each group as it is. The GP may evaluate many hundreds of candidate output functions in each run. If we were to attempt to evolve update functions for each of these, the runtime would be prohibitively long for most realistic systems. Hence, I chose to stage the inference of output and update functions separately. A more closely coupled inference of the two is left for future work.

7.5.5 Standardisation

As I explained in Subsection 7.5.2, transitions are grouped not only by their structure, but also by their *history* to ensure that the GP has clean training data to work with. Because the GP works independently for each group, we can end up with different output and update functions for transitions with the same structure. When this happens, we ideally want to find a configuration of output and update functions which works for all the different subgroups so that we can recentralise the output and update functions of transitions. Standardisation is an attempt to achieve this.

There are three stages of standardisation. The first is to delay register initialisation. The second is to try to find common output and update functions across transitions with the same structure. Finally, we attempt to merge registers which appear to be used in the same way.

Delay Initialisation

The first stage of standardisation is to try to delay the initialisation of registers to be as late as possible. Before a register can be used, it must first be initialised to a value. Often, this will be a literal assignment to a constant value, for example $r_1 := 0$. Because register updates are inferred along branches of the PTA, we are often left with the first transition being a "set up" transition, which initialises all the registers in the model, most of which are not subsequently used until much later on. This is problematic for two reasons. Firstly, it is unsightly and unintuitive to have transitions behave this way. It is generally considered bad programming practice to initialise variables before they are used, so we should try not to do this in our model either. Secondly, it could cause the inference process to miss transition merges.

Example 7.5.5. Consider the EFSM fragment in Figure 7.11. Here, the transition $q_0 \xrightarrow{f} q_1$ initialises r_1 to zero. There are then many transitions (denoted by a dashed line connecting the states q_1 and q_2), none of which read from or write to r_1 until is used in the output of $q_3 \xrightarrow{h} q_4$. Transition $q_4 \xrightarrow{g} q_5$ then reinitialises r_1 back to zero, before it is used again in the output of $q_5 \xrightarrow{h} q_6$.

$$\begin{tikzcd}[column sep=large]
{} \arrow[r] & q_0 \arrow[r, "f : 0{[r_1 := 0]}"] & q_1 \arrow[r, dash, dashed] & q_2 \arrow[r, "g : 1"] & q_3 \arrow[r, "\substack{h : 1/o_0 := r_1 + i_1 \\ {[r_1 := r_1 + i_1]}}"] & q_4 \arrow[r, "g : 1/{[r_1 := 0]}"] & q_5 \arrow[r, "\substack{h : 1/o_0 := r_1 + i_1 \\ {[r_1 := r_1 + i_1]}}"] & q_6 \arrow[r, dash, dashed] & {}
\end{tikzcd}$$

Figure 7.11: Part of an EFSM with a "set up" transition.

Here, the $q_2 \xrightarrow{g} q_3$ does not change the value of r_1 where $q_4 \xrightarrow{g} q_5$ reinitialises it to zero. This means that the two transitions do not have consistent updates, so cannot be merged without the help of heuristics. The reason for this is that r_1 is initialised to zero by $q_0 \xrightarrow{f} q_1$, so it already holds the correct value when $q_2 \xrightarrow{g} q_3$ is reached, meaning that there is no need for it to be updated again.

Intuitively, though, it does not matter whether r_1 is initialised to zero by $q_0 \xrightarrow{f} q_1$ or by any of the other transitions up to and including $q_2 \xrightarrow{g} q_3$. Delaying the initialisation of r_1 until just before it is needed not only makes sense from a human perspective, but also gives the two g transitions consistent updates such that they can more easily be merged.

Note that here, the term "initialisation" is used to mean "setting to a literal constant" rather than assigning the value of an input. Clearly if we are assigning an input to a register, we cannot arbitrarily move this assignment around because inputs are instantaneous. By contrast, if we are simply assigning a constant value, we can do this anywhere so there is no sense in initialising a register earlier than necessary. That is not to say that there are not occasions where it might be beneficial to bring forward the initialisation of a register, but there is no way of knowing in advance the optimal point at which to initialise a particular register, and preliminary experiments revealed that delaying initialisation seemed to lead to better models.

Centralise Configuration

Recall from the example in Subsection 7.5.1 that the structural group of *coin* transitions was split into two historical subgroups. While this is necessary for the GP to work correctly, it is problematic because it means that transitions which were initially identical in the PTA may end up with different output and update functions. This feels somewhat unclean and is likely to cause problems for the inference process later on, which may be unable to merge the two behaviours because of their differing functions.

This problem can be resolved by grouping transitions by structure, and then trying to make all the outputs and updates the same within each structural group. To do this, I collect together the various output and update functions and try to find a combination of them which accounts for the behaviour of all the transitions in the group. If such a configuration can be found, all the transitions in the group are given this configuration. If not, the group remains unchanged.

Example 7.5.6. For the PTA in Figure 7.8, the standardisation of the *coin* subgroups proceeds as follows. First, the two groups are joined to form the group

$$\begin{aligned}&\mathit{coin}: 1[i_0 = 50]/o_0 := i_0 + r_1[r_1 := i_0 + r_1],\\&\mathit{coin}: 1[i_0 = 100]/o_0 := i_0 + r_1[r_1 := i_0 + r_1],\\&\mathit{coin}: 1[i_0 = 100]/o_0 := 100\end{aligned}$$

Next, a list of candidate functions for each output is generated. In this case, we have one output with two possible candidate functions: 100 and $i_0 + r_1$. There is only one possible update function, $r_1 := i_0 + r_1$, which we can either include or not. This gives us four combinations, tabulated below.

outputs	updates
$o_0 := 100$	$[r_1 := i_0 + r_1]$
$o_0 := 100$	$[]$
$o_0 := i_0 + r_1$	$[r_1 := i_0 + r_1]$
$o_0 := i_0 + r_1$	$[]$

For each combination in the table, we replace the outputs and updates of each transition in the structural group and see if the new PTA accepts the original traces. If it does, we keep this configuration of outputs and updates for the structural group. In this instance, outputting $o_0 := i_0 + r_1$ and updating r_1 to $i_0 + r_1$ works for all *coin* transitions.

Merge Similar Registers

The third stage of standardisation is to merge registers which are used in the same way. As we have seen, transitions with the same structure are split into subgroups according to the previous transition. If a particular structural group cannot be standardised, it may be the case that transitions in the same structural group end up using different registers for the same thing. It can be extremely helpful to merge these registers, if we can, not only to reduce the number of registers used by the model, but also to make transitions with the same structure consistent with each other.

To do this, I first enumerate the registers of the PTA to form the set R and, for each $(r_1, r_2) \in (R \times R)$, I check to see if there are two transitions in the PTA such that r_1 and r_2 are used in the same way, i.e. if the transitions which use the respective registers are isomorphic up to register renaming. If so, r_2 is renamed to r_1 if the resulting PTA still accepts the traces used to build it.

7.5.6 Generalisation

Having now inferred output and update functions for each transition, we now want to remove the literal input guards so that the model can accept as many inputs as possible. If we have done our job correctly, the output and update functions should generalise not only to the inputs in the training set, but also to other inputs which we have not seen. The generalisation step involves dropping the guard for each transition and then attempting to resolve the resulting nondeterminism. This is done in exactly the same way as in the main loop of inference, using the resolveNondeterminism function detailed in Algorithm 4 with whatever additional heuristics have been given to the main inference process.

The dropping of guards is not strictly necessary for model inference, but doing so makes transitions which express the same behaviour with different data exactly identical, allowing this fact to manifest itself in the form of nondeterminism during state merging. This allows us to merge many more transitions. An alternative to this would be to retain the guards, run the state merging process, and drop (or generalise) them *after* this has finished (resolving any resulting nondeterminism as per Algorithm 4). We could even add guards for the literal register values needed to satisfy the original traces, and then generalise these after state merging. An investigation into the effect of this is left for future work.

7.6 Distinguishing Guards

Thus far, we have been concerned with the merging of transitions as a means to resolve the nondeterminism which arises as a result of merging states, but there are sometimes occasions where we explicitly do *not* want to do this. In this section, I propose a *heuristic* (like those in Chapter 6), to be applied during the merging process, to infer guards which distinguish the behaviour of transitions which cannot be merged.

Example 7.6.1. Consider the EFSM in Figure 7.12 which has been inferred from the PTA in Figure 7.7. Here, we can do *vend* transition and a *coin* transition from both q_1 and q_2 so we might consider merging those two states. This then introduces nondeterminism between pairs of *coin* and *vend* transitions. Since the two *coin* transitions are exactly identical, they can be trivially merged. The same cannot be said for the two *vend* transitions.

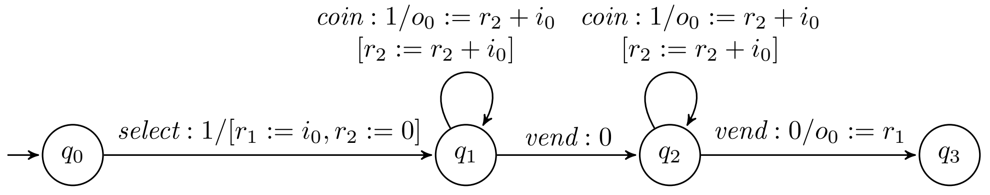


Figure 7.12: An EFSM in which merging q_1 and q_2 introduces nondeterminism between two *vend* transitions which cannot be resolved by merging them.

In Figure 7.12, the $q_1 \xrightarrow{vend} q_2$ transition has no output, signifying that the customer has not inserted enough money to pay for their drink. By contrast, the $q_2 \xrightarrow{vend} q_3$ transition represents the user receiving their drink after having inserted sufficient payment, so *does* have an output. There is clearly no way for these transitions to be merged into a single behaviour as it is impossible to simultaneously have different numbers of outputs.

If we cannot resolve nondeterminism by merging transitions, we might then conclude that we should not merge q_1 and q_2 into a single state. Alternatively, we could consider the possibility of *value-dependent* behaviour. Recall the Java code from Figure 2.1 which implements the simple drinks machine. Here, the `vend` method has an `if` statement which affects the return value. If the user has inserted sufficient payment, the selected drink is dispensed. If not, they get `null`. It would be extremely useful if we could somehow express the value-dependent nature of the *vend* method in our model. Transition *guards* allow us to do this and, if we can infer a mutually exclusive pair of guards to distinguish the two transitions, we can resolve the nondeterminism which arises from merging states q_1 and q_2 in a meaningful way.

In Example 7.6.1, we have two different kinds of *vend* transition which cannot ever be merged: one which outputs the selected drink, and one which outputs nothing. In the underlying system, the difference in behaviour depends on the value of a register. We would like to infer guards for the different behaviours to reflect this. This is a function which takes inputs and registers and returns either *true* or *false*. We have already seen how GP can be used to infer functions, and can apply the same technique to infer guards. Indeed, the task of inferring guards is much easier than inferring output functions as we do not need to use latent variables. The task is further simplified by the fact that the implementation from [150], which I used as a foundation for my own work, already supports boolean functions.

With most of the necessary infrastructure already in place, the task of inferring transition guards is then as simple as determining the training set for the GP. To do this, the original traces are run through the nondeterministic model up to the point of reaching the newly merged state. If one of the offending transitions should be taken, the inputs, current register values, and transition are recorded. This produces two sets, one for each of the two transitions we are trying to distinguish.

Since we are trying to infer mutually exclusive guards, it is sufficient to infer a single guard function for one transition and then apply its negation to the other. Thus, for one of the two sets, the target output value is set *true*, and for the other set it is *false*. The GP is then called with the supported arithmetic and guard operators from Chapter 4 as non-terminals, and the observed input and register variables and values (plus 0, 1, and 2) as terminals. This should then produce a guard function which returns *true* for one set of inputs and registers, and *false* for the other. This can then be added to the list of guards for the transition corresponding to the *true* training set, and its negation added to the guards of the *false* transition.

Example 7.6.2. The EFSM in Figure 7.13 is the result of merging q_1 and q_2 of Figure 7.12. Here, we have two nondeterministic *vend* transitions from $q_{1,2}$ which we cannot merge.

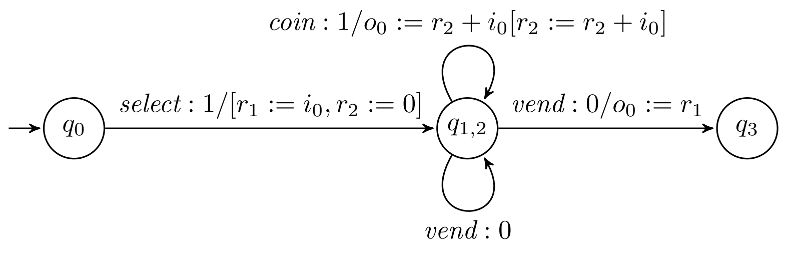

$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
q_0
$select : 1/[r_1 := i_0, r_2 := 0]$
$q_{1,2}$
$vend : 0/o_0 := r_1$
q_3
$vend : 0$

Figure 7.13: The result of merging states q_1 and q_2 of the EFSM in Figure 7.12.

Running the traces used to build the PTA in Figure 7.7 through this model, we end up with the sets $\{([], \langle r_1 := 100, r_2 :=$ "tea" $\rangle, \mathit{vend} : 0/o_0 := r_2), ([], \langle_1 := 100, r_2 :=$ "coffee" $\rangle, \mathit{vend} : 0/o_0 := r_2)\}$ and $\{([], \langle r_1 := 0, r_2 :=$ "coffee" $\rangle, \mathit{vend} : 0)\}$. Thus, we wish to infer a guard that returns *true* when $r_1 = 100$ and $r_2 =$ "tea" and when $r_1 = 100$ and $r_2 =$ "coffee" , but returns *false* when $r_1 = 0$ and $r_2 =$ "coffee" . The guard $r_1 = 100$ will do this nicely, as will a multitude of other guards. We then add this guard to $\mathit{vend} : 0/o_0 := r_2$ and its negation to *vend*:0, which resolves the nondeterminism.

7.7 Implementation Speed-ups

Running the GP is quite computationally expensive, especially with latent variables, but there are certain steps which can be taken to safely reduce the amount of work that needs to be done. This section details the steps I took to reduce the computational expenditure.

7.7.1 Population Seeds

GP works by evolving a population of candidate solutions over time. To begin with, the population is initialised with random expressions. We can give the GP a head start by adding some hand-picked expressions which it might find helpful. To this end, when inferring output and update functions, the population is seeded with all terminal values. This is every input and register variable, as well as all literal constants available to the GP. This ensures that, if the training set can be explained by either by a literal value or a variable, this fact will not only be discovered, but will be discovered quickly. As expressions tend to grow over time, without seeding in the trivial expressions, the GP sometimes runs for many iterations before the correct one enters the population and may not even discover it at all.

When there are a lot of terminal values, this may end up creating an initial population which is larger than the specified desired size. In this instance, after the first generation of GP, there will be a "cull" of the weakest members such that the population returns to the specified size.

7.7.2 Memoisation

One of the most obvious steps to speeding up computation is memoisation. This technique involves storing the results of expensive function calls in a lookup table such that functions only need to be evaluated once for each set of inputs. Since GP is called with a training set, we can apply memoisation to each training set such that the keys of the lookup table are training sets and the values are arithmetic expressions. This is particularly applicable to distinguishing guards, as they are needed throughout the inference process. Before running the GP, the lookup table is checked to see if it already contains a guard for the training set at hand. If it does, the guard from the lookup table is returned. If not, the GP is run and a new expression is inferred. If that expression is correct, it is added to the lookup table. If it is not, we record the failure in the lookup table so that we need not waste time calling the GP again.

Because the result of GP is stochastic, it may seem odd to store failure in the lookup table, however the GP is configured such that this is subject to a random *seed.* Since this remains constant throughout each run of inference, the result of the GP depends purely on the training set. Thus, it will produce the same expression for the same training set every time it is called.

Outputs and updates are only inferred prior to inference, and transitions are grouped such that it is not particularly likely that the GP will be called lots of times with the same training set. It is still worth attempting to memoise expressions, though, as structural transition groups may be split historically, meaning that the same expression can account for multiple training sets. Rather than building a lookup table, each correctly inferred expression is simply added to a list. When the time comes to infer a new expression, the list of previously inferred expressions is first checked to see if it contains an expression which accounts for the training set. Because we would like to infer the most general function possible for each training set, literal values are not memoised. This does not affect runtime much since the initial population of the GP is seeded with terminal values anyway so, if the current training set can be explained by a single literal value, this will be discovered by the first iteration of GP anyway.

7.7.3 Latent Variables

As we saw in Section 7.4, some training sets simply cannot be explained without latent variables. This occurs when there are more possible outputs for a given set of inputs than there are inputs. For update function and guard inference, we do not have access to latent variables so, if we have more possible outputs than inputs, we can fail prematurely as we know the GP will not find a suitable expression. In the case of output function inference, which is first called without access to latent variables in order to discourage their use, at the point of discovering their necessity, we can simply provide access without having to first try and fail to infer a function without them.

As mentioned in Section 7.5, if we have a training set which has multiple outputs without any inputs, we do not call the GP. Instead, we simply say that the output must be the content of a register. This pushes work into the inference of update functions but makes that job easier since the value of the register is simply output as-is.

7.8 Conclusion

This chapter presented a technique involving GP to infer expressions to relate sets of input-output pairs. The novelty here is that we do not need all the input values, as my technique allows *latent variables* to be used to account for this. I then apply my technique to tackle the problem of inferring output and update functions for EFSM transitions, and propose a preprocessing technique to take the place of the simple pattern-recognition heuristics of Chapter 6. The technique of GP is also applied to infer guards to distinguish the behaviour of transitions when it is clear that they should not be merged. This is vital to account for *value-dependent* behaviour, which would otherwise mean that states and transitions could not be merged.

It still remains to evaluate the effectiveness of my technique. We do not yet know its limitations or how applicable it is in general. Since the technique is intended to act as a preprocessor for the inference technique in Chapter 6, it makes sense to perform the evaluation in this context. An empirical evaluation of both techniques is the subject of Chapter 8.

It is interesting, at this stage, to note down some possible directions of future research. Currently, output and update functions are evolved in isolation from a training set. It may be beneficial to make use of co-evolution [69] to evolve updates alongside their respective output functions. Co-evolution may also be beneficial when inferring guards to distinguish transition pairs. Currently, only one guard is inferred, and its negation is added to the other transition. It may be that actively evolving two separate mutually exclusive guards gives better results.

Another possible area of exploration would be the use of classifiers in the state merging process, like in MINT [152]. The inference of generalised functions prior to merging means that models then potentially have internal registers as well. It is these which are fed to the classifiers in MINT to help them predict the next action. Because MINT works with white-box traces, these values are readily available, but my technique is designed to work with black-box traces which only contain input and output values. While we could certainly feed the input and output values from the traces to classifiers, these values are instantaneous and bound to individual events. By contrast, internal registers persist throughout the lifetime of the model so, like the control flow state, hold information about the history of the model so are more likely to have a determining effect on subsequent behaviour.

Chapter 8

Experimental Evaluation

Chapter 6 presented a technique to infer EFSM models from black-box execution traces using state merging and heuristics. Chapter 7 presented an additional preprocessing step for this technique which uses GP to infer the functions on transitions that relate inputs, outputs, and register values. This chapter presents an evaluation of the system in the context of three realistic case studies: liftDoors, drinks, and spaceInvaders.

8.1 Introduction

As discussed in Section 3.13, it is often the evaluation of the proposed tools and techniques where the literature on EFSM inference falls short. There does not appear to be a standard evaluation process and, due to the fact that most works begin with a new EFSM definition, it is often impossible to meaningfully compare the models produced by different tools.

The task of objectively evaluating the quality of (E)FSM models is inherently difficult, especially if there is no reference model available. This is even more problematic for EFSM models than classical FSMs since we can arbitrarily move information between the control and data states. Thus, for every system, there is a potential infinitude of trace equivalent models. To further complicate matters, the intended use-case of the model can affect what we would like to infer. That is, different kinds of models may be more or less suited to particular tasks meaning that there is no single optimum model. This applies not just to automatically inferred (E)FSMs, but also to handcrafted models and more widely to other modelling techniques.

The most similar tool in the literature to my own is that presented in [150], which uses GP to infer variable update functions on transitions of EFSMs inferred by MINT [152]. Taking inspiration from the evaluation sections of these works, I have identified four research questions to be investigated here:

RQ1 How accurate are the models produced by my inference tool?

RQ2 How does eliding variables affect the accuracy of the models produced by my inference tool?

RQ3 How does the ability to discover value-dependent behaviour affect the accuracy of the models produced by my inference tool?

RQ4 How well does my tool scale?

4a. How large are the inferred models in terms of states and transitions?

4b. How long does model inference take?

RQs 1-3 are concerned with the *accuracy* of the models we can infer in various scenarios. RQ1 seeks to ascertain how well the models produced by my tool compare to the underlying systems and the models produced by MINT [150], the current state of the art of passive EFSM inference, in situations where the two techniques are comparable. Additionally, I compare the inferred models to two baseline approaches, the original PTA built from the traces, and the model which can be inferred from this without any preprocessing or additional heuristics.

RQs 2 and 3 are concerned with situations where my tool goes beyond MINT. RQ2 is an investigation of how my tool copes when the outputs of transitions depend on the values of variables which do not appear in the traces, and the factors which affect the accuracy of the inferred models. RQ3 aims to evaluate how the ability to infer *guards* during inference to distinguish transitions which cannot affects the accuracy of the models my tool can infer of systems which exhibit *value-dependent behaviour.* That is, when a particular action can produce different classes of behaviour depending on the value of a variable.

Finally, RQ4 seeks to evaluate the performance of the inference tool and how well it is able to cope with realistic systems. There are two aspects to evaluate here. The first concerns the *technique*, and the factors that affect the complexity of the models we can infer. The second is to do with the *implementation* and how fast my tool runs. While optimising runtime is not a major objective of this work, it is clearly important that my tool is able to run in reasonable time on case studies which are large enough to effectively evaluate it.

To answer these research questions, I first identified some suitable subject systems and obtained traces from them for the inference tools to work with. I then ran the tools (both my own tool and MINT) with these traces as input, and computed various quality metrics of the output models as detailed in Section 8.3.

Unfortunately, it is only possible to compare my tool with MINT for RQ1 and RQ4. Since MINT is designed to work with white-box traces, it can only operate in a purely functional setting. That is, when the output (or posterior state) of each transition depends entirely on its input (or anterior state). MINT cannot handle situations where there are additional variables at play which do not appear in the traces. Thus, RQ2 considers my inference tool only.

For RQ3, while MINT does infer guards for transitions, this does not really compare to what I am attempting to do here. The guards inferred by MINT play a descriptive role, aggregating the observed data values. The guards inferred by my tool are designed to distinguish transitions where the observable behaviour depends on the value of an input or register. Since the models inferred by MINT do not have explicit outputs, there is no observable behaviour to distinguish.

The remainder of this chapter is laid out as follows. Section 8.2 outlines the systems I used in this evaluation and how traces were obtained from each system. Next, Section 8.3 describes each of the accuracy metrics I used to evaluate the quality of the inferred models. Section 8.4 describes the design and setup of the experiments I performed to answer my research questions. The results of these experiments are presented in Section 8.5. In Section 8.6, I provide an informal discussion on the relative utility and understandability of the inferred models. Finally, in Section 8.7, I discuss some threats to the validity of these results.

8.2 Subject Systems

To evaluate an inference technique, we must first find some systems from which to obtain traces. What we need here are sequential systems that make use of a *data state.* As discussed in Section 3.13, many works in the literature use their own case studies for evaluation, which are often not openly available. This makes coming up with suitable case studies a difficult task in and of itself. Here, we have the additional constraint that inputs and outputs must be either integers or strings, since these are currently the only data types supported by my implementation. For this evaluation I use two systems from the literature and an extended version of the simple drinks machine from Section 1.1.

Ideally, it would have been good to choose systems for which an (E)FSM model already exists, as this would provide a "model solution" against which to compare the output of the inference tools. As discussed in Section 3.11, the nature of EFSMs is such that there is often no single optimal model, but it is helpful nonetheless to have a rough idea of the behaviour of each system so we can tell at a glance whether the inference has got the right idea. Unfortunately, none of these case studies came with a suitable existing model, and the traces are sufficiently long and numerous that it is infeasible to try to infer a model from them by hand. We can, however, form a rough idea of what we might expect to infer by inspecting the artefacts of the systems, since they are all relatively simple.

8.2.1 Lift Doors

The first case study, liftDoors, comes from a model of a lift door controller originally published in [136]. This case study was also used in [150] to evaluate their system.[1] In this model, requests to open or close the door are sent from the central control system. When it receives this signal, liftDoors opens the door, waits for passengers to enter or leave the lift, and closes the door again. The variable *timer* is used as a counter to ensure the door stays open for a certain amount of time before closing again. When the door is closed, liftDoors sends a signal back to the central control system.

To ensure that nobody is crushed by the doors, the lift door system is equipped with an optical sensor which triggers the reopening of the doors if a passenger is stood in the way while they are closing. Passengers can "hold the lift" for other people by pressing a button inside the lift which also triggers the reopening of the doors.

Since there was no implementation of this model available, the authors of [150] created a simple Java implementation from which to obtain traces. This implementation is not openly available but, since the traces used in [150] are,[2] I was able to use them for this evaluation.

A typical trace of the liftDoors system is shown in Figure 8.1. Here, events take a single input (which represents the anterior value of the *timer* variable) and produce a single output (which represents the posterior value). As with most realistic traces, it is quite difficult to read, but this only serves as motivation for the inference of a model.

$$\begin{aligned}&\langle \mathit{setTimer}(0)/[5], \mathit{waitTimer}(5)/[4], \mathit{waitTimer}(4)/[3], \mathit{waitTimer}(3)/[2], \mathit{waitTimer}(2)/[1],\\ &\mathit{waitTimer}(1)/[0], \mathit{systemInitReady}(0)/[10], \mathit{closingDoor}(10)/[9], \mathit{closingDoor}(9)/[8],\\ &\mathit{closingDoor}(8)/[7], \mathit{closingDoor}(7)/[6], \mathit{buttonInterrupted}(6)/[3], \mathit{openingDoor}(3)/[2],\\ &\mathit{fullyOpen}(2)/[1], \mathit{fullyOpen}(1)/[0], \mathit{timeout}(0)/[5], \mathit{closingDoor}(5)/[4], \mathit{closingDoor}(4)/[3],\\ &\mathit{closingDoor}(3)/[2], \mathit{closingDoor}(2)/[1], \mathit{closingDoor}(1)/[0], \mathit{fullyClosed}(0)/[0],\\ &\mathit{fullyClosed}(0)/[0], \mathit{requestOpen}(0)/[10], \mathit{openingDoor}(10)/[9]\rangle\end{aligned}$$

Figure 8.1: A typical trace of the liftDoors system.

[1]Two case studies are used in the evaluation of [150]: liftDoors and cruiseControl. Of these, only liftDoors is applicable to my own work, since cruiseControl relies heavily on the use of floating-point numbers, which are not currently supported by my implementation.

[2]`http://www.cs.le.ac.uk/people/nw91/Files/ICSMEData.zip` (Accessed 15/05/20)

The traces of liftDoors record ten actions and can be divided up into two parts. The first part of the trace, up to the *systemInitReady* event in Figure 8.1 is common to all traces of the liftDoors system and appears at the start of each trace, differing only in the input to *setTimer*, which initialises the timer to five. The *waitTimer* action decrements the timer and is used to make the lift wait a given amount of time. When the timer reaches zero, the *systemInitReady* action signals that the lift door is ready, and sets the timer to ten.

After *systemInitReady*, the system will make an attempt to close the doors. The input is always ten, and the system will issue repeated *closingDoor* events to decrement the timer until either it reaches zero or a *buttonInterrupted* action occurs. This represents a passenger pressing the button to reopen the doors to "hold the lift" for other people and sets the timer to three. Since there is no *sensorInterupted* action in the traces, it seems as though the optical sensor did not make it into the Java implementation. The system will then issue repeated *openingDoor* events until the door is *fullyOpen*, at which point the system will issue *timeout*. This sets the timer to 5 and represents the lift having waited sufficient time with the doors open to begin closing them again. When the doors are fully closed, the system will then issue *fullyClosed* events until a *requestOpen* action is performed, whereupon the cycle begins again. This functionality can be expressed as the EFSM in Figure 8.2.

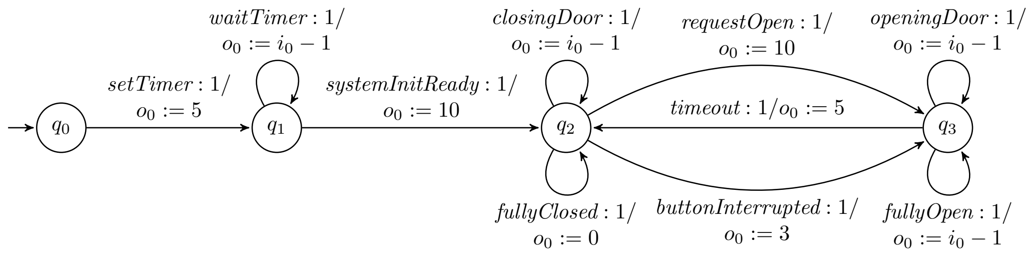

q_0
q_1
q_2
q_3
$setTimer : 1/$ $o_0 := 5$
$waitTimer : 1/$ $o_0 := i_0 - 1$
$systemInitReady : 1/$ $o_0 := 10$
$closingDoor : 1/$ $o_0 := i_0 - 1$
$requestOpen : 1/$ $o_0 := 10$
$openingDoor : 1/$ $o_0 := i_0 - 1$
$timeout : 1/o_0 := 5$
$fullyClosed : 1/$ $o_0 := 0$
$buttonInterrupted : 1/$ $o_0 := 3$
$fullyOpen : 1/$ $o_0 := i_0 - 1$

Figure 8.2: A model that we might expect to infer of liftDoors.

The model in Figure 8.2 has four states. As always, q_0 represents the initial state. State q_1 represents the lift waiting for passengers to enter or leave the car. State q_2 represents the door closing and being fully closed. State q_3 represents the door opening and being fully open. This model is quite compact and it is relatively easy to manually run traces like the one in Figure 8.1 through the model to verify acceptance.

An acceptable alternative would be to do as is done in [136] and split states q_2 and q_3 up into states representing the lift doors being in motion and being stationary at their respective positions. This makes the model slightly bigger but no harder to understand. It would also be acceptable to have states q_2 and q_3 representing moving and stationary rather than closing/closed and opening/open. Again, this would not make the model harder to interpret.

8.2.2 Space Invaders

The second case study is a simplified version of the 1978 arcade game Space Invaders, created by Tomohiro Nishikado. Here, the user of the system controls a gun turret and tries to shoot aliens on the screen. Their score increases each time they hit an alien, but aliens can also shoot back. If the gun turret is hit by an alien, the player loses a "life". The game is over either when the user has killed all the aliens, or has lost all their lives.

The implementation I used here is an openly available[3] accompaniment to [108] written in Java. This implementation differs from the original arcade game in several ways. Firstly, in the original game, the gun turret can only move horizontally across the screen. Here, the user can move both horizontally and vertically. Secondly, the aliens in the Java implementation do not shoot back. The player loses a life (here called a "shield") only if they are directly hit by an alien. Thirdly, in the arcade game, there is a fixed number of aliens on the screen which the user must shoot. When they have shot all the aliens, they win the game. In the Java version, there is no fixed number of aliens. They simply keep spawning until the user runs out of shields, at which point a test is performed to see if they have shot enough aliens to have won. This means that the only way to end the game is to run out of shields. Finally, the aliens in the arcade game only move down the screen towards the gun turret. In the Java version, the aliens have a random trajectory and can move across the screen as well, bouncing off the edges.

This implementation takes the form of a Java applet, so there was no way to collect traces automatically. Instead, I collected traces manually by playing the game. To help with this, I made a few changes to the source code. Like the original arcade game, I made it so that the gun turret only moves horizontally and the aliens only move vertically. I also made it so that the gun turret moves across the screen in increments of 50 pixels, meaning that there is only a small number of positions in which the turret can be. I then made it so that aliens only spawn above these locations. This makes it easier for the player both to hit and be hit by the aliens so the games progress faster. Finally, I made it such that once the user has killed five aliens, victory is declared and the game ends automatically. I also reduced the number of shields from five to three, again to make the games progress faster.

When recording traces, I chose to record seven events. The *start* event initialises the game. The gun turret is spawned in the middle of the screen at the bottom, the score is set to zero, and the player is given three shields. The *moveEast* and *moveWest* actions move the gun turret 50 pixels right and left respectively. The *alienHit* action increases the player's score by one. The *shieldHit* action takes a "life" from the player. Finally, the *win* and *lose* actions show appropriate text and play audio clips[4] to signal whether the player has won or lost.

A typical trace of the spaceInvaders system is shown in Figure 8.3. Like with liftDoors, most events take a single input which represents the anterior value of a variable, and produce a single output which represents its posterior value. Unlike liftDoors, there are three variables at work here: x, *shields*, and *aliens*, which respectively record the x coordinate of the gun turret, the number of shields the player has left, and the number of aliens the player has hit. The two *move* events take in and modify the x variable by adding or subtracting 50. The *alienHit* event increments the *aliens* variable, and the *shieldHit* event decrements the *shields* variable.

$$\begin{aligned}&\langle \mathit{start}(200,3,0)/[200,3,0], \mathit{moveWest}(200)/[150], \mathit{alienHit}(0)/[1], \mathit{moveEast}(150)/[200],\\ &\mathit{moveEast}(200)/[250], \mathit{shieldHit}(3)/[2], \mathit{alienHit}(1)/[2], \mathit{alienHit}(2)/[3], \mathit{moveEast}(250)/[300],\\ &\mathit{alienHit}(3)/[4], \mathit{moveWest}(300)/[250], \mathit{moveWest}(250)/[200], \mathit{moveWest}(200)/[150],\\ &\mathit{moveWest}(150)/[100], \mathit{alienHit}(4)/[5], \mathit{win}()/[]\rangle\end{aligned}$$

Figure 8.3: A typical trace of the spaceInvaders system.

[3]`http://www.doc.ic.ac.uk/~jnm/book/book_applets/concurrency/invaders` (Accessed 15/04/20)

[4]I removed all audio from the implementation in the interest of preserving my sanity.

The general form of traces here is much simpler than for LIFTDOORS. Every trace begins with a *start* event, which initialises the three variables. The game is then active, and the player can move east and west, and shoot or be hit by aliens. When the player hits an alien and the output is 5, the next event is always *win* and the game ends. When the player gets hit by an alien and the output is 0, the next event is always *lose* and the game ends. The EFSM in Figure 8.4 shows this graphically.

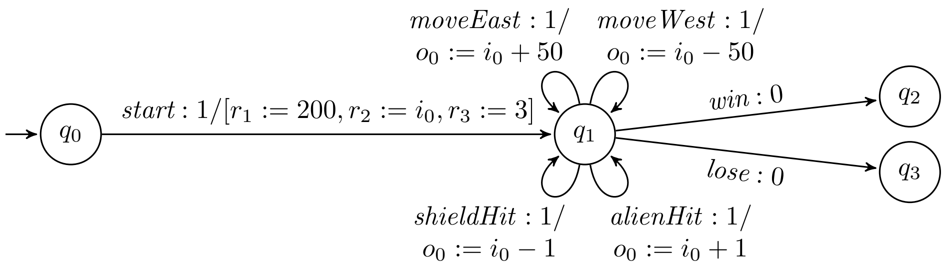


Figure 8.4: A model that we might expect to infer of SPACEINVADERS.

The model in Figure 8.4 has four states, with q_0 representing the initial state. State q_1 represents an active game. State q_2 represents the user having won the game and q_3 represents the user having lost. The model is compact and summarises the functionality of the system nicely. The only alternative to this model which is really acceptable would be to merge q_2 and q_3 into a single "end of game" state rather than having separate states for winning and losing, but this would not affect the model's utility in most applications.

While this model has fewer actions than the lift controller, it is more complex in other ways. There are three state variables at work here and the system is much more reactive. That is, from the active game state, there are six possible actions which can occur. As mentioned in [150], states with many outgoing transitions are capable of producing many different trace suffixes, which makes it difficult to match sequences of events.

8.2.3 Value-Dependent Behaviour

RQ3 is concerned with how the ability to infer guards during inference to distinguish transitions which cannot be merged affects the accuracy of the models we can infer of systems which exhibit value-dependent behaviour. That is, systems like the simple drinks machine in Figure 2.1 where an action has different classes of observable behaviour depending on the value of a variable, be that an input or an internal register.

To answer this question, we need slightly different case studies since neither LIFTDOORS nor SPACEINVADERS exhibits any value-dependent behaviour. For SPACEINVADERS, we know the winning and losing conditions, but this is not what is meant by "value-dependent behaviour". Here, as in Example 7.6.1, by "value-dependent behaviour" I mean that the *observable behaviour* (i.e. the output) of a transition depends on the value of either an input or a register. Essentially, we are looking for situations where two transitions with the same label and arity, which cannot be merged, can be distinguished by imposing appropriate guard conditions. Since the *win* and *lose* actions have different labels and do not have any recorded output, they do not exhibit value-dependent behaviour.

