An Overview of Specification Inference

This report formed part of my End of Year report detailing work completed during the first year of a program studying towards a PhD award in Computer Science at the University of Manchester

2011

Giles Reger

School Of Computer Science

Contents

List of Figures

List of Tables

Abstract

This report has been extracted from an end of year report written during the first year of PhD study. For this reason it may seem oddly edited in places. The report gives an extensive overview of the area of specification inference, focussing on grammar inference, specification mining and genetic techniques.

Disclaimer. The majority of this report was written a while ago (Summer 2011) and some of my opinions are sure to have changed in that time. Therefore, any opinions expressed in this report are not necessarily those I currently hold.

Chapter 1

Introduction

The purpose of this report is to present the information I have gathered about the field of **Specification Inference**. In this chapter I attempt to explain *why* the area of specification inference is interesting.

Organisation

As this report was extracted from a different report the organisation may seem odd in places. There are two main chapters in this report - Chapter 3 outlines specification inference techniques based on *grammar inference* and *specification mining*, whereas Chapter 4 presents specification inference techniques utilising genetic algorithms. Chapter 2 is mainly full of definitions, it can be skipped.

1.1 Motivation

For a topic of research to be interesting it must be applicable - solve an existing problem that somebody cares about. Most people working in the area of Grammar Inference[1] use the problem of inferring a model from some data as motivation enough - finding ways to infer more expressive models from less informative data. However, many applications have been successful. People working in Specification Mining [2] usually begin by motivating these techniques by stating that most systems currently developed do not come with full specifications, and that specifications that do exist are often incomplete or out-of-date. This first motivation for specification inference in general is *Program Understanding*. So that a developer can improve, check and communicate the behaviour of a system it is important that they fully understand what the actual behaviour of that system is. Modern systems are often large and complex and automatic techniques for abstracting the actual behaviour of a system can reduce the effort and margin for error in program development. When we construct systems we usually want to make sure they do what they are supposed to do. To check if a program does what it is supposed to do we must have some model or description of what that is. This is another motivation for specification inference techniques - *Checking Program Correctness*. Another use for inferred specifications are in locating errors - if we know that a program exhibits undesired behaviour a model of what it *actually* does can help find the problem. Inferred specifications can either be used to help methods in *testing* or *formal verification*. Sometimes to ensure correctness, but more often efficiency, it is necessary to restrict or control the behaviours of systems. To do this we need a complete formal understanding of the correct behaviours of the system - this is where an inferred specification can be used. Therefore a third motivation for specification inference techniques is that of *Controlling Programs*. This builds on program understanding and checking program correctness

[1] An inductive learning approach - see Chapter 3

[2] A more pragmatic technique for inferring models based on an extraction rather than construction - see Chapter 3

- to control a program we must understand what it should do and if it is doing it. Beyond these three motivations there may be other fields within computer science that could make use of a specification of intended behaviour for some system - for example, in the field of computer security *intrusion detection* systems have recently become more reliant on models of expected and unexpected behaviour. In Section 3.5 I consider further uses for inferred specifications, with references from the literature - these can be taken as further motivation for specification inference techniques. In the following I explore one area of application in a little more detail and discuss whether all the problems have been solved.

The Software Development Process

To consider a potential application of an inferred specification I consider the software development process. Figure 1.1 illustrates how specification inference techniques can be used to improve the software development process. The first use is in *checking if a program is written well*. This is usually carried out using informal and possibly inconsistent programming standards and code checks. However, a programmer has a number of rules in mind when writing code. By inferring a specification from the program which captures these rules, it is possible to check whether they have been applied consistently. Such a specification inference technique must be able to deal with these inconsistencies and only extract rules that *usually* hold rather than *always* hold. The next use is in *replacing legacy systems*. This can be very time consuming and expensive - especially if the system specification is incomplete, which it will almost definitely be if it has been modified to take account of new functionality. A specification inferred from the old system could be used to direct the design of the new system and then to test the new system once it has been constructed. In this case a specification inference technique must be able to infer specifications which are expressive and readable enough to be workable as design documents. Another use is in the ongoing development of a system - particularly when *making small changes*. Small changes to the system made when optimising code or adding small pieces of functionality can have large repercussions on the rest of the code. Currently regression test suites are used to check that these small changes have not broken any properties of the system. An alternative approach would be to infer a specification before and after the change and check for changes in the specification of the system. Finally, specification inference can be used as a *feedback tool*. A client will typically give a software developer a list of requirements and the developer will generate a prototype to present back to the client. An inferred specification can be used as a 'free' form of feedback which the client can use to explore the existing functionality of the system and check against their requirements. For this approach specification inference should focus on specification of observable behaviour, rather than implementation details.

So what is there left to do?