The traces of spaceInvaders can be modified to introduce value-dependent behaviour since the last event of every trace is either *win* or *lose*. In the traces, the *win* event always follows an *alienHit* event, while the *lose* event always follows a *shieldHit* event. Consequently, we can make the output of the penultimate events either "win" or "lose" and drop the last event of each trace. Thus, we end up with the *alienHit* and *shieldHit* actions exhibiting value-dependent behaviour. Either they will produce the posterior value of their respective variables, or the literal constant "win" or "lose". The EFSM of the modified system is shown in Figure 8.5.

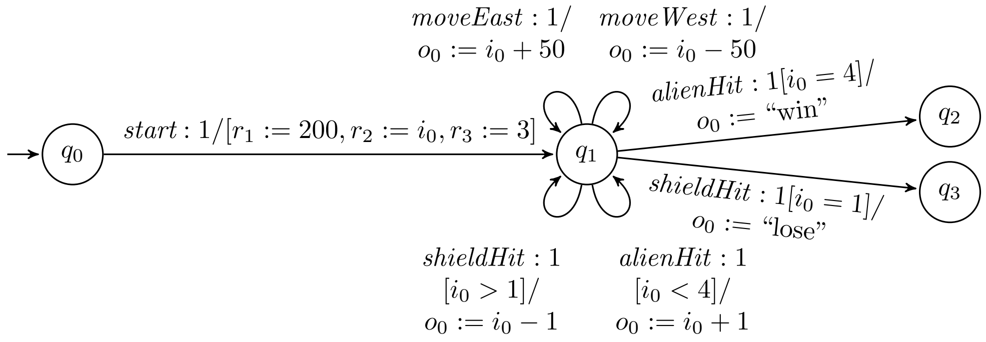

$moveEast : 1/$ $o_0 := i_0 + 50$
$moveWest : 1/$ $o_0 := i_0 - 50$
$alienHit : 1[i_0 = 4]/$ $o_0 :=$ "win"
$start : 1/[r_1 := 200, r_2 := i_0, r_3 := 3]$
q_0
q_1
q_2
q_3
$shieldHit : 1[i_0 = 1]/$ $o_0 :=$ "lose"
$shieldHit : 1$ $[i_0 > 1]/$ $o_0 := i_0 - 1$
$alienHit : 1$ $[i_0 < 4]/$ $o_0 := i_0 + 1$

Figure 8.5: A model that we might expect to infer of spaceInvaders with guards.

Unfortunately, there is no sensible transformation which can be applied to the liftDoors system to introduce similar value-dependent behaviour. Thus, we need another case study. For this, I used a modified version of the drinks machine in Figure 2.1, in which there are several drinks of differing prices. Thus, the output of the *vend* action depends on which drink has been selected and how much money has been inserted so far. This actually makes the behaviour of the system quite complicated, despite initially seeming like a very simple example.

We have already seen several examples of traces of the drinks machine, such as in Figure 3.3. The general format is that a user first selects their drink, inserts coins to pay for it, and dispenses their drink once they have paid sufficiently. The value-dependent behaviour comes from the *vend* action. If a customer tries to dispense their drink before they have paid enough money, they receive nothing. In this modified machine, whether the customer receives their selected drink when they press *vend* depends not just on how much money has been inserted but also on which drink has been selected. Here, there are three drinks available: tea (which costs 80p), coffee (which costs £1), and soup[5] (which costs £1.20). The EFSM in Figure 8.6 depicts the behaviour of the system.

The model in Figure 8.6 looks very similar to Figure 1.5, but the guards on the *vend* transitions are much more substantial to account for the fact that the different drinks have different prices. An alternative but acceptable version of this model would be to separate the disjunctive guards into distinct transitions. This does complicate the model somewhat, and makes it significantly more difficult to draw in an aesthetically pleasing way, but does not make it any more difficult to understand the underlying behaviour.

[5]Whether or not soup should be classed as a drink is left up to the reader.

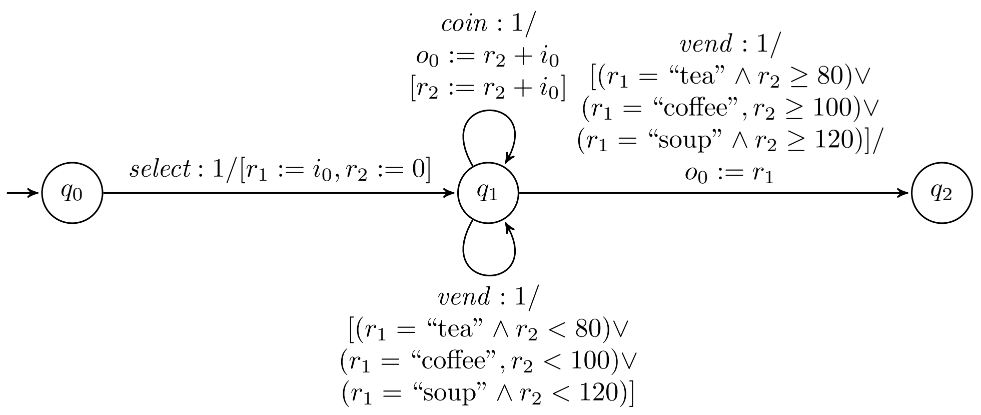

$coin : 1/$
$o_0 := r_2 + i_0$
$[r_2 := r_2 + i_0]$
$vend : 1/$
$[(r_1 = \text{“tea”} \wedge r_2 \geq 80)\vee$
$(r_1 = \text{“coffee”}, r_2 \geq 100)\vee$
$(r_1 = \text{“soup”} \wedge r_2 \geq 120)]/$
$o_0 := r_1$
q_0
$select : 1/[r_1 := i_0, r_2 := 0]$
q_1
q_2
$vend : 1/$
$[(r_1 = \text{“tea”} \wedge r_2 < 80)\vee$
$(r_1 = \text{“coffee”}, r_2 < 100)\vee$
$(r_1 = \text{“soup”} \wedge r_2 < 120)]$

Figure 8.6: A model that we might like to infer of DRINKS.

8.3 Evaluation Metrics

RQs 1, 2, and 3 are all concerned with the *accuracy* of the models my tool can infer. Thus, I shall be using the same metrics for all three of these questions. As discussed in Section 3.11, there is no definitive metric which can be used to evaluate the accuracy of EFSM models. Instead, a myriad of different metrics are available, each giving a slightly different view of the system such that a model which scores well by one metric may score poorly by another. To get a good idea of the quality of the models output by my inference technique, we must consider several different angles. This is complicated further by the fact that the structural and functional characteristics that are desirable of a model are often related to how the model will be used. If we are primarily interested in a visualisation of a system's control flow, getting the number and functionality of states correct is the top priority. By contrast, if we are more interested in data-dependency, it is much more important that we can correctly infer the use of *registers.*

As discussed in Section 3.11, most (E)FSM inference techniques in the literature are evaluated with respect to *traces.* That is, one set of traces (the *training set*) is used to infer a model, and then a second set of traces (the *test set*) is used to evaluate the predictive power of the model in terms of some metric. Evaluations are often carried out using a popular machine learning technique called k-folds cross validation [95]. This involves partitioning the data set into k sets then, over k iterations, a model is inferred using the union of $k - 1$ of these sets as the training data and the remaining set as the test set. A different test set is used for each iteration, thus mitigating bias towards any one in particular. The final accuracy score is taken as the average over all the iterations. I apply a similar technique here to evaluate my tool, and will discuss this in Section 8.4.

An alternative technique we could use here is to hand write a “gold standard” model of each system and then evaluate with respect to these in terms of either the structure or the accepted language. While Section 8.2 provides figures to illustrate the systems, these can be thought of as “artists' impressions” rather than a gold standard which must be upheld. Indeed, both Figures 8.2 and 8.4 were produced by my inference technique while investigating RQ1. While we could evaluate with respect to these models, it is difficult to get a view of their predictive power for realistic traces. Moreover, it is very difficult to quantitatively measure the differences between two EFSM models in a meaningful way without reference to traces. Thus, while I make informal structural comparisons in Section 8.6, the quantitative metrics I use to answer RQs 1, 2, and 3 are all trace based. I shall be using the following four metrics.

- Sensitivity = $\frac{\text{number of accepted positive traces}}{\text{total number of positive traces}}$
- NRMSE = $\frac{\sqrt{\frac{\Sigma_{t=0}^{n}(X_t - Y_t)^2}{n}}}{\max X - \min X}$ where X is the observation produced by the model in response to a trace and Y is the observation produced by the system in response to the same trace.
- Proportion of correctly executed actions = $\frac{\text{number of correctly executed actions}}{\text{length of trace}}$
- Proportional length of accepted prefix = $\frac{\text{length of accepted prefix}}{\text{length of trace}}$

8.3.1 Sensitivity

The first evaluation metric I use is *sensitivity.* This is a popular metric in the literature on classical FSM inference and is calculated as the number of positive traces in the test set accepted by the model divided by the number of positive traces in the test set. Thus, it represents the proportion of traces in the test set which were correctly accepted by the model. Models with a high sensitivity are considered more accurate than those with a low sensitivity.

Another popular metric in classical FSM inference is *specificity.* This is the dual of sensitivity and represents how many negative traces the inferred model correctly rejects. The two are then combined to calculate the binary classification rate as discussed in Section 3.11. I do not use specificity here as it requires negative traces, which are quite hard to obtain for realistic systems. A semi-automated technique is presented in [152], but it is still quite involved.

While sensitivity is an excellent metric for classical FSMs, it falls a little short for EFSMs. Since classical models have atomic actions, traces are accepted or rejected based on whether or not they are *recognised* by the model. Since EFSM transitions can produce observable outputs, a trace can be recognised but not accepted if the outputs in the trace do not match those produced by the model. Given that part of the objective of the inference technique presented in Chapter 6 is to automatically infer output and update functions, if these are inferred incorrectly, it is unlikely that any trace in the test set will be fully accepted. This means that the sensitivity of the inferred models may be very low, but does not necessarily mean that the models should get an accuracy score of zero. Not all EFSMs with zero sensitivity are equally bad models of the system. Indeed, some models with zero sensitivity can actually score quite highly on other metrics which view the system differently.

8.3.2 Normalised Root Mean Square Error

Despite the flaws highlighted in Section 3.11, the NRMSE of the recognised prefixes of the traces in the test set is one such metric. As discussed in Section 3.11, NRMSE aggregates the difference between the outputs produced by the system and those produced by the model. Thus, in contrast to the other metrics I use here, *lower* scores indicate better models. This is not a particularly trustworthy metric on its own, however, as it does not take into account the rejected suffix. Thus, in situations where the point of deviation is also the point of non-recognition, the NRMSE will look perfect even for poor models. If used in conjunction with other metrics, though, NRMSE can help to give an idea about the output behaviour of the inferred model. Since it was used as the primary metric in the evaluation of [150], I will also be using it here.

8.3.3 Accepted Prefix Length

Another evaluation metric I use here is the length of the accepted prefix of each trace. Like RMSE, the accepted prefix length is *scale-dependent,* in this case depending on the lengths of the traces used to evaluate the model. To normalise this, I divide by the length of each trace to give a value between zero and one. This then represents the proportion of the trace which was processed before the point of deviation, and indicates how far along a trace we can expect to get before we notice a different between the behaviour of the model and the behaviour of the system. As with sensitivity, higher scores indicate better models.

8.3.4 Proportion of Correct Events

The final accuracy metric I use is the proportion of correctly executed events in each trace. This is calculated as the number of events in each trace where the output of the model matches that of the system divided by the total number of events in the trace. This is is conceptually similar to NRMSE in that an event is considered "incorrect" if the expected output is different to the actual output. The difference here is that this metric gives the same weight to each event. It also measures correct events, so higher scores indicate better models. Another advantage over NRMSE is that we are able to take the rejected suffix into account by treating the events within it as incorrect. Thus, this metric can distinguish between a perfect model and a completely unreactive model, where NRMSE cannot.

The value of this metric likely to be very similar to the accepted prefix length in most cases but, as mentioned in [150], when models go awry, it is often because of only a few particular transitions rather than the entire model. In terms of Figure 3.12, this means that the yellow midsection between the accepted prefix and the unrecognised suffix could contain many correctly executed actions. While it is obviously desirable to increase the length of the accepted prefix, models which are able to correctly execute many actions after the point of deviation are obviously more desirable than models which cannot do this. This metric takes these actions into account, so is able to give extra credit to such models.

8.3.5 Evaluating Scalability

RQ4 is concerned with the scalability of my tool. The first part of this RQ is about the *size* of the inferred models. To answer this question, I will compare the various configurations among themselves with respect to the numbers of states and transitions in the inferred models. Section 8.6 provides a more informal qualitative discussion relating the inferred models to the exemplary models in Section 8.2.

The second part of the RQ concerns the runtime of the tools. For each experiment, I recorded the wall clock runtime from start to finish and will use this to compare the different configurations. While this is certainly not as rigorous as calculating the asymptotic complexity of each technique, optimising runtime is not a major part of this thesis. Here, I am more interested in the factors which affect runtime, for which wall clock runtime is a sufficient to be able to compare the different configurations.

8.4 Experimental Setup

The output of my inference tool is determined by two things: the training set and the *random seed* with which any GP operation (either preprocessing or guard inference) is called. To get an accurate view of the system, we need to exercise it with different trace sets and different random seeds. Because this is an empirical study, we need to ensure that we run the system enough times for the results to reflect the behaviour of the system sufficiently for us to draw conclusions. According to [11], an acceptable compromise between feasibility and reliability is 30 runs. Thus, we need to run the inference tool with 30 different trace sets. Further, for each trace set, we should run the tool using 30 different random seeds for the GP. Consequently, the tool is run 900 times for each configuration of each system. To achieve some comparability of wall clock runtime for RQ4b, each experiment was run as a separate job on the University of Sheffield's Bessemer HPC cluster and was allocated a single CPU core and up to 8 GB of RAM.

In addition to the random seed, there are several other parameters which can be adjusted to affect the performance of the GP system. These parameters are the size of the population μ, the number of new individuals generated per generation λ, and the maximum number of generations for which the GP runs. In this experiment, I ran the GP for 100 generations with $\mu = 100$ and $\lambda = 10$. There may be values of μ and λ which enable the GP to find optimal functions in fewer generations, but there is no efficient way to determine these values. Preliminary experiments revealed that these settings usually enabled the GP to find a function with an optimal fitness if one existed, but a full parameter optimisation investigation is left for future work.

As well as comparing the models inferred by my tool to those of MINT, I also compare to two baseline approaches: the initial PTA and the model which my tool infers from this purely by merging states, i.e. without GP or additional heuristics. In these cases, the output is entirely dependent on the input traces. No GP occurs here so there is no need to provide random seeds. Consequently, I only ran these configurations once for each set of traces, i.e. 30 times rather than 900. This somewhat reduces the reliability of the runtime data for these configurations, but runtime is not the primary aspect of this investigation and, as we will see in RQ4b, the runtimes without GP are often too long for 900 repeats to be feasible. Even as it is, it still took nearly three months to run the entire suite of experiments.

In terms of the training and test sets of traces, k-folds is not directly applicable here. The main reason for this is the disparity between the number of traces available for liftDoors and spaceInvaders. liftDoors has an openly available set of over 300 traces where spaceInvaders had to be run manually to obtain traces. While this does not impose a theoretical limit on the number of traces available, my own time and patience was a significant limiting factor. It is obviously unfair to divide the respective trace sets evenly if they are of different sizes, since this would give the larger trace set an unfair advantage.

Instead of partitioning the trace sets evenly, I made up the respective training and test sets by drawing random samples of 60 traces and then splitting them into two sets of 30. In the case of liftDoors, I also discarded traces of length less than five because the vast majority of the available traces are much longer than this, meaning that short traces are extremely likely to be prefixes of other traces in the training set and so contribute no new information. This left 348 traces. For spaceInvaders, I played the game 100 times to produce 100 traces, all of which were of a reasonable length. Because the drinks machine was specifically implemented for the task of evaluating my inference tool, I made it very easy to automatically obtain traces. Thus, I simply ran the program on a loop to obtain 30 separate training and test sets of 30 traces each.

8.4.1 Obtaining Traces

To compare my system with MINT [150], we need traces in a format which can be accepted by both systems. Unfortunately, the input format of MINT is slightly different to the one required by my inference tool. MINT requires traces in the following form.

```
trace
  setTimer 5
  waitTimer 5
  waitTimer 4
  waitTimer 3
  waitTimer 2
  waitTimer 1
  systemInitReady 0
  closingDoor 10
  closingDoor 9
  buttonInterrupted 8
```

Each event is on a new line which consists of the action label followed by the anterior variable values. Recall that the models inferred in [150] have no distinct inputs or outputs, so we can consider the anterior variable values to be an action's "inputs". Additionally, since MINT uses white-box traces to infer models, the variables which appear here are actually internal variables of the system, so would not visible to an outside observer.

My tool from Chapter 6 infers models with inputs, outputs, and internal variables from *black-box* traces. Here, traces are lists of JSON objects of the form `{"label":"action", "inputs":[`$i_0, i_1, \ldots$`], "outputs":[`$o_0, o_1, \ldots$`]}`. There are no internal variables here so, to convert traces in the MINT format to my own JSON format, I model the variables as inputs and output. Each event in a MINT trace reports the *anterior* value of every variable. Assuming there is no interference from the environment between transitions, we can treat the anterior values of event e_{n+1} as the posterior values of event e_n. I then encode the anterior values for each event as the input and the posterior values as the output. For the above MINT trace, this leads to the following.

```
{"label": "setTimer",  "inputs":[5], "outputs":[5]},
{"label": "waitTimer", "inputs":[5], "outputs":[4]},
{"label": "waitTimer", "inputs":[4], "outputs":[3]},
{"label": "waitTimer", "inputs":[3], "outputs":[2]},
{"label": "waitTimer", "inputs":[2], "outputs":[1]},
{"label": "waitTimer", "inputs":[1], "outputs":[0]}
{"label": "systemInitReady", "inputs":[0], "outputs":[0]}
{"label": "closingDoor", "inputs":[0], "outputs":[10]}
{"label": "closingDoor", "inputs":[10], "outputs":[9]}
```

The available traces of the liftDoors system are already in the MINT format and only make use of a single variable, so could be easily converted to the JSON format required by my tool. For spaceInvaders, I recorded traces in the MINT format with each event containing the anterior values of every variable. I then converted the traces to my JSON format, keeping only the variable values relevant to each event. For example, the *shieldHit* event from the

SPACEINVADERS system only makes use of the *shields* variable so x and *aliens* are removed from the event. DRINKS is only used by my tool, so I recorded the traces in my JSON format.

The conversion from the MINT format to JSON does mean that we lose the last event in each trace since there is no next action from which we can infer the posterior variable values. For LIFTDOORS, the available traces are sufficiently long and varied that this does not matter. For SPACEINVADERS, I recorded a sacrificial dummy action at the end of each trace.

The fact that variable values are being modelled in the JSON trace format as inputs and outputs makes the investigation of RQ2 relatively simple. For this, we can simply remove one of the inputs from the traces while keeping the output. Thus, the output behaviour of certain transitions depends on the value of a variable which is hidden. For example, we can choose to remove the *aliens* variable input from the SPACEINVADERS traces to make all the *alienHit* events zero arity. Apart from *start*, whose arity decreases to two, all other events remain unchanged.

8.5 Results

This section presents the results of the experiments I performed to investigate the four research questions identified in Section 8.1.[6]

RQ1 How accurate are the models produced by my inference tool?

This research question is concerned with the accuracy of the models we can infer when the traces contain all the values used to compute the outputs. Here, I am interested in whether and how GP preprocessing affects the accuracy of the models inferred by my technique, and how these models compare to those which can be inferred by MINT [150].

Accepted Prefix Length

The first accuracy metric I consider is the accepted prefix length. Figure 8.7 shows the distributions for each configuration of both programs with the median shown in red. For LIFTDOORS, we can see that the PTA is able to process 55% of the average trace before there is a difference between there is a deviation between the behaviour of the system and that of the model. This is a strong starting point for inference and makes it more likely that we will end up with a good model of the system. Indeed, the "LIFTDOORS none" plot shows that, even without any preprocessing, my inference technique is able to produce a model which accepts the entirety of the average test trace, with relatively few outliers. With GP preprocessing, my inference tool is *always* able to achieve this. We can also see from Figure 8.7 that MINT also does a good job for LIFTDOORS, often inferring a model which can accept the average trace in its entirety, although there are quite a few outliers here where this was not achieved.

For SPACEINVADERS, the story is quite different. Here, the PTA is not so good a model of the system, on average only getting through about 20% of a trace before the point of deviation. Without GP preprocessing, my inference process was always able to produce a model which accepted every trace in its respective test set. Interestingly, the GP preprocessing slightly *decreases* performance here as there are some outlying models for which certain traces have a point of deviation before the end.

[6]The raw data is openly available at `https://figshare.com/s/9badfc5bff3643d1f02f`.

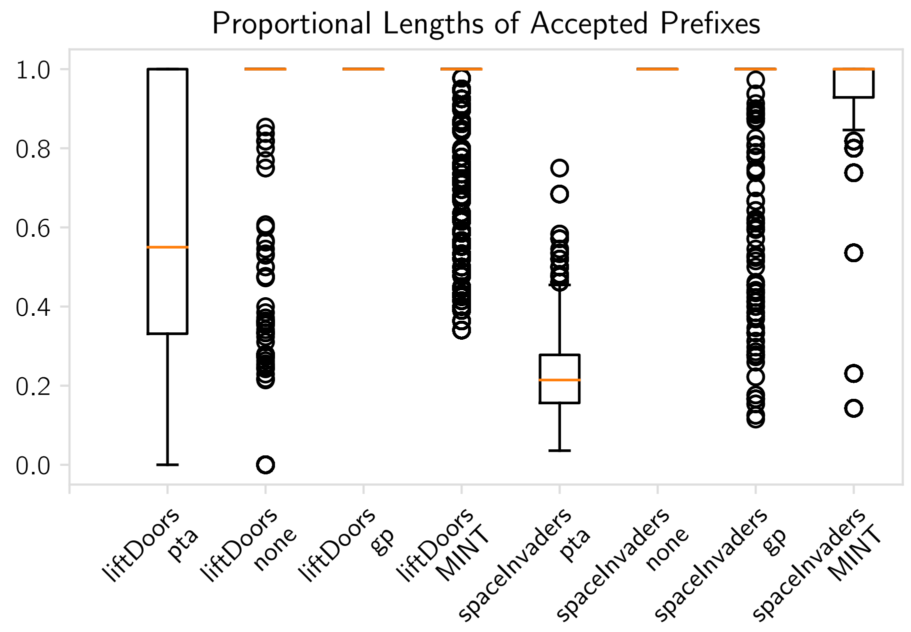


Figure 8.7: Proportional accepted prefix lengths.

The reason for this is that the output of the GP depends on both the training set and the random seed it is called with. Recall that the GP attempts to abstract away concrete values in favour of symbolic functions that explicitly compute output from input. If the initial set of traces contains every valid literal value, we can obtain a very accurate model without any kind of generalisation whatsoever. Even in these situations, certain random seeds can still cause the GP to produce inaccurate output or update functions, or even fail to discover them entirely. In such cases, we are forced to default back to the literal outputs from the PTA.

Since the generalisation step of my GP preprocessing technique involves dropping the guards on transitions, in cases where we have had to retain the literal values, we end up with transitions which produce concrete literal outputs but have no guards to restrict the input values. This severely limits the number of states we can merge because there is likely to be a lot of unresolvable nondeterminism. When this happens, we end up with states in the model that divide the full functionality of the underlying program state between them and are thus not able to respond to as many actions as they should.

This ties into the conclusion from [150] that program states with many outgoing transitions are harder to infer. In Figure 8.4, state q_1 represents an active game and has six outgoing transitions. Since we can either *win* or *lose* a game (but not both in the same trace), we know that no trace can contain every event which the system can perform. Thus, to infer the full functionality of an active game, we need to combine information from different traces by merging states. If we are forced to keep certain states separate which should really be merged, we cannot amalgamate their transitions. Thus, these states only have partial functionality — there are certain actions which they should be able to perform that are missing — meaning that some valid system traces will not be recognised.

Further evidence for this explanation can be found in Figure 8.11a. Here, we can see that there are certain instances of spaceInvaders for which the model inferred with my GP preprocessor has a slightly higher than average number of states. If we examine these cases individually, we can see that the instances with a higher number of states are exactly the instances with shorter accepted prefixes. While the actual models are too large to display effectively here, it is indeed the case that certain events have retained their literal outputs meaning that the functionality of the active game state has been split between several states.

MINT also does not do as well for spaceInvaders as it did for liftDoors. While its models are still, on average, capable of accepting the traces in the test set in their entirety, they do this less often than models inferred for liftDoors. The explanation for this is that MINT requires every event to report the value of every variable. Since spaceInvaders uses three state variables but each event only works with one, there is a lot of irrelevant information. In [150], it is identified that MINT struggles in situations like this because it gets confused by the surplus variables, and this is what is happening here. By contrast, my technique does not require that every variable be present in every event. Thus, each event only needs to record the inputs which are relevant, so my GP has much less surplus information to mislead it.

Analysing the individual experimental instances reveals that there are only three runs for which my GP technique was outperformed by MINT. In all other instances, my technique does at least as well as MINT, outperforming it in 779 experimental runs out of the 900. An inspection of these outlying instances reveals that, even here, the models produced by MINT are meaningless, even though two out of the three instances have perfect score. The transition guards inferred by MINT during inference severely hampered its ability to merge states and transitions, meaning that the structure of the model does not reflect the control flow of the underlying system. The models are too large to be displayed effectively in the format of this thesis, but I discuss the situation in more detail and provide figures in Section 8.6.

Having said that, the models inferred by my own tool in these instances are considerably larger than those inferred by MINT. As discussed above, when my GP fails to come up with a function, it must fall back the concrete values from the traces. Unfortunately, my preprocessing technique drops the guards on transitions regardless of the GP's success. In situations where the GP fails, we end up producing a concrete output without restricting the input. This leads to a lot of unresolvable nondeterminism when merging states, meaning we must keep more states separate and the inferred models are closer to the original PTA than the model in Figure 8.4. This turns out to be a common theme throughout this evaluation and is a limitation of my GP preprocessing technique in its current form. I discuss this in more detail in Chapter 10.

Sensitivity

Figure 8.8 shows the different sensitivities of the various configurations. The figure mostly tells a similar story to Figure 8.7. For liftDoors, we can see that the PTA generally accepts just over 20% of the traces in the test set, again indicating that the PTA is a strong start for the inference process. Here, the performance gain from my GP is more apparent than in Figure 8.7. Where GP preprocessing always gives the inference process the ability to infer a model with perfect sensitivity, if GP is not used, we average about 0.9. That is, 90% of the traces in the average test set are accepted in full. Here, MINT is able to infer models whose sensitivity is comparable with the ones inferred by my technique. In the majority of cases these models achieve perfect sensitivity, although there are a few outliers which do not.

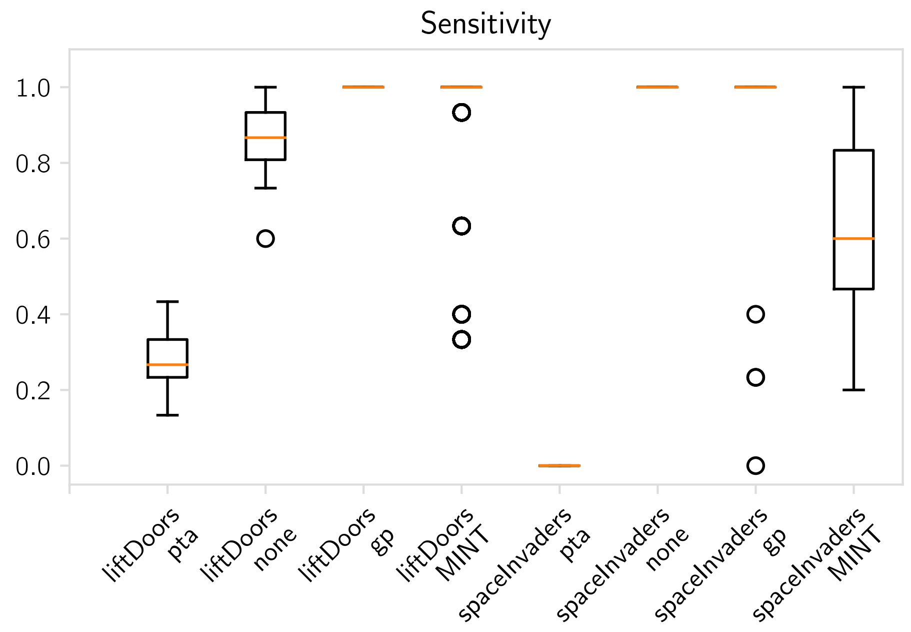


Figure 8.8: Sensitivity.

For spaceInvaders, the sensitivity of the PTA is always zero. No PTA built from any of the training sets accepts any of the traces in the respective test set in full. Despite this, my inference process can still infer a model with perfect sensitivity, even without any preprocessing. With GP, there are a few outliers with a less than perfect score. The explanation for this is the same as above. Analysing the individual runs reveals that there are only four instances where MINT outperforms my technique, three of which are the same as above. In all other instances, my technique does at least as well as MINT, outperforming it in 778 runs of the 900.

We can see that there are much fewer outliers in Figure 8.8 than for Figure 8.7. The reason for this is that each test set of traces gives us a single sensitivity data point, in Figure 8.7 we see one outlier for every trace which was not accepted in full and in Figure 8.8 we see one outlier for every model which did not accept the entirety of its test set. The fact that there are much fewer outliers on Figure 8.8 indicates that there are a few really bad models which reject a lot of traces rather than there being slight imperfections in lots of models, which sheds a more favourable light on my GP preprocessing technique.

Figure 8.8 reveals that the models produced for spaceInvaders by MINT have a much lower average sensitivity than those produced by my technique. While Figure 8.7 shows comparable performance between the techniques, this is because there are only a few outlying prefix lengths plotted on top of each other many times, meaning that their frequency is not visible.

The difference in performance between my system and MINT is, again, due to its requirement for every trace to show every variable value. This leads the inference to come up with spurious guards which restricts the number of actions the model can respond to. My preprocessing technique works only with the variables relevant to each event and explicitly drops the guards on transitions, making them able to respond to every input value.

Proportion of Correctly Executed Actions

Figure 8.9 shows the proportion of correctly executed events in each trace. There is relatively little new information here. Again, for liftDoors the PTA does relatively well on average, my technique with GP preprocessing produces a perfect score, and MINT appears to be comparable. The story is consistent for spaceInvaders as well. The PTA performs very poorly, but my tool is able to infer a perfectly accurate model without GP. The GP preprocessor introduces a few erroneous models, which are the same instances as for the previous metrics.

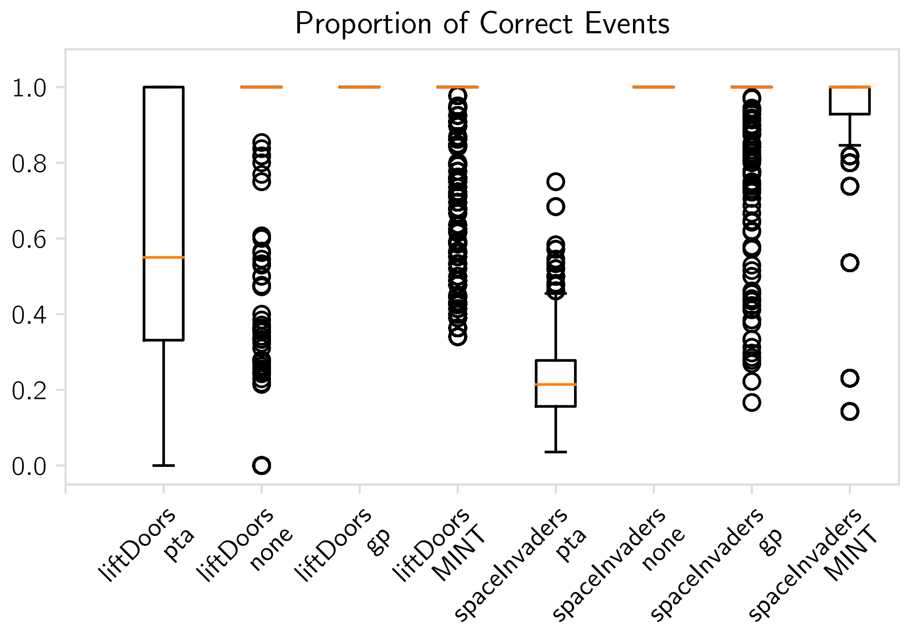


Figure 8.9: Proportions of correctly executed events.

Again, we can see that my technique performs better than MINT for the spaceInvaders system. This is not quite so apparent as in Figure 8.8, but the difference is still visible. Looking at the individual experimental runs, my technique is outperformed by MINT by the same three experimental runs, and outperforms it in 779 of the 900. This difference in performance is, again, explained by the fact that MINT requires every event to record the value of every variable, thus giving it a lot of surplus information which may mislead it when inferring both guards and update functions.

NRMSE

Finally, let us examine the NRMSE. The box plots in Figure 8.10 are slightly different from the other metrics. As mentionned in Section 8.3, lower NRMSE scores represent better models. Despite the fact that we know that neither PTA is a perfect model, all configurations achieve a perfect NRMSE apart from the plot for spaceInvaders with GP. The explanation for this is helped by considering the different trace parts, as discussed in Section 3.11. In situations where a trace can be written in the form xz, where x is the accepted prefix and z is the rejected suffix, no events are recognised after the point of deviation. This makes for a perfect NRMSE since the rejected suffix is not included in the metric.

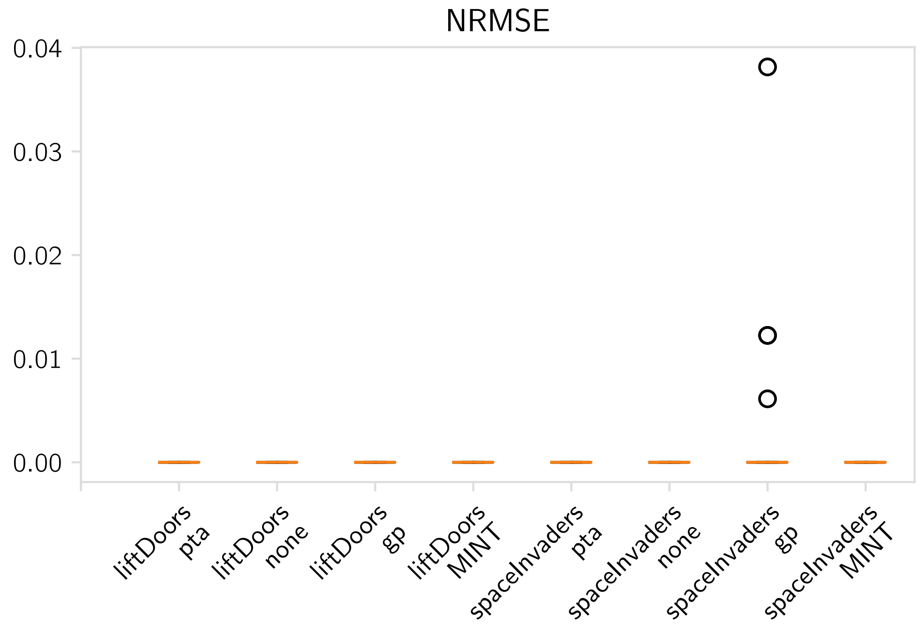


Figure 8.10: NRMSE.

The only configuration which does not score a perfect NRMSE every time is the one for my inference technique for spaceInvaders with GP preprocessing. Again, there are only a few outlying instances where this is the case, and these are the same instances as for the other metrics. The lack of a perfect NRMSE indicates that there is a portion of the trace after the point of deviation and before the unrecognised suffix where at least one event is *recognised* but is not *accepted.* That is, the output produced by the model does not match that produced by the system. This indicates that either the GP has come up with an output function which accounts for the training set but is not sufficiently general or could not find an output function so was forced to default to the literal values from the PTA. In the latter scenario, since guards are removed from transitions as part of preprocessing, the transition no longer represents a literal input-output pair and, instead, is able to respond to any action with the correct label.

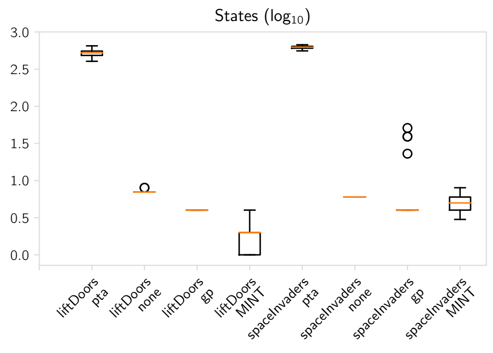
Interestingly, MINT is able to achieve a perfect NRMSE here despite the fact that the other metrics show that its models are not always perfect. Again, the explanation for this is that the traces have no recognised mid-portion after the point of deviation. This is interesting because my own GP sometimes leads to models with a non-zero NRMSE. The explanation for this is that the models inferred by MINT often still have guards on the transitions. This means that transitions cannot be taken unless they have an appropriate input. This makes a model more likely to reject an action rather than recognise it and produce an incorrect output. When this happens, the remainder of the trace does not contribute to the NRMSE. Unlike MINT, my GP removes guards from transitions so they can be taken with any input. While this widens the applicability of transitions, it makes it much more likely that incorrect output functions will reveal themselves as such.

Looking at the individual runs, my technique is outperformed by MINT for the same four instances where MINT achieves a higher sensitivity than my technique. What is happening here is that my GP has failed, meaning we have had to default back to the concrete values from the PTA. The transition guards are dropped regardless of this, meaning that certain states in the final model are able to react to inputs which elicit an incorrect literal output. By contrast, the guards inferred by MINT act in its favour here, limiting the inputs to which each state can react to those which elicit the correct response.

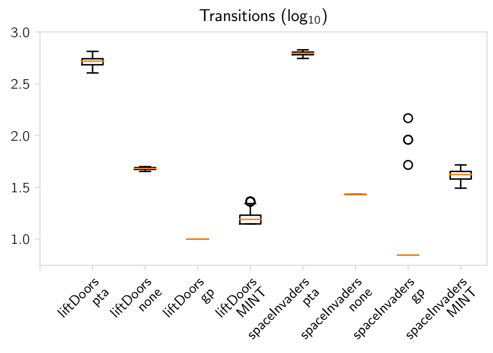
This does not mean the models are necessarily better, though. As discussed above, the models produced by both tools in these outlying instances do not adequately reflect the control flow of the underlying system. Additionally, placing guards on transitions makes the model less reactive, so the point of non-recognition is likely to occur sooner. Events after this cannot contribute to the NRMSE, so the model looks better even though it is less reactive. This is exactly the problem discussed in Section 3.11.

Model Complexity

Figure 8.11 shows the numbers of states and transitions in the inferred models for the various configurations, plotted using a log scale to account for the large range of values. While *scalability* is the subject of RQ4, it is worth considering model complexity here as well in the context of *accuracy*. Ideally, we want to infer a model with the same number of states and transitions as the underlying system. Additionally, models with fewer states and transitions are often easier to understand than those with many. This is discussed more in Section 8.6.



(a) Numbers of States.



(b) Numbers of Transitions.

Figure 8.11: States and transitions.

We can see from Figure 8.11 that my GP preprocessing technique often enables us to infer models with fewer transitions than we can without it, although the number of states is often very similar. The reason for this is that GP allows us to abstract away concrete values in favour of more generic functions. This allows us to merge a lot more transitions, so we end up with a smaller model. Without GP, we retain the concrete values from the original traces. Here, output is not *calculated* from input, rather, a literal value is produced as output as a mere *consequence* of the input. This means that we end up retaining a lot of transitions which are all *instances* of the same behaviour because there is no nondeterminism when we merge states. Not only does this lead to cluttered models that are difficult to follow, it is also not likely to be a particularly accurate representation of the system. Most programmers do not code functions as a series of *if input is x then return y* rules. Instead, they write more general and meaningful functions. It is these which the GP helps us to discover. This is discussed more in Example 8.5.3.

Figure 8.11 also shows that there are a few outlying model instances for spaceInvaders with GP preprocessing that have more states and transitions than the rest. As mentioned in RQ1, these are the same models which are less accurate. Here, the GP was unable to successfully come up with suitable output and update functions, meaning that more states and transitions had to remain distinct.

Figure 8.11 shows that, while MINT infers models with a similar number of states to my inference technique (with GP preprocessing), it often infers models with many more transitions. The reason for this is that MINT often retains guards on transitions. This means that it encounters less nondeterminism when merging states than my tool so has less difficulty merging states, but is forced to keep more transitions separate if their guards are not compatible. For my own technique, the inference of guards is the subject of RQ3, but it comes by default with MINT. Unfortunately, as we will see in Section 8.6, these transition guards are often not particularly readable and can clutter the inferred models.

Conclusions

This research question was concerned with the accuracy of the models we can infer when the traces contain all the values used to compute the output. For liftDoors, the ability to use GP to generalise away the concrete values on average led my inference technique to produce better models than without this ability. The models produced by MINT for this case study are comparable in terms of accuracy.

For spaceInvaders, my inference process was able to infer perfectly accurate models even without GP preprocessing. Consequently, the stochastic nature of GP was revealed as a weakness here. On the other hand, GP preprocessing did allow us to infer models with fewer states and transitions. The discovery of functions which explain and generalise the concrete values observed in the traces meant that we could merge more behaviour. Not only are these models likely to be easier to understand, they should also be more accurate representations of the underlying systems because the output is *computed* from input, rather than simply being produced as a consequence. This will be discussed in more detail in Section 8.6.

In this case study, the difference in performance between my system and MINT was quite apparent, with my technique producing significantly more accurate models. This is because MINT requires every event to report the value of every variable. Thus, it ends up with much more data to work with, much of which is irrelevant. This is identified as a weakness in [150] and leads to spurious transition guards being inferred during state merging, severely limiting the predictive power of the models.

RQ2 How does eliding variables affect the accuracy of the models produced by my inference tool?

This research question is concerned with how well my GP preprocessing technique from Chapter 7 is able to utilise *latent variables* to infer output functions for transitions when certain variable values do not appear in the traces. To answer this question, I modified the traces I used to answer RQ1 by removing one input variable at a time. This results in traces where the output of certain events depends on the value of a variable which is hidden. The task is for the GP firstly to discover that a register needs to be introduced, and secondly to infer *how* to use this register. No comparison between my technique and MINT is possible here since MINT is only designed to work with white-box traces and does not consider the possibility that there may be additional variables which are not visible.

Before examining the different accuracy metrics, let us first look at the bigger picture. When we elide variables, we make the task of generalising concrete values into functions much harder for the GP. This makes it much more likely that it will produce erroneous functions that force us to fall back on the concrete values from the traces. Because we drop transition guards as part of preprocessing, we must keep more states distinct to avoid nondeterminism. As before, this often means we end up with a model containing states that share between them the functionality of the underlying program state. Thus, the model will not recognise as many traces as it should.

Sensitivity

Figure 8.12 shows the sensitivity distributions for the various liftDoors and spaceInvaders configurations. For liftDoors, there is only one variable: the system *timer*. We can see here that obfuscating this variable has a slight negative impact on the average sensitivity of the inferred model, both with and without GP preprocessing. This is unsurprising since the GP has less information to work with. This makes it less likely to come up with the correct output and update functions, making it more likely that we will end up with a less accurate model.

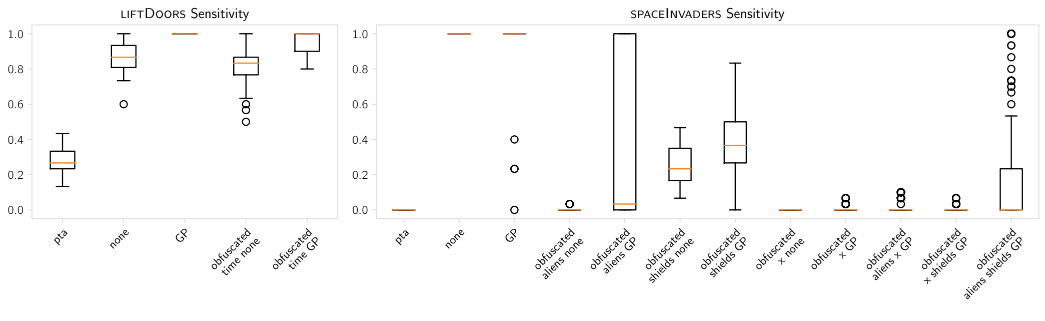


Figure 8.12: Sensitivities.

For spaceInvaders, there are three different variables which can be elided: the x coordinate of the gun turret, the number of *aliens* the player has hit, and the number of *shields* they have left. Figure 8.12 shows that any obfuscation gives a considerable drop in performance when compared to the original traces. We can also see that, no matter which variable is obfuscated, models inferred with GP perform at least as well (and usually better than) those inferred without it. Here, the GP plot with *aliens* obfuscated looks a little odd. This is due to the fact that the distribution is bimodal — either the inference does very well or very badly. It does badly slightly more often, thus the median is very low, but the split is roughly 50:50.

What is interesting in Figure 8.12 is the plots when *two* variables are obfuscated. Since obfuscating the x variable alone brings the median sensitivity down to zero, it is not surprising that further obfuscation leaves this unchanged. What is surprising, though, is the outliers in the plot with x and *aliens* obfuscated. These appear to be slightly *more accurate* than with the *aliens* variable present in the traces.

The reason for this is due to the update functions inferred by the GP. With the *aliens* variable present as an input to *alienHit* events, it just so happens that the input value is exactly the value which should be assigned to a particular register. For example, we might need to assign the value 2 to register r_2 to meet the obligations of a subsequent output function. When the *aliens* variable holds value 2, the GP can produce either $r_2 := 2$ or $r_2 := i_0$ as the update. In the latter case, this forces us to keep more transitions separate than if we had used the constant value since i_0 is not always 2. With the *aliens* variable elided, we no longer have access to the i_0 variable, so the constant is always used, enabling us to merge more transitions, and thus infer a more reactive model.

Accepted Prefix Length

Figure 8.13 shows the lengths of the accepted prefixes for the various models. This gives a more favourable picture than the sensitivity metric. Indeed, for liftDoors, we can see that, in the average case, there is no point of deviation in the traces. There are quite a few outlying instances where this is not the case, but the vast majority of these cases still get through at least half of the trace before the point of deviation. We can also see that there are much fewer such instances when GP preprocessing is used.

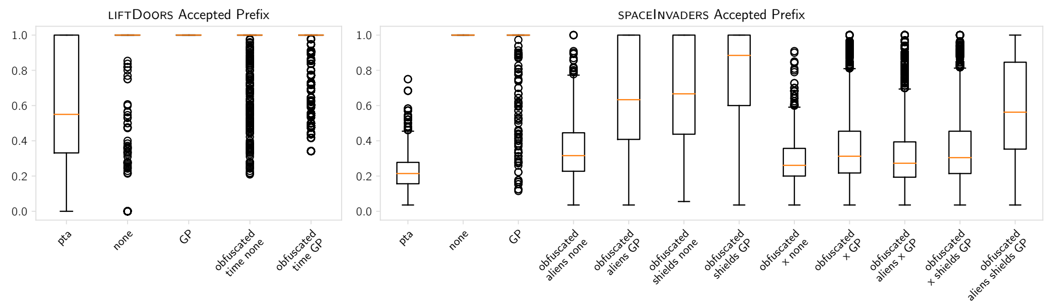


Figure 8.13: Proportional accepted prefix lengths.

The story is similar for spaceInvaders, with Figure 8.13 giving a much more favourable picture than Figure 8.8. Again, we can see that models inferred with GP preprocessing are always at least as accurate as those inferred without. What is very clear here is that *which* variable we elide has a much greater effect on model accuracy than the *number* of elided variables. When we obfuscate *aliens* and *shields* together, the drop in accuracy is roughly the sum of the respective drops when we obfuscate the two variables separately. The same, however, cannot be said when we obfuscate these variables in combination with x. Here, there is very little change in model accuracy. The reasons for this are discussed in detail at the end of this section.

Proportion of Correct Events

Looking at the proportions of correctly executed events shown in Figure 8.14, we get an even more favourable picture. For liftDoors with *time* obfuscated, the GP preprocessor often allows us to infer a model which can execute every event correctly, with relatively few outliers. This indicates that the GP is generally able to cope well with the elision of the *timer* variable. While it is still possible (and indeed quite likely) that my tool will infer a perfect model without GP preprocessing, there are many more outliers here and it is even possible to infer a model which can only execute 20% of events correctly. By contrast, nearly every model inferred with GP preprocessing achieves greater than 80%.

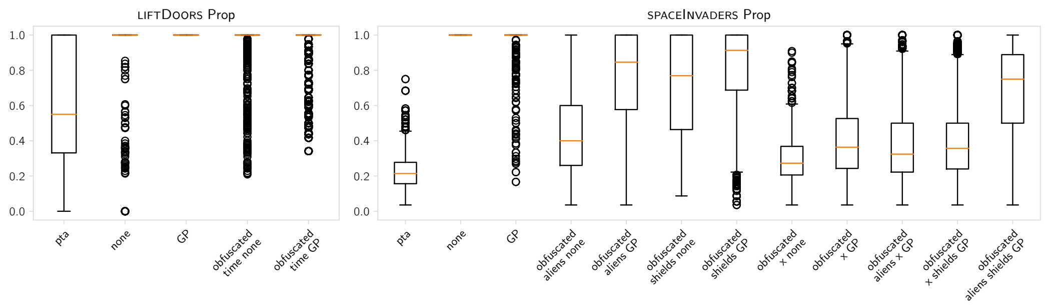


Figure 8.14: Proportions of correctly executed events.

For spaceInvaders, we again see that the accuracy of the inferred model depends primarily on which variable we have elided, rather than how many. We also see that the proportion of correctly executed events is usually very similar to the length of the accepted prefix. This indicates that not many events are executed correctly after the point of deviation, which means that the portion of the trace between the point of deviation and the point of non-recognition is either very short or contains many incorrectly executed events. The exception to this is the "obfuscated aliens GP" plot, which has a much higher proportion of correctly executed events. This means that the model was able to correctly execute many events after the point of deviation, so only a few transitions were problematic.

NRMSE

Looking at the NRMSEs shown in Figure 8.15, we can see that the accumulated difference between expected and actual outputs is usually quite small. As before, we cannot take this metric in isolation, but it can be helpful when considered in conjunction with the other metrics discussed above. This is particularly pertinent for the spaceInvaders case study. Here, we can see that GP preprocessing usually leads to models with perfect NRMSEs when *aliens* and *shields* are obfuscated, subject to the usual outliers. When taken in conjunction with the accepted prefix length and proportion of correctly executed events, this indicates that the point of non-recognition and the point of deviation are usually the same, meaning that the main source of inaccuracy is usually the inability to recognise events from certain states.

When the x variable is obfuscated, the NRMSE is much higher. This is interesting as it means that there is a substantial mid-section between the point of deviation and the point of non-recognition where the model is able to recognise events but is not able to correctly process them. It seems that the GP generally struggles much more to generalise behaviour when we obfuscate the x variable. The reasons behind this are discussed in the conclusions of this RQ.

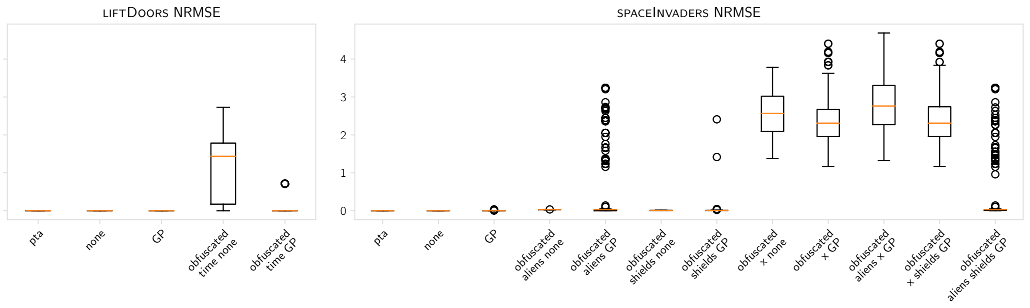


Figure 8.15: NRMSEs.

Model Complexity

Again, let us briefly consider model complexity in the context of accuracy. In Figure 8.16, we can see that the size of a model is related to its accuracy in terms of the other metrics. As for RQ1, it seems that the configurations where the inferred models were less accurate generally had models with more states. The explanation for this is the same as before: if we are forced to keep states separate which should be merged, we end up with functionality being split between sets of states, and models which cannot accept as many traces as they should.

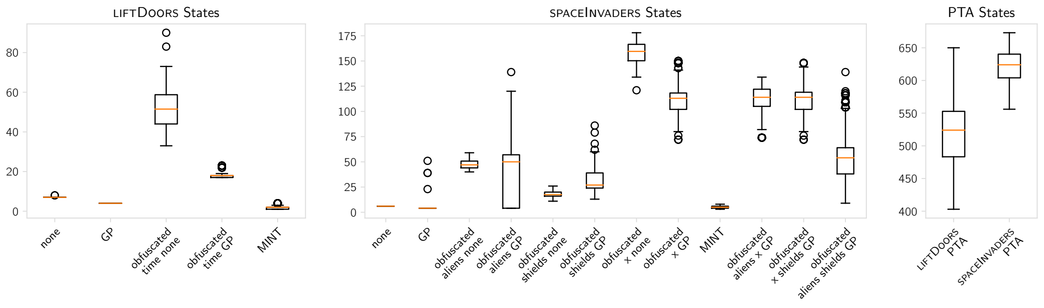


Figure 8.16: Numbers of States.

Particularly noteworthy here are configurations for spaceInvaders where the x variable has been obfuscated, as these models have by far the most states. This provides further evidence that the GP was not often able to infer a suitable function for these events and was forced to default to the original literal outputs from the PTA. The reasons behind this are discussed in the conclusions of the RQ. Because the majority of events are *move* events, if we were forced to default back to the literal outputs of these events, the inferred model would be almost identical to the one inferred without any preprocessing. This is also why obfuscating additional variables gives very little change in the size of the model.

Also noteworthy is the configuration with the *shields* variable obfuscated. Like the other configurations, GP preprocessing leads to more accurate models, but they have slightly more states than without preprocessing. This is because the GP preprocessor drops the guards on transitions. Thus, if it is unable to infer an output function for the *shieldHit* transitions, it will have to leave more states separate. On the other hand, it is still able to infer output functions for the *move* events since the x input to these transitions has not been obfuscated. Because the vast majority of events in the traces are *move* events, the ability to correctly infer the appropriate output functions for these transitions has a much greater impact on the accuracy of the model.

Conclusions

This research question was concerned with how obfuscating variables affects the accuracy of the models my tool can infer. The first conclusion we can draw from the above results is that GP preprocessing generally enables my technique to infer more accurate models when this is the case. This is in keeping with the observation from [106] that using literal equality as a merging criterion tended to produce less accurate models. While this observation was made in a slightly different context, it is relevant here as well since, without the ability to generalise away concrete values, only syntactically equivalent transitions can be merged. With GP preprocessing, we drop the guards on transitions, thus enabling models to continue processing traces even if the transitions are not able to generate the correct output in response to the input.

There second conclusion we can draw is that *which* variable is elided seems to have a much greater effect on the accuracy of the inferred model than *how many* variables we elide. When we elide just one variable, my technique generally does much when the *aliens* or *shields* variable is elided than for x. When we elide two variables, if one of these is the x variable, there is generally very little change in model size or accuracy.

Part of the reason for this is the relative frequencies of the events in the traces which depend on these variables. Both the *alienHit* and *shieldHit* events occur very infrequently in the traces, with the majority of events being either *moveEast* or *moveWest*. This means that problems with these events have a much greater effect on the inferred model and are more likely to be revealed by the traces in test set. The events concerning *aliens* or *shields* occur much less frequently, so problems with the corresponding transitions have much less of an effect on the model.

This is especially true when we have elided variables in addition to x. Because *alienHit* and *shieldHit* events occur so infrequently, the GP often ends up with very small or singleton training sets. This means it is often fails to produce an output function or simply sticks to the original concrete values, even when the corresponding variable is present. Thus, obfuscating it in addition to x has very little effect on the inferred model. The effect of obfuscation is further diminished by the fact that it is quite likely that we will encounter a *move* event which the model fails to execute correctly (or at all) before we have encountered an *alienHit* or *shieldHit* event. Thus, problems with these events are more likely to be masked when x has been obfuscated.

The reason that obfuscating the x variable has such a detrimental effect on the accuracy of the inferred models is that there are *two* actions which are affected by the obfuscation of x (*moveWest* and *moveEast*) where the other two variables are used by only one event each. Without the obfuscation, both transitions take in a single input which represents the anterior state of the variable and produce a single output which is its updated value. The two actions are each other's inverse: one adds 50 to x and the other takes 50 away.

While LIFTDOORS also has multiple transitions which update the *time* variable, traces are laid out in such a way that *time* is always reset before it is mutated, meaning that it is possible to infer update functions which "play well" with each other. With SPACEINVADERS, this is not the case since both *moveWest* and *moveEast* mutate the x coordinate, respectively decreasing and increasing it by 50. Since there is no transition to reset x between blocks of *moveWest* and *moveEast* actions, update functions cannot be inferred for the two transitions in isolation.

Subdividing structural groups by history mitigates this a little, since the GP is often able to infer the correct output and update functions for the first few *move* transitions. More specifically, it can infer output and update functions up to the point where we have a *moveWest* after a *moveEast* after a *moveWest* or vice versa. Other non-*move* actions can obviously occur at any point in between since these do not affect the x variable.

The reason for this is that each transition group gets its own register, meaning that *moveWest* and *moveEast* update distinct variables. The GP is usually smart enough to realise that at the start of the first block of *move* actions, x holds the value 200 which can be set by the *start* transition. After this, *move* transitions either increase or decrease the variable. When the direction switches, the second register representing x can also hold a value such that increasing or decreasing it by 50 produces the correct output, as this can also be set by the *start* transition.

Example 8.5.1. Consider the following traces where the x variable has been obfuscated.

$$\langle start(3,0)/[200,3,0], moveWest()/[150], moveWest()/[100], moveEast()/[150], moveEast()/[200], moveWest()/[150]\rangle$$
$$\langle start(3,0)/[200,3,0], moveWest()/[150], moveWest()/[100], moveWest()/[50], moveEast()/[100], moveWest()/[50]\rangle$$

In a PTA built from these traces, there would be two *moveWest* groups: those which follow *start* and those which follow *moveEast*. These are coloured red and blue respectively. There is only one *moveEast* group. The three groups are each given their own register when output functions are inferred, and each simply outputs the content of this register. The problem comes when we try to infer updates. All goes to plan for the *moveWest* group that follows *start*. The GP can successfully infer that the register r_1 must be updated to $r_1 - 50$. Similarly for the *moveEast*, the GP infers that the register r_2 must be incremented by 50 each time.

Because a third register, r_3, is used for the second group of *moveWest* transitions (blue), the GP is left with inconsistent training sets when it tries to infer updates. For the top trace, we need r_3 to be 150 when it is evaluated by the blue *moveWest*. For the bottom trace, we need it to be 50. There is no single update expression which can be added to any of the transition groups to achieve this, so update inference fails here. This means we are left with the default output values from the PTA for the blue *moveWest* group. The same principle applies for longer traces which contain multiple direction switches.

When the direction changes for a second time, the various registers representing the underlying x variable are no longer in sync. The GP cannot work out what happens after this, so defaults back to literal outputs for all subsequent *move* events. This reveals a limitation in the current way of processing which is that we cannot recognise when two or more transitions need to update the same variable. For systems like SPACEINVADERS with relatively simple output and update functions, the underlying behaviour is reasonably obvious to a human, but it is extremely difficult to automate this intuition in a way which applies generally.

RQ3 How does the ability to discover value-dependent behaviour affect the accuracy of the models produced by my inference tool?

Let us now examine how the ability to discover and distinguish *value-dependent behaviour* affects the models produced by my inference tool. As discussed in Section 8.2, neither liftDoors nor spaceInvaders exhibits any value-dependent behaviour. For this RQ, I used a slightly modified version of the spaceInvaders case study and a more complex version of the simple drinks machine. Again, I cannot compare my tool to MINT here since its transitions do not have outputs, so there is no observable behaviour to distinguish.

Sensitivity

Figure 8.17 shows the sensitivities for the different configurations. The original PTAs have a sensitivity of zero in all but a few outlying cases of drinks. The interesting case study here is spaceInvaders, since the same (slightly modified) traces were used for inference here as for RQ1. Particularly interesting is the drop in accuracy when GP is used without the ability to infer guard functions. For RQ1, GP achieved a perfect sensitivity, subject to a few outliers. Here, we only have a sensitivity of about 0.1. This indicates that even a small amount of value-dependent behaviour significantly increases the difficulty of the inference challenge.

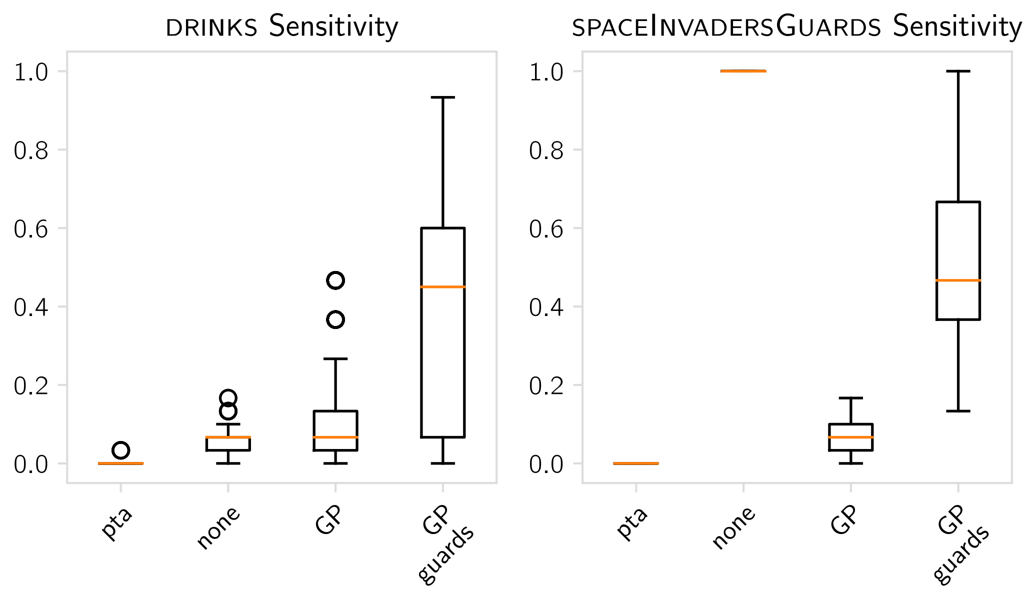


Figure 8.17: Sensitivities.

Even more interestingly, inferring models of spaceInvaders without any kind of additional support is quite clearly the best strategy here and achieves a perfect sensitivity. The reason for this is that every output is purely determined by its input. Even the value-dependent behaviour on the *alienHit* and *shieldHit* transitions is determined by the input. Without any preprocessing, the literal input guards from the PTA are retained, and the inference process infers pretty much the same model as it did for RQ1, except that the *win* and *lose* transitions are replaced with $alienHit : 1[i_0 = 4]/o_0 :=$ "win" and $shieldHit : 1[i_0 = 1]/o_0 :=$ "lose" respectively.

The critical thing here is that the single active-game state can be inferred as the literal input guards stop there from being any nondeterminism between non-identical transitions. Thus, the inferred EFSM has the same shape as Figure 8.4. The GP preprocessor drops the guards on transitions to increase generality, meaning that we must keep more states separate as we have no way to resolve the value-dependent behaviour of the *alienHit* and *shieldHit* transitions. As previously discussed, this leads to less accurate models since the functionality of an active game is spread across many states rather than being condensed into one.

For DRINKS, the story is very different. Here, models inferred with GP preprocessing achieve, on average, the same sensitivity as those inferred without, but higher sensitivities are not unusual. Despite this, the sensitivities are still very low without the ability to infer distinguishing guards. This is because the value-dependent behaviour here is much more complicated than for SPACEINVADERS. As discussed in Section 8.2, the output of the *vend* action depends not just on how much money has been inserted so far, but also on which drink was selected. In the real system, both of these values are stored in variables which do not appear in the traces. Consequently, *vend* takes no input and it is much harder to infer what is going on.

The problem here is that the GP preprocessor must come up with exactly the correct output and update functions for all the transitions. Even if this is the case, the value-dependent behaviour of the *vend* transition often means that the nondeterminism introduced by dropping the guards on transitions often cannot be resolved by merging transitions alone. The initial set of traces must be conducive to this. If it is not, the inference process must default back to the original PTA, and we end up with the same model we would get without GP preprocessing. Even when we can resolve the nondeterminism introduced at the preprocessing stage, we often cannot merge many more states because of the nondeterminism this introduces.

For both case studies, the ability to infer guards to distinguish transitions during inference results in a huge increase in the sensitivity of the final model when GP preprocessing is used. This should not come as too much of a surprise since models of systems with value-dependent behaviour must also contain value-dependent behaviour in order to be truly accurate. The only way to achieve this is with transition *guards*. If we do not provide the ability to infer these, we significantly limit the accuracy of any model we can infer.

The underlying explanation as to why the ability to distinguish transitions leads to an increase in accuracy is, again, to do with how states are merged. More specifically, when we merge states, we need to resolve any nondeterminism which arises by merging further states and transitions. If a resulting pair of nondeterministic transitions cannot be merged, we cannot merge the original state pair. By providing the ability to infer guards which *distinguish* transitions that cannot be merged, we provide an alternative tool for resolving nondeterminism. This means that more states can be successfully merged. As discussed above, if we can merge more state pairs, we generally end up with a model which can accept more traces.

Accepted Prefix Length

Figure 8.18 shows the lengths of the accepted prefixes for the different configurations. Again, we can see that the ability to infer guards during inference gives a significant increase in accuracy when GP preprocessing is used. Looking at this figure we can see that, despite initially seeming very simple, it appears that DRINKS is the more difficult case study. The sensitivity values in

Figure 8.17 are relatively similar between the case studies for the two GP configurations, but Figure 8.18 shows a real difference between them. While the average accepted prefix lengths for the respective PTAs are similar, the average accepted prefix for spaceInvaders are noticeably longer than for drinks both with and without the ability to distinguish transitions.

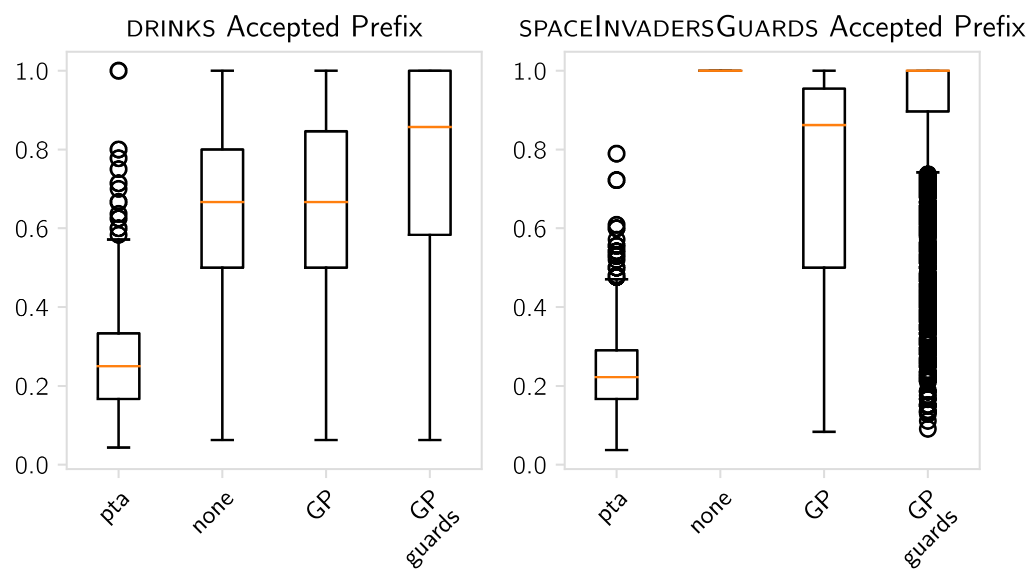


Figure 8.18: Proportional accepted prefix lengths.

There are two main factors behind this difference in performance. The first is the *position* of the value-dependent behaviour within the traces. With spaceInvaders, the value-dependent behaviour only occurs at the end of each interaction. Either a player hits an alien and wins the game, or the player is hit by an alien and loses the game. Thus, it is the final event in each trace where the behaviour needs to depend on a value. This make it possible to do quite a lot of merging without needing to infer any guards. With drinks, the customer may attempt to *vend* their drink any time after selecting it. This means that we are much more likely to encounter value-dependent behaviour earlier on in the merging process. If we have the ability to infer guards, and do so incorrectly, the point of deviation will occur much earlier in the traces than for spaceInvaders. If we do not have the ability to infer guards, we will be able to merge far fewer states and thus end up with many states which share functionality between them.

The second factor is the *complexity* of value-dependent behaviour exhibited by the two systems. While spaceInvaders has two value-dependent actions (*alienHit* and *shieldHit*), the behaviour of these events is relatively simple and depends on the value of a single variable. By contrast, for drinks, only the *vend* action exhibits any value-dependent behaviour, but it's behaviour is dependent on both the drink which was selected and the price of that drink. This makes it much more likely that the GP will fail to find a distinguishing guard and, as a result, will fail to resolve nondeterminism when merging states. Alternatively, if the GP is able to find a distinguishing guard, it is much more likely to be incorrect or not sufficiently general.

Something else which is interesting here is that, while the *sensitivity* of models inferred for spaceInvaders with GP preprocessing but without distinguishing guards is very low, the average length of the accepted prefix is quite high, around 90%. This is about the point in the traces where we might start to expect to encounter value-dependent behaviour. That is, when it becomes important to distinguish the behaviour of the *alienHit* and *shieldHit* transitions.

The same is true for drinks. The average point of deviation occurs towards the end, when the amount of money the user has inserted begins to approach the cost of the drinks. At this point, the output of the *vend* transition depends on the exact values of the two variables. Without the ability to infer guards on data values, we must to encode these conditions within the control state of the model. As with outputs, this is not what we really want to do as it leads to overly specific models. When we can distinguish behaviours by inferring guards, we can push back the point of deviation. Even if the guards we infer are not very accurate, they enable us to merge states which could not be merged otherwise. This means that there are fewer states between which the underlying functionality is split, so the model can accept more traces.

Proportion of Correct Events

Figure 8.19 tells a similar story to Figure 8.18. In fact, the two figures are almost identical. This indicates that the point of non-recognition comes very quickly after the point of deviation in most cases, which indicates that the main source if inaccuracy in the inferred models is the inability to *recognise* events rather than the inference of inaccurate output or update functions. The main explanation for this is the fact that the value-dependent behaviour means we cannot merge as many states as we would like to, even when we do have the ability to infer guards. This means that our inferred models still have lots of states which share the behaviour of the underlying program state between then, so lack certain transitions.

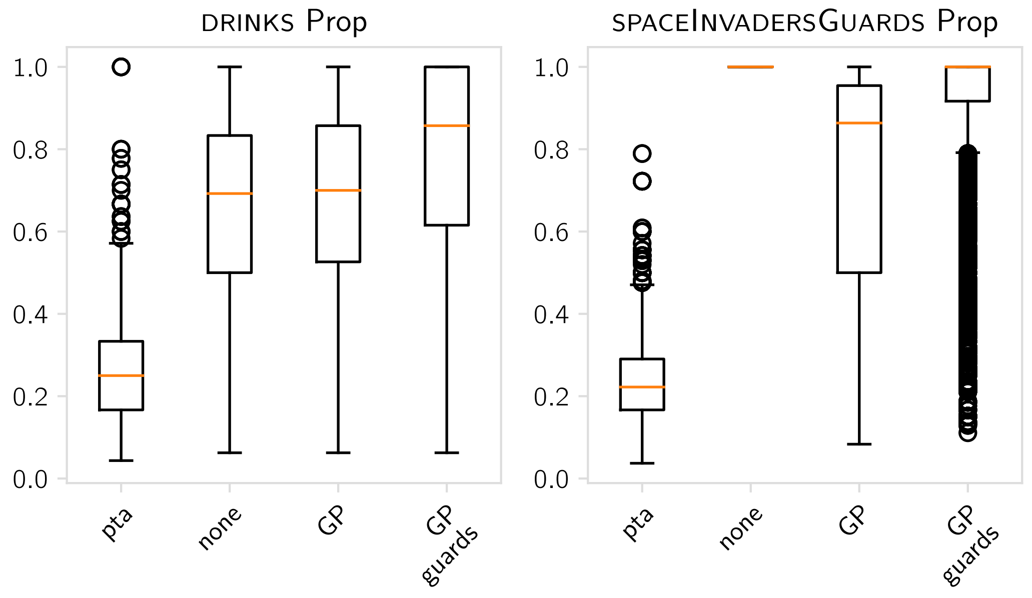


Figure 8.19: Proportions of correctly executed events.

NRMSE

Figure 8.20 shows the NRMSEs of the configurations of each case study. The story here is consistent with the other metrics in that we get a much better average score when we provide the ability to infer guards. The figure also provides further evidence that DRINKS is the more difficult case study, since its NRMSEs are significantly higher than those of SPACEINVADERS. Because the NRMSEs for SPACEINVADERS are so low, this provides further evidence that very few events are recognised after the point of deviation.