So far I have shown that specification inference is useful and desired. However, people have been working on this problem for half a century - so is there anything left to do? I discuss what has been achieved so far in Chapter 3 but summarise some high level conclusions here

- The specification inference problem is NP-hard (for many intuitive descriptions of the problem)
- There exist approaches which can weaken this hardness
- Grammar Inference techniques for inferring models for regular languages, in the form of finite state machines, are very advanced. Techniques for inferring more expressive models exist but are still underdeveloped
- Grammar Inference techniques have only been used to infer program specifications in a small number of instances. However, where they have been used they have been successful

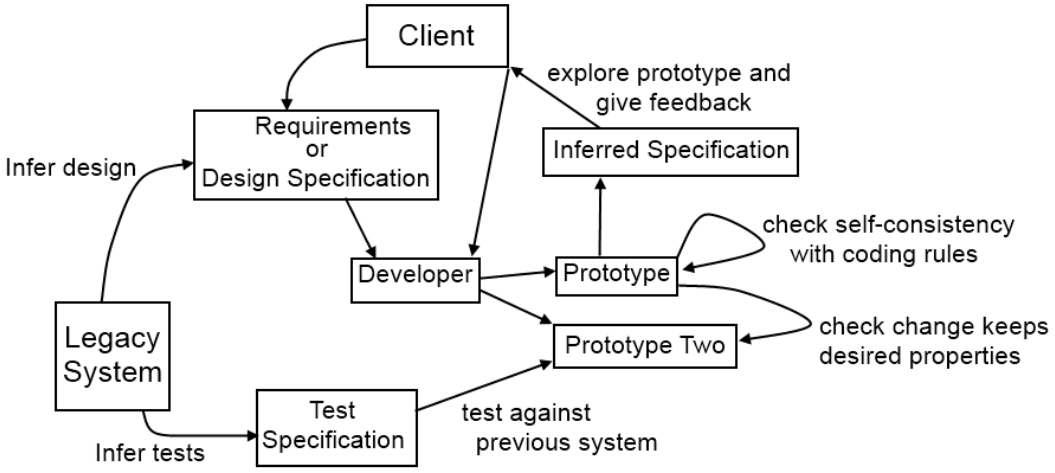

Figure 1.1: Specification Inference and the Software Development Process

- Specification Mining techniques are relatively new and still focus on reasonably limited specifications
- There exist few approaches that infer parametric specifications (trace specifications with free variables) and those that do have certain limitations

So there is a high level problem to be solved, which has been attacked considerably to uncover a number of interesting and approachable subproblems. More importantly, there exist many areas of application which have not been explored - therefore there exist many problems which may be addressed using these techniques.

Chapter 2

Preliminaries

This chapter covers necessary preliminary material, which a reader familiar with these topics may skip over. I begin by discussing formal program specifications (Section 2.1) and then give some preliminary definitions on languages, grammars and automata (Section 2.2). I finish by discussing the related areas of testing (Section 2.3) and instrumentation (Section 2.4).

2.1 Formal Program Specification

This section discusses formal program specifications, attempting to briefly capture some intuition about what a program specification is. At this point I note that I am primarily interested in *trace* specifications.

2.1.1 What is a Formal Program Specification

A formal program specification is a mathematical model that specifies some intended or forbidden behaviour of a program. Here we focus on *functional* behaviour in the form of *allowed states*, but a specification may capture non-function behaviour such as *performance*.

To consider how this behaviour may be modelled let us first consider how we might model a program. At runtime, a program receives *inputs*, generates *outputs* and updates a *memory* (Figure 2.1). We can choose to model a program at two levels of abstraction - either through *observable* behaviour, or *internal* behaviour. We can model the *state* of a program by a 'snapshot' of its memory, letting two states be equivalent if they capture the same memory snapshots - abstractly a memory is a mapping of addresses or program variables to values. We can assume that the memory captures the inputs to the program and that an output will be the result of the program being in some state. A program will, typically, have the expressive power of a Turing Machine and although the number of actual possible states a program can take is bounded by the machine's physical memory the possible states a program can be in is infinite.

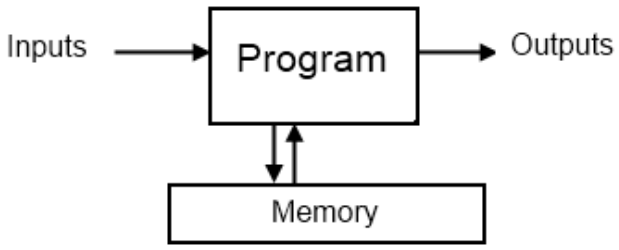

Figure 2.1: Abstractly modelling programs

When considering *observable* behaviour then the program transitions between states based on *inputs*, and states are observed via the outputs they emit. When considering *internal* behaviour the program transitions between states based on some internal state-modifying *operations*, we can additionally assume a set of non-state-modifying operations that can be used to observe internal state. The feasible transitions

are captured statically by the program's source code. Additional information can be added to this model - for example the amount of time it takes to perform certain operations, the times between inputs or communication links.