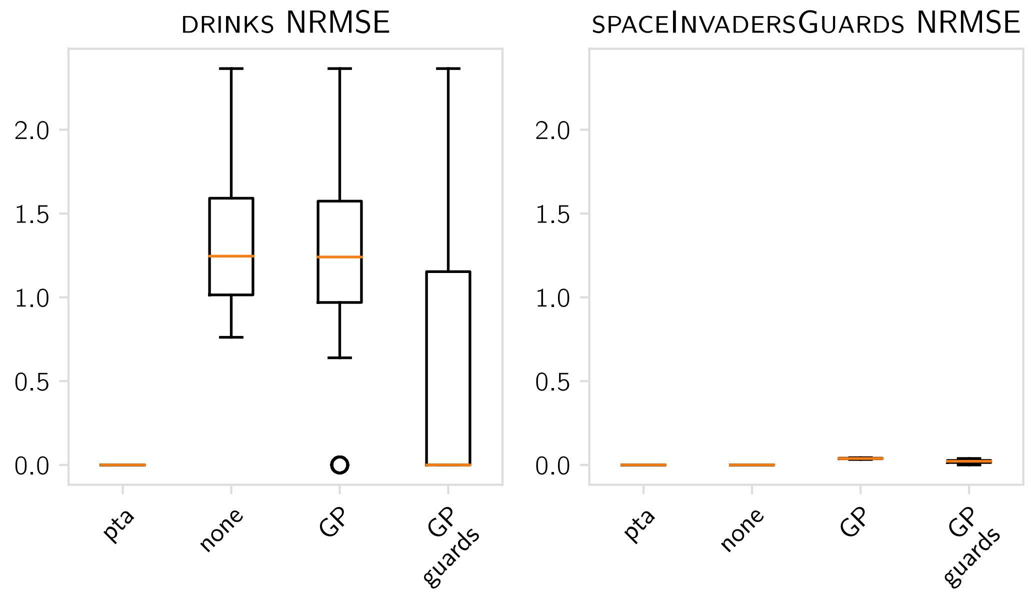


Figure 8.20: NRMSE.

The fact that the NRMSEs for DRINKS are quite high indicates that the models are able to continue processing at least some events after the point of deviation. Figure 8.19 shows that the number of events is not high, but it is certainly at least one. Assuming we are always able to select a drink, the only transitions which can be responsible for this are *coin* and *vend*.

Without any GP preprocessing, the NRMSE is quite high, but when we preprocess and provide the ability to infer guards the average NRMSE is very low. This suggests that the main source of error is the *vend* transitions, which have zero arity so are unguarded. By contrast, the *coin* transitions have inputs which (without GP) are guarded to ensure that they produce the correct output for each input. When we provide the ability to infer guards, the average NRMSE drops significantly, implying that the guarding the *vend* transitions helps to increase the accuracy of the model, at least until the point of non-recognition.

Model Complexity

Let us now consider model complexity in the context of accuracy. The aim of inferring guards during inference is to allow value-dependent behaviour to appear explicitly in transition guards, rather than having to implicitly encode it within the control flow states. This means that, when we have the ability to infer guards during inference, we should be able to infer smaller as we are able to resolve more of the nondeterminism which arises when states are merged.

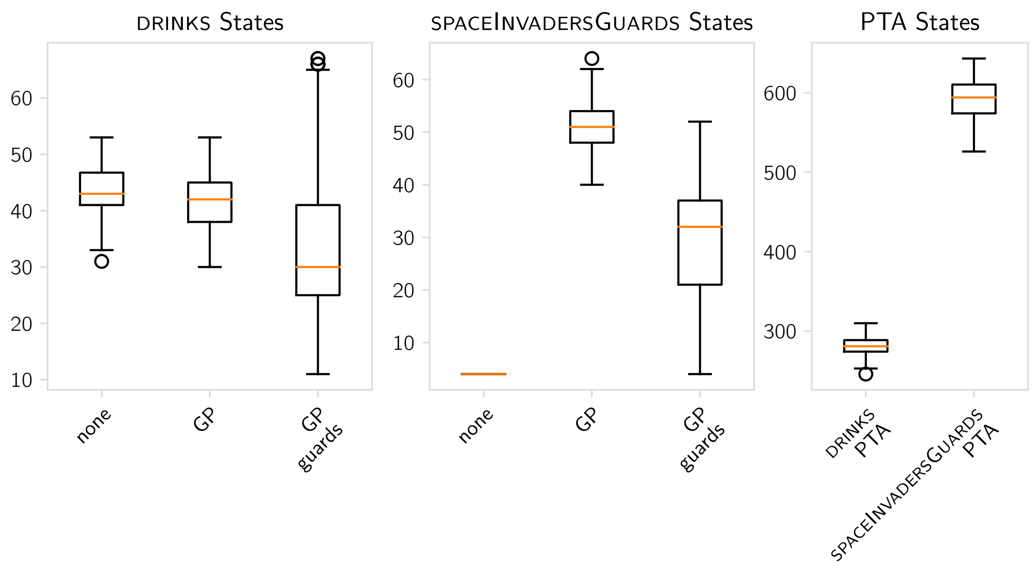


Figure 8.21: Numbers of States.

Figure 8.21 shows that this is, indeed, the case. For both case studies, the average number of states in the inferred model is nearly halved when we provide the ability to infer guards. As for RQ1 and RQ2, we can see that the size of the model is related to its accuracy in terms of the other metrics, with smaller models seeming to be more accurate. The story is similar for the numbers of transitions, so the graphs are omitted here.

While the ability to infer guards gives us a *smaller* model, this does not necessarily mean that the inferred model is easier to understand. Because we call the GP to infer guards at merge time, this often means that we end up with more guards on transitions than we need.

Example 8.5.2. When inferring the EFSM for one particular configuration of the simple drinks machine, we end up with the following *vend* transition.

$$\mathit{vend} : 0[r_1 > 59, r_1 \neq 0, r_2 \neq \text{“coffee”}, r_1 \neq 110]/o1 := r2$$

While the guards on this transition may be sufficient to distinguish it from its immediate neighbours, they are not particularly *meaningful* and are not sufficiently *general* that it can be successfully merged with all similar transitions. We can also remove the guard $r_1 \neq 0$ since, if r_1 is greater than 59, it cannot be equal to zero.

The cause of this issue is that guards are inferred at merge time. This means that the GP often has to work with very small training sets, so we are likely to end up with oddly specific guards that solve the problem at hand but do generalise well. When applied to the same transition several times throughout the inference process, we end up with quite a collection of guards, as illustrated in Example 8.5.2. To resolve this, we need the ability to *simplify* transition guards such that we can remove redundant guards from earlier on in the inference process. An implementation of this would be relatively easy to achieve with something like the Z3 simplifier, which we have already seen used in Chapter 7, but is left for future work.

Another weakness with the way guards are inferred is that we only infer a guard for one transition and apply its negation to the other. In Example 8.5.2, this means that there is a transition with guard $r_1 = 0$ and another with $r_1 = 110$. This kind of guard works well in the short term to resolve nondeterminism, but does not work well in the long term since literal equalities like these are too specific. In such situations, we may end up with many parallel transitions which are not merged together because there is no nondeterminism between them.

What we would ideally like is to provide the GP with larger training sets so that we can infer better guards. The only way to achieve this is to group certain transitions and infer guards for them together. Essentially, we would be partitioning transitions into equivalence classes and then inferring guards to keep these classes separate where this may be required. While this is certainly a desirable feature, its realisation would be an entirely different technique to the one presented in this thesis. Consequently, this is left for future work.

Conclusions

This research question addressed how the ability to discern value-dependent behaviour affects the accuracy of the model we can infer. From the accuracy metrics considered above, the results seem quite positive. For systems with value-dependent behaviour, the ability to infer guards which distinguish transitions allows us to infer a much more accurate model when GP is used. Providing this ability means that we can merge more states in the model and, thus, not have to keep separate states in the model which represent the same program state.

Having said that, if the output of every action depends purely on the input, GP preprocessing may not lead to such a good model even if we give the ability to infer guards. This is because preprocessing drops the guards on transitions which means that, unless it can do a perfect job of inferring the output functions, it will be unable to merge as many states. The value-dependent behaviour here not only makes this task more difficult, but is also much less forgiving when the GP makes mistakes. Without GP, we keep the literal guards from the PTA, so merging states very rarely introduces nondeterminism. This means that many more states can be merged, thus condensing the underlying functionality into fewer states which can perform more actions.

RQ4 How well does my tool scale?

Let us now investigate how my tool scales. There are two aspects to this: the complexity of the inferred models in terms of the number of states and transitions, and the runtime of the tool. Model complexity is a property of the *technique* detailed in Chapter 6. Runtime is a property of the *implementation*, which is also affected by the environment in which the tool is run.

RQ4a How large are the inferred models in terms of states and transitions?

We have already seen model complexity explored in the context of *accuracy.* In situations where we can merge many states, our models tend to be more accurate than when we must keep separate states which should really be merged. While the accuracy of a model is often the top priority, it is not the only metric by which we can measure the quality of an inference technique. If the intended use of a model involves it being inspected by a human, it is obviously helpful if the model is not only accurate but also easy to *understand.* This is discussed more in Section 8.6, but models with fewer states are often easier to comprehend.

Figure 8.16 shows the numbers of states for the various configurations used to answer RQs 1 and 2. Without obfuscated variables, my inference process almost always manages to infer models of both systems with less than 10 states, whether or not the GP preprocessor is used. When it can successfully infer output and update functions, the GP preprocessor enables slightly more states to be merged, but the improvement is not particularly significant since the number of states we can achieve without GP is so low to start with.

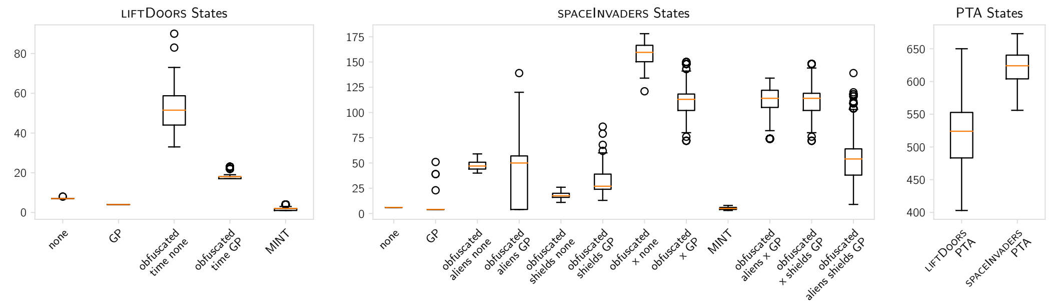


Figure 8.16: Numbers of States.

The advantage of GP preprocessing is more apparent when we consider the numbers of *transitions* in the inferred models, as shown in Figure 8.22. Here, we can see that models inferred with GP preprocessing have, on average, much fewer transitions than those inferred without it. The reason for this is that the preprocessing technique aims to abstract away concrete values into output functions. This has the effect of making many transitions in the PTA identical, meaning they can be trivially merged.

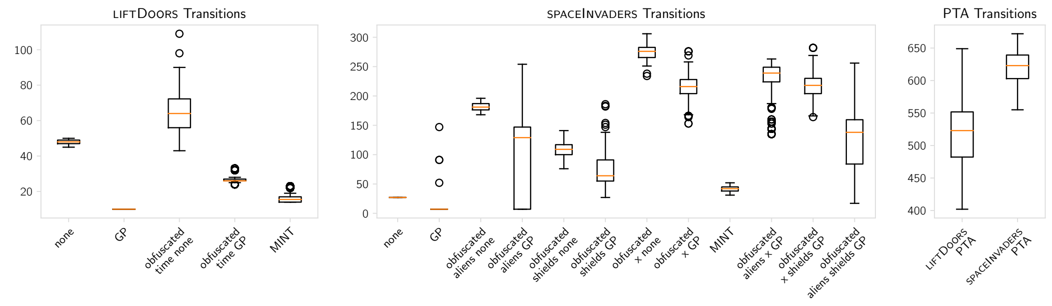
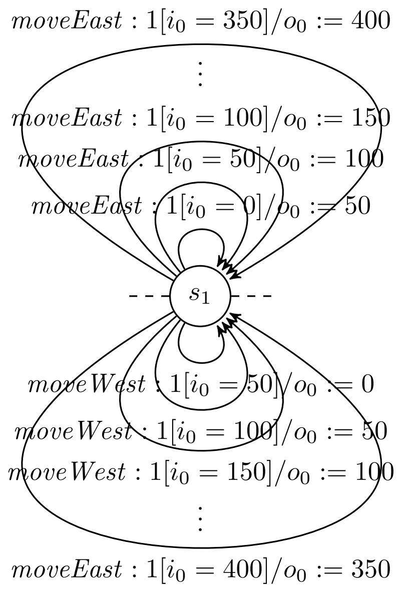


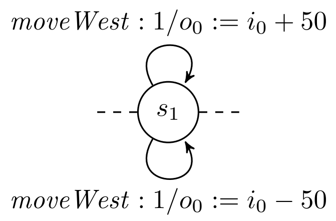
Figure 8.22: Numbers of Transitions.

Without GP preprocessing, transitions retain their literal input guards and output values. Very little nondeterminism arises when we merge states, so transitions which perform the same action with different data remain distinct in the inferred model. This often leads to final models which contain a lot of "parallel" transitions. GP preprocessing does not affect the number of *states* we can merge, but it does affect what happens to the *transitions*. Here, we generalise behaviour and remove guards, meaning that many more transitions can be merged.

Example 8.5.3. Figure 8.23a contains a fragment of a model of spaceInvaders inferred without GP preprocessing. The *moveEast* and *moveWest* transitions respectively increase and decrease the value of their input by 50, but do this by producing a literal output value in response to a literal input input. This means that we need a separate transition for each possible input/output pair. Since the play area of the game is 400px wide, we need eight of each transition.


$moveEast : 1[i_0 = 350]/o_0 := 400$
⋮
$moveEast : 1[i_0 = 100]/o_0 := 150$
$moveEast : 1[i_0 = 50]/o_0 := 100$
$moveEast : 1[i_0 = 0]/o_0 := 50$
s_1
$moveWest : 1[i_0 = 50]/o_0 := 0$
$moveWest : 1[i_0 = 100]/o_0 := 50$
$moveWest : 1[i_0 = 150]/o_0 := 100$
⋮
$moveEast : 1[i_0 = 400]/o_0 := 350$

(a) Literal transitions for spaceInvaders.


$moveWest : 1/o_0 := i_0 + 50$
s_1
$moveWest : 1/o_0 := i_0 - 50$

(b) General transitions for spaceInvaders.

Figure 8.23: Parallel transitions for spaceInvaders.

Figure 8.23b shows the same fragment of a model inferred with GP preprocessing. Here, we have generalised the output functions and removed the guards such that we can use a single transition to represent each of the *move* actions. Not only does this model have fewer transitions, but it is also much easier to see how the system uses the data values.

If we halved the x increment from 50 to 25, we end up with twice as many distinct input-output pairs for *moveWest* and *moveEast*. Without GP preprocessing, the final inferred model would have twice as many *move* transitions as in Figure 8.23a. With GP, the output functions would change from $i_0 \pm 50$ to $i_0 \pm 25$, so there would be no change in the number of transitions in the inferred model.

In terms of scalability, what Example 8.5.3 shows is that the GP preprocessor allows us to infer models for which the number of distinct input-output pairs in the original traces does not affect the number of transitions in the model. Thus, the difference in transitions between using GP and not using it is proportional to the number of distinct inputs with which each action in the traces is called. This is, of course, dependent on the fact that the GP is able to successfully infer a function which correctly generalises the input-output pairs in the training set.

While MINT is often able to infer models with the same number of states as my technique, those models often have more transitions. This is especially true for SPACEINVADERS. As previously discussed, this is to do with the fact that MINT is less effective in situations where there are many state variables. Where my technique is forced to keep states distinct if it cannot merge transitions, MINT is able to infer distinguishing guards by default so is able to resolve nondeterminism this way. As we will see in Section 8.6, the guards MINT infers are often overly specific, so it often keeps more transitions separate than is really necessary.

Let us now examine how the obfuscation of variables affects the complexity of the models inferred by my technique. We cannot compare with MINT here since it is only designed to work with white-box traces. Figure 8.22 shows that obfuscating the *time* variable in LIFTDOORS leads to larger models than without the obfuscation. The explanation for this is the same as in Example 8.5.3. If we are able to generalise the behaviour of transitions, we are better able to resolve the nondeterminism which arises as a result of merging states.

What Example 8.5.3 does not explain is the reduction in the number of *states* in the models inferred using GP with *time* obfuscated. For this, we must take a closer look at how obfuscation works. When we obfuscate a variable, we remove its values from the traces. Transitions with arity one, with their input obfuscated, become transitions with arity zero. Input guards are lost as a result of this, but the literal outputs are retained. This means that there is much more potential nondeterminism when states are merged. Without the ability to generalise outputs into functions, we cannot resolve nondeterminism by merging transitions, so states which could be merged with the inputs present must now remain distinct. Consequently, with variables obfuscated, the ability to generalise behaviour is critical if we want to infer small models.

Figures 8.11a and 8.22 show that the situation is similar for SPACEINVADERS. Like with LIFTDOORS, the models inferred for SPACEINVADERS with GP preprocessing seem to have significantly fewer transitions than those inferred without it. Figure 8.11a shows that the difference in the numbers of states of models inferred with GP and without is not so great for SPACEINVADERS. The reason for this varies depending on the variable which has been obfuscated.

For *aliens* and *shields*, the explanation is to do with the relative frequencies of the respective events in the traces. Each variable is only involved in a single action, neither of which occurs particularly frequently. This means that the GP often has insufficient training data to infer a general function and instead keeps the literal output values, so fewer states can be merged.

For the x variable, the explanation lies in the fact that, as mentioned in the discussion of RQ2, the GP does not do a very good job at generalising the output behaviour with this variable elided and we are often forced to default back to the literal outputs from the PTA. Because *move* events make up the vast majority of the events in the SPACEINVADERS traces, the model we can infer with GP is relatively similar to the one we can infer without it.

We can also see from Figures 8.16 and 8.22 that obfuscating additional variables does not appear to have a huge effect on the size of the model. The main difference in model size seems to come from obfuscating one variable. This is not particularly surprising for this case study since the majority of events in the traces are *move* events, which depend on the x variable. The corresponding events for the other two variables are relatively infrequent. Thus, when we obfuscate either of them as a second variable, there is relatively little change in the traces and the corresponding model.

Let us now consider how the complexity of the inferred models is affected by the ability to infer transition guards during inference. Again, we cannot compare with MINT here since the transitions which make up its models do not have any observable behaviour. We have already discussed how the ability to distinguish transitions during inference enables us to merge more states as we have a greater capacity to resolve the resulting nondeterminism. In terms of scalability, this means that models inferred with this ability scale much better in complexity than those inferred without.

There is still room for improvement in this respect, though. As discussed above, transitions often accumulate guards as inference proceeds. The more merges a transition is involved in, the more guards it is likely to accumulate. Thus, larger PTAs are likely to infer transitions with more guards. If we could, instead, infer more generally applicable guards, this would not only reduce the number of guards on transitions in the inferred model, but should also allow us to merge more transitions, making for smaller models.

Conclusions

This research question was concerned with the factors that affect the complexity of the models we can infer and how my technique copes with these. Traces with a lot of distinct input-output pairs that are related in the same way often end up leading to models with a lot of "parallel" transitions if we are not able to abstract away the concrete values into functions. While this does not necessarily affect the number of *states* in the inferred models, they are cluttered with a lot of extra *transitions*. The results presented above suggest that my GP preprocessing technique helps to mitigate this by generalising behaviour such that more transitions can be merged.

An alternative to applying GP as a preprocessing technique would be to do as in [150] and apply GP as a *postprocessing* technique. This has the advantage that we can group "parallel" transitions together rather than having to guess groups based on structure and history like in Chapter 7, but means that the inferred output and update functions would not get to play a part in the inference process.

This is particularly important when variables are elided, because there are fewer data values which can be guarded. Without first generalising transition outputs, we encounter lots of unresolvable nondeterminism during merging, so need to keep a lot of states separate. If we were to run GP after merging states, we would likely then be able to merge many more states, as the generalised transitions would enable us to resolve much of the nondeterminism which was previously unresolvable. Thus, we would need to run a second pass of state merging after running GP. In such situations, it makes much more sense to perform GP as a preprocessing step as I do here rather than having to run two phases of state merging.

While MINT is able to infer models with the same number of states as my technique, its models seem to have more transitions. This is to do with the fact that MINT infers guards on transitions by default, meaning it encounters less nondeterminism when merging states but is forced to keep more transitions separate.

Obfuscating variables makes the task of inference much harder. There are fewer inputs to be guarded, so we encounter much more nondeterminism when we merge states, forcing us to keep them separate. Again, my GP preprocessor helps to mitigate this. By generalising behaviour, we are able to merge more transitions. Thus, we are more able to resolve the nondeterminism which arises as a result of merging states so are able to merge more of them. For the spaceInvaders case study at least, it seems that the main difference in model size is between eliding zero and one variables. Additional elisions do not seem to make much of a difference, although this is likely to be due to the events which correspond to certain variables occurring infrequently in the traces, thus having little effect on the overall structure of the models.

RQ4b How long does model inference take?

While optimising runtime was not a high priority for this project, it is clearly important that the tool runs sufficiently quickly that it can be run on examples large enough to test its limitations. Figure 8.24 shows the runtimes for the different configurations.

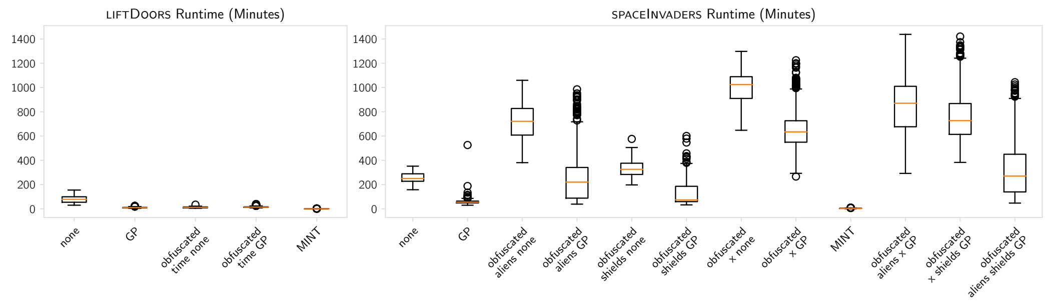


Figure 8.24: Runtimes.

Looking first at liftDoors, we can see that the runtime of my technique is usually pretty fast, even with *time* obfuscated. Interestingly, running the system without any preprocessing is the slowest configuration by quite a long way. The reason for this can be explained with a little insight into how the inference program runs. There are two main bottlenecks here. The first is the resolution of nondeterminism, and the second is the scoring of possible state merges. For a model with s states, there are ${}^{s}C_2$ state pairs to score. This obviously grows very quickly with s, and must be recomputed each iteration. Without GP, the number of states which get merged each generation will likely be quite small because there is much less associated nondeterminism. This makes for a lot of time spent scoring potential merges, much of which is actually duplicated work since most states will remain unchanged between iterations.

The resolution of nondeterminism can be a computationally intensive process but, as it proceeds, further states are merged. With GP preprocessing, since we end up with many identical transitions, we will also end up with much more nondeterminism when we merge states. This means the inference process spends much more time on nondeterminism resolution but also means that many more states can be merged per iteration. Consequently, we do not need to spend as much time scoring state merges. Additionally, GP preprocessing makes for a lot of identical transitions when it is successful. The resolution of nondeterminism between such transitions is computationally trivial as one can simply be deleted.

Looking at spaceInvaders, the story is relatively similar for the average runtimes, but with a much greater spread and a much higher average. The higher average can be explained by the fact that the PTAs are usually much larger, as shown in Figure 8.11a. Thus, there are more states to score in the early iterations of inference. The larger spread in runtimes is explained by the fact that, if an attempt to resolve nondeterminism fails, the algorithm has to backtrack and attempt to merge a different pair of states. This problem is exacerbated when the GP preprocessor goes wrong. It is clearly much more likely that an attempt to resolve nondeterminism will fail if we have inferred the wrong output function. Conversely, if the GP is able to correctly infer output and update functions, it is much less likely that an attempt to resolve nondeterminism fails as nondeterministic transitions should be trivially equal.

From Figure 8.24, we can see that the slowest configuration, on average, is when we obfuscate the x variable without any preprocessing. Here, the scoring bottleneck is really working against us. Because we have the literal input guards, very little nondeterminism arises when we merge states. This means that we spend very little time on nondeterminism resolution, but does mean that we only usually merge one or two states per iteration. This means that we score the potential merges many times during inference. Since models shrink relatively slowly, the time spent on this step remains long throughout the inference process.

We can see from Figure 8.24 that MINT is *much* faster than my tool for both case studies. There are several explanations for this. The first is that much more time has been spent on optimising the various operations of MINT than my tool. MINT was written from scratch in Java, and was designed to run quickly. By contrast, my technique was written in Isabelle and is implemented by automatically generated code which is not particularly fast or well-optimised. This code is in Scala, the compiler of which is notoriously bad at optimising recursive functions, which make up the vast majority of my implementation. It is therefore not surprising that MINT is faster than my tool purely for this reason.

The second reason that MINT is faster than my tool is the way it scores potential state merges. Where my technique scores every state pair at each iteration, MINT uses the Blue-Fringe approach [98] discussed in Section 4.6, which considers only a few merges each iteration. This means that there is much less work to do at each stage, and also means that MINT scales much better with the number of traces.

Let us now examine how the ability to infer guards to distinguish transitions during inference affects the runtime of my tool. Figure 8.25 shows the runtimes for the different configurations of the drinks machine. We can see that the average runtime is very similar with and without the ability to infer guards. There is, however, the possibility for the runtime to be much greater if guards can be inferred. A likely explanation for this is that the inference of guards is done using GP, which takes time to run. If the GP has to be run a lot, this will obviously increase the runtime. As detailed in Chapter 7, this can be mitigated with *memoisation*, but this dependent on there being a memoised guard for the given training set. If the traces are structured such that there are a lot of training sets, this cannot occur so the memoisation provides limited benefit.

We can see from Figure 8.25 that both spaceInvaders and drinks have a similar average runtimes whether or not guards are inferred during inference. This indicates that there is not much computational cost associated with guard inference. Combining this information with the fact that guard inference often leads to much more accurate models, this seems to advocate in favour of the inference of distinguishing guards in situations where value-dependent behaviour is known to be present.

Conclusions

This research questions was concerned with the runtime of both my inference tool and MINT, and the factors which affect this. The main conclusion that can be drawn from the results presented above is that MINT is much faster than my tool, owing mainly to the fact that the way it scores state merges means it has much less work to do at each iteration. MINT has also had much more time spent on its optimisation.

The fact that such a large part of my tool's runtime is spent scoring state merges makes it relatively easy to improve its runtime. As mentioned in Section 6.4, my tool uses Scala's parallel list processing library. Since the scoring of merges is just a map over a list of pairs, it can make

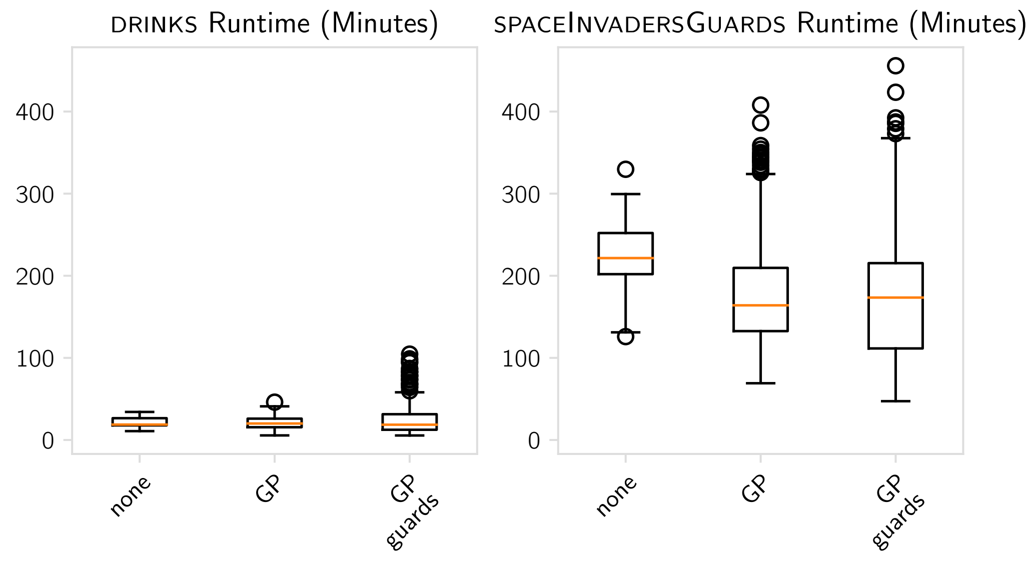


Figure 8.25: Runtimes.

heavy use parallelism. This means that running the tool with more CPU cores, or even with access to a GPU, would give a substantial speed up with very little implementational effort. A second way to speed up scoring would be to use *memoisation.* As the algorithm proceeds, we can record which states have changed in some way such that each iteration, we only rescore the states we need to. Because most states will not change in a given iteration, this cuts down hugely on the amount of duplicated work each iteration, making the tool run much faster.

An alternative to both of these methods is to use a more advanced scoring metric like Blue-Fringe, which does not consider every state pair each iteration. Because of this, there is inherently much less work to do, so the inference can proceed much faster. As discussed in Subsection 6.2.2, I deliberately chose to keep things as simple as possible here, but an investigation into the potential benefits of more complex scoring functions is desirable future work.

The speeding up of nondeterminism resolution is much more difficult, as this is a key feature of my inference algorithm. As mentioned in Section 7.7, I already make use of memoisation here to stop the inference trying repeatedly to merge states which it has already failed to merge. Beyond this, there is very little else which can be done to speed up this step in the algorithm.

8.6 Utility

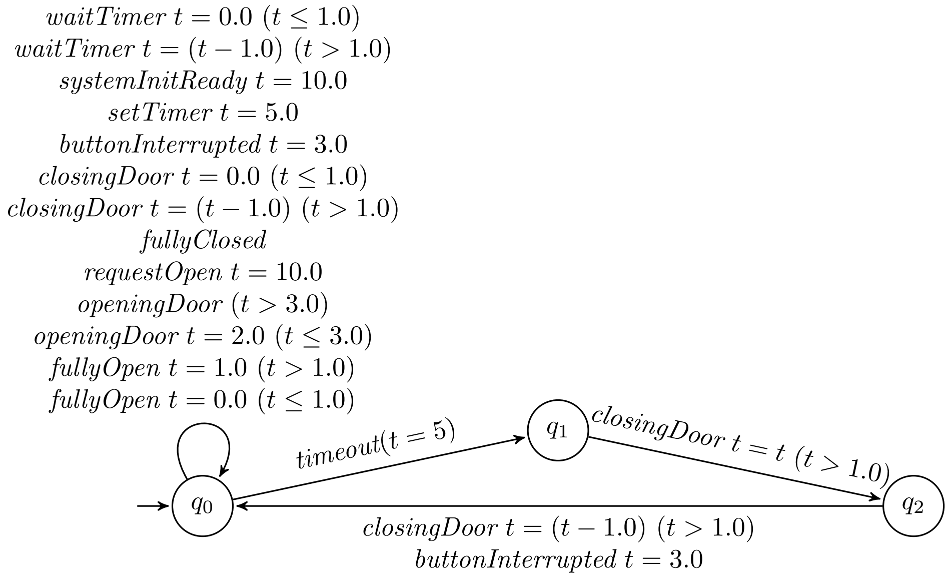
In addition to the *accuracy* of models we can infer, it is also important to judge how *useful* the inferred models are. To some extent, this depends on the intended application, but models are often used to give an intuitive overview of how the underlying system functions. It is therefore important that our inferred models are easy to understand. This is obviously very subjective, and there are no meaningful metrics to quantitatively evaluate this. Hence, this section provides only an informal discussion.

8.6.1 Lift Doors

Let us first examine the utility of the models inferred for the liftDoors system. For RQ1, I used traces in which all variable values are present, so the output of each transition is always expressible purely in terms of inputs and constants. Both MINT and my own inference technique achieved good accuracy here, but what do the inferred models actually look like?

With GP preprocessing, my inference technique infers a model very similar to the one in Figure 8.2. There was a slight variation in the exact phrasing of the output functions in that the GP tended to prefer the phrasing "$-1 + i_0$" to $i_0 - 1$, but the structure and semantics of the model were there. The inference never chose to split the moving/moved and closing/closed into separate states but this is not surprising since, from an inference point of view, there is no reason to split them. Splitting the states also requires a guard on the *openning*- and *closingDoors* transitions, which the inference was not given the capability to infer for this RQ.

Let us now examine the models inferred by MINT. Figure 8.26 shows an EFSM for the liftDoors system inferred by MINT. Because MINT uses a slightly different formalism, as discussed in Chapter 3, the syntax of the transitions is slightly different. Here, transitions take the form $q_m \xrightarrow{label\ update\ guards} q_n$, so the transition *waitTimer* $(t - 1.0)$ $(t > 1.0)$ has label *waitTimer*, updates the variable t (which represents the *timer* variable) to its anterior value less one, and can only be taken if the anterior value of t is greater than one. For aesthetic reasons, each $q_m \to q_n$ arrow is only drawn once with each transition being on a new line.



waitTimer t = 0.0 (t ≤ 1.0)
waitTimer t = (t − 1.0) (t > 1.0)
systemInitReady t = 10.0
setTimer t = 5.0
buttonInterrupted t = 3.0
closingDoor t = 0.0 (t ≤ 1.0)
closingDoor t = (t − 1.0) (t > 1.0)
fullyClosed
requestOpen t = 10.0
openingDoor (t > 3.0)
openingDoor t = 2.0 (t ≤ 3.0)
fullyOpen t = 1.0 (t > 1.0)
fullyOpen t = 0.0 (t ≤ 1.0)
q0
timeout(t = 5)
q1
closingDoor t = t (t > 1.0)
q2
closingDoor t = (t − 1.0) (t > 1.0)
buttonInterrupted t = 3.0

Figure 8.26: An EFSM for the liftDoors system inferred by MINT.

The model in Figure 8.26 only has three states, where the one in Figure 8.2 has four. It also has a lot more transitions, with some behaviour (such as *closingDoor*) appearing with both a literal update and a general function. The majority of activity takes place from state q_0. Indeed MINT quite often merges down to a single state which performs the full functionality of the system. While this can still be very accurate in terms of traces, it does not effectively illustrate the control flow of the system. This is done much better by the model in Figure 8.2 and by my own inference technique, where states in the models are more clearly linked to the control flow states of the underlying system.

Something else that is noteworthy here is that MINT occasionally infers redundant updates such as $t = t$ on the $q_1 \to q_2$ transition. Such updates are not incorrect, but they are unnecessary since the semantics of the models are such that variables remain unchanged unless they are explicitly updated. Updates like this clutter models with unnecessary details and make them more difficult to understand. As discussed in Chapter 7, my technique only introduces update functions when they are necessary so will never include redundant updates like this.

A key advantage of my system over MINT is that it is able to cope in situations where certain variables do not appear in the traces. Indeed, it is even able to infer very accurate models for this case study with the *timer* variable taken away. Ideally, in this situation, we would hope to infer a model very similar to the one in Figure 8.2 except that all transitions would have an arity of zero instead of one and a register would be used and updated in place of the inputs. This is not quite what happens. In order to achieve such a model, the GP must perfectly infer register usage. With the variable elided, there is much more freedom for the GP to "experiment" with different output and update functions. This means that the inference is then unable to merge as many states as we would like, and we end up with a model which, although accurate, is large and unwieldy. An example of such a model is shown in Appendix B.

8.6.2 Space Invaders

The next case study is the SPACEINVADERS system. Here, my tool is able to infer much more accurate models than MINT when the GP preprocessor is used. Indeed, my tool infers the model in Figure 8.4 with very few exceptions. As mentioned in Section 8.2, this is the only model of the system that gives a good overview of the underlying behaviour. There are not really any other acceptable alternatives.

The models inferred by MINT for SPACEINVADERS are generally too large to be effectively displayed in the format of this thesis. This is not because the models themselves are too large — they often have four states or fewer — but because the transition expressions are unreasonably long. An example of such a transition is the following.

$$\begin{aligned}
&\mathit{alienHit}\ (\mathit{aliens} = (1.0 + \mathit{aliens}))\\
&((\mathit{aliens} \leq 3.0) \wedge (x \leq 250.0) \wedge (\mathit{aliens} \leq 0.0))\vee\\
&((\mathit{aliens} \leq 3.0) \wedge (x \leq 250.0) \wedge (\mathit{aliens} > 0.0) \wedge (x \leq 100.0))\vee\\
&((\mathit{aliens} \leq 3.0) \wedge (x \leq 250.0) \wedge (\mathit{aliens} > 0.0) \wedge (x > 100.0) \wedge (\mathit{aliens} > 1.0))
\end{aligned}$$

Here, the guard expression alone is so long that it must be broken onto three lines. Guards like this do not make for readable models, especially when MINT tends towards models with a single state that performs the entire functionality of the system.

Not only are the guard expressions inferred by MINT overly verbose, they are often unnecessary. None of the transitions in SPACEINVADERS exhibit any value-dependent behaviour, so there is no real need to impose a guard on them at all. While it can be informative to aggregate the data values observed in the traces into a concise guard expression, these guard functions should only reference variables which are relevant to the transition. The value of the x variable in the above guard expression is purely incidental and has nothing to do with the transition itself. This kind of bloat is an unfortunate product of MINT's requirement that every event include the value of every variable.

Having said that, when I modified the traces to introduce value-dependent behaviour for RQ3, my inference technique did not fare any better. What we ideally want is a model like the one in Figure 8.5. Here, the guards on the *alienHit* and *shieldHit* transitions decide which of the two respective transitions is taken for a given action. To be able to infer a model like this, we would have to perfectly infer the guard of each transition such that the inference process is able to safely merge all the transitions without introducing nondeterminism. Unfortunately, this is rarely the case, and we tend to end up with large and unwieldy models. While the guards inferred by my technique are much simpler expressions, they are still overly specific as discussed in Section 8.5. This means that the inference cannot merge as many states as we would like, and we end up again with models which are too large to be effectively displayed on an A4 page.

Let us now consider how my inference technique fares when variables are obfuscated. We have already seen in Section 8.5 that the models we are able to infer are much larger and less accurate. What we would like to obtain is a model similar to the one in Figure 8.4, but where a register is used instead of a given input. What we actually end up with are models which are again too large to be effectively displayed on an A4 page. Again, these models are not necessarily *inaccurate* representations of the system in that they may be (close to) trace equivalent, but they no longer give an intuition of how the system behaves.

8.6.3 Drinks Machine

The final case study I examined was a more complicated version of the drinks machine, the EFSM for which is shown in Figure 8.6. The ability to infer a model like this again requires the GP to infer perfect guards. Without this, the nondeterminism which is introduced by merging states cannot be resolved in the correct way, meaning that certain states cannot be merged.

Unfortunately, the GP heuristic from Subsection 4.7.5 is simply not powerful enough to infer suitable guards. Instead, we end up with literal equalities which are sufficient to resolve local nondeterminism but do not generalise across the model. The reason for this is that we do not have sufficient training data. As we merge more states, we accumulate more training data but, because we have no way to revise transition guards once they have been added, these literal guards are retained throughout the inference process. Like with spaceInvaders, we tend to end up with models which are, to some extent, accurate with respect to the test traces but are too large to be effectively displayed and understood as a whole.

8.7 Threats to Validity

The main external threat to the validity of the results presented here is that I was only able to test my system on a very small number of systems. Consequently, the accuracy and scalability results presented here cannot be taken to be representative of *all* software systems. The main reason for this, as discussed in Section 3.13, is that it is difficult to find suitable subject systems and extremely labour-intensive to obtain traces from these systems.

This threat is somewhat mitigated by the fact that the systems and traces thereof are reasonably diverse. They differ in their reactivity, the number of different possible actions, and trace length. Nevertheless, it is important to avoid drawing general conclusions about accuracy and scalability and, like [152], to focus instead on the factors which *affect* these.

Another threat to external validity is that the GP preprocessor used as part of my inference technique is subject to a number of different configuration parameters, such as population size and the number of new individuals created per generation. I did not spend much time optimising these parameters for this evaluation. I simply chose arbitrary values that gave acceptable performance on preliminary experiments. MINT also has a number of configurable parameters which were all left at their default values. This avoids the internal threat to validity of overfitting configuration parameters to these particular case studies and biasing the results in favour of either technique. It does, however, open up the external threat to validity that there may be more suitable configurations than the ones I used, meaning the behaviour of the techniques is somewhat misrepresented. Without an extensive parameter-optimisation investigation involving many different case studies, it is impossible to mitigate this threat.

The main threat to construct validity is that the inferred models are not evaluated with respect to specificity (or similar metric) to ensure that they do not wildly overgeneralise the original traces. The reasons I did not use such a metric in this evaluation are detailed in Section 3.11. In summary, the ability to use specificity is reliant on the existence of negative system traces which are extremely difficult to obtain from real programs, especially if we want to make these traces *meaningful* to the underlying system in some way. It also seems unjust to use negative traces to evaluate a technique which only has access to positive traces.

The threat of excessive overgeneralisation is mitigated to an extent by the kind of models inferred by my tool. Since traces in the test set are only accepted if the output produced by the model matches that produced by the system in response to the same action, there is no risk of inferring models which trivially accept any trace as there is for classical FSM models. There is, however, the risk that the inferred models are more reactive than the original system, and are more permissive about which actions can be called from which states with which inputs.

Another threat to construct validity is that models for the spaceInvaders case study are inferred from traces which come from manually playing the game rather than randomly generated traces from the implementation of a model. I mitigated this threat by playing the game as randomly as possible, i.e. without the objective of winning. An alternative to this would have been to implement a model of the system and randomly generate training and test traces from that, as I did with the simple drinks machine case study. As the majority of my evaluation is done in terms of system traces, rather than by comparison to a "gold standard" model, this would not significantly change the results presented in this section as the structure of the traces would be almost identical. In both situations, the first event would be *start* and the last event would be either *win* or *lose.* Between these events, the player would be free to move left and right and hit or be hit by aliens.

In the real system, the game does not end until the player has hit five aliens or been hit by three. Implementing Figure 8.4 directly would have removed this restriction, allowing the game to end prematurely. Alternatively, a guard could be added to prevent this. Either way, the main source of inaccuracy in models inferred by my technique comes from the limitation of my GP technique exemplified in Example 8.5.1, which would still be given plenty of opportunity to reveal itself as it only requires a switch from moving in one direction to the other and then back to the first. With MINT, the problem is the fact that spaceInvaders uses three system variables, which would be the same for a reference model as for the program.

The final threat to the validity of the results in this section is an absence of statistical tests demonstrating the significance of the performance difference between my GP technique and MINT. As discussed above, I only evaluated my system with three systems. Thus, the significance of any tests performed here would be limited. The threat to validity is nonetheless mitigated by the fact that my technique outperformed MINT in the vast majority of cases, only being outperformed by it in three or four outlying instances of spaceInvaders, depending on the metric. For the liftDoors case study, my technique inferred a perfect model every time where MINT had a few outliers.

8.8 Conclusion

This chapter presented an empirical evaluation of the techniques presented in Chapters 6 and 7 in the context of several case studies. The results presented in this section show that my inference tool with GP preprocessing is generally able to infer models which are better able to predict system behaviour for unseen traces than those inferred by MINT [150], the current state of the art. A major contributing factor to this is that MINT requires that every event reports the value of every variable. Where there are multiple system variables at work, the inference ends up with a lot of irrelevant information which can mislead it.

In addition to my formal research questions, I presented an informal discussion of the utility of the models we can infer for each case study with respect to what we might call the "optimal" model. Models like the ones inferred here are often intended to give a high-level overview of the subject system, so we need our inferred models to intuitively capture this. My technique seems to be able to achieve this when all variables are present in the traces. The models produced by MINT tend to be less intuitive, and transitions often have spurious guards which clutter the model. When we obfuscate variables from traces, MINT cannot cope with this at all, and the models produced by my own technique become very large and difficult to understand, even though they often remain relatively "accurate" with respect their predictive power. In terms of my four main research questions, the main conclusions are as follows.

RQ1 How accurate are the models produced by my inference tool? This research question was concerned with the accuracy of the models we can infer when the traces contain all the values used to compute the output. The ability to generalise the concrete values observed in the traces into computational functions allows my tool to merge more behaviour, enabling it to infer models with fewer states and transitions. In general, both my tool and MINT were able to infer accurate models of the subject systems, with those inferred by my tool generally being more accurate and intuitive.

RQ2 How does eliding variables affect the accuracy of the models produced by my inference tool? This research question was concerned with how my inference tool copes when the output of certain events in the traces depends on a variable which is hidden. The main conclusion we can draw from the results presented here is that which variable is elided has a much greater effect on the accuracy of the inferred model than how many variables are elided. For the spaceInvaders case study, the inferred models were relatively accurate when the *aliens* and *shields* variables were elided but were significantly less accurate when the x variable was elided. This is because my current approach struggles when the same variable is mutated by multiple transitions because it currently has no way to determine which events should share variables. Indeed, eliding the x variable has such a detrimental effect on the accuracy of the inferred models that it overshaddows the effect of further elision.

RQ3 How does the ability to discover value-dependent behaviour affect the accuracy of the models produced by my inference tool? This research question addressed how the ability to discern value-dependent behaviour affects the accuracy of the models we can infer. The accuracy results in this chapter seem to be in favour of this when GP preprocessing is used. For systems with value-dependent behaviour, the ability to infer guards which distinguish transitions allows us to merge more states in the model because we are better able to resolve the resulting nondeterminism. This means that we are more able to merge states in the model which represent the same program state, thus leading to smaller and more accurate models.

RQ4a How large are the inferred models in terms of states and transitions? This research question was concerned with the various factors which affect the complexity of the inferred models. Traces which have a lot of distinct input-output pairs which are related in the same way often end up leading to models with a lot of “parallel” transitions if we are not able to generalise these. While this does not necessarily affect the number of *states* in the inferred models, they contain more *transitions* than is necessary. The results presented in this chapter suggest that my GP preprocessing technique helps to mitigate this by generalising behaviour such that more transitions can be merged.

When we obfuscate variables, there is much more potential for nondeterminism to arise when we merge states. This often means we are not able to merge as many states. Again, my GP preprocessor helps to mitigate this. By generalising behaviour, we are able to merge more transitions. Thus, we are more able to resolve the nondeterminism which arises as a result of merging states so are able to merge more of them.

There appears to be very little increase in model size when we elide more than one variable. This is not particularly surprising for the spaceInvaders case study since the majority of events in the traces are *move* events, which depend on the x variable. The events which correspond to the other two variables occur very infrequently so have relatively little effect on the size and structure of the inferred models. More generally, a “diminishing return” effect on model size as we obfuscate additional variables is not surprising. When we obfuscate variables, the inference process is unable to merge as many states because it is unable to resolve the resulting nondeterminism. As we obfuscate successive variables, the number of new state pairs which must be kept separate as a result of each obfuscation will diminish because there will be overlap with previous variables.

RQ4b How long does model inference take? This research question was concerned with the runtime of the inference tools. The main conclusion here is that MINT runs much faster than my tool. The main reason for this is that my tool spends a lot more time on the scoring of potential state merges. Cases where it is possible to merge many states in one iteration of inference tend to be the fastest to run, since my tool spends much less time scoring potential state merges. It also seems that, in general, providing the ability to infer guards for transitions during inference does not significantly affect the average runtime of my tool meaning that, most of the time, the benefits that this ability affords in terms of size and accuracy come without extra computational cost.

Concluding Remarks

Having now proposed and evaluated my EFSM inference technique, it only remains to examine in a little more detail the applications of the inferred models. As well as being used to show system functionality, models are also used for more formal analysis. Here, it is critical that the model accurately captures the behaviour of the system, preferably in a way which is easy to reason about. The next chapter shows how we might use the models we have inferred to prove certain behavioural properties of the underlying systems and to reveal potential flaws.

Chapter 9

Formal Analysis of EFSM Properties

In previous chapters, I showed how EFSMs can be inferred from black-box traces using state merging, and how GP can be used to infer functions to relate inputs, outputs, and register values. This chapter is based on work currently in draft [138], and examines how we can analyse and verify properties of EFSM models once they have been inferred, the primary contributions being the following.

- The extension of my Isabelle formalisation of EFSMs such that properties in Linear Temporal Logic can be specified and proven.
- A semantically equivalent EFSM formalisation in the model checker SAL to enable counterexamples to be easily generated for untrue properties.
- A framework of function definitions in both tools to ease the specification of properties and enable automated translation between the two representations.

9.1 Introduction

When we infer a model of a system, we usually do so because we have a specific purpose in mind. One such purpose is the analysis and verification of system properties, the idea being that it is often extremely difficult to verify *systems.* Instead, our model serves as a suitable *abstraction* of the system such that we can prove properties more easily.

As an example, consider again the simple drinks machine. At the end of Chapter 5, I gave examples of two interesting properties that we wish to hold true of the system, namely that the customer will always receive the drink they selected, and that they cannot receive this drink without first inserting a sufficient amount of money. While a formal verification of these proofs is perhaps a little overkill in this setting, there are many systems where such strong assurance is vital. Consider the simple lift door controller from Chapter 8. Here, people may be put in danger if the controller misbehaves, and there are various safety-critical properties we may want to prove about this system, for example that passengers will not be crushed in the doors, or that the lift must be stationary before the doors can open.

What we need to be able to do here is take a model, specify properties over it, and determine whether those properties hold. This is the process of formal verification, and there are many tools and techniques which can help with this. In addition, where a given property is untrue, it can be extremely helpful to have a concrete counterexample. For EFSM models, these take the form of traces which violate the given property. Such traces provide insight into *why* a given property is untrue, and can be invaluable when trying to fix faults.

The remainder of this chapter is structured as follows. Section 9.2 provides some background on formal verification such that the remainder of this chapter can be understood. Section 9.3 discusses how LTL is implemented in Isabelle, and how I applied this to EFSMs. Section 9.4 discusses how EFSMs and LTL properties are represented in the model checker SAL and argues semantic equivalence between the two representations. In Section 9.5, I briefly discuss how we can automate the translation between Isabelle and SAL. Finally, Section 9.6 provides examples of how SAL and Isabelle can be used in tandem to develop and prove properties of models.

9.2 Background

Formal verification is the process of using formal methods to show that a system exhibits certain properties. As discussed in Section 2.3, it is desirable to verify both *safety* (nothing bad happens) and *liveness* (eventually something good happens) properties. Both of these classes of properties are *temporal properties.* They refer to something happening at a particular instance in time. For example, we may want to verify safety properties like "the doors of a lift are never open while the lift is moving" and liveness properties like "the lift will eventually arrive after pressing the *call* button". There are three main techniques used to verify system models:

Simulation is a process often used in the verification of hardware and has nothing to do with the simulation we have seen so far, which is about finding a relation between the states of two models. Here, the process of simulation involves running a model of the system to be verified with many inputs to see if it breaks. This technique is relatively straightforward and can often be easily automated, but it does have some drawbacks. The main limitation of the technique is that its effectiveness is related to the number of simulations run. We can only be completely confident that a system will perform properly under those circumstances which have been simulated. It is often impossible to run simulations for *all* circumstances, though, so the simulation can only be run on a subset of inputs. This means that there may exist inputs for which the system breaks that have not been tested. It is often unclear how many runs are necessary to determine a reasonable degree of confidence that a given property holds.

Theorem Proving is at the other end of the scale to simulation and involves mathematically proving properties of a system model from axioms and inference rules. Proof techniques such as induction make it possible to prove properties for *all* inputs, so a proof guarantees that a system exhibits a given property. The cost of this level of assurance is that it is often extremely difficult to prove even simple properties of real systems. Proof assistants such as Isabelle [120], Coq[1], and Agda[2] aim to make this process easier, but the learning curve is often steep.

Model Checking can be thought of as a compromise between simulation and theorem proving. A key advantage of model checking is that, under circumstances where a property does not hold, it is often possible to yield a concrete counterexample in the form of a path through the transition system which violates the given property. Unfortunately, most model checking tools can only work effectively with finite systems, or finite subsets of infinite systems. This has the same limiting effect as simulation in that there may exist circumstances which can occur in real life for which the model has not been checked.

It is also worth mentioning *model-based testing*, a technique in which models are used as a basis to generates test cases for implementations. This differs from simulation in that here, it is the actual system being tested rather than just a model. While model-based testing is very much a *testing* technique rather than a *verification* technique like those above, it is an attempt to integrate the rigour of formal methods to the often more ad-hoc field of software testing.

[1]`https://coq.inria.fr` (Accessed 24/03/20)
[2]`https://github.com/agda/agda` (Accessed 24/03/20)

9.2.1 Temporal Logic

Temporal logics provide a means of reasoning about properties of systems with reference to discrete steps through time. In addition to the standard logical operators of conjunction, disjunction, negation, and implication, temporal logics provide operators to express information about *when* properties need to hold. This enables the values of variables to change over time such that properties may be true at some points and false at others.

The correspondence between model state and variable interpretation is captured via a labelling function $L : S \rightarrow \mathbb{P}(V)$, in which S is the set of states in the model, V is the set of variables, and $\mathbb{P}$ is the interpretation. A variable $v \in V$ is true in a system state $s \in S$ iff $v \in L(s)$. In addition to the set of states, there is also a set $I \subseteq S$ of initial states, and a transition relation $T \subseteq S \times S$ between states. Such a model is called a Kripke Structure [97].

The idea of states being connected by a transition relation is very similar to FSMs, but the notion of a *state* is very different. Here, states are more analogous to "program states" or ASM states from Subsection 2.2.6 than the simple control flow states of FSMs. That is, states are simply the values of variables at different points in time. Transitions then place guards on these variables and can apply updates when they fire, just like EFSM transitions. Essentially, we can think of this as a data-only EFSM.

It is important to note here that the transition function is assumed to be total, meaning $\forall s \in S.\, \exists s' \in S.\, (s, s') \in T$. This is because the semantics of most temporal logics are only well-defined over necessarily infinite traces. There have been attempts to define temporal semantics over finite traces [128] but these are not particularly well studied. The necessity for infinite traces presents some interesting problems when we wish to verify systems like our running drinks machine example from Figure 1.5, which have been designed with finite interaction in mind. This is discussed in more detail in Subsection 9.3.3.

Linear Temporal Logic

Linear temporal logic (LTL) [123] is a temporal logic that provides the ability to express properties in terms of linear time where each input event corresponds to one time step. LTL provides the following temporal operators.

$G(p)$ for *globally* – the given property, p, is always true.

$F(p)$ for *eventually* – the given property, p, becomes true at some point.

$X(p)$ for *next* – the given property p is true in the next state.

$p \;\mathcal{U}\; q$ for *strong until* – q eventually becomes true and p holds up to that point.

$p \;\mathcal{W}\; q$ for *weak until* – like $p\,\mathcal{U}\,q$ except that q need not become true if p holds globally.

There should be obvious parallels between the above temporal operators and the two main classes of properties we wish to verify. The *globally* operator allows us to verify *safety* properties. If we want to be certain that bad things never happen, we need to verify $G(\neg b)$, where b represents a bad thing happening. Conversely, the *eventually* operator allows us to verify *liveness* properties. If we want to ensure that good things eventually happen, we need to phrase the property as $F(g)$, where g represents a good thing happening.

Example 9.2.1. Consider a simple pedestrian crossing controlled by traffic lights. An obvious safety property here is that the light for cars and the light for pedestrians should never be green at the same time. We can phrase this in LTL as follows.

$$G(\neg(\mathit{greenForCars} \wedge \mathit{greenForPedestrians}))$$

A simple liveness property of the system is that if a pedestrian presses the button to cross the road, they should eventually be able to do so. A naive phrasing of this would be as follows.

$$\mathit{buttonPressed} \implies F(\mathit{greenForPedestrians})$$

This is not quite the property we had in mind, though, as this property only applies if the *first* event is *buttonPressed*. We really want the liveness property to hold throughout the lifetime of the crossing so we need to wrap the property within the *globally* operator.

$$G(\mathit{buttonPressed} \implies F(\mathit{greenForPedestrians}))$$

The $G(p \implies F(q))$ is a very common form for liveness properties.

Example 9.2.2. Let us return to our simple drinks machine example. Say that we would like to verify the property that a customer will only receive the drink they have selected. We can phrase this property as $G(\mathit{output} \neq [d]\ \mathcal{U}\ \mathit{label} = \text{“select”} \wedge \mathit{input} = [d])$. This property states that, for some d, the output will never be equal to $[d]$ until we have first called the *select* action with input $[d]$. This ensures that we will not get a drink until we have first selected it.

The semantics of LTL are defined along traces (or *paths*) of the model. A path π is an infinite sequence of states $s_0, s_1, s_2, \ldots$ with $s_i \to s_{i+1}$. We can refer to single states of the path using the notation $\pi(i)$ which represents the i^{th} state in the path. A path is said to be *initialised* if $\pi(0) \in I$. A suffix of a path, π^i, represents all events after $\pi(i)$. We can then define the semantics of LTL as follows.

$$\begin{array}{lcl}
\pi \models p & \iff & p \in L(\pi(0)) \\
\pi \models X(p) & \iff & \pi^1 \models p \\
\pi \models G(p) & \iff & \forall j \in \mathbb{N}.\, \pi^j \models p \\
\pi \models F(p) & \iff & \exists j \in \mathbb{N}.\, \pi^j \models p \\
\pi \models p\, \mathcal{U}\, q & \iff & \exists j \in \mathbb{N}.\, \pi^j \models q \wedge \forall k < j.\, \pi^k \models p \\
\pi \models p\, \mathcal{W}\, q & \iff & p\, \mathcal{U}\, q \vee G(p)
\end{array}$$

The standard logical operators $p \wedge q$, $p \vee q$, $\neg p$, and $p \implies q$ have the usual semantics.

Computational Tree Logic

In contrast to LTL, which operates over linear time, computational tree logic [36] (CTL) is a *branching time* logic. CTL provides the same basic temporal operators as LTL but each operator must be immediately preceded by a *path quantifier*, either “all” (A) or “exists” (E). This allows us to express properties in terms of paths such as “there exists a path such that property p eventually holds”. It may seem that this would make CTL more expressive than LTL however, this is not true as there exist LTL formulae which cannot be expressed in CTL, and vice versa [38]. The logic CTL* [37] subsumes both LTL and CTL as this allows temporal operators to optionally be preceded by a path quantifier.

9.2.2 Theorem Proving and Model Checking

If we want to *check* temporal properties of *models*, the obvious tool to use is a *model checker*. Here, a formal model of the system is specified (which is often some form of abstract state machine) along with the desired properties. To verify a property, the model checker explores all possible traces of the specified system in search of one which *violates* it. If such a trace is found, this can be presented to the user as a clear, and usually succinct, counterexample which indicates that the specified property does not hold.

If the model checker cannot find a trace that violates a property, this provides reasonable assurance of its validity but does not guarantee it as it is usually infeasible to explore *every* possible trace of a reasonably sized system. To ensure that properties can be checked in finite and reasonable time, model checkers often take shortcuts, for example working with finite subsets of infinite datatypes. This means that the absence of a counterexample does not guarantee the validity of a property.

To guarantee that a property holds, we need to prove it formally. It is here that theorem proving takes over. We have already seen in Chapter 4 how EFSMs can be formalised in Isabelle. We can also use Isabelle to state and prove properties phrased in LTL. Isabelle does not take any of the shortcuts commonly used by model checkers so, if we can prove in Isabelle that a property holds, this is a much stronger assurance of its validity. The problem here is that the proofs are often difficult and time-consuming to obtain, so we only want to attempt to prove properties which we are confident are true.

LTL is notoriously difficult to work with, meaning that it often takes several iterations to properly capture our intuition. Counterexamples are extremely helpful during this process. When faced with a counterexample, we know that there is either a problem with the *model* or the *property*. We can then examine the counterexample and decide whether it is valid — in which case we must fix the model — or whether we need to rephrase our property such that the given trace no longer violates it. Another use for counterexample traces is to compare a model to the real implementation of a system. If we can execute each step of the trace in the real system, we can easily investigate to see if there really is a problem.

While Isabelle has two notable counterexample generators — QuickCheck [24], and Nitpick [17] — they are not guaranteed to find a counterexample for every false statement [17]. Indeed, they are rarely able to find counterexamples to properties involving EFSMs and are certainly not intended to generate traces which violate LTL properties. To make matters worse, Nitpick can sometimes produce *spurious counterexamples* meaning that it may sometimes find counterexamples to properties which are, in fact, true.

When working with EFSMs and LTL properties, we really want to be using a model checker as our counterexample generation tool. Here, we can quickly and easily generate counterexamples in the form of traces which violate our property. This allows us to rapidly iterate our formalisation. Only when our model checker cannot find any more counterexamples do we want to shift to a formal proof. At this stage we can be reasonably confident that our property holds, so it is worth putting in the time and effort necessary to prove it properly.

In order to use theorem proving and model checking tools in harmony, we need to be able to represent models and properties in both in a way which is semantically consistent. The next two sections detail the two representations and argue semantic equivalence. It is also helpful to automate the translation between the two representations to save time and reduce the chance of human error. This is the main contribution of [138].

9.3 Using Isabelle for LTL

As part of Chapter 4, I described how I formalised EFSMs in Isabelle. We can also state and prove temporal properties of these models using Isabelle's formalisation of LTL on streams. This section details how this formalisation works and how I used it to create a framework of functions which simplifies the phrasing and proof of LTL properties over EFSM models.

9.3.1 LTL and Streams

Rather than expressing properties over *models*, like with most model checking tools, Isabelle LTL predicates apply directly to *paths.* Here, properties are functions that take a path and return boolean *true* or *false*. Paths take the form of infinite streams of states, where a *stream* is a coinductive datatype defined similarly to an inductive list but with only a CONS operator, i.e. no empty base case. This makes them necessarily infinite. Like inductive lists, coinductive streams have both a *head* element (accessed by SHD) and a *tail* (accessed by STL) which is the rest of the stream. For properties φ and ψ, and a stream π, Isabelle defines the temporal operators *coinductively* as follows.

$X(\phi)$	$\mathit{nxt}\ \varphi\ \mathit{xs} = \varphi\ (\mathit{stl}\ \mathit{xs})$
$F(\phi)$	*base:* $\varphi\ \pi \Longrightarrow \mathit{ev}\ \varphi\ \pi$
	step: $\mathit{ev}\ \varphi\ (\mathit{stl}\ \pi) \Longrightarrow \mathit{ev}\ \varphi\ \pi$
$G(\phi)$	$\varphi\ \pi \wedge \mathit{alw}\ \varphi\ (\mathit{stl}\ \pi) \Longrightarrow \mathit{alw}\ \varphi\ \pi$
$\varphi\ \mathcal{U}\ \psi$	*base:* $\psi\ \pi \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ \pi$
	step: $\varphi\ \pi \wedge (\varphi\ \mathit{suntil}\ \psi)\ (\mathit{stl}\ \pi) \Longrightarrow (\varphi\ \mathit{suntil}\ \psi)\ \pi$
$\varphi\ \mathcal{W}\ \psi$	*base:* $\psi\ \pi \Longrightarrow (\varphi\ \mathit{until}\ \psi)\ \pi$
	step: $\varphi\ \pi \wedge (\varphi\ \mathit{until}\ \psi)\ (\mathit{stl}\ \pi) \Longrightarrow (\varphi\ \mathit{until}\ \psi)\ \pi$

Note that here the semantics are "backwards", i.e. defining the conditions necessary for the temporal properties to hold, rather than the other way round. Note also that the semantics of weak and strong *until* appear to be the same. The difference here is that strong until is defined *inductively* where weak until is defined *coinductively.* This has the effect of necessitating that the base case be true (i.e. the release operator holds) for strong until, where weak until can continue forever if ϕ holds globally.

Given that the semantics of weak and strong until are so similar, we should be able to phrase them in terms of each other. Indeed, these are fairly basic identities in LTL semantics. We have $\phi\,\mathcal{U}\,\psi \iff \phi\,\mathcal{W}\,\psi \wedge F(\psi)$ and $\phi\,\mathcal{W}\,\psi \iff \phi\,\mathcal{U}\,\psi \vee G(\phi)$. From this second identity, we have the property $G(\phi) \implies \phi\,\mathcal{W}\,\psi$. While these are key LTL identities, their proofs in Isabelle were surprisingly challenging and are an, albeit small, contribution of this chapter. They have now been integrated into the Isabelle sources and are now included as part of the core distribution.

9.3.2 EFSMs as Event Streams

When we express an LTL property in Isabelle, it is phrased as "property φ holds on stream π". This is a problem for us as we would like to express properties over *models.* Model checkers verify properties by exploring every possible *trace* of the model in search of one which violates the property. If no such trace is found, the property is said to hold of the model. We can apply a similar technique here. Instead of saying "property φ holds of model m", we can say "property φ holds of every *trace* of model m".

To apply this implementation of LTL to EFSMs, we must find a way to express them as streams. Since LTL operators effectively act over traces of models, this seems like a good place to start, but the traces we have been working with so far are not suitable for this. Firstly, LTL semantics are defined over necessarily infinite streams where the traces we have seen so far have all been finite. The second problem is that up to this point, we have been working with *black-box* traces which only contain information visible to the outside observer. If we want to represent EFSMs as traces, we need to include everything, including the current control flow state and the values of registers, in the traces.

Example 9.3.1. Returning to our simple drinks machine example, we might like to verify the property that the customer cannot get their drink without paying for it. We can phrase this informally as "if we do a *vend* action which dispenses a drink, r_2 must be greater than or equal to 100".

Here, the desired property is phrased in terms of the value of register r_2. This means that we need more comprehensive traces than have been used so far. In previous chapters, we have been concerned with inferring models from *black-box* traces since it might not be possible to modify or look inside the real system. When verifying properties like this, we need more comprehensive *white-box* traces to reveal the values of system variables.

Thinking of each action as a step forward in time, there are five components which characterise a given point in the execution of an EFSM. At each point, the model has a current *control state* and *data state*. Each action has a *label* and possibly some *input* parameters, and its execution may produce some observable *output* and update the data state. It is therefore sufficient to provide a stream of 5-tuples containing the current control state, data state, the label and inputs of the action, and the computed output.

Simply quantifying a property over every conceivable trace is likely to lead to a lot of spurious counterexamples. Consider the property from Example 9.2.2. If we were to say that this property holds over *all* traces, it is easy to falsify this claim with the trace which begins $f()/[d]$. This counterexample is spurious though, since there is no way for our simple drinks machine to generate such a trace. What we need is a way to specify that we are only interested in traces that our model can actually produce.

Traces of models are generated by observing executions. If a particular trace is a trace of a given model, there must exist an execution which generates it. If, instead of quantifying directly over traces, we instead quantify over *executions,* this gives us what we want. We can then phrase properties as "for all executions of model m, property φ holds on the corresponding trace".

To express this in Isabelle, I define the `make_full_observation` function, which is very similar to the `observe_execution` function in Subsection 4.3.1 except that it operates over infinite streams of actions rather than finite lists, and produces white-box traces rather than just an observation. The `make_full_observation` function is defined below, with an additional function `watch` defined on top of this which starts the `make_full_observation` off in the initial control state with the empty data state.

In order to be as consistent as possible with the world view of SAL described in the next section, the output component of each point in the path is that of the *previous* transition. That is, the output produced by the model in response to the action of π^n appears as the output component of π^{n+1}. This is a result of the fact that SAL uses an abstract state machine semantics where the "output" is just another local variable which must be updated by transitions. Thus, its updated value does not appear until the next state.

```
record state =
  statename :: nat option
  datastate :: registers
  action :: action
  output :: outputs

type-synonym whitebox-trace = state stream

fun ltl-step :: transition-matrix ⇒ cfstate option ⇒ registers ⇒ action ⇒ (nat option × outputs × registers)
where
  ltl-step - None r - = (None, [], r) |
  ltl-step e (Some s) r (l, i) = (let possibilities = possible-steps e s r l i in
                 if possibilities = {||} then (None, [], r)
                 else
                   let (s', t) = Eps (λx. x |∈| possibilities) in
                   (Some s', (evaluate-outputs t i r), (evaluate-updates t i r))
                )

primcorec make-full-observation :: transition-matrix ⇒ cfstate option ⇒ registers ⇒ outputs ⇒ action stream
⇒ whitebox-trace where
  make-full-observation e s d p i = (
    let (s', o', d') = ltl-step e s d (shd i) in
    (|statename = s, datastate = d, action=(shd i), output = p|)##(make-full-observation e s' d' o' (stl i))
  )

abbreviation watch :: transition-matrix ⇒ action stream ⇒ whitebox-trace where
  watch e i ≡ (make-full-observation e (Some 0) <> [] i)
```

Example 9.3.2. Consider again the property "if we do a *vend* action which dispenses a drink, r_2 must be greater than or equal to 100". Intuitively, we would phrase this in LTL as $G((label = vend \land output = [d]) \implies r_2 \geq 100)$. Since the *output* of the current transition is not visible until the next state, however, we actually need to phrase the property as $G((label = vend \land X(output = [d])) \implies r_2 \geq 100)$ so that we are looking one step ahead to examine the output of the *vend* transition we are interested in, rather than that of its immediate predecessor.

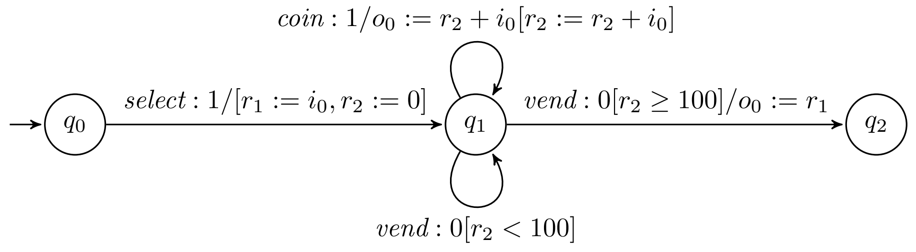
9.3.3 Making EFSMs Complete

Careful inspection of the definition reveals another way that `make_full_observation` differs from `observe_execution`. Rather than taking a `cfstate`, it takes a `cfstate option`. The reason for this is that we need to make our EFSM models *complete*. That is, we need them to be able to respond to every action from every state, like a DFA.

When inferring models from traces, it does not make sense to try to infer complete models as we are only likely to have a subset of behaviour. If a model does not recognise a given action in a given state, execution simply terminates and the EFSM is said to have *rejected* the action. In the context of LTL, however, this cannot happen because we are working with necessarily infinite traces. Since these traces are generated by observing action sequences, the `make_full_observation` function must keep processing whether there is a viable transition or not. To support this, `make_full_observation` adds an implicit "sink state" to every EFSM it processes by lifting control flow state indices from `nat` to `nat option` such that state `n` is seen as state **Some** n. The control flow state **None** represents the sink state.

When processing streams of actions, `make_full_observation` proceeds in the normal way unless and until the model is unable to recognise a particular action from its current state. At this point, rather than stopping processing, the model moves into the **None** sink state. Like the DFA in Example 2.2.2, once this state is entered, there is no escape. From here, the behaviour is constant for the rest of the time — the control flow state remains **None**; the data state does not change, and no output is produced. This allows `make_full_observation` to continue processing even after the model has stopped recognising actions.

Example 9.3.3. Consider again the EFSM representing our simple drinks machine. From q_0, if we receive any action which is not of the form *select*(d), the model moves into the sink state. Similarly from q_1, if it receives any action not of the form *coin*(n) or *vend*$()$. From q_2, any action moves the model into the sink state, since there are no outgoing transitions.



$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select : 1/[r_1 := i_0, r_2 := 0]$
$vend : 0[r_2 \geq 100]/o_0 := r_1$
q_0
q_1
q_2
$vend : 0[r_2 < 100]$

Figure 1.5: An EFSM model of the drinks machine.

The sink state becomes quite important when proving properties of EFSMs. For models like our simple drinks machine in Figure 1.5, interactions are intended to be finite. Any property involving the "globally" quantifier is essentially trying to carry out an infinite interaction. Thus, the model is almost guaranteed to end up in the sink state at some point, unless we happen to come upon an immortal individual of infinite wealth who selects a drink and then proceeds to insert coins for the rest of time without ever dispensing their chosen beverage.

An alternative to a global sink state would be to have unrecognised actions simply loop back to the current control state without modifying the data state or producing any input. This is not a particularly sensible course of action, however, since it effectively changes the semantics of EFSMs such that every trace is accepted. What we do by introducing a sink state is make it such that every execution is recognised, but we still know whether a trace is accepted or not because if we are in the sink state, we know we have done something invalid.

9.3.4 Expressing and Proving Properties

Since, in Isabelle, both the temporal operators and the properties over which they operate are functions from streams to boolean values, this can make even the simplest of properties difficult both to express and to understand. For example, the property from Example 9.2.2 is expressed in Isabelle as the following.

```
lemma LTL-output-vend:
  alw (λxs. (label (shd xs) = STR ''vend'' ∧
           nxt (λs. output (shd s) = [Some d]) xs) ⟶
             ¬? value-gt (Some (Num 100)) (datastate (shd xs) $ 2) = trilean.true)
    (watch drinks t)
```

Here, we first test to see whether the label at the head of the stream is *vend* and whether, in the next state, the output is $[d]$. This should imply that, in the current state, register r_2 holds a value which is greater than or equal to 100. The intuition of this property is quite simple, but its corresponding expression is almost unreadable to the untrained eye. In order to simplify the expression and understanding of properties, as well as to facilitate the automated translation from Isabelle to SAL [138], part of the contribution of this chapter involves the definition of a number of named functions which can be used to express certain properties of EFSMs.

state_eq takes a natural number representing a control flow state index and returns *true* if this is the control flow state at the head of the stream.

label_eq takes a string and returns *true* if this is equal to the label at the head of the stream.

input_eq takes a `value` list and returns *true* if it equals the input at the head of the stream.

output_eq takes a `value option` list and returns true if this is equal to the output at the head of the stream.

check_exp takes a guard expression and returns *true* if it holds at the head of the stream.