Let the *behaviour* of a program to be given by the states the program passes through. We can then either specify some behaviour by placing a predicate on the allowed/disallowed states of the program. This predicate can either be on the memory snapshot captured by the state, or, as a state can be described by the transitions leading to that state, on a sequence of transitions. Based on this intuition I define the following high-level distinction between specifications - a specification is either a *state* or *trace* specification.

- *State specifications.* A state specification is a function $P : State \to \mathbb{D}$ that assigns, to each state, a verdict from the verdict domain $\mathbb{D}$. Among other things, these are also known as *state invariants*.
- *Trace specifications.* A trace specification is a function $P : Trace \to \mathbb{D}$ that assigns, to each trace, a verdict from the verdict domain $\mathbb{D}$. Among other things, these are also known as *protocol* specifications, *typestate* properties or *interaction* invariants.

It should be noted that here I talk about *formal* program specifications, there are of course *informal* program specifications, and these are generally more common. An informal program specification might detail the expected behaviour of the program given a certain input or discuss certain sequences of operations that should not occur, but will generally not cover all behaviours.

Finally, we can draw a distinction between *positive* (or *validation*) specifications describing desired behaviour and *negative* (or *violation*) specifications describing *un*desired behaviour. These will not necessarily be the complement of each other for a number of reasons - for example, they may be incomplete (containing some 'don't care' behaviours). Additionally, the specification language of the specification may not be complete under complement, that is the complement of a specification may not be in the specification language.

2.1.2 Specification Languages

Specifications must be written in a *specification language*, which will have a certain *expressive power* limiting the behaviours that can be expressed in that language. For some domains we are also concerned with the completeness, soundness and complexity of related decision procedures for that language. These properties of the specification language may not only effects the number of properties which can be specified in the language but also the usability and related effectiveness of the written specifications. A specification language must have well-defined syntax and semantics - although in some cases, for example graphical languages, this can be very difficult to capture. Based on descriptions in [66] I describe a number of approaches taken to specifying program behaviour.

Model-Based. Where an *abstract model* of the program is constructed that describes program states and state changing operations. Operations are defined *axiomatically* using pre and post conditions- for example, a stack pop function might be specified axiomatically by precondition $|stack| > 0$ and postcondition $|stack| = |stack'| + 1$. Examples of specification languages for model-based specifications are VDM or Z. These specifications are often part of a larger software development process making use of *refinement* or *retrenchment* techniques to progress from an abstract specification of program behaviour to a concrete implementation.

Finite-State-Based. A program is described using a finite number of states and labelled transitions between these states that represent state-changing actions that the program can perform - a common formalism is a Deterministic Finite Automata. This approach can also capture disallowed transitions using a notion of non-accepting states. Finite-state based specifications have been very popular as their related decision procedures are tractable. Dwyer et al. [42] have described a number of common specifications that are useful to consider when reasoning about usable finite-state based specifications. Some specifications that have a finite-state based structure but capture internal data (such as Extended Finite State Machines) represent specifications with infinite states (if the domains of internal data are infinite) but are still referred to as finite-state based. Note that the advantage of finite-state based specifications is that it is possible to consider all paths through the specification and, even though there may be a state-explosion, check that two finite-state specifications capture the same paths.

Process Algebra State-Based. Concurrent programs can be captured elegantly in process algebra such as Communicating Sequential Processes (CSP) and Calculus of Communicating Systems (CCS) which can be presented as labelled-transition systems (LTS). In finite-state based specifications equivalence is usually defined in terms of *trace*-equivalence, the traces which are accepted, labelled-transition systems have a richer sense of conformance in the form of bisumulation which checks whether two processes can simulate each other.

Hybrid. To capture systems with discrete and continuous components a hybrid specification consists of discrete states with continuous behaviour. These are particularly useful for describing systems from engineering and the physical sciences.

Algebraic. A program can be specified completely algebraically using a set of axioms. An algebra consists of a set of symbols denoting values of some type and a set of operations on this set. For example, the specification of a stack pop function might include the axiom `pop(push(Stack,Object)) = Object`. Given appropriate semantics, axioms can be used as rewrite rules in a process called *term-rewriting* to reduce sentences from the algebraic language into some canonical form.

2.1.3 Summary

A formal program specification is a mathematical definition that renders the behaviour of the program as unambiguous as possible. There exist many specification languages with varying levels of usefulness in different domains. In this report I am, unless otherwise stated, concerned only with finite-state based specifications.