Of these functions, it is `check_exp` which is the most interesting. Here, we can supply an arbitrary guard expression as per Subsection 4.7.5 to be evaluated. Here, though, we may also wish to express properties over the *outputs* of the EFSM. We can do this easily by defining a new `vname` datatype, `ltl_vname`, as follows.

datatype *ltl-vname* = *Ip nat* | *Op nat* | *Rg nat*

Here, we have inputs represented as `Ip`, outputs as `Op`, and registers as `Rg`. Because the `gexp` datatype takes a type parameter which is used to index variables, we simply use `ltl_vname gexps` in place of the `vname gexps` used in transitions as per Section 4.7, and we can then define expressions in terms of inputs, outputs, and registers.

The above functions allow us to define the property from Example 9.2.2 as the following.

lemma *LTL-output-vend*:
  alw (((*label-eq* ''*vend*'') *aand* (*nxt* (*output-eq* [*Some d*]))) *impl*
    (*check-exp* (*Ge* (*V* (*Rg 2*)) (*L* (*Num 100*))))) (*watch drinks t*)

Here, the logical operations $\wedge$ and $\longrightarrow$ are respectively represented by the `aand` and `impl` operators, which are themselves syntactical constructs from the Isabelle formalisation of LTL which allow us to integrate logical operators into stream expressions. For example $(p\texttt{aand}q)s$ is equivalent to $(\lambda s.ps \wedge qs)$.

Where inductive lists require inductive proofs with a base case and a step case, LTL properties must be proven co-inductively. This is very similar to induction except without the base case. The general form of these proofs is to first prove that the property holds true in the current state and then apply the coinduction principle that *if* the property holds true in the current state, then it holds true in the next state. This then leads to "unrolling" type proofs where we simply prove the property for each control-flow state in the model using an arbitrary register valuation, and then apply the coinduction principle with each outgoing transition. In this instance, we first explore what happens if we attempt to do a *select* action. This takes us into state q_1 and updates the data state. We then prove that the property holds in this state using an auxiliary lemma, the proof of which proceeds similarly.

The second case is what happens if we try any action which is not *select*. Since there is only one outgoing transition from the initial state, any action which is not of the form *select*(d) is invalid, so the model enters the sink state. From here, there is never any output so the property is trivially true. This is proven using the `alw_mono` rule, which states the following.

$$alw\ \varphi\ xs \wedge (\bigwedge xs.\ \varphi\ xs \Longrightarrow \psi\ xs) \Longrightarrow alw\ \psi\ xs$$

That is, if we have a predicate φ, which we know to hold globally over the stream, and for all streams xs, $\varphi\ xs \Longrightarrow \psi\ xs$, then we know that ψ holds on our stream too.

The `alw_mono` rule is extremely useful when we reach the sink state as it often allows us to simplify the predicate hugely, meaning that a proof can often be found automatically. In this example, we can have $\varphi \equiv G(output = [])$ which is known to hold in the sink state.

9.4 Model-Checking with SAL

The previous section showed how we can use Isabelle to prove LTL properties of EFSM models. An Isabelle proof of a property provides conclusive evidence that the property holds, but we only really want to attempt such a proof if we are confident that our property is true. Before this, it is helpful to use a model checker to help iron out any problems, either with the model or the phrasing of our property.

The Symbolic Analysis Laboratory (SAL) is a framework which provides several formal analysis tools, including both a bounded and a symbolic model checker. Of course, SAL represents both models and properties differently to Isabelle so, in order to use the two tools in harmony, we need a translation between the two representations which is semantically consistent. There are many existing model checking tools such as SPIN [86] and nuSMV [35]. SAL was chosen for this work to facilitate the automated translation tool discussed in Section 9.5. Since the implementation of this tool is based on a previous work [49], which translates Z specifications to SAL models, the use of SAL for this work allows some of the original code to be reused.

An alternative to using a model checker would be to implement a model-checking type algorithm using an SMT solver like Z3. Here, we could construct symbolic traces and check these for satisfiability against the specified properties. The advantage of this is that we would not need to limit ourselves to finite types. Unfortunately, this is not feasible for a number of reasons. Firstly, the encoding of temporal properties in Z3 would be rather difficult to implement and would likely be a much clumsier specification language than classical LTL. Secondly, Z3 cannot cope with variable reassignment. In addition to needing a separate variable for every input and output value, each register update would lead to the creation of a new variable. For traces of a reasonable length, this would lead to a very large number of variables which would lead to a very long runtime. For the purposes of this work, I deemed it most appropriate to use an existing model checking tool.

This section details my representation of EFSMs in SAL and argues semantic equivalence between the two representations insofar as is necessary to provide counterexamples to untrue LTL properties. I also briefly discuss a tool to be presented in [138] which is able to automatically perform the conversion between both representations.

9.4.1 Input and Output Sequences

In the Isabelle formalisation of EFSMs presented in Section 4.7, inputs and outputs take the form of lists of values. A major problem which must be overcome here is that SAL has no support for lists out of the box. To get around this, I make use of the implementation of finite *sequences* presented in [49]. These differ from Isabelle's `list` datatype in several ways. Firstly, Isabelle represents lists recursively, with a *head* and *tail*. While Isabelle lists are necessarily finite, they can be arbitrarily long. By contrast, SAL sequences are of a fixed size specified in their type declaration and are implemented as finite functions of fixed domain from natural numbers to elements, like an *array*. Thus, for a sequence s, we have $s[0]$ being the first element and $s[n-1]$ being the n^{th} element.

To enable an exhaustive search for counterexamples, SAL can only work with finite data types. If we allow lists to be an arbitrary length, we can create infinitely many distinct lists, thus making the search space for counterexamples infinitely large. While the SAL language manual[3] does describe how to define recursive lists in SAL, any attempt to check a model involving them results in an error message stating that the type is not finite.

To represent inputs as fixed-length SAL sequences, we must apply a lifting such that all inputs are the same length. To do this, a *bottom element* $\perp$ is required for any data type we would like to form a sequence of. This is used to pad out sequences with a length less than the maximum specified in the type declaration, such that the function from indices to elements is total (i.e. every element has a value). The "length" of the sequence is then defined as the minimum index for which the corresponding element is $\perp$. When representing EFSMs, the length of the input sequences used in the traces is set to the maximum arity of any transition in the EFSM. Similarly, the length for the output sequences is the maximum number of outputs produced by any transition in the EFSM.

Example 9.4.1. Consider the execution ⟨f(1, 2), g(1), h(4, 5, 6)⟩. To represent this in SAL, we need to use input sequences with a maximum length of length three or more, since action h has three inputs. We then have the execution $\langle f(1,2,\perp),\ g(1,\perp,\perp),\ h(4,5,6)\rangle$.

The necessity to represent lists as fixed-length sequences in SAL leads to a slight difference in semantics between the Isabelle and SAL representations. While EFSMs defined in Isabelle can process inputs of arbitrary length, this is not so in SAL since input sequences of length longer than the specified maximum are not members of the datatype. Since the main purpose of using a model checker is to generate *counterexamples*, that is, traces of the model which violate a given property, we do not need to consider actions which take more inputs than the maximum arity of the model since any trace involving such actions is not a valid trace of the model so cannot serve as a counterexample.

Example 9.4.2. Consider again the simple drinks machine EFSM from Figure 1.5. Here, the maximum arity of any transition in the model is one. Thus, any trace containing an event which takes more than one input is not a valid trace of the model, even if the action label is *select*, *coin*, or *vend*. Invalid traces cannot serve as counterexamples to an untrue property, so the fact that we cannot generate them does not affect our ability to refute untrue properties.

[3]`http://SAL.csl.sri.com/doc/language-report.pdf` (Accessed 24/03/20)

9.4.2 Values, Integers, and Strings

In the Isabelle formalisation, we aggregate integers and strings into the sum type `value`. This is the same in SAL. As discussed above, in order to facilitate input sequences of type `value`, we additionally need a bottom element. To achieve this we can first define the datatype `B_value`, which has three cases.

```
B_value : TYPE = DATATYPE
  ValueBB,
  Str(stringOf: STRING),
  Num(intOf: BOUNDED_INT)
END;
```

In `B_value`, the atomic element `ValueBB` represents the bottom element. Clearly we do not want this element to occur in the `value` type, but we do want the other two. Consequently, we define the `value` type as {`g : B_value | g /= value_BB`} meaning that `value` is a subtype of `B_value` which does not include `ValueBB`.

Note that integers are of the type `BOUNDED_INT` rather than SAL's native `INTEGER` type. The reason for this again comes from the fact that, in order to effectively check for the existence of counterexample traces, SAL must work with finite types. Since there are infinitely many integers, we must work with a finite subset of the datatype. The range of the `BOUNDED_INT` type is a parameter passed to the model when it is checked. More specifically, when calling SAL to check a model, it is required that a minimum and maximum integer are specified. SAL then only considers integers within this range.

The necessity to use a finite subset of integers when checking properties introduces two semantic differences from the Isabelle implementation. Firstly, Isabelle is able to handle the infinite integer type, so properties proven in Isabelle over this type hold true for all integers. When SAL declares a property to hold, we can only be certain that it holds for integers within the specified range. This is not just a limitation of SAL, but applies to most established model checkers in one form or another so is basically unavoidable. The way to mitigate this limitation is to use a suitable subset of integers when checking properties. Unfortunately, there is no reliable way to determine this, and it is very much a compromise between reliability and runtime as SAL obviously takes longer to check properties with more possible values.

The second problem that stems from using a finite subset of the integer type is that we open ourselves up to *arithmetic overflow.* That is, when the value of an expression exceeds the maximum integer, it "loops around" to the beginning of the range. For example, if our `BOUNDED_INT` type spans the integers $-10 \ldots 10$ and we have an expression that evaluates to 12, it actually evaluates to -9. Consequently, we effectively have a strange version of modulus arithmetic. To stop SAL producing spurious counterexamples that exploit over- or underflow of variable values, we must explicitly guard against this. This is discussed further in Subsection 9.4.7.

While the Isabelle formalisation uses strings both as labels and as data values, SAL has no support for strings whatsoever. The normal way of defining strings is as a list of characters, which we obviously cannot do here. We could use finite sequences of a maximum length, but this does not make for particularly readable models. Instead, the two different uses of strings in the Isabelle formalisation require two different solutions for SAL.

To represent transition labels, we can form a finite data type `LABEL`. This excludes the possibility of actions with an invalid label but, like with input sequences of a length longer than the maximum transition arity in the model, events with an invalid label cannot appear in genuine counterexamples since traces involving them are not accepted by the model.

Example 9.4.3. Consider again our simple drinks machine. Here, we have transition labels *select*, *coin*, and *vend*. This can be transformed into the following SAL datatype. Actions with a label other than *select*, *coin*, or *vend* are not recognised from any state in the model so cannot appear in any counterexample trace.

```
LABEL : TYPE = DATATYPE
  select, coin, vend
END;
```

Strings also appear as part of the `value` datatype. Here, the necessity for finite datatypes presents more of a problem as we effectively need to enumerate exactly the strings we want to allow as inputs and outputs in counterexamples as a `STRING` datatype. To tackle this, let us consider the PTA in Figure 7.7. Here, some transitions produce literal strings as output, and others have guards which test for particular string inputs. Thus, if we want to represent this EFSM in SAL, we need to include at least these in our datatype. Since SAL does not allow empty datatypes, in EFSMs where no string literals are used in any expression, we need a dummy string. In fact, this is always included as part of the `STRING` type such that there is always at least one string literal which is not explicitly mentioned in the EFSM definition. This helps mitigate the risk of missing counterexamples due to the string datatype being too small.

Example 9.4.4. For the PTA in Figure 7.7, the `STRING` datatype is defined as the following.

```
STRING : TYPE = {String__tea, String__coffee, String__dummy};
```

Of course, there is always a risk of missing counterexamples which require more dummy values than we have supplied but, again, choosing a good set of strings to use is a non-trivial problem. Ideally, the string enumeration would be made an additional parameter when checking properties, but SAL unfortunately does not support this. The way around this is to manually edit the `STRING` datatype to add more strings as necessary.

Example 9.4.5. Consider the situation where our string datatype only contains one dummy element and we called SAL with a `BOUNDED_INT` range of one, any counterexample which required more than two input `values` would be missed. Here, we could add in a second dummy string as follows.

```
STRING : TYPE = {String__dummy1, String__dummy2};
```

Again, this misses counterexamples requiring more than three values, but we can add arbitrarily many strings as necessary.

9.4.3 Control Flow States

In the Isabelle formalisation, control flow states are indexed by natural numbers. As discussed in Chapter 4, this is primarily so that states can arbitrarily be added and removed during the inference process. In SAL, EFSMs are fixed — we do not need to worry about adding or removing states — thus we can index states with a finite enumeration. To do this, we have a data type `states` which enumerates all the states mentioned in the EFSM transition matrix such that a state indexed as n in Isabelle becomes `State__n` in SAL.

Since the intention is to check LTL properties, we need to make our EFSMs *complete*, as discussed in Section 9.3. This is done implicitly by the `make_full_observation` function in the Isabelle formalisation. In SAL, we need to be explicit since properties are defined over *models* rather than *traces.* To do this, we need to define and name an additional sink state. This is indexed as `NULL_STATE`.

Example 9.4.6. The simple vending machine has three states q_0, q_1, and q_2. In SAL, this becomes the following datatype.

```
states : TYPE = {State__0, State__1, State__2, NULL_STATE};
```

As with `make_full_observation`, we need to keep track of the current control flow state. Since we do not process executions explicitly in SAL, we need a local variable, `cfstate`, instead. Continuing the convention established in Chapter 4 that the initial state is always q_0, we initialise `cfstate` to `State__0`. It is then updated by transitions as they are taken.

9.4.4 Data State

In Isabelle, the data state is defined as a finite function from register index (`nat`) to register value (`value option`). SAL does not support an `option` datatype out of the box, so we must define one. Since outputs are sequences of type `value option`, we need an additional bottom value. I therefore follow the same strategy as for the `value` datatype, first defining the `B_option` datatype and then defining `option` as a subset of this without the `OptionBB` bottom element. It has already been proven in Chapter 4 that the set of registers used by any given EFSM is finite, thus we can simply define each register used by a given EFSM as a local variable that can be updated by transitions. Similarly to the Isabelle formalisation, all registers are initialised to **None** and remain unchanged unless they are explicitly updated.

9.4.5 Arithmetic

Like in my Isabelle formalisation, since we are using dynamically typed `values`, we cannot rely on standard arithmetic and must define our own. The semantics of this are identical to those specified in Chapter 4 however, because SAL struggles with recursively defined datatypes, I decided to use a *shallow embedding* instead of deeply embedding expressions as their own datatype like I did in Isabelle. This means that arithmetic expressions are represented using functions over SAL's existing logical and mathematical constructs rather than by defining a syntactic datatype for expressions which must then be explicitly evaluated. It would be possible to define such a datatype in SAL but, since we do not need to recognise and modify the different expressions here, a shallow embedding is the better approach.

The evaluation of arithmetic functions is very similar in SAL to how it is done by the `aval` function in Isabelle. The only difference is that there is no intermediate datatype. Expressions simply appear as functions which can be evaluated directly. Literal values simply appear as they are, as do variable names, which are evaluated in the current context. Addition, subtraction, and multiplication are evaluated by the auxiliary functions `value_plus`, `value_minus`, and `value_times`, each of which takes two `value_options` as input and returns a `value_option`. Because constants and inputs are `values` rather than `value_options`, we need to `Some` them to make expressions type check. For example, rather than writing the expression $i_1 + 5$ as `value_plus(i1, Num(5))`, we must write `value_plus(Some(i1), Some(Num(5)))`.

9.4.6 Guards

The implementation of guards in SAL required the same three-valued Bochvar logic discussed in Subsection 4.7.5. Unlike in the Isabelle formalisation, the number of recursive cases is not a problem here, so I used the conventional logical operators of conjunction, disjunction, and negation as they are more intuitive. To ease translation of the `Nor` element of the `gexp` datatype from Isabelle to SAL, I also define a function `maybe_nor` which is evaluated as "not or". Functions to check equality, and numeric less than, greater than, etc. which take in two `value_options` and return a `Trilean` are also defined with the same semantics as those in Subsection 4.7.5 such that any comparison which is not between two defined numeric values (for example a number and a string or `None`) returns `invalid`.

As in Isabelle, the decision of whether a transition may or may not be taken must be a boolean one. The fact that a guard expression evaluates to `Trilean` *true* means nothing to SAL. To handle this, I defined a function `gval` in SAL which checks if the evaluation of a guard is `Trilean` *true*, returning boolean *true* if it does and *false* otherwise. This is not to be confused with the function of the same name in the Isabelle formalisation which takes a `gexp` and a context and evaluates to a `Trilean`.

9.4.7 Transitions

Isabelle transitions were described in Chapter 4. They are defined using Isabelle's built-in `record` type and have five components: label, arity, guards, outputs, and updates. In Isabelle, an EFSM is defined entirely by its transition matrix, which consists of a set of tuples each of the form $((\mathit{origin}, \mathit{dest}), \mathit{transition})$, where *origin* and *dest* are natural numbers that index the origin and destination states.

Models in SAL use an abstract state machine semantics, as described in Subsection 2.2.6. This means that we do not have explicit states and transitions, rather we have a set of variables and a series of "**if** *condition* **then** *updates*" rules. To represent the control flow states of the EFSM, we define a variable `cfstate`, which is initialised to zero and updated by transitions. Each "transition" has an explicit test to see if the `cfstate` variable holds the correct value representing its origin state. We must also test to see if the given `label` is that of the transition, and that the correct number of inputs have been supplied. This is effectively what the `observe_execution` function does in Isabelle, except that SAL does not give the `cfstate` variable special semantic significance.

Example 9.4.7. The Isabelle representation of the `coin` transition from our simple drinks machine in Figure 1.5, is represented in SAL as shown in Figure 9.1. In SAL a variable followed by a prime indicates its posterior state, and the unprimed version is its anterior value.

In addition to checking the control flow state, and the label and arity of the action, the transition also uses the auxiliary function `check_bounds` to ensure that the result of evaluating the output (and update) expression $r_2 + i_1$ is within the range of `BOUNDED_INT`. As discussed in Subsection 9.4.2, this is to ensure that the transition cannot be taken if doing so would result in an arithmetic overflow.

```
          COIN :
            cfstate = State__1 AND
            label = coin AND
            input_sequence!size?(i) = 1 AND
            check_bounds(value_plus(r__2, Some(i(1))))
          -->
            cfstate' = State__2;
            r__1' = r__1;
            r__2' = value_plus(r__2, Some(i(1)));
            o' = output_sequence!
                insert(value_plus(r__2, Some(i(1))),
                output_sequence!empty)
        []
```

Figure 9.1: A SAL representation of the *coin* transition.

As well as the transitions from the EFSM, we must also add a transition to the `NULL_STATE` which makes the EFSM complete, as described in Section 9.3. To implement this in SAL, we can make use of SAL's keyword `ELSE` which allows the sink-hole transition to fire if and only if no other transition is able to process the current input from the current state. Like with the `make_full_observation` function, in this case, the model moves into the `NULL_STATE` from which it is unable to escape.

```
SINK_HOLE :
   ELSE
 -->
   cfstate' = NULL_STATE;
   o' = output_sequence!empty
```

9.4.8 LTL

Being a model checker, SAL has good support for LTL out of the box. This makes translating between Isabelle syntax and SAL syntax quite straightforward. One noteworthy difference between the ways Isabelle and SAL handle LTL properties is that Isabelle properties are expressed over *traces* where SAL properties are expressed over *models.* In Isabelle, we use the `make_full_observation` function to express properties over models, but we can also express properties of the traces themselves — known as *hyperproperties* [60] — in the same way. For example, we could use Isabelle to express the property of the drinks machine that, if there exists a trace in which we can select a drink, pay for it, and receive it, then there exists a different trace in which we do the same thing with a different drink. There is no way to do this explicitly in SAL. Instead, we would need to use a hyperproperty model checker.

It is also possible, in Isabelle, to leave variables *free,* as has been done with *d* in Example 9.2.2. Here, this has the same effect as using the "for all" quantifier, but free variables can also act as concrete instances of an existential. SAL does not support free variables. Instead, all variables must be quantified either with "for all" or "exists". This can make translating expressions which involve free variables from Isabelle to SAL quite difficult. In such situations, it is preferable to use quantified variables in the first place.

It is important to note that, while it is common to translate *models* from Isabelle to SAL, for *properties* the reverse is more common. The syntax of SAL is much more pleasant to work with, and its speed and efficacy at generating counterexamples makes it the perfect tool when developing properties. Once SAL can no longer find a counterexample, we then translate the property to Isabelle such that we can build a formal proof that it holds. The fact that it is more common to translate properties from SAL to Isabelle means that the fact that SAL is more restricted in the properties it can express is not a problem since we are never faced with properties which can be expressed in SAL which cannot be expressed in Isabelle.

9.5 Automated Translation

Inspired by a similar tool Z2SAL [49] which translates specifications in Z to models in SAL, a major contribution of [138] is a tool which is able to automatically translate EFSMs and LTL properties between Isabelle and SAL. The motivation behind this is that performing this translation manually is both time-consuming and error prone. It therefore makes sense to have an automated tool to do this for us.

My main contribution to this work is the formalisation of EFSMs in both Isabelle and SAL, and the arguments presented above that the two representations are sufficiently semantically equivalent for the purposes of counterexample generation. The implementation of the tool was inspired by and uses much of the original code from [49] but, since I was not involved with this part of the work, I will not discuss it further here.

Before continuing, however, it is important to note one important limitation of this work with regard to LTL expressions. The tool can only work with a subset of LTL expressions that are expressible in Isabelle. Most notably, the tool can only translate expressions in terms of concrete values. While both Isabelle and SAL have the capability to quantify variables, the automated translation tool does not yet support either free or quantified variables. That is to say that the expression from Example 9.2.2 cannot be translated, but the similar expression $G((\mathit{label} = \text{“vend”} \wedge X(\mathit{output} = [\text{“d”}])) \implies r_2 \geq 100)$ can be, since “d” here is a literal string rather than a free variable.

9.6 Case Studies

We have now seen how EFSMs are represented in Isabelle and SAL, how the two representations differ, and that it is possible to automatically translate EFSMs and LTL properties between the two representations. This section illustrates the application of this through three case studies.

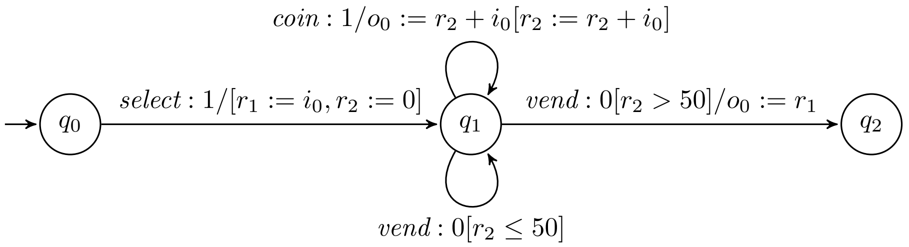
9.6.1 Drinks Machine

We have already seen much of the simple drinks machine example which runs throughout this work. Figure 9.2 shows the full Isabelle definition of the EFSM from Figure 1.5. We can specify properties of the drinks machine model using the LTL framework detailed in Section 9.3. If these properties hold, their proofs are usually reasonably straightforward and, assuming some auxiliary lemmas about the possible steps for each action from each state have already been proven, can often be found with a reasonably high degree of automation.

```
definition select :: transition where
select ≡ ⦇
        Label = STR ''select'',
        Arity = 1,
        Guards = [],
        Outputs = [],
        Updates = [(1, V (I 0)),(2, L (Num 0))]
      ⦈

definition coin :: transition where
coin ≡ ⦇
        Label = STR ''coin'',
        Arity = 1,
        Guards = [],
        Outputs = [Plus (V (R 2)) (V (I 0))],
        Updates = [(2, Plus (V (R 2)) (V (I 0)))]
      ⦈

definition drinks :: transition-matrix where
drinks ≡ {|
          ((0,1), select),
          ((1,1), coin),
          ((1,1), vend-fail),
          ((1,2), vend)
         |}

definition vend:: transition where
vend≡ ⦇
        Label = STR ''vend'',
        Arity = 0,
        Guards = [(Ge (V (R 2)) (L (Num 100)))],
        Outputs = [(V (R 1))],
        Updates = []
      ⦈

definition vend-fail :: transition where
vend-fail ≡ ⦇
        Label = STR ''vend'',
        Arity = 0,
        Guards = [(Lt (V (R 2)) (L (Num 100)))],
        Outputs = [],
        Updates = []
      ⦈
```

Figure 9.2: The Isabelle formalisation of the simple drinks machine EFSM.

Consider now the EFSM in Figure 9.3. This is identical to the EFSM in Figure 1.5 except for the guards on the *vend* transitions. Here, drinks are half price and only cost 50p instead of a pound. If we attempt to prove the property stated in Example 9.2.2, we obviously cannot, since it asserts that r_2 must be greater than or equal to 100 before a user can receive their drink. Here, r_2 only needs to be greater than or equal to 50. The problem with this is that the absence of a proof does not mean that the property does not hold, just that it has not yet been proven to hold. In this instance, we know the property to be false, so we could attempt to prove its negation, but this does not hold for the model either as it asserts that we *never* get a drink if r_2 holds a value greater than or equal to 100.



$coin : 1/o_0 := r_2 + i_0[r_2 := r_2 + i_0]$
$select : 1/[r_1 := i_0, r_2 := 0]$
$vend : 0[r_2 > 50]/o_0 := r_1$
q_0
q_1
q_2
$vend : 0[r_2 \leq 50]$

Figure 9.3: Half price drinks machine.

In this situation, we can use SAL to generate a counterexample to the property in the form of a trace of the model which violates it. This includes concrete input and output values where necessary so it would be possible to run the trace on the real implementation if desired. In this case, the trace should represent an interaction in which the user receives their drink when r_2 holds a value less than 100. To produce a counterexample, SAL must be provided with a definition of the EFSM, and of the LTL property to be checked. Since we already have a specification of both the model and the property in Isabelle, we can convert these to SAL automatically using the tool from [138] mentioned in Section 9.5. We can then call SAL's *symbolic model checker*, which produces the trace shown in Figure 9.4.

```
Counterexample:
========================
Path
========================
Step 0:
--- Input Variables (assignments) ---
label = select
i(0) = Str(String__d)
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__0
r__1 = None
r__2 = None
o(0) = OptionBB
------------------------
Transition Information:
((label SELECT transition at
  [Context: drinks_machine,
   line(39), column(10)]))
------------------------
Step 1:
--- Input Variables (assignments) ---
label = coin
i(0) = Num(90)
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__1
r__1 = Some(Str(String__d))
r__2 = Some(Num(0))
o(0) = OptionBB
------------------------
Transition Information:
((label COIN transition at
  [Context: drinks_machine,
   line(48), column(10)]))
------------------------
Step 2:
--- Input Variables (assignments) ---
label = vend
i(0) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__1
r__1 = Some(Str(String__d))
r__2 = Some(Num(90))
o(0) = Some(Num(90))
------------------------
Transition Information:
((label VEND transition at
  [Context: drinks_machine,
   line(60), column(10)]))
------------------------
Step 3:
--- Input Variables (assignments) ---
label = coin
i(0) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 1
cfstate = State__2
r__1 = Some(Str(String__d))
r__2 = Some(Num(90))
o(0) = Some(Str(String__d))
```

Figure 9.4: A counterexample produced by SAL.

Although Figure 9.4 is a little verbose, it depicts the trace ⟨*select*("d"), *coin*(90)/[90], *vend*()/["d"], *coin*()/[]⟩.[4] Since we have only inserted one coin with value 90 before receiving our drink, this trace clearly violates the property that r_2 must be greater than or equal to 100 before a drink is dispensed. Armed with this trace, we now know the transitions which are involved in violating the property, so can inspect these to investigate where the problem is. Since this example is very small, it is easy to see where the error is and fix the problem by changing the guards on the two *vend* transitions to use 100 rather than 50.

Once the flaw is repaired, we can run the symbolic model checker again. This time it produces the message `proved`, indicating that it was unable to find a trace that violates the LTL property. Because the drinks machine is not a particularly critical system, this is probably sufficient evidence of the property's validity but, as discussed earlier, due to the bounded nature of model checking is a lesser degree of assurance than an Isabelle proof. Fortunately, we can use the translation tool from [138] to convert both the fixed SAL model and the LTL property back to Isabelle and prove our property here, safe in the knowledge that it is now probably true since SAL did not find any counterexamples.

[4]Note that this trace is four elements long. The final *coin*()/[] event allows us to observe the final output ["d"]. Since output is set by transitions as they execute, its value can only be observed in the next state.

While this example is a little contrived, it serves to demonstrate the utility of counterexample traces. The fault is fairly obvious here, but it is easy to see how less obvious faults can be detected with this too. For example, we can correctly phrase the guard $r_2 \geq 100$ as $r_2 > 99$. While both are equivalent, the second guard opens us up to an "off by one" error if we incorrectly type $r_2 \geq 99$ instead. Here, there is only one value of r_2 which is in violation of the property in Example 9.2.2 but SAL is easily able to find this.

9.6.2 LinkedIn Security Exploit

In 2014, the popular social networking site LinkedIn was shown to have a flaw in its behaviour [58]. The site allows professionals to record details of their experience and qualifications, and to form networks with others. Users can join and create profiles for free, and can view the profiles of other users in their network, however they can only view summary information about users who are outside their network. LinkedIn also offers paid subscriptions which allow users to view detailed information about any user inside or outside their network.

User profiles contain a link to download the information they contain as a PDF file. This link is generated dynamically and includes parameters that tell the server whether to provide detailed or summary information depending on the accessing user's status (paid or free) and their relationship (friend or not). One of these parameters is an authentication token generated by the server on a per-session basis. Although they could not directly generate the required session tokens, the researchers in [58] discovered that modifying the parameters in the URL to view someone's profile to make it look like the user was in their network caused the server to generate links to access full user profiles instead of just summary information.

To obtain a model of the LinkedIn system, I converted the steps used in the published vulnerability description [58] into traces and ran them through my inference tool from Chapter 6.[5] The output produced by the tool was a DOT file representing the EFSM in Figure 9.5, which I converted to an Isabelle representation using the translation tool from [138].[6]

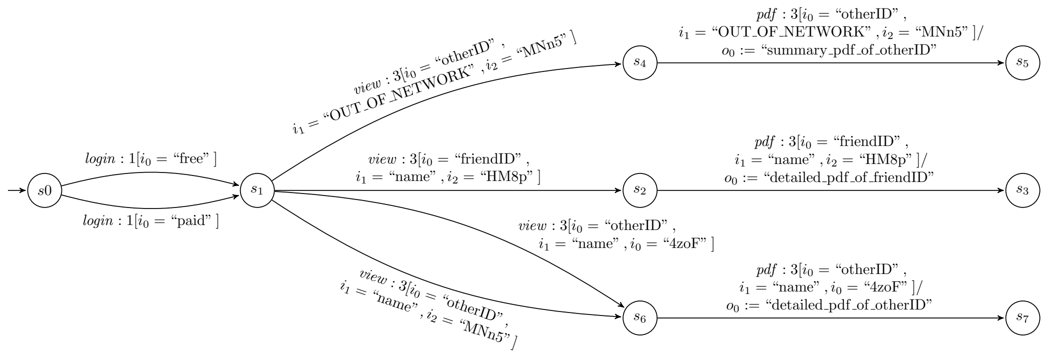

s0
s1
s2
s3
s4
s5
s6
s7
login : 1[i0 = "free"]
login : 1[i0 = "paid"]
view : 3[i0 = "otherID" , i1 = "OUT_OF_NETWORK" , i2 = "MNn5"]
view : 3[i0 = "friendID" , i1 = "name" , i2 = "HM8p"]
view : 3[i0 = "otherID" , i1 = "name" , i0 = "4zoF"]
view : 3[i0 = "otherID", i1 = "name" , i2 = "MNn5"]
pdf : 3[i0 = "otherID" , i1 = "OUT_OF_NETWORK" , i2 = "MNn5"]/ o0 := "summary_pdf_of_otherID"
pdf : 3[i0 = "friendID" , i1 = "name" , i2 = "HM8p"]/ o0 := "detailed_pdf_of_friendID"
pdf : 3[i0 = "otherID" , i1 = "name" , i0 = "4zoF"]/ o0 := "detailed_pdf_of_otherID"

Figure 9.5: An abstract EFSM model of the LinkedIn site protocol

[5]Since this system makes heavy use of string values, generates pesudorandom authentication tokens, and has only four published traces, I did not deem it to be a suitable system to use to evaluate my inference tool in Chapter 8.

[6]All the input and generated files for this example are available at `https://github.com/jmafoster1/efsm-sal/tree/master/linkedin`.

In the traces, each page request is an event, with the label being the page accessed, the inputs being the data parameters, and the output being an abstraction of the returned page. There are three principal actions in the system: *login*, *view*, and *pdf*. The *login* event takes one argument which represents the user ID. These are pseudo-random strings in the real system, but I abstracted them to represent either a `free` or a `paid` user. Only traces involving the `free` user were published in [58], however, the written description of the `paid` user functionality was sufficient to infer traces of what would happen if a paid user had attempted the same process, namely that they should be able to view the full profiles of users outside their network.

The *view* and *pdf* actions each have three inputs. The first input is the ID of the profile to be accessed. These are pseudo-random strings in the real system so, to improve readability, I replaced them with `friendID` (representing the ID of the friend's profile), and `otherID` (representing the ID of the non-friend's profile). The second parameter represents whether the target profile is a friend or not. This appears as `name` if the target profile is a friend and `OUT_OF_NETWORK` if they are not. The third parameter is a pseudorandom authentication token.

The *pdf* action represents generating a PDF version of the viewed profile, and it was this that was exploited. My inference tool has correctly represented the fact that both types of login lead to the same state (s_1) and it should be the other parameters that enact the difference in behaviour. The top three *view* transitions in Figure 9.5 represent the "correct" behaviour. The session parameters `HM8p`, `MNn5`, `4zoF` are the tokens generated for a free user viewing a friend, a free user viewing someone out of their network, and a paid user viewing someone out of their network respectively. The bottom *view* transition represents the attack: by replacing the second input with `name` instead of `OUT_OF_NETWORK`, but leaving the session parameter with the value `MNn5` the attacker is able to cause the system to generate what should be a paid user's session token. This leads the model to the state from which it can take the *pdf* transition which produces the output abstracted to `detailed_pdf_of_otherID`.

The fact that free users can access detailed information of users outside their network of friends is clearly not what was intended. The required property of the system could be expressed as, "*After a user has logged in as a free user, they should never be able to get the `pdf` action to output the detailed report for users who are not their friend.*" Given the abstraction of the millions of potential user IDs into just `friendID` and `otherID`, this becomes the requirement that after logging in as `free`, the *pdf* action called with the input `otherID` should *not* output `detailed_pdf_of_otherID`. Figure 9.6 shows this as an Isabelle lemma.

```
lemma LTL-neverDetailed:
  (((label-eq  ''login'' aand input-eq [Str ''free'']) impl
   (nxt (alw ((label-eq ''pdf'' aand check-exp (Eq (V (Ip 0)) (L (Str ''otherID'')))) impl
   (not (nxt (output-eq [Some (Str ''detailed-pdf-of-otherID'')]))))))))) (watch linkedIn i)
```

Figure 9.6: An LTL property that defines the expected behaviour.

We can attempt to prove the lemma in Figure 9.6 in Isabelle but, since the property is not true of the system as it stands, we will inevitably fail to find a proof. As before, the fact that we cannot prove the property does not show that it is false. With considerable human effort, it is possible to reach a contradictory proof state which reveals that the lemma is indeed untrue. This indicates that there is a flaw in the system, but does not provide any insight into where the problem lies or how we might repair the system. What we really need is a counterexample trace from SAL. Fortunately, this is easy to obtain as we can use the translation tool from [138] once again to convert the Isabelle model and property to SAL. We can then run the symbolic model checker on this to produce the counterexample shown in Figure 9.7.

```
Counterexample:
========================
Path
========================
Step 0:
--- Input Variables (assignments) ---
label = login
i(0) = Str(String__free)
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 3
cfstate = State__0
o(0) = OptionBB
------------------------
Transition Information:
((label LOGIN  transition at
  [Context: linkedin_ext,
   line(42), column(10)]))
------------------------
Step 1:
--- Input Variables (assignments) ---
label = view
i(0) = Str(String__otherID)
i(1) = Str(String__name)
i(2) = Str(String__MNn5)
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__1
o(0) = OptionBB
------------------------
Transition Information:
((label VIEW3  transition at
  [Context: linkedin_ext,
   line(94), column(10)]))
------------------------
Step 2:
--- Input Variables (assignments) ---
label = pdf
i(0) = Str(String__otherID)
i(1) = Str(String__name)
i(2) = Str(String__4zoF)
--- System Variables (assignments) ---
ba-pc!1 = 2
cfstate = State__6
o(0) = OptionBB
------------------------
Transition Information:
((label PDF2  transition at
  [Context: linkedin_ext
   line(134), column(10)]))
------------------------
Step 3:
--- Input Variables (assignments) ---
label = view
i(0) = Str(String__4zoF)
i(1) = Str(String__4zoF)
i(2) = Num(-7)
--- System Variables (assignments) ---
ba-pc!1 = 1
cfstate = State__7
o(0) = Some(Str(String__detailed_pdf_of_otherID))
```

Figure 9.7: The counter-example generated by SAL.