2.2 Languages, Grammars and Automata

This report makes extensive reference to formal languages, grammars and automata. To avoid these definitions being spread throughout the report I have gathered these together here. I begin by discussing formal languages in general (Section 2.2.1) and then grammars (Section 2.2.2) and automata (Section 2.2.3). A good introductory text to this material is [70].

2.2.1 Languages and the Chomsky Hierarchy

A word is a (possibly infinite) sequence of symbols taken from some alphabet and a language is a set of words. Let Σ be an alphabet of symbols, a word w over Σ is a sequence of symbols from Σ. The set of all finite words is given by Σ^* and the set of all infinite words is given by Σ^ω. The empty word is given by ϵ

and $|w|$ gives the length of w. A language $\mathcal{L}$ over Σ is a set of finite words $\mathcal{L}_\Sigma \subseteq \Sigma^*$. An ω-language $\mathcal{L}^\omega$ over Σ is a set of infinite words $\mathcal{L}^\omega_\Sigma \subseteq \Sigma^\omega$. The alphabet Σ is omitted where obvious from context. A language can be defined by a model, typically a grammar G or automata A, that can generate (or accept) all words in that language. We denote the language of a model $\mathcal{M}$ as $L(\mathcal{M})$. The expressiveness of different languages is given by the Chomsky Hierarchy [22] and different forms of grammars and automata can be related to this, I given an overview of this hierarchy in Table 2.1. The least expressive form of languages in this hierarchy are *regular languages*, as captured by regular expressions, finite state automata and regular grammars. A proper subset of the regular languages are the *finite languages* consisting of only a finite number of words.

Level	Language	Automata	Logic
0	Recursively-Enumerable	Turing Machine	Typed λ-calculus
1	Context-Sensitive	Linear-bounded non-deterministic Turing Machine	First Order Logic
2	Context-Free	Non-deterministic Pushdown Automata	Combinatory logic
3	Regular	Finite State Automata	Combinatory logic restricted to **I**

Table 2.1: The Chomsky Hierarchy, based partly on [76]

I make the following definitions about languages in general (taken from [123])

- The prefixes of $\mathcal{L}$ are $Pref(\mathcal{L}) = \{w \mid wu \in \mathcal{L}\}$
- The suffixes in $\mathcal{L}$ of a word w are $L_w = \{u \mid wu \in \mathcal{L}\}$
- The short prefixes of $\mathcal{L}$ are $S_p(\mathcal{L}) = \{w \in Pref(\mathcal{L}) \mid \nexists u \in \Sigma^* : L_w = L_u \wedge |u| < |w|\}$
- The kernel of $\mathcal{L}$ is $N(\mathcal{L}) = \{\epsilon\} \cup \{wa \in Pref(\mathcal{L}) \mid w \in S_p(\mathcal{L}) \wedge a \in \Sigma\}$

The prefixes and suffixes of a language should be obvious. The short prefixes of a language are all those prefixes such that there is no shorter word with the same set of suffixes. The kernel of a language is the set of all prefixes that are one symbol longer than a shortest prefix, and the empty string. Later (Section 3.2.1) we will see that the short prefixes of a regular language represent the states of a DFA and that the kernel captures the transitions of a DFA.

2.2.2 Grammars

A grammar is a set of *productions* consisting of *terminals* and *non-terminals*. The terminals of a grammar are drawn from the grammar's alphabet and non-terminals are meta-symbols used to rewrite a *start symbol* into a string of terminals by applying the productions. For example, given the production $aA \to b$ the string aAb could be rewritten as bb. If a grammar can generate the same string in more than one way it is said to be *ambiguous*.

Definition 1 (Grammar)**.** *A grammar is a 4-tuple $\langle N, \Sigma, P, S\rangle$, where N is a finite set of non-terminals, Σ is a finite set of terminals, P is a finite set of productions of the form $p : (\Sigma \cup N)^* N (\Sigma \cup N)^* \to (\Sigma \cup N)^*$ and $S \in N$ is a start symbol.*

We can define different forms of grammars, such as *regular grammars* or *context-sensitive grammars*, by replacing restrictions on the kinds of productions that are allowed. An unrestricted grammar may represent a recursively-enumerable language. The constraints are as follows:

- *Context-Sensitive* grammars may only have productions with a single non-terminal on the lefthand-side. For example, the language $a^n b^n$ can be given by $S \to aSB \mid ab$.
- *Regular* grammars may only have productions with a single non-terminal on the lefthandside and either the empty string, a single terminal or a single terminal followed by a non-terminal on the righthandside. For example, the language ab^*c can be given by $S \to aT, T \to bT \mid c$.