Although the trace is quite verbose, SAL has correctly identified the exploit. First, the user logs in as user `free`. Next, they call the `view` action with the parameters `otherID`, `name`, and `MNn5`. This takes the model into state s_6 in Figure 9.5, from which the `pdf` action can be called with the same parameters to obtain the detailed PDF of the other user's profile.

Armed with a concrete counterexample, the analyst can propose an improvement to the system that prevents this flaw being exploited. In this case, the flaw was in assuming that session tokens in the URL were trustworthy. The solution is to include session information in the server itself. We can add a register r_1 to record whether the user logged in as `paid` or `free` and then use this in the guard expressions of subsequent transitions. The resulting system is shown in Figure 9.8. When we run the symbolic model checker for the same property on the fixed model, it simply states `proved`. Now we know that there is no obvious counterexample, it is worth attempting a proof in Isabelle. This is rather arduous, but does go through successfully.

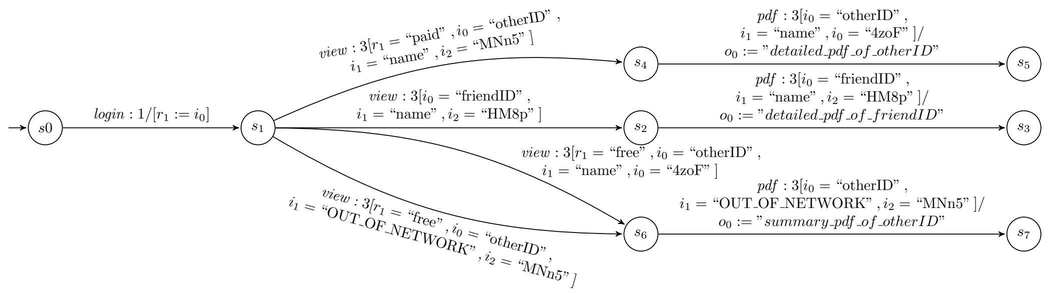

s0
login : 1/[r1 := i0]
s1
view : 3[r1 = "paid" , i0 = "otherID" , i1 = "name" , i2 = "MNn5"]
s4
pdf : 3[i0 = "otherID" , i1 = "name" , i0 = "4zoF"]/ o0 := "detailed_pdf_of_otherID"
s5
view : 3[i0 = "friendID" , i1 = "name" , i2 = "HM8p"]
s2
pdf : 3[i0 = "friendID" , i1 = "name" , i2 = "HM8p"]/ o0 := "detailed_pdf_of_friendID"
s3
view : 3[r1 = "free" , i0 = "otherID" , i1 = "name" , i0 = "4zoF"]
view : 3[r1 = "free" , i0 = "otherID" , i1 = "OUT_OF_NETWORK" , i2 = "MNn5"]
s6
pdf : 3[i0 = "otherID" , i1 = "OUT_OF_NETWORK" , i2 = "MNn5"]/ o0 := "summary_pdf_of_otherID"
s7

Figure 9.8: The EFSM model of the fixed systems

9.6.3 Lift Controller

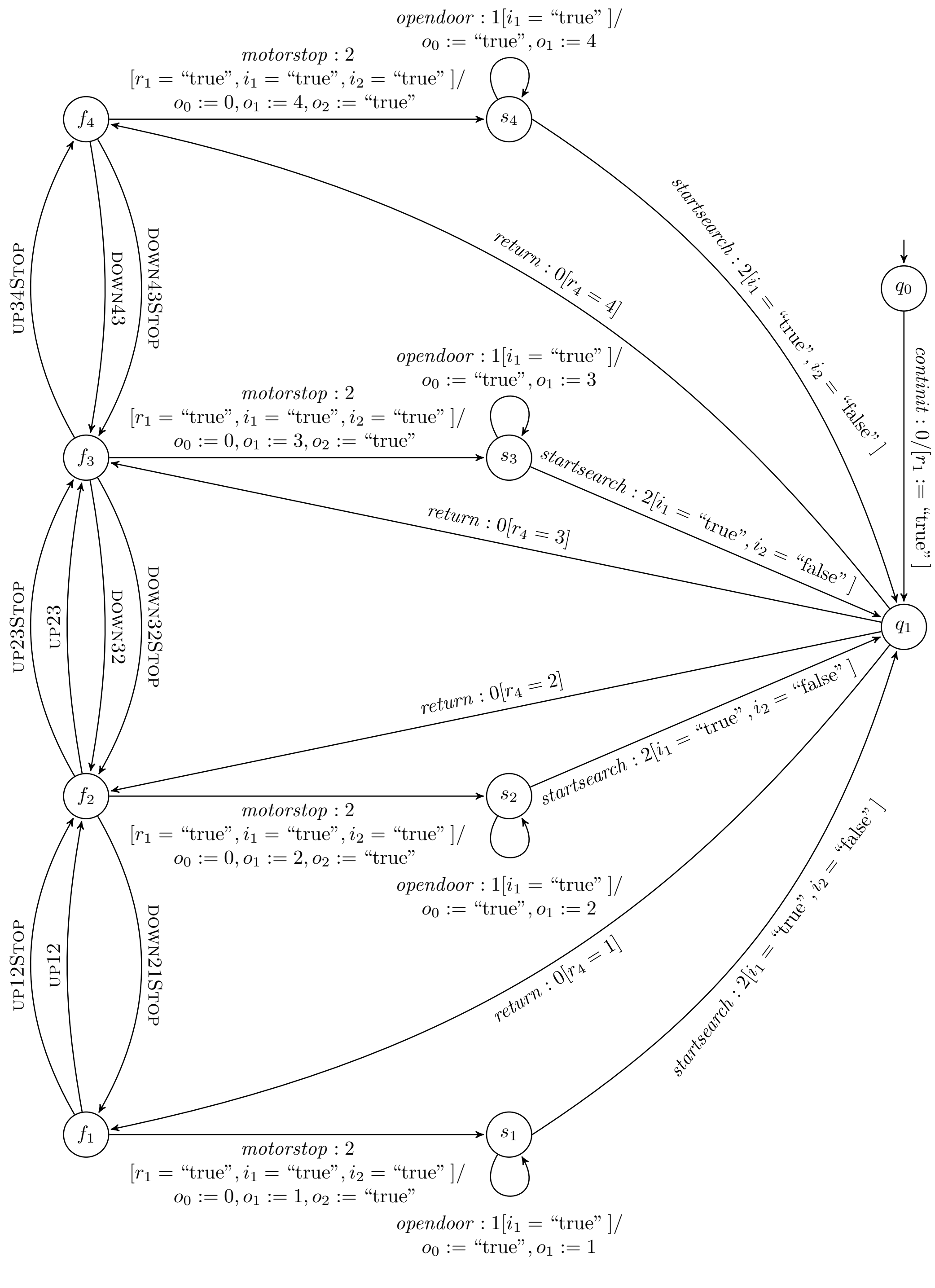
Let us now apply the same technique to a more complex case study. For this, we look again to [136], the source of the liftDoors case study in Chapter 8. This work contains several EFSMs which together model the full functionality of a realistic elevator system with four floors. While the liftDoors system is a popular model for evaluating inference tools, it is actually not a particularly interesting model when it comes to proving properties. Here, I use the "Central Elevator Control" model [136, Figure 3.12] which is shown in Figure 9.9.[7] In the interest of clarity, the *up* and *down* transitions have been assigned abbreviations in the figure and are shown in full in Table 9.1.

down43	$down : 3[r_2 = 2, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"true"}]/$ $o_0 := 2, o_1 := \text{"true"}\ [r_4 := 3, r_1 := \text{"true"}]$
down43Stop	$down : 3[r_2 = 2, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"false"}]/$ $o_0 := 2, o_1 := \text{"false"}\ [r_4 := 3, r_1 := \text{"false"}]$
up34Stop	$up : 2[r_2 = 1, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}]/$ $o_0 := 1, o_1 := \text{"true"}\ [r_4 := 4, r_1 := \text{"true"}]$
down32	$down : 3[r_2 = 2, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"true"}]/$ $o_0 := 2, o_1 := \text{"true"}\ [r_4 := 2, r_1 := \text{"true"}]$
down32Stop	$down : 3[r_2 = 2, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"false"}]/$ $o_0 := 2, o_1 := \text{"false"}\ [r_4 := 2, r_1 := \text{"false"}]$
up23	$up : 3[r_2 = 1, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"true"}]/$ $o_0 := 1, o_1 := \text{"true"}\ [r_4 := 3, r_1 := \text{"true"}]$
up23Stop	$up : 3[r_2 = 1, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"false"}]/$ $o_0 := 1, o_1 := \text{"false"}\ [r_4 := 3, r_1 := \text{"false"}]$
down21Stop	$down : 2[r_2 = 2, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}]/$ $o_0 := 2, o_1 := \text{"true"}\ [r_3 := 1, r_1 := \text{"true"}]$
up12	$up : 3[r_2 = 1, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"true"}]/$ $o_0 := 1, o_1 := \text{"true"}\ [r_1 := \text{"true"}, r_4 := 2]$
up12Stop	$up : 3[r_2 = 1, r_1 = \text{"false"}, i_1 = \text{"true"}, i_2 = \text{"true"}, i_3 = \text{"false"}]/$ $o_0 := 1, o_1 := \text{"false"}\ [r_4 := 2, r_1 := \text{"false"}]$

Table 9.1: The *up* and *down* transitions from Figure 9.9.

As always, state q_0 is the initial state. In fact, the initialisation of the lift is quite complex and is abstracted out into a separate EFSM model which sits within state q_0. This model is not shown here but can be found in full in [136, Figure 3.13]. In this example, only the top-level behaviour illustrated in Figure 9.9 needs to be considered. This thesis does not cover how EFSMs behave when arranged hierarchically, as it is somewhat outside of the scope of this work. In essence, the initialisation sub-model serves to ensure that the lift is properly initialised before operation. This means that the doors are closed and it is on the first floor. In Figure 9.9, the current floor of the lift is held by the register r_4.

[7] I make a slight deviation here from [136] in that all of my *up* and *down* transitions have the labels *up* and *down* respectively. This is not the case in [136]. Here, these transitions are uniquely labelled according to their origin and destination floors and whether the lift is to stop at the relevant floor. This deviation has little effect on the model itself but does make Isabelle proofs simpler since there are fewer unique labels.



opendoor : 1[i_1 = "true"]/ o_0 := "true", o_1 := 4
motorstop : 2 [r_1 = "true", i_1 = "true", i_2 = "true"]/ o_0 := 0, o_1 := 4, o_2 := "true"
f_4
s_4
UP34STOP
DOWN43
DOWN43STOP
return : 0[r_4 = 4]
startsearch : 2[i_1 = "true", i_2 = "false"]
q_0
continit : 0/[r_1 := "true"]
opendoor : 1[i_1 = "true"]/ o_0 := "true", o_1 := 3
motorstop : 2 [r_1 = "true", i_1 = "true", i_2 = "true"]/ o_0 := 0, o_1 := 3, o_2 := "true"
f_3
s_3
startsearch : 2[i_1 = "true", i_2 = "false"]
return : 0[r_4 = 3]
UP23STOP
UP23
DOWN32
DOWN32STOP
q_1
return : 0[r_4 = 2]
startsearch : 2[i_1 = "true", i_2 = "false"]
f_2
s_2
motorstop : 2 [r_1 = "true", i_1 = "true", i_2 = "true"]/ o_0 := 0, o_1 := 2, o_2 := "true"
opendoor : 1[i_1 = "true"]/ o_0 := "true", o_1 := 2
UP12STOP
UP12
DOWN21STOP
return : 0[r_4 = 1]
startsearch : 2[i_1 = "true", i_2 = "false"]
f_1
s_1
motorstop : 2 [r_1 = "true", i_1 = "true", i_2 = "true"]/ o_0 := 0, o_1 := 1, o_2 := "true"
opendoor : 1[i_1 = "true"]/ o_0 := "true", o_1 := 1

Figure 9.9: The EFSM for the lift controller.

State q_1 represents the lift in its "idle" state. It is stationary with its doors closed. It does not contain any passengers and awaits a call to a particular floor. Again, this state contains a sub-model (which can be found in [136, Figure 3.14]) to handle the control logic of summoning the lift. This is actually quite complex since this model has a directional priority element, meaning that the lift will service all requests from the direction it is travelling before changing direction. The lift does not store floors to visit in a list, rather, it stores the current direction of travel, the floor it is currently at, and whether it should stop at the next floor it reaches.

States f_1 to f_4 in Figure 9.9 represent the lift being in motion. The lift can travel between floors arbitrarily, but the control flow is such that it cannot ascend above floor four, nor descend below floor one. This is because the only incoming transitions to the respective states stop the lift. It is also impossible to select a floor which is not in the range [1..4]. States s_1 to s_4 represent the lift being stationary at a particular floor. Here, the doors may open to allow passengers to enter or alight, after which the lift awaits instruction to visit a particular floor.

A basic safety property of most modern lifts is that, under normal circumstances, they must be stationary at a floor before the doors can be opened. The desire for this is obvious: if we can open the doors while the lift is moving, there is a risk that body parts, pets, or items of clothing could get caught on the wall the lift shaft, potentially causing damage either to the lift mechanism or the passengers. To make the lift as safe as possible, we would like to verify that our lift controller does not allow this to happen.

To verify our property, we must first formalise the intuition in LTL. To do this, we need to specify what it means to "stop the motor" and to "open the doors". In Figure 9.9, the stopping of the motor is done by the *motorstop* transitions, and the opening of the door is done by the *opendoor* transitions. The action of successfully opening the doors can be characterised by calling the *opendoor* action and receiving the output $[n, \text{“true”}]$, where n is the current floor number. We can then phrase the property "we cannot open the door until we have first stopped the lift" in LTL as the following, in which n has been left as a free variable to make the property independent of the floor the lift is currently on.

$$((\neg(\mathit{label} = \mathit{opendoor} \wedge X(\mathit{output} = [n, \text{“true”}]))) \ \mathcal{W} \ (\mathit{label} = \mathit{motorstop})) \tag{9.1}$$

This phrasing of the property omits the outputs of *motorstop* and the inputs to both actions. This is because they do not affect the validity of the property. Calling either action with invalid inputs will cause the model to enter an implicit sink state, from which our property trivially holds since we can never successfully open the doors. Similarly, we do not care about the outputs of *motorstop* either since, if it is called unsuccessfully, the model goes into the sink state, from which we cannot open the doors.

Note also that in Equation 9.1, we check the output of *opendoor* in the next state rather than the current state. Recall from Section 9.3 that this is because the output of the current action can only be observed once that action has been completed, i.e. in the next state. This is common both to the Isabelle framework and to SAL.

Here, we need to use the weak variant of the *until* operator since we do not necessarily want to enforce that the motor is eventually stopped. If the lift doors never open, we do not need to ever stop the lift. If we were to use the strong variant of *until*, we would eventually have to stop the lift. This would make the property trivially untrue, since we could simply never call the *motorstop* action. At some point, this would result in us ending up in the sink state from which we can neither stop the motor nor open the doors, but this does not affect the invalidity of the property. We could phrase our property to explicitly exclude the sink state, but this is not as elegant or intuitive as the phrasing in Equation 9.1.

Looking at the Figure 9.9, we can see that the property in Equation 9.1 holds. We can only open the doors in states s_1-s_4, which can only be reached by calling the *motorstop* action. The proof of this property in Isabelle is a relatively straightforward unrolling proof, showing that the model enters the sink state if the *opendoor* action is called before *motorstop*.

Unfortunately, Equation 9.1 does not really capture the intuition of what we want to verify as it does not operate over the entire lifetime of the lift controller. More specifically, once we have stopped the motor for the first time, the *until* operator is released meaning that anything can happen after this point. It may be the case that, once the motor starts up again, we are then free to open the doors while the lift is in motion. Equation 9.1 does not pick this up, so we need to phrase our property in such a way that it operates *globally.* An intuitive way to do this is to simply wrap a "globally" operator around Equation 9.1, as in Equation 9.2.

$$G((\neg(label = opendoor \wedge X(output = [n, \text{“true”}]))) \; \mathcal{W} \; (label = motorstop)) \tag{9.2}$$

While Equation 9.2 is *intuitive*, it is not actually true. It is here that SAL comes into its own. Instead of wasting days of effort trying to prove this untrue property, we can simply use the tool from [138] to convert the model and property to the SAL representation and generate a counterexample. Unfortunately, while working with this example, I was so confident that Equation 9.2 was true that I did not do this. Instead, I attempted to jump straight into an Isabelle proof, only realising that the property was untrue after a couple of days of work. While frustrating, this serves as both a cautionary tale and a demonstration of the utility of counterexample generation.

To generate counterexamples for the lift controller in SAL, we first need to apply small workaround to account for the fact that we are ignoring the hierarchical structure of the model. This workaround is to initialise r_4 (the register which holds the current floor) to 1, as is done by the initialisation sub-EFSM. This could be seen as "cheating" but is sufficient for our purposes here and saves us the trouble of having to condense the hierarchy in [136] into a single model.

With the workaround applied, SAL's symbolic model checker can then generate the counterexample shown in Figure 9.10. In situations where SAL produces a counterexample, we know that there is either a problem with the *model* or the *property.* Here, the fault lies in phrasing of the property. The problem is that *opendoors* is a reflexive transition, meaning that we can call it as many times as we like. Since, in s_1, the motor is already stopped, we do not have to explicitly stop it again before opening the doors. We must therefore phrase our property differently. Counterexample generation is invaluable at this stage, as it allows us to quickly iterate our property until it holds. At each stage, we get a concrete counterexample which allows us to improve the property.

We can see from the model in Figure 9.9 that we can only successfully perform an *opendoor* action from a state where the motor is stopped, i.e. s_1 to s_4. We could, therefore, check this as part of the statement. This phrasing is shown in Equation 9.3, and states that it is always the case that we cannot open the door until the control flow state is s_1, s_2, s_3, or s_4.

$$\begin{aligned}
&G(\\
&\quad F(label = opendoor \wedge X(output = [n, \text{“true”}])) \implies \\
&\quad ((\neg(label = opendoor \wedge X(output = [n, \text{“true”}])) \; \mathcal{W} \; (cfstate \in \{s_1, s_2, s_3, s_4\}))) \\
&\quad)
\end{aligned} \tag{9.3}$$

```
Counterexample:
========================
Path
========================
Step 0:
--- Input Variables (assignments) ---
label = continit
i(0) = ValueBB
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 9
cfstate = State__0
r__1 = None
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = OptionBB
o(1) = OptionBB
o(2) = OptionBB
------------------------
Transition Information:
((label CONTINIT transition at
  [Context: liftcontroller3,
   line(42), column(10)]))
------------------------
Step 1:
--- Input Variables (assignments) ---
label = return
i(0) = ValueBB
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 9
cfstate = State__9
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = OptionBB
o(1) = OptionBB
o(2) = OptionBB
------------------------
Transition Information:
((label RETURN1 transition at
  [Context: liftcontroller3,
   line(221), column(10)]))
------------------------
Step 2:
--- Input Variables (assignments) ---
label = motorstop
i(0) = Str(String__true)
i(1) = Str(String__true)
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 9
cfstate = State__1
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = OptionBB
o(1) = OptionBB
o(2) = OptionBB
------------------------
Transition Information:
((label MOTORSTOP1 transition at
  [Context: liftcontroller3,
   line(95), column(10)]))
------------------------
  Step 3:
  --- Input Variables (assignments) ---
  label = opendoor
  i(0) = Str(String__true)
  i(1) = ValueBB
  i(2) = ValueBB
  --- System Variables (assignments) ---
  ba-pc!1 = 9
  cfstate = State__5
  r__1 = Some(Str(String__true))
  r__2 = None
  r__3 = None
  r__4 = Some(Num(1))
  o(0) = Some(Num(0))
  o(1) = Some(Num(1))
  o(2) = Some(Str(String__true))
  ------------------------
  Transition Information:
  ((label OPENDOOR1 transition at
    [Context: liftcontroller3,
     line(183), column(10)]))
  ------------------------
Step 4:
--- Input Variables (assignments) ---
label = opendoor
i(0) = Str(String__true)
i(1) = ValueBB
i(2) = ValueBB
--- System Variables (assignments) ---
ba-pc!1 = 8
cfstate = State__5
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = Some(Str(String__true))
o(1) = Some(Num(1))
o(2) = OptionBB
------------------------
Transition Information:
((label OPENDOOR1 transition at
  [Context: liftcontroller3,
   line(183), column(10)]))
------------------------
Step 5:
--- Input Variables (assignments) ---
label = up
i(0) = Num(-1)
i(1) = Str(String__true)
i(2) = Str(String__false)
--- System Variables (assignments) ---
ba-pc!1 = 20
cfstate = State__5
r__1 = Some(Str(String__true))
r__2 = None
r__3 = None
r__4 = Some(Num(1))
o(0) = Some(Str(String__true))
o(1) = Some(Num(1))
o(2) = OptionBB
```

Figure 9.10: The SAL counterexample for the property in Equation 9.2.

This property is not particularly robust, however, since it is not *model independent.* That is, it is tied to how we have modelled the lift controller. If we rename the states, the property may no longer hold. It also requires us to have some knowledge of the model such that we know that the motor is stopped in the relevant states. Ideally, we would like to avoid this kind of property and instead phrase things purely in terms of input and output. This way, our properties are not tied to the model, and we can arbitrarily rename states and registers. We can even change the structure of the model completely if we want to. If, at some point, we change the model and the property no longer holds, we know that the fault lies with the new model not the property.

When working with this example, it took several iterations before I eventually settled on the property shown in Equation 9.4, for which SAL's symbolic model checker produces the output `proved`. Equation 9.4 is rather convoluted and needs a little disentanglement. The outer *globally* operator states that the property must hold true throughout the entire execution of the model. The inner predicate states that if eventually there is a next state in which we successfully open the doors, this does not occur until either the label is *motorstop* or the output is $[n, \text{“true”}\,]$. It is this second disjunct which is the key. What this is doing is exploiting the fact that, when we open the doors, the outputs are the current floor n and the string “true”. We could make this more explicit by writing $label = opendoor \wedge X(output = [n, \text{“true”}\,]$ instead, but this does not affect the validity of the property since *opendoors* is the only transition which could ever produce this output.

$$
\begin{aligned}
&G(\\
&\quad F(X(label = opendoor \wedge X(output = [n, \text{“true”}\,]))) \implies \\
&\quad (\neg(X(label = opendoor \wedge X(output = [n, \text{“true”}\,]))) \ \mathcal{W} \\
&\quad (label = motorstop \vee X(output = [n, \text{“true”}\,]))) \\
&)
\end{aligned} \tag{9.4}
$$

The *next* operations somewhat obfuscate the meaning of Equation 9.4. These are necessary to get round the fact that LTL only looks into the future. What this property is effectively saying is that, if we successfully open the doors, the *previous* event was either stopping the motor or opening the doors. We do not have a “previously” operator in LTL though, so we must look one step into the future to effectively treat the current action as the “previous” event.

With SAL unable to find a counterexample for this property, we can now embark on an Isabelle proof. The first step is to translate the SAL property into Isabelle syntax. This can be done automatically using the tool from [138], which produces the lemma in Figure 9.11.

```
lemma alw-must-stop-to-open:
  alw ((ev (nxt ((label-eq ''opendoor'') aand
        (nxt (output-eq [Some (Str ''true''), Some  n]))))) impl
       ((not (nxt ((label-eq ''opendoor'') aand
        (nxt (output-eq [Some (Str ''true''), Some  n]))))) until
       (((label-eq ''motorstop'') or (nxt (output-eq [Some (Str ''true''), Some  n]))))))) (watch lift i)
```

Figure 9.11: The Isabelle representation of the property in Equation 9.4.

To prove this, it is helpful to strengthen the property so it applies to all control flow and data states. The reason for this is that the model contains many *cycles*, so we can easily loop back around to states which we have already visited with a different register state. If we do not generalise our property, we end up with potentially infinite proof goals, which we obviously cannot fulfil. To do this, we rephrase the lemma in Figure 9.11 to that in Figure 9.12.

```
lemma alw-must-stop-to-open-gen:
  assumes ∃ s r p t. j= make-full-observation lift (Some s) r p t
  shows alw ((ev (nxt ((label-eq ''opendoor'') aand
            (nxt (output-eq [Some (Str ''true''), Some n]))))) impl
           ((not (nxt ((label-eq ''opendoor'') aand
            (nxt (output-eq [Some (Str ''true''), Some n]))))) until
           (((label-eq ''motorstop'') or (nxt (output-eq [Some (Str ''true''), Some n])))))) j
```

Figure 9.12: A generalised version of the property in Figure 9.11.

Recall that the coinduction rule for `alw` is that the property must hold true in the current state and globally henceforth. Phrasing proof goals as in Figure 9.12 makes the coinductive step trivial, as we simply need to prove that we can take a step from any state. Because our EFSM is implicitly complete, we can easily prove this. This means that we only need to think about the current situation. That is, we need to prove that the inner predicate holds in every state for every register configuration. It is helpful to phrase this as the lemma in Figure 9.13.

```
lemma alw-must-stop-to-open-aux:
  assumes ∃ s r p t. j= make-full-observation lift (Some s) r p t
  shows ((ev (nxt ((label-eq ''opendoor'') aand
            (nxt (output-eq [Some(Str ''true''), Some n]))))) impl
       ((not (nxt (label-eq ''opendoor'' aand
        (nxt (output-eq [Some(Str ''true''), Some n]))))) until
       (((label-eq ''motorstop'') or (nxt (output-eq [Some(Str ''true''), Some n])))))) j
```

Figure 9.13: The inner predicate of Figure 9.12.

Proving the lemma in Figure 9.13 turns out to be quite an undertaking as we must consider each state in the model as a separate case. This leaves us with ten subgoals: one for each state and an extra one for an arbitrary invalid state. While none of these subgoals are particularly intellectually challenging, the sheer number of them makes the proof both long and tedious.

Having proved that the property in Equation 9.4 holds of the lift controller, we now know that it is impossible to open the doors while the lift is in motion at any point during its operation. Because we made the exact floor a *free variable*, we only need to have proved this once rather than for each floor. Additionally, because we have proven that the property holds for all control flow states and register configurations, we needn't concern ourselves with how either of the submodels modifies the data state. If we were to work these models into our EFSM, the property would still hold since neither of them involves opening the doors.

9.7 Conclusion

This chapter has illustrated how two different techniques, theorem proving and model checking, can be used to prove LTL properties of EFSM models. The ability to do this is an important step in the certification of many safety-critical systems, so there is a clear motivation for this. I have shown how the two techniques can be used to complement each other by presenting a bidirectional translation between the representations of Isabelle and SAL which is sufficiently semantically equivalent that we can use SAL to find *counterexamples* to untrue properties. This is made easier by the use of an automated tool [138] to convert between the two representations.

The counterexamples generated by SAL can then be used to improve either the model or the property until no counterexamples can be found. At this point, we can then convert the model and the property to Isabelle and use *coinduction* to prove that the property holds.

While it is impossible to quantitatively evaluate the current approach, it is nonetheless possible to identify some of its strengths and weaknesses. The major strength of the translation approach is ease of expression. While my framework of definitions for expressing LTL properties from Section 9.3 is certainly an aesthetic improvement to the lambda notation, the syntax of SAL and its ability to quickly find counterexamples to untrue properties make a better tool for developing models and properties. The ability to automatically translate between SAL and Isabelle enables us to make use of this without extra effort.

The flip side of this is that the tool can only translate properties involving the predicates in Section 9.3, so is only able to translate a subset of what can be expressed in Isabelle and SAL individually. Users cannot make full use of Isabelle's lambda notation to express functions, nor can they express properties using SAL's literal mathematical operators. The *check_exp* function allows the use of arbitrary guard expressions, all of which are translatable, meaning that this should rarely be a problem, but users are still limited both in what they can express and how they can express it. Further to this, there is currently no translation support for quantifiers and free variables, which means that we cannot yet translate properties which operate over all possible values or assert that there exists a value for which a property holds.

Another limitation is that, while it is technically possible to express and prove any LTL property in Isabelle, the proofs are often extremely long and arduous. The fact that it is difficult to prove interesting properties of complex models should not be surprising, but expert-level knowledge of Isabelle is required in order to prove even relatively simple properties. Because Isabelle's coinduction package is relatively new, there is much less automation than for standard inductive proofs. This means that the exact phrasing of auxiliary lemmas is much more important to Isabelle's proof tactics than for standard inductive proofs, and it is often necessary to explicitly substitute for schematic variables. This is not likely to improve any time soon, but could be mitigated to an extent by the creation of *proof tactics* which can be applied to automatically break down proofs into subgoals that are easier to work with. The implementation of such tactics is left for future work.

It is also worth relating the work presented in this chapter back to Chapter 6. There, it was mentioned that we can use SAL to check the various properties which imply direct subsumption. Indeed, the automated translation tool has Java bindings and is able to produce SAL representations of the intermediate models during inference. As discussed in Chapter 6, though, for realistically sized models the use of SAL during inference is too slow to be practical.

Chapter 10
Conclusion

This thesis aimed to advance the field of model inference from traces, specifically the inference of *extended* finite state machines. Thus, the thesis aimed to answer the high level research question "What strategies can we apply to automatically infer extended finite state machine models from black-box traces?". The thesis answered this question by addressing the following main objectives.

- To bring together desirable characteristics from the various existing EFSM definitions in the literature to form a new definition which is well suited to inference.
- To establish a technique which can be used to determine whether one EFSM transition can account for the behaviour of another such that they can be merged.
- To establish a state merging technique to infer extended finite state machine models, including functions to relate inputs and outputs and mutate the data state, from black-box software execution traces.
- To evaluate this technique with respect to a baseline and the current state of the art.
- To establish techniques to aid the process of the verification of properties of EFSM models once they have been inferred.

10.1 Summary of the Thesis and its Contributions

In Chapter 1 of this thesis, I set out the above objectives and motivated the work with a simple example. Chapters 2 and 3 went on to provide the necessary background material which makes up the foundation of the research presented here.

Chapter 4: Extended Finite State Machines

This chapter presented my novel definition of EFSMs which incorporates the desirable aspects from various existing definitions into a single model definition. This definition was then formalised in Isabelle/HOL to serve as a foundation for the work presented in subsequent chapters. In addition to this, some key properties of EFSMs were proven to show that the formalised definition is consistent with the intuition of how EFSMs should behave.

C1: The formulation of a new EFSM definition that combines desirable characteristics from the literature.

Chapter 5: Formalising EFSM Transition Merging

This chapter introduced *contexts* as a means of recording the values of registers at various points during the execution of an EFSM model. It went on to use contexts to define the *subsumption in context* relation as a way to determine whether one EFSM transition can account for the behaviour of another given a particular register valuation. The *direct subsumption* relation was defined on top of this, to determine whether it is safe to merge a given pair of transitions. These definitions were formalised in Isabelle/HOL using my EFSM formalisation from Chapter 4 as a basis and various properties, including reflexivity and transitivity, were proven.

C2: The *subsumption in context* and *direct subsumption* relations.

Chapter 6: EFSM Inference from Traces

This chapter presented a state merging technique to infer EFSM models from black-box traces using the direct subsumption relation from Chapter 5 as a foundation. As part of this technique, simple *heuristics* were used to recognise certain data usage patterns and abstract away concrete values in favour of more general functions to *compute* output from input. This technique was implemented as a prototype tool using the Isabelle formalisation from Chapter 4 as a basis.

C3: The use of direct subsumption to formulate a state merging algorithm for EFSM inference.

Chapter 7: Using Genetic Programming to Infer Computation

The work in this chapter aimed to overcome the limitations of the heuristics used in Chapter 6. The chapter first introduced the concepts of evolutionary computation and genetic programming before using these ideas as part of a novel technique to infer functions which relate inputs and outputs, and recognise when and how registers need to be used. This technique can also be used to infer guards to distinguish transitions with value-dependent behaviour.

C4: A technique to infer functions which relate inputs, outputs, and registers as well as transition guards.

Chapter 8: Experimental Evaluation

This chapter carried out an empirical investigation into the performance of my inference technique in comparison with a baseline approach and the current state of the art.

C5: An empirical evaluation of my inference technique.

Chapter 9: Formal Analysis of EFSM Properties

This chapter was concerned with the verification of models once they have been inferred. The chapter first introduced the field of formal verification and temporal logics before going on to discuss how Isabelle/HOL can be used to prove properties of EFSMs specified in LTL. I also presented my framework of functions designed to make this process easier. Next, I discussed how we can use SAL to find counterexamples to untrue properties and why this is useful. Finally, the chapter demonstrated the use of this in the context of three case studies.

C6: A framework to allow model checking and theorem proving to be used in tandem to prove properties of EFSMs.

10.2 Limitations

This thesis includes some limitations in both its artefacts and their evaluation. For instance, the current implementation of EFSMs only supports integer and string values. Floating point numbers, and more complex datatypes such as lists are not supported. The primary focus is on numeric operations, with the only supported operations being addition, subtraction, and multiplication. Division is not supported, as this really needs to be an operation over floats.

The GP preprocessing technique presented in Chapter 7 has limited support for shared variables. It is only able to succeed in cases where a variable is mutated by at most one action. Other actions can *reset* the variable to a constant value, but only certain actions can have a non-constant update.

The GP is also highly dependent on the values which occur in the traces. The assumption is made that suitable values for any latent variables appear at some point in the traces. If this is not the case, the performance of the GP is severely inhibited.

Another limitation the GP preprocessing technique is faced with is the fact that it only introduces one latent variable per transition, and is not allowed to use external registers as part of update functions. This means that it is likely to perform poorly for systems with more complex data dependencies.

The evaluation of my techniques in Chapter 8 is limited in the number of case studies which could be examined. Since finding suitable systems is extremely difficult, I was only able to use three case studies in my evaluation. These systems may not adequately represent larger real-world systems, so the confidence in the results is relatively low.

10.3 Future Work

This section presents several ideas for future work.

10.3.1 Increased Datatype and Operation Support

My current implementation of EFSMs only supports integers and strings. To improve the applicability of my inference tool, I recommend extending the supported datatypes to at least incorporate floating-point numbers. Ideally, the supported datatypes would be parametric such that users could easily create custom datatypes.

The situation is similar for operations over the different datatypes. Currently, the focus is purely numeric and has only three supported operations. I recommend increasing support to at least include numeric division and string concatenation. Again, the supported operations should ideally be parametric so that users can easily define their own.

10.3.2 Inference of Output and Update Functions

The current GP-based approach suffers from several limitations. One direction for future research is to increase support for shared variables, possibly by attempting to "share" existing registers in the model with the GP before introducing new ones. It is also desirable to take steps to increase the complexity of systems for which the GP will perform well. Allowing more than one latent variable to be used as part of output functions and allowing other registers to influence update functions is highly desirable if it can be performed reliably. Additionally, in the conclusions of Chapter 7, I identified co-evolution as a possible strategy to infer output and update functions at the same time. If this could be implemented successfully, this may be better able to cope with more complex systems since the ability to infer suitable updates can then affect the fitness of a given output function.

Another limitation to the current approach is that the literal input guards are dropped on *all* transitions, whether or not an output function was successfully inferred. In situations where an output function could not be inferred, this can be extremely problematic for the inference process as it increases the potential for nondeterminism to arise during state merging without having mitigated this by generalising the output behaviour. Consequently, I recommend an alternative technique where the guards on transitions are only dropped if an output function has successfully been inferred. This is not as easy as only dropping guards on transitions which do not produce a literal output value, though. We must distinguish those literal outputs which are representative of a particular behaviour (i.e. successful generalisations in and of themselves) from those which have simply been retained from the original PTA because we could not successfully generalise the behaviour. It is only in the second case where we wish to retain the guard.

An alternative to this approach would be to allow the GP to use "**if** ... **then** ... **else** ..." conditions. This would then allow it to infer functions to generalise *subsets* of the training set, if it cannot be generalised in its entirety. The **if** conditions then afford us a relatively easy way to retain guard conditions while still generalising as much as possible.

10.3.3 Inference of Guards

The current GP-based approach for the inference of guards suffers from two main limitations. Firstly, transitions often end up accumulating redundant guards as the inference process proceeds. Secondly, guards are often inferred with too small a training set, making them too specific to be generally applicable. As a result of this, I recommend an additional preprocessing technique be implemented. In this technique, we would form groups of transitions which may be nondeterministic with each other but can clearly never be merged. For example, transitions with the same label and arity may be nondeterministic, but we clearly cannot merge transitions which produce different numbers or types of outputs. We could then attempt to infer guards to distinguish these groups of transitions. Doing this would give the GP access to larger training sets and, thus, we would be more likely to end up with more generally applicable guards.