The above demonstrates two common shorthand notations used in writing grammars - firstly, where two productions have the same lefthandside they may be written together by placing a | between the two righthandsides and secondly, the *kleene star* can be used to represent a (possibly empty) sequence of the same symbol(s).

2.2.3 Automata

Automata are machines that have *states* and may make *transitions* between states based on receiving (or generating) symbols from some *alphabet*. Here I discuss different forms of automata with different expressive power.

Finite State Automata

A finite-state automata (FSA) or finite-state machine (FSM) can accept or generate a regular language. A FSA may be *deterministic* if given a symbol a unique transition is available or *non-deterministic* otherwise. A FSA is *complete* if given a symbol it may always make a transition, and *incomplete* otherwise. What should happen for incomplete FSA is not usually defined, however two common approaches are to allow *skip* behaviour where the FSA stays in the same state or *next* behaviour where the FSA goes to some failing state.

Definition 2 (Deterministic Finite Automata (DFA))**.** *A DFA is a 5-tuple $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ where Q is a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \to Q$ a transition function, q_0 an initial state and $F \subseteq Q$ a set of accepting states.*

A word $w \in \Sigma^*$ is accepted by a DFA if and only if $\delta(q_0, w) \in F$ where the transition function δ is lifted to words by $\delta(q, \epsilon) = q$ and $\delta(q, aw) = \delta(\delta(q, a), w)$.

Definition 3 (Nondeterministic Finite Automata (DFA))**.** *A NDFA is a 5-tuple $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ where Q is a finite set of states, Σ a finite alphabet, $\delta \subseteq Q \times \Sigma \times Q$ a transition relation, q_0 an initial state and $F \subseteq Q$ a set of accepting states.*

A word $w \in \Sigma^*$ is accepted by a NDFA if and only if $\delta(q_0, w, S)$ and $S \cap F \neq \emptyset$ where the transition function δ is lifted to words by $\delta(q, \epsilon, \{\})$ and $\delta(q, aw, S)$ iff $\delta(q, a, T) \wedge S = \bigcup_{p \in T} : R \; st. \; \delta(p, w, R)$. There exist algorithms for converting from NDFA to DFA, minimising DFA and telling whether two DFA are equivalent. A DFA with N states can be encoded as a binary string of length $O(NlogN)$.

When modelling finite state machines we may want to take *outputs* into account as well as *inputs*, these are often call *finite state transducers* as they transduce/transform input into output. These do not explicitly capture a notion of acceptance, instead a finite state transducer can be characterised by the words that it outputs.

Definition 4 (Moore Machine)**.** *A moore machine is given by a 6-tuple $\langle Q, \Sigma, \Lambda, q_0, \delta, \gamma\rangle$ where Q is a finite set of states, Σ is a finite input alphabet, Λ is a finite output alphabet, q_0 is an initial state, $\delta : Q \times \Sigma \to Q$ is a transition function and $\gamma : Q \to \Lambda$ is an output function.*

Definition 5 (Mealy Machine)**.** *A mealy machine is given by a 6-tuple* $\langle Q, \Sigma, \Lambda, q_0, \delta, \gamma \rangle$ *where* Q *is a finite set of states,* Σ *is a finite input alphabet,* Λ *is a finite output alphabet,* q_0 *is an initial state,* $\delta : Q \times \Sigma \to Q$ *is a transition function and* $\gamma : Q \times \Lambda \to Q$ *is an output function.*

Lastly I should note that the fact that these machines only have a *finite* number of states is important for reasons discussed above (Section 2.1.2).

Probabilistic Finite State Automata

The following definitions come from [37]. A probabilistic language is a distribution over Σ^*, that is $\mathcal{L}^P : \Sigma^* \to [0, 1]$ and $\sum_{u \in \Sigma^*} \mathcal{L}^P(u) = 1$.

Definition 6 (Probabilistic Finite Automata (PFA))**.** *A PFA is a 5-tuple* $\langle \Sigma, Q, \phi, \iota, \tau \rangle$ *where* Σ *is a finite alphabet,* Q *is a finite set of states,* $\phi : Q \times \Sigma \times Q \to [0, 1]$ *is the transition probability function,* $\iota : Q \to [0, 1]$ *is the initial probability distribution,* $\tau : Q \to [0, 1]$ *is the final probability function.*

A PFA must satisfy $\sum_{q \in Q} \iota(q) =$ and $\forall q \in Q : \tau(q) + \sum_{a \in \Sigma} \sum_{q' \in Q} \phi(q, a, q') = 1$. We can lift ϕ to words u by $\phi(q, \epsilon, q') = \begin{cases} 1 & \text{if } q = q' \\ 0 & \text{otherwise} \end{cases}$ and $\phi(q, ua, q') = \sum_{q'' \in Q} \phi(q, u, q'')\phi(q'', a, q')$. A state q is accessible if $\exists q' : \iota(q') > 0 \wedge \exists u \in \Sigma^* : \phi(q', u, q) > 0$. So that a PFA represents a distribution it must satisfy the constraint that for all accessible q we have that $\exists u \in \Sigma^*, \exists q \in Q : \phi(q, u, q')\tau(q') > 0$. This states that the probability of reaching a final state from any accessible state is strictly positive.