10.3.4 State Merging

My current state merging approach ranks merges based on the number of common outgoing transitions from pairs of states. As discussed in Chapter 6, this thesis focusses on the ability to infer the functions on transitions which relate inputs, outputs, and internal variables rather than on the impact of different scoring metrics. Since the results in Chapter 8 show that, even with this simple metric, the models my technique infers are often more accurate than those inferred by MINT [152], I did not feel the need to explore this avenue further here. Nevertheless, there is still much work which can be done in this area. For example, we could try to lift the Blue-Fringe algorithm [98] to EFSM models, or make use of data classifiers like MINT [152].

10.3.5 Deep Learning

As mentioned in Chapter 3, traditional machine learning techniques such as deep learning can infer very accurate predictive models given sufficient training data. While these models are generally completely black-box, if the primary application of inferring EFSM models is something like automated regression testing (where the models do not have to be human readable), there is no reason not to use these techniques to relate input and output functions. This would likely require significant changes to the current implementation, which is built with human readable functions in mind, however, it is certainly an interesting avenue to pursue.

10.3.6 Combining Active and Passive Inference

Another interesting potential research area would be to combine active and passive learning as mentioned in Chapter 3. That is, we could use passive techniques, like those presented in this thesis, to come up with an initial model which could be fed to an adaptive learner. Moreover, because the output of my inference tool depends on both the input traces and a random seed (if GP is used), we could infer several different models for the adaptive learner to work with. Assuming the models inferred by my technique could easily be converted to an acceptable format for an existing adaptive learning tool, this process would likely be relatively straightforward.

10.3.7 Evaluation

Given that it is extremely difficult to evaluate inference techniques, I recommend that several steps be taken to mitigate this. Firstly, I recommend that a “gold standard” dataset be established. This would consist of a large set of programs and associated traces and would enable the creators of subsequent inference tools to more easily evaluate and compare their approaches.

In the field of classical FSM inference, techniques are often evaluated using traces obtained by randomly walking through models which have been generated automatically [98, 151]. This enables an arbitrary number of sample systems to be generated with configurable levels of difficulty. It is much harder to apply a similar approach here since EFSM traces and transitions are much more complex than their classical counterparts. While approaches do exist for generating feasible *traces* from existing EFSM models [92], the area is not well explored and there is no work in the literature which investigates the generation of random *models.*

Clearly it is only worth evaluating systems with respect to randomly generated models if the results obtained by doing this translate to real-world systems. There is currently no work in the literature, even for classical FSMs, which investigates this. I recommend an in-depth investigation be carried out into whether inference techniques which perform well for randomly generated systems perform similarly well when inferring models for real-world programs.

Chapter 8 was very focussed on the use of quantitative metrics to evaluate models, however the utility of a model is often also based on how easy it is for a person to *understand.* Consequently, I recommend that a human study be carried out to investigate this. Possible research topics could involve whether EFSMs are easier to understand than their classical counterparts, and how the inference of guards and output and update functions affect understandability.

10.4 Concluding Remarks

Models which accurately reflect the behaviour of software systems are extremely valuable for a number of software engineering tasks. Despite their value, they are often neglected during the development process. It is therefore desirable to be able to reverse engineer them from existing systems. There are various tools and techniques which support this, however, most are only able to infer classical FSM models with atomic transitions. These models show the control flow of systems, but do not give a picture of how the systems work with data. For this, we need an EFSM model. The field of EFSM modelling is much less well-established than that of classical FSMs. Consequently, the first contribution of this thesis is the definition (and formalisation in Isabelle/HOL) of a new EFSM definition which brings together desirable characteristics from a number of existing EFSM definitions.

The majority of modern inference techniques involve some kind of state merging. As part of this, it is also necessary to merge transitions which account for each other's behaviour. The second contribution of this thesis is a relation called *direct subsumption* which is used to identify when it is safe to merge EFSM transitions. The third contribution involved using this relation as the basis for a state merging technique to infer EFSM models, complete with output and update functions, from black-box execution traces. As part of this, the fourth contribution of this thesis was a generally applicable approach to infer the output and update functions for transitions. The final contribution of the thesis was a framework to support the verification of models once they have been inferred. The EFSM formalisation in Isabelle was extended to support the specification and proof of properties phrased using LTL, and a parallel representation for the model checker SAL was proposed such that counterexamples to untrue properties can easily be found. Overall, the work presented in this thesis makes up an effective set of tools for the inference and verification of EFSM models.

References

[1] SAP system trace. `https://help.sap.com/doc/saphelp_nw70/7.0.31/en-US/1f/83114c4bc511d189750000e8322d00/content.htm`, 2016 (Accessed 16/09/2020).

[2] Percepio tracealyzer – Stop guessing! `https://percepio.com/tracealyzer`, 2020 (Accessed 16/09/2020).

[3] Fides Aarts. *Tomte : Bridging the gap between active learning and real-world systems.* PhD thesis, Radboud University Nijmegen, 2014.

[4] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM 2012: Formal Methods*, pages 10–27, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[5] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 673–686, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Jean Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[7] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 74–100. Springer International Publishing, 2018.

[8] Enrique Alba, Gabriel Luque, and Sergio Nesmachnow. Parallel metaheuristics: Recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[9] Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. Constructing call graphs of scala programs. In *ECOOP 2014 – Object-Oriented Programming*, pages 54–79, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[10] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[11] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[12] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 324–341, New York, NY, USA, 1996. Association for Computing Machinery.

[13] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[14] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach.* Springer Berlin Heidelberg, 2006.

[15] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In *Fundamental Approaches to Software Engineering*, volume 4961 LNCS, pages 317–331. Springer Berlin Heidelberg, 2008.

[16] Alan W. Biermann and Jerome A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, C-21(6):592–597, 1972.

[17] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving*, pages 131–146. Springer Berlin Heidelberg, 2010.

[18] Stefan Bleuler, Martin Brack, Lothar Thiele, and Eckart Zitzler. Multiobjective genetic programming: Reducing bloat using SPEA2. In *Proceedings of the IEEE Conference on Evolutionary Computation*, volume 1, pages 536–543, 2001.

[19] D. A. Bochvar. On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. *History and Philosophy of Logic*, 2(1-2):87–112, 1981.

[20] Egon Börger and Robert Stärk. *Abstract state machines.* Springer Berlin Heidelberg, 2003.

[21] Matko Botinčan and Domagoj Babić. Sigma*: Symbolic learning of input-output specifications. *SIGPLAN Not.*, 48(1):443–456, 2013.

[22] Markus F. Brameier and Wolfgang Banzhaf. *Linear genetic programming.* Genetic and Evolutionary Computation. Springer US, 2007.

[23] L.C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *International Conference on Software Engineering*, pages 86–95. IEEE Comput. Soc., 2004.

[24] Lukas Bulwahn. The new Quickcheck for Isabelle. In *Certified Programs and Proofs*, pages 92–108. Springer, Berlin, Heidelberg, 2012.

[25] Donald S. Burke, Kenneth A. De Jong, John J. Grefenstette, Connie Loggia Ramsey, and Annie S. Wu. Putting more genetics into genetic algorithms. *Evolutionary Computation*, 6(4):387–410, 1998.

[26] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[27] Sofia Cassel, Falk Howar, and Bengt Jonsson. Ralib: a learnlib extension for inferring EFSMs. *DIFTS 2015*, 2015.

[28] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Learning extended finite state machines. In *Software Engineering and Formal Methods*, pages 250–264. Springer, Cham, 2014.

[29] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. *Formal Aspects of Computing*, 28(2):233–263, 2016.

[30] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. *Extending Automata Learning to Extended Finite State Machines*, pages 149–177. Springer International Publishing, Cham, 2018.

[31] Sofia Cassel, Bengt Jonsson, Falk Howar, and Bernhard Steffen. A succinct canonical register automaton model for data domains with binary relations. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis*, pages 57–71, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[32] Kwang Ting Cheng and Avinash S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th International Design Automation Conference*, pages 86–91. ACM Press, 1993.

[33] Raymond Chiong, Thomas Weise, and Zbigniew Michalewicz. *Variants of evolutionary algorithms for real-world applications.* Springer Berlin Heidelberg, 2012.

[34] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of the 20th USENIX Conference on Security*, page 10. USENIX Association, 2011.

[35] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An opensource tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification*. Springer, 2002.

[36] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, pages 52–71. Springer-Verlag, 1982.

[37] Edmund M. Clarke, E. Allen Emerson, and Aravinda P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 117–126. ACM Press, 1983.

[38] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30. Springer Berlin Heidelberg, 2012.

[39] Hila Cohen and Shahar Maoz. The confidence in our k-tails. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Wngineering*, pages 605–610. ACM Press, 2014.

[40] Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel Van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[41] Carlos Diego N. Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simao. Learning to reuse: Adaptive model learning for evolving systems. In *Integrated Formal Methods*, pages 138–156. Springer International Publishing, 2019.

[42] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 273–289, Cham, 2015. Springer International Publishing.

[43] Edwin D. de Jong, Richard A. Watson, and Jordan B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 11–18. Morgan Kaufmann Publishers Inc., 2001.

[44] Colin de la Higuera. *Grammatical inference*. Cambridge University Press, 2010.

[45] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[46] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer Aided Verification*, pages 496–500. Springer Berlin Heidelberg, 2004.

[47] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Efficient formal verification for the Linux kernel. In *Software Engineering and Formal Methods*, pages 315–332. Springer International Publishing, 2019.

[48] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, pages 77–101. Springer Berlin Heidelberg, 1995.

[49] John Derrick, Siobhán North, and Anthony J. H. Simons. Z2SAL: A translation-based model checker for Z. *Formal Aspects of Computing*, 23(1):43–71, 2011.

[50] Benjamin Doerr, Carola Doerr, and Franziska Ebel. From black-box complexity to designing new genetic algorithms. *Theoretical Computer Science*, 567(C):87–104, 2015.

[51] Benjamin Doerr, Huu Phuoc Le, Régis Makhmara, and Ta Duy Nguyen. Fast genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, page 777–784. Association for Computing Machinery, 2017.

[52] Samuel Drews and Loris D'Antoni. Learning Symbolic Automata. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–189, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[53] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22(1-2):77–115, 2008.

[54] Greg Durrett, Frank Neumann, and Una-May O'Reilly. Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics. In *Proceedings of the 11th Workshop Proceedings on Foundations of Genetic Algorithms*, page 69–80. Association for Computing Machinery, 2011.

[55] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computing.* Natural Computing Series. Springer Berlin Heidelberg, 2$^{\text{nd}}$ edition, 2003.

[56] Samuel Eilenberg. *Automata, languages, and machines.* Academic Press, Inc., 1974.

[57] Anikó Ekárt and S. Z. Németh. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, 2001.

[58] Casey Ellis. Bypassing 3rd-degree profiles in LinkedIn by Osanda Malith. `https://www.bugcrowd.com/blog/bypassed-3rd-degree-profiles-linkedin`, 2014 (Accessed 23/09/2019).

[59] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[60] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Computer Aided Verification*, pages 30–48. Springer Cham, 2015.

[61] Patrick C. Fischer. Some universal elements for finite automata. automata studies, edited by C. E. Shannon and J. McCarthy, Annals of Mathematics studies no. 34, lithoprinted, Princeton University Press, Princeton 1956, pp. 117–128. *Journal of Symbolic Logic*, 35(3):480–481, 1970.

[62] John Fitzgerald and Peter Gorm Larsen. *Modelling systems: Practical tools and techniques in software development.* Cambridge University Press, 2009.

[63] Michael Foster, Achim D. Brucker, Ramsay Taylor, Siobhán North, and John Derrick. Incorporating data into EFSM inference. In *Software Engineering and Formal Methods*, pages 257–272. Springer International Publishing, 2019.

[64] Michael Foster, Ramsay Taylor, Achim D. Brucker, and John Derrick. Formalising extended finite state machine transition merging. In *Formal Methods and Software Engineering*, pages 373–387. Springer International Publishing, 2018.

[65] Michael Foster, Ramsay Taylor, Achim D. Brucker, and John Derrick. A formal model of extended finite state machines. *Archive of Formal Proofs*, 2020 (Accessed 19/09/2020). `http://isa-afp.org/entries/Extended_Finite_State_Machines.html`, Formal proof development.

[66] Michael Foster, Ramsay Taylor, Achim D. Brucker, and John Derrick. Inference of extended finite state machines. *Archive of Formal Proofs*, 2020 (Accessed 19/09/2020). `http://isa-afp.org/entries/Extended_Finite_State_Machine_Inference.html`, Formal proof development.

[67] Michael Foster, Neil Walkinshaw, and John Derrick. Using genetic programming to infer output and update functions for efsm transitions. *DRAFT.*

[68] Gordon Fraser and Neil Walkinshaw. Behaviourally adequate software testing. In *International Conference on Software Testing, Verification and Validation*, pages 300–309. IEEE, 2012.

[69] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[70] Bharat Garhewal, Frits Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. Grey-box learning of register automata. In Brijesh Dongol and Elena Troubitsyna, editors, *Integrated Formal Methods*, pages 22–40, Cham, 2020. Springer International Publishing.

[71] Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In Antoine Miné and David Schmidt, editors, *Static Analysis*, pages 248–264, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[72] Georgios Giantamidis and Stavros Tripakis. Learning Moore machines from input-output traces. In *FM 2016: Formal Methods*, pages 291–309, Cham, 2016. Springer International Publishing.

[73] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[74] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[75] Brian W. Goldman and William F. Punch. Parameter-less population pyramid. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, page 785–792. Association for Computing Machinery, 2014.

[76] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. Springer Berlin Heidelberg, 2002.

[77] Florian Haftmann and Lukas Bulwahn. Code generation from Isabelle/HOL theories. Technical report, TUM, 2013.

[78] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10, 2009.

[79] Thomas Haynes. Collective adaptation: The exchange of coding segments. *Evolutionary Computation*, 6(4):311–338, 1998.

[80] M. Hentschel, R. Hähnle, and R. Bubel. The interactive verification debugger: Effective understanding of interactive proof attempts. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 846–851, 2016.

[81] Martin Hentschel, Richard Bubel, and Reiner Hähnle. The symbolic execution debugger (sed): A platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer*, 21(5):485–513, 2019.

[82] Thomas A. Henzinger. *The Theory of Hybrid Automata*, pages 265–292. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[83] Robert M. Hierons. Generating candidates when testing a deterministic implementation against a non-deterministic finite-state machine. *Computer Journal*, 46(3):307–318, 2003.

[84] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2):1–76, 2009.

[85] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[86] Gerard Holzmann. *The SPIN model checker: Primer and reference manual.* Addison-Wesley Professional, 1st edition, 2011.

[87] Falk Howar, Malte Isberner, Bernhard Steffen, Oliver Bauer, and Bengt Jonsson. Inferring semantic interfaces of data structures. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 554–571, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[88] Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *Verification, Model Checking, and Abstract Interpretation*, pages 251–266, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[89] David Huistra, Jeroen Meijer, and Jaco van de Pol. Adaptive learning for learn-based regression testing. In *Formal Methods for Industrial Critical Systems*, pages 162–177. Springer International Publishing, 2018.

[90] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 720–725, 2016.

[91] Abdul Salam Kalaji, Robert M. Hierons, and Stephen Swift. Generating feasible transition paths for testing from an extended finite state machine (EFSM) with the counter problem. In *International Conference on Software Testing, Verification, and Validation Workshops*, pages 232–235, 2010.

[92] Abdul Salam Kalaji, Robert M. Hierons, and Stephen Swift. An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. *Information and Software Technology*, 53(12):1297–1318, 2011.

[93] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[94] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[95] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, pages 1137–1143. Morgan Kaufmann Publishers Inc., 1995.

[96] John R. Koza. *Genetic programming: On the programming of computers by means of natural selection*. MIT Press, 1992.

[97] Saul A. Kripke. Semantical analysis of modal logic i normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9(5-6):67–96, 1963.

[98] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Grammatical Inference*, pages 1–12. Springer Berlin Heidelberg, 1998.

[99] Joel Lehman, Jeff Clune, Dusan Misevic, Christoph Adami, Lee Altenberg, Julie Beaulieu, Peter J. Bentley, Samuel Bernard, Guillaume Beslon, David M. Bryson, Nick Cheney, Patryk Chrabaszcz, Antoine Cully, Stephane Doncieux, Fred C. Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Fŕenoy, Christian Gagńe, Leni Le Goff, Laura M. Grabowski, Babak Hodjat, Frank Hutter, Laurent Keller, Carole Knibbe, Peter Krcah, Richard E. Lenski, Hod Lipson, Robert MacCurdy, Carlos Maestre, Risto Miikkulainen, Sara Mitri, David E. Moriarty, Jean-Baptiste Mouret, Anh Nguyen, Charles Ofria, Marc Parizeau, David Parsons, Robert T. Pennock, William F. Punch, Thomas S. Ray, Marc Schoenauer, Eric Schulte, Karl Sims, Kenneth O. Stanley, François Taddei, Danesh Tarapore, Simon Thibault, Richard A. Watson, Westley Weimer, and Jason Yosinski. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *Artificial Life*, 26(2):274–306, 2020.

[100] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[101] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In *Formal Techniques for Networked and Distributed Systems*, pages 436–450. Springer Berlin Heidelberg, 2006.

[102] Wenbin Li, Franck Le Gall, and Naum Spaseski. A survey on model-based testing tools for test case generation. In *Tools and Methods of Program Analysis*, pages 77–89. Springer International Publishing, 2018.

[103] Lennart Ljung. *System Identification*, pages 163–173. Birkhäuser Boston, Boston, MA, 1998.

[104] Andreas Lochbihler. Formalising FinFuns – Generating code for functions as data from Isabelle/HOL. In *Theorem Proving in Higher Order Logics*, pages 310–326. Springer Berlin Heidelberg, 2009.

[105] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Inferring state-based behavior models. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, page 25. ACM Press, 2006.

[106] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 13th international conference on Software engineering*, page 501. ACM Press, 2008.

[107] Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 829–836. Morgan Kaufmann Publishers Inc., 2002.

[108] Jeff Magee and Jeff Kramer. *State models and Java programs.* Wiley Hoboken, 2$^{\text{nd}}$ edition, 2006.

[109] Felix Mannhardt. *Multi-perspective process mining.* PhD thesis, TU Eindhoven, 2018.

[110] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. *DAIMI Report Series*, 6(78), 1977.

[111] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[112] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[113] Ramy Medhat, Sethu Ramesh, Borzoo Bonakdarpour, and Sebastian Fischmeister. A framework for mining hybrid automata from input/output traces. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 177–186, 2015.

[114] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1996.

[115] Julian F. Miller, editor. *Cartesian genetic programming.* Natural Computing Series. Springer Berlin Heidelberg, 2011.

[116] Edward F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies. (AM-34)*, pages 129–154. Princeton University Press, 1956.

[117] Alberto Moraglio, Krzysztof Krawiec, and Colin G. Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature - PPSN XII*, pages 21–31. Springer Berlin Heidelberg, 2012.

[118] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541, 1958.

[119] Oliver Niese. *An integrated approach to testing complex systems.* PhD thesis, TU Dortmund, 2003.

[120] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL — A proof assistant for higher-order logic.* Springer-Verlag, Berlin, Heidelberg, 2002.

[121] José Oncina and Pedro García. *Identifying regular languages in polynomial time*, pages 99–108.

[122] Fabrizio Pastore, Daniela Micucci, and Leonardo Mariani. Timed k-tail: Automatic inference of timed automata. In *2017 IEEE International Conference on Software Testing, Verification and Validation*, pages 401–411, 2017.

[123] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[124] Aleš Procházka, Magdaléna Kolinová, and Jaroslav Stříbrský. Signal segmentation using time-scale signal analysis. In *9th European Signal Processing Conference*, pages 1–4, 1998.

[125] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. Learnlib: a framework for extrapolating behavioral models. *International Journal on Software Tools for Technology Transfer*, 11(5):393, 2009.

[126] Lucio R. Ribeiro and Neusa Maria F. Oliveira. UAV autopilot controllers test platform using Matlab/Simulink and X-Plane. In *2010 IEEE Frontiers in Education Conference (FIE)*, pages S2H–1–S2H–6. IEEE, 2010.

[127] Ken A. Robinson. Introduction to the B method. In *Program Development by Refinement*, pages 3–37. Springer London, 1999.

[128] Grigore Roşu. Finite-trace linear temporal logic: Coinductive completeness. In *Runtime Verification*, pages 333–350. Springer International Publishing, 2016.

[129] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):216–226, 1979.

[130] Dominic Scheurer, Reiner Hähnle, and Richard Bubel. A general lattice model for merging symbolic execution branches. In *Formal Methods and Software Engineering*, pages 57–73. Springer International Publishing, 2016.

[131] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning parameterized state machine model for integration testing. In *31st Annual International Computer Software and Applications Conference - Vol. 2 - (COMPSAC 2007)*, volume 2, pages 755–760. IEEE, 2007.

[132] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 215–223. MIT Press, 1996.

[133] William M. Spears and Vic Anand. A study of crossover operators in genetic programming. In Z. W. Ras and M. Zemankova, editors, *Methodologies for Intelligent Systems*, pages 409–418. Springer Berlin Heidelberg, 1991.

[134] J. M. Spivey. *The Z notation: A reference manual.* Prentice-Hall, Inc., 1989.

[135] Amitabh Srivastava. Unreachable procedures in object-oriented programming. *ACM Lett. Program. Lang. Syst.*, 1(4):355–364, 1992.

[136] Frank Strobl and Alexander Wisspeintner. Specifcation of an elevator control system. Technical report, TUM, 1999 (Accessed 15/05/20). `https://wwwbroy.in.tum.de/publ/papers/elevator.pdf`.

[137] Ramsay Taylor, Kirill Bogdanov, and John Derrick. Automatic inference of Erlang module behaviour. In *Integrated Formal Methods*, pages 253–267. Springer Berlin Heidelberg, 2013.

[138] Ramsay Taylor, Michael Foster, and Siobhán North. Automatically verifying or refuting trace properties of extended finite state machines. *DRAFT*.

[139] Ramsay Taylor, Mathew Hall, Kirill Bogdanov, and John Derrick. Using behaviour inference to optimise regression test sets. In *Testing Software and Systems*, pages 184–199. Springer Berlin Heidelberg, 2012.

[140] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[141] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[142] Frits Vaandrager and Abhisek Midya. A Myhill-Nerode theorem for register automata and symbolic trace languages. In *Theoretical Aspects of Computing*, pages 43–63. Springer International Publishing, 2020.

[143] Alfonso Valdes and Keith Skinner. Adaptive, model-based monitoring for cyber attack detection. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, page 80–92, Berlin, Heidelberg, 2000. Springer-Verlag.

[144] Wil van der Aalst, Arya Adriansyah, Ana Karla Alves de Medeiros, Franco Arcieri, Thomas Baier, Tobias Blickle, Jagadeesh Chandra Bose, Peter van den Brand, Ronald Brandtjen, Joos Buijs, Andrea Burattin, Josep Carmona, Malu Castellanos, Jan Claes,

Jonathan Cook, Nicola Costantini, Francisco Curbera, Ernesto Damiani, Massimiliano de Leoni, Pavlos Delias, Boudewijn F. van Dongen, Marlon Dumas, Schahram Dustdar, Dirk Fahland, Diogo R. Ferreira, Walid Gaaloul, Frank van Geffen, Sukriti Goel, Christian Günther, Antonella Guzzo, Paul Harmon, Arthur ter Hofstede, John Hoogland, Jon Espen Ingvaldsen, Koki Kato, Rudolf Kuhn, Akhil Kumar, Marcello La Rosa, Fabrizio Maggi, Donato Malerba, Ronny S. Mans, Alberto Manuel, Martin McCreesh, Paola Mello, Jan Mendling, Marco Montali, Hamid R. Motahari-Nezhad, Michael zur Muehlen, Jorge Munoz-Gama, Luigi Pontieri, Joel Ribeiro, Anne Rozinat, Hugo Seguel Pérez, Ricardo Seguel Pérez, Marcos Sepúlveda, Jim Sinur, Pnina Soffer, Minseok Song, Alessandro Sperduti, Giovanni Stilo, Casper Stoel, Keith Swenson, Maurizio Talamo, Wei Tan, Chris Turner, Jan Vanthienen, George Varvaressos, Eric Verbeek, Marc Verdonk, Roberto Vigo, Jianmin Wang, Barbara Weber, Matthias Weidlich, Ton Weijters, Lijie Wen, Michael Westergaard, and Moe Wynn. Process mining manifesto. In Florian Daniel, Kamel Barkaoui, and Schahram Dustdar, editors, *Business Process Management Workshops*, pages 169–194. Springer Berlin Heidelberg, 2012.

[145] R. J. van Glabbeek. The linear time - branching time spectrum. In *CONCUR '90 Theories of Concurrency: Unification and Extension*, pages 278–297. Springer Berlin Heidelberg, 1990.

[146] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic finite state transducers: Algorithms and applications. *SIGPLAN Not.*, 47(1):137–150, 2012.

[147] Vivek Vishal, Mehmet Kovacioglu, Rachid Kherazi, and Mohammad Reza Mousavi. Integrating model-based and constraint-based testing using specexplorer. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, pages 219–224. IEEE, 2012.

[148] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257. IEEE, 2008.

[149] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *14th Working Conference on Reverse Engineering*, pages 209–218. IEEE, 2007.

[150] Neil Walkinshaw and Mathew Hall. Inferring computational state machine models from program executions. In *2016 IEEE International Conference on Software Maintenance and Evolution*, pages 122–132. IEEE, 2016.

[151] Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. STAMINA: A competition to encourage the development and assessment of software model inference techniques. *Empirical Software Engineering*, 18(4):791–824, 2013.

[152] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

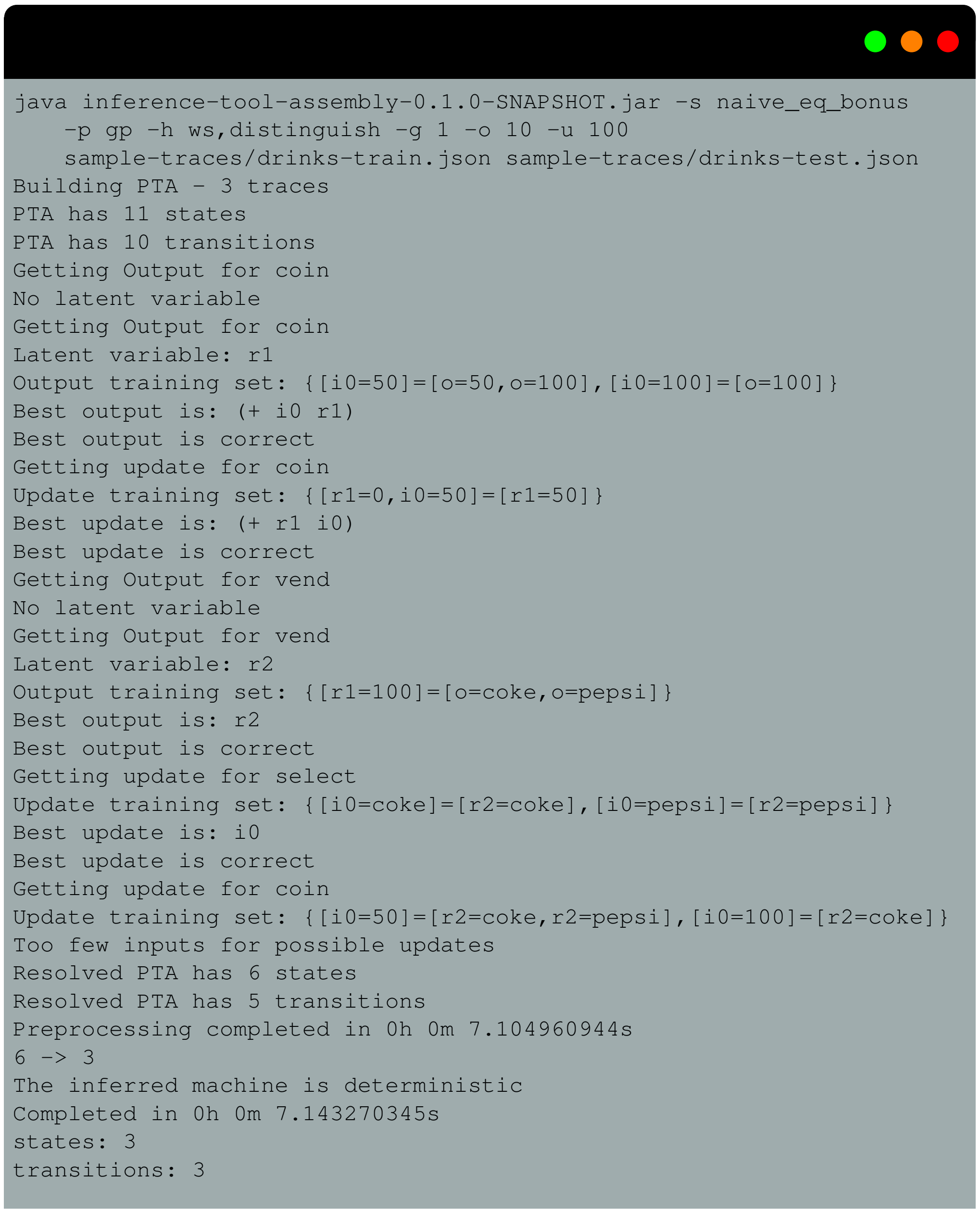
[153] Elaine J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems*, 5(4):641–655, 1983.

Appendix A
Tool Use

Figure A.1 is an example of how the inference tool is called. After compiling the code with `sbt`, the resulting `jar` file can then be called with the `java -jar` command, passing in the relevant options. Here, the `-s naive_eq_bonus` option represents the scoring function used to determine whether states are compatible for merging. The `naive_eq_bonus` scoring function is as described in Subsection 6.2.2. The `-p gp` argument indicates that the GP preprocessor, as described in Chapter 7 is to be used. The `-h ws,distinguish` argument indicates that the heuristics to be used at merge time are *weak subsumption* as described in Subsection 6.4.3 and *distinguishing guards* as described in Section 7.6. The `-g`, `-o`, and `-u` options allow the user to supply random seeds for the *distinguishing guards* heuristic, output GP, and update GP, respectively. Finally, the two arguments `sample-traces/drinks30-train.json` and `sample-traces/drinks30-test.json` are the files containing the training and test logs. The model will be inferred using the traces in `sample-traces/drinks-train.json` and then evaluated according to the traces in `sample-traces/drinks-test.json`. These traces are all in JSON format as exemplified below.

```
[
  [
      {"label":"select","inputs":["coke"],"outputs":[]},
      {"label":"coin","inputs":[50],"outputs":[50]},
      {"label":"coin","inputs":[50],"outputs":[100]},
      {"label":"vend","inputs":[],"outputs":["coke"]}
  ],
  [
      {"label":"select","inputs":["coke"],"outputs":[]},
      {"label":"coin","inputs":[100],"outputs":[100]},
      {"label":"vend","inputs":[],"outputs":["coke"]}
  ],
  [
      {"label":"select","inputs":["pepsi"],"outputs":[]},
      {"label":"coin","inputs":[50],"outputs":[50]},
      {"label":"coin","inputs":[50],"outputs":[100]},
      {"label":"vend","inputs":[],"outputs":["pepsi"]}
  ]
]
```

As it runs, the inference tool prints output to both the terminal window and to a separate log file. It first outputs the fact that it is building a PTA from the training traces. Once the PTA is built, the tool outputs the number of states and transitions it has. The next stage is to run the preprocessor, in this case GP. For each group of transitions with label `label`, the tool outputs `Getting output for label`, whether or not a latent variable is being used, the training set, the best output function, and whether it is correct. In the case where latent variables are used, the preprocessor must also search for update functions. The printed output here is very similar to that for the output functions. Finally, the preprocessor outputs how many states and transitions the resolved PTA has, and how long it took to run.



```
java inference-tool-assembly-0.1.0-SNAPSHOT.jar -s naive_eq_bonus
    -p gp -h ws,distinguish -g 1 -o 10 -u 100
    sample-traces/drinks-train.json sample-traces/drinks-test.json
Building PTA - 3 traces
PTA has 11 states
PTA has 10 transitions
Getting Output for coin
No latent variable
Getting Output for coin
Latent variable: r1
Output training set: {[i0=50]=[o=50,o=100],[i0=100]=[o=100]}
Best output is: (+ i0 r1)
Best output is correct
Getting update for coin
Update training set: {[r1=0,i0=50]=[r1=50]}
Best update is: (+ r1 i0)
Best update is correct
Getting Output for vend
No latent variable
Getting Output for vend
Latent variable: r2
Output training set: {[r1=100]=[o=coke,o=pepsi]}
Best output is: r2
Best output is correct
Getting update for select
Update training set: {[i0=coke]=[r2=coke],[i0=pepsi]=[r2=pepsi]}
Best update is: i0
Best update is correct
Getting update for coin
Update training set: {[i0=50]=[r2=coke,r2=pepsi],[i0=100]=[r2=coke]}
Too few inputs for possible updates
Resolved PTA has 6 states
Resolved PTA has 5 transitions
Preprocessing completed in 0h 0m 7.104960944s
6 -> 3
The inferred machine is deterministic
Completed in 0h 0m 7.143270345s
states: 3
transitions: 3
```

Figure A.1: Calling the inference tool for the drinks machine.

Having run the preprocessor, state merging can begin. For each iteration, the tool outputs `s -> s'` where `s` is the number of states before the merge took place, and `s'` is the number of states after merging and resolving nondeterminism. When no more merges can take place, the tool outputs `Completed in xh ym zs`, where `x`, `y`, and `z` are the hours, minutes, and seconds respectively. The number of states and transitions in the final model are also printed.

As well as the logfile, the inference tool also produces a DOT [69] representation of the inferred model like the one shown below. This textual representation of the model can be compiled to various graphical formats including png, jpg, and svg. It can also be translated to either Isabelle or SAL by the tool mentioned in Chapter 9. The inference tool also produces DOT files of the PTA before and after preprocessing and after each successful merge, so we have a step-by-step reconstruction of the inference process.

```
digraph EFSM{
  s0[fillcolor="gray", label=<s0>];
  s1[label=<s1>];
  s4[label=<s4>];

  s0->s1[label=<select:1/[r1:=0, r2:=i0>];
  s1->s1[label=<coin:1/o1:=i0 + r1[r1:=r1 + i0>];
  s1->s4[label=<vend:0/o1:=r2>];
}
```

The final outputs of the inference tool are records of how well both the PTA and inferred model performed with respect to the test set. These take the form of JSON files very similar to the input traces, except that the expected and actual outputs are recorded separately, as well as the current state of the model and the ID of the transition taken in response to each action.

Appendix B

Lift Doors EFSM

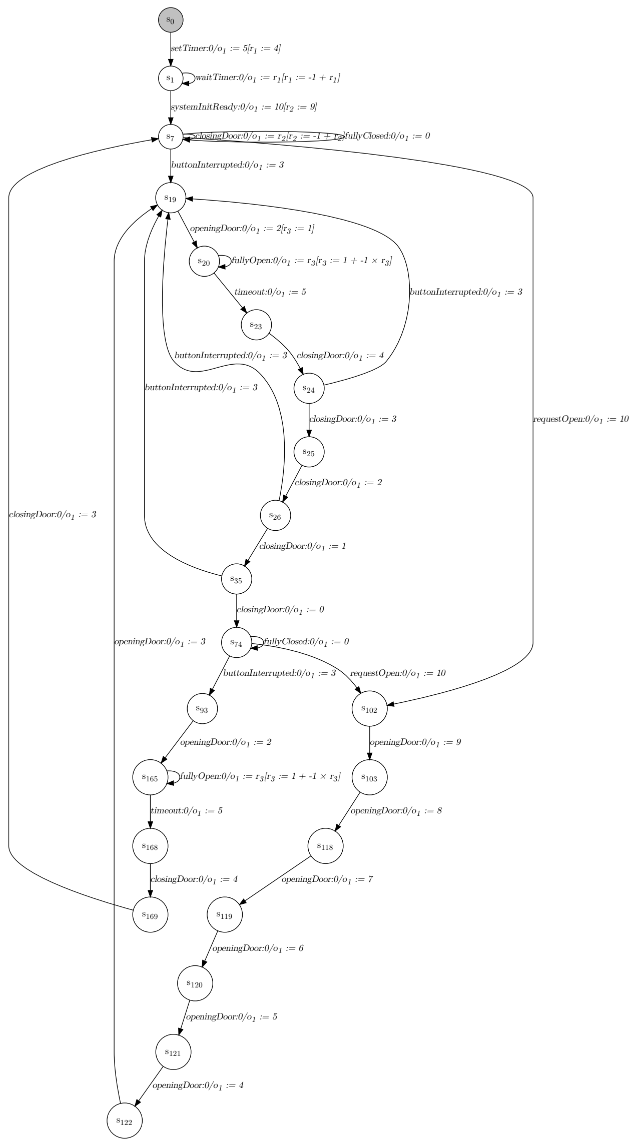


Figure B.1: A model of the LIFTDOORS system inferred by my tool from traces with the *timer* variable obfuscated.