A state q is initial if $\iota(q) > 0$ and final if $\tau(q) > 0$. The probability of generating word u is therefore given by $P(u) = \sum_{q, q' \in Q} \iota(q)\phi(q, u, q')\tau(q')$. A probabilistic language is regular if it can be generated by a PFA. A PFA is deterministic if for every state there exists at most one transition out of that state for each symbol in the alphabet with a probability larger than zero.

Definition 7 (Hidden Markov Model))**.** *A discrete HMM is a 5-tuple* $\langle \Sigma, Q, A, B, \iota \rangle$ *where* Σ *is a finite alphabet,* Q *is a finite set of states,* $A : Q \times Q \to [0, 1]$ *is a probability transition distribution,* $B : Q \times \Sigma \to [0, 1]$ *is a emission probability distribution and* $\iota : Q \to [0, 1]$ *is an initial probability function.*

A path v on a HMM is a word on Q^* the probability of a HMM emits the word u whilst taking the path v is given by

$$P(u, v) = \begin{cases} \iota(v_1) \sum_{i=1}^{l-1} [B(v_i, u_i)A(v_i, v_{i+1})]B(v_l, u_l) & \text{if } l = |u| = |v| > 0 \\ 1 & \text{if } |u| = |v| = 0 \\ 0 & \text{otherwise} \end{cases}$$

The probability of a HMM emitting u by any path is therefore $P(u) = \sum_{v \in Q^*} P(u, v)$. Dupont [37] showed that a HMM is equivalent to a PFA with no final probabilities, that is $\forall q \in Q : \tau(q) = 0$.

More Expressive Automata

Here I detail a few more variants of more expressive automata. As automata become more expressive there are more possible variants and therefore definitions are more likely to change in the literature. I present examples here.

Definition 8 (Pushdown Automata (PDA))**.** *A PDA is a 7-tuple* $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ *where* Q *is a finite set of states,* Σ *is a finite input alphabet,* Γ *is a finite stack alphabet,* $\delta \subseteq (Q \times \Sigma \times \Gamma \times Q \times \Gamma^*)$ *is a transition relation,* q_0 *is a start state,* Z_0 *is a start symbol and* $F \subseteq Q$ *is a set of accepting states.*

A configuration of a PDA is a triple $t \in (Q \times \Sigma^*, \Gamma^*)$. We define a step of a PDA by $\vdash$ such that for all words w, stacks β, input symbols a and stack symbols X if $\delta(q, a, X, p, \alpha)$ holds then $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$. We can either say that a word w is accepted by a PDA if $\exists q \in F : (q_0, w, Z_0) \vdash^* (q, \epsilon, \alpha)$ or if $\exists q : (q_0, w, Z_0) \vdash^* (q, \epsilon, \epsilon)$, where $\vdash^*$ is the transitive closure of $\vdash$. The first definition is termed acceptance by final state and the second acceptance by empty stack, they are equivalent in the sense that if a language can be accepted by a PDA with one there exists a PDA that will accept it with the other. Pushdown automata accept the context-free languages.

A timed word w_T is a finite sequence from $(\Sigma \times \mathbb{R}^{\geq 0})$ such that $\forall (a_i, t_i), (a_j, t_j) \in w : j > i \Rightarrow t_j \geq t_i$. A timed language is a (possibly infinite) set of timed words.

Definition 9 (Timed Automata (TA))**.** *A TA is a 5-tuple $\langle \Sigma, Q, q_0, \delta, F \rangle$ where Σ is a finite input alphabet, Q is a finite set of states, q_0 is an initial state, $\delta \subseteq Q \times \Sigma \times G_\Sigma \times Q$ is a transition relation where $G_\Sigma : \mathbb{R}^{\geq 0^{|\Sigma|}} \to \mathbb{B}$ is a set of guards, and F is a set of accepting states.*

A TA defines a set of guarded words $w_G \in (\Sigma, G_\Sigma)$. The set of timed words accepted by a TA is the set of all w_T such that $w_T \models w_G$ where the judgement $\models$ checks that the times in the timed word satisfy the guards in the guarded word. This can be defined by translating timed words into clocked words where each symbol in Σ is given a clock a time represents a predicate on a clock and then judgement $\models$ holds if the predicates from the clocked word satisfy the predicates defined by the guard.

To allow for parameters on input symbols, guards on transitions and a notion of local state, extended finite state machines were developed. These machines are very close to computer programs and are Turing-complete (can simulate a Turing machine).

Definition 10 (Extended Finite State Machine (EFSM))**.** *An EFSM is a 7-tuple $\langle Q, \Sigma, \Lambda, D, C, U, \delta \rangle$ where Q is a set of states, Σ is a finite set of input symbols, Λ is a finite set of output symbols, D is an n-dimensional linear space of parameter domains $D_1 \times \ldots \times D_n$, C is a set of constraint functions of the form $c : D \to \mathbb{B}$, U is a set of update functions of the form $u : D \to D$ and $\delta : Q \times C \times \Sigma) \to (Q \times U \times \Lambda)$ is a transition function.*

2.3 Testing

Testing is the process of checking if inputs to a program give expected outputs, for a finite number of inputs. There are some areas of testing which are of interest to this work, I summarise these here without going into great detail.

Integration Testing

Integration testing is the process of testing a number of modules or components as a combination. This either tests that the behaviours of the components are preserved after combination, or it tests some larger behaviour related to the combination. The area of *component-based design* relies on a concept of integrating separate components to achieve some larger behaviour. We might also view integration testing as testing at the interface between the different components, where we view the *communication* between components as conforming to some protocol.

Regression Testing

Regression testing is the process of testing a program against a previous version to make sure that the changes have not introduced any undesired behaviour. This will usually be achieved by using a large (growing) test suite to test each version of the program and ensuring that all new versions of the program satisfy this test suite.

Automatically Generating Test Suites

Test suites and test cases can be difficult to generate by hand - time consuming and prone to error. There have been a number of approaches to automatically generate test suites. These include the following:

- *Model Based Testing* [30, 137] A (formal) model of a system is used to generate a test suite, models are typically given as finite state machines or UML diagrams and the test suite can be constructed using a number of techniques such as model checking [50] or constraint solving [35].
- *Symbolic execution* - The main idea behind symbolic execution is to use symbolic instead of concrete values. In test generation symbolic execution is used to compute a representation of all the inputs that execute a certain path in the system. This input constraints must then be solved to create concrete input data. A number of approaches exist that make use of symbolic execution [165, **?**].
- *Random testing* - This idea is very basic, but has been shown to work, the program is simply randomly explored (or randomly with heuristics) to generate test cases [112].
- *Mutating an existing test suite* - Linked to the concept of mutation testing [73] which can measure the adequacy of a test suite by mutating the test target, one proposed approach to test generation it to mutate an existing test suite to explore behaviours similar to those captured by the original test suite.

There are other techniques and if I were to look at how this area could be used in my work I should look at these in more detail - a good starting point would be [11], which looks at combining test case generation with runtime verification.

Conformance Testing

Conformance testing [87] involves checking of a program conforms exactly to a specification. This differs from model-base testing slightly as the goal is slightly different - in conformance testing the focus is ensuring that the model has been adequately covered in tests so that we have a high confidence that the program and specification capture the same behaviour, whereas in model-based testing the goal is to ensure that the program is free of errors. The process is very similar however. There are a number of approaches to conformance testing which take models and generate test suites with certain coverage guarantees . These consist of the W-method [25, 164], partial W-method [52], transition tour [116], distinguishing sequence method [63] and UIO method [138]. I discuss the W-method below.

The Vasilevski/Chow W-Method (description following [167, 17]) This takes an implementation DFA $\mathcal{P}$ and a specification DFA $\mathcal{S}$ with the same alphabet Σ and attempts to construct $Y \subseteq \Sigma^*$ such that $(L(\mathcal{P}) \cap Y = L(\mathcal{S}) \cap Y) \Rightarrow L(\mathcal{P}) = L(\mathcal{S})$. Let C be a prefix-closed subset of Σ^* that visits every state in $\mathcal{P} = \langle Q, q_0, \Sigma, \delta, F \rangle$.

$$\forall q \in Q \backslash \{q_0\}, \exists c \in C : \delta(q_0, c) = q$$

Call C a *state cover set* - this contains all strings needed to visit every state in $\mathcal{P}$. Let W be a subset of Σ^*. States$q_1, q_2 \in Q$ are W-distinguishable iff $(L(\mathcal{P})(q_1) \cap W) \neq (L(\mathcal{P})(q_2) \cap W)$ where

$$L(\mathcal{P})(q) = \{w \in \Sigma^* \mid \exists q_f \in F : q \xrightarrow{w} q_f\}$$

Let W be a set such that any two distinct states of $\mathcal{P}$ are W-distinguishable. Call W a *characterisation set*. If we know, or can guess, that the model has m states and the implementation has n states then let $k = n - m$. Let the test set Y be given by $Y = C(\{\epsilon\} \cup \Sigma \cup \ldots \cup \Sigma^{k+1})W$. The state cover set contains a string to reach any state and the characterisation set can tell any two states apart. In the case where $m = n$ then the test set compares all states.

2.4 Instrumentation

To generate dynamic traces a program must be instrumented. There are different levels of instrumentation

- Manual instrumentation at the source code level - this is no longer practical for large programs and is not generally accepted as a practical form of instrumentation
- Automatic instrumentation at the source code level - this can be achieved through a process called Aspect Orientated Programming (AOP) [43]. An example of AOP is AspectJ [1], an aspect-orientated extension to Java. AspectJ *weaves advice* into a program at user defined *point cuts* defined in terms of *join points*. A *join point* is a well-defined point in the program and might be a method or constructor invocation or execution, the handling of an exception, field assignment or access, etc. A *point cut* represents a pattern of join points that may match a number of points in the program, and is used to select these points and collect context at these points. The user can then define *advice*, standard Java code, to add before, after or around those points. The *advice* is added to the program when it is compiled using the AspectJ weaver, resultant bytecode contains the advice in-place. For a more in-depth discussion of AspectJ and all its uses see [82].
- Automatic instrumentation at the byte code level - practically this has the same implementation as instrumentation at the source code level, however the focus is different. At the source level instrumentation is defined in terms of source code objects. Whereas at the byte code level instrumentation is in terms of byte code objects. A source code level instrumentation tool will generally translate instrumentation details into the byte code level. Instrumentation at this level can either be static or dynamic - an example of a dynamic byte code instrumentation tool is given in [16]
- Automatic instrumentation at the machine level - specialised hardware (or software) at the architectural level can be used to record operations made by the program. This will not generally be program specific and would usually be used for debugging low-level correctness or performance issues.

The method that I use throughout this project is AspectJ, as I am mainly dealing with Java (and Scala) programs. Some runtime monitoring tools automatically create AspectJ aspects containing instrumentation and monitoring code.

2.5 Summary

This chapter covered some preliminary material relevant to the rest of the report. I discussed different forms of program specifications, languages, automata and grammars, first order linear temporal logic, runtime verification (focussing on the runtime verification tool RuleR), testing and instrumentation.

Chapter 3

The Specification Inference Problem

This chapter explores the problem of inferring trace specifications in general, although I focus on inferring these specifications from execution traces. This has mainly been covered by the field of Grammar Inference but there exist alternative approaches that exist within other fields. I begin by giving an overview of the area (Section 3.1) including a map of the different approaches, potential formalisations of the problem, how solutions to the problem may be evaluated and some complexity results. I then describe existing techniques for Grammar Inference (Section 3.2), Dynamic Specification Mining (Section 3.3) and Static Specification Mining (Section 3.4). I then describe some applications of specification inference techniques (Section 3.5). I finish by summarising the key concepts and issues (Section 3.6).

3.1 An Overview of the Area

The problem of deciding whether a program satisfies a specification is one of *deduction* - from the facts that *the program exhibits the behaviours X* and *the correct program behaviours are Y* we can deduce either that the program is correct or incorrect. Here we are interested in a problem of *induction* - from the facts *the program exhibits the behaviours X* and *the program is correct* we wish to induce the specification *the correct program behaviours are Y* - we may additionally include the fact that the program exhibits the behaviours Z and should not. This is often referred to as the problem of *Inductive Inference* and in the general case is a problem of *generalisation* - the resulting specification will necessarily *approximate* the actual specification. However, sometimes it is desirable to infer the *exact* specification and in this case the facts about the program must contain enough information to exactly construct the specification.

In the case where our problem is one of generalisation there will be a number of possible solutions and we will wish to have some means of selecting one of these. To do this we can apply the principle of Occam's razor, to tend towards the simplest such model. To describe simplicity of such models we could use Kolmogorov complexity [169, 79, 21, 153, 154], a measure of the computational resources required to describe an object, or some other concept of minimality such as the number of states in a sate machine.

The question of learnability asked by the field of computational learning theory considers whether there exists a polynomial algorithm that can learn a suitable model to describe a set of inputs under certain conditions. In the following I give a map of the area, describing its different components, formalise the specification inference problem, discuss how solutions can be evaluated and consider the learnability issue.

3.1.1 A Map

Figure 3.1 attempts to capture the different approaches to solving the specification inference problem. In the following I describe the main dimensions given in the map. The techniques referred to on the map can be found later in the chapter (Sections 3.2 to 3.3).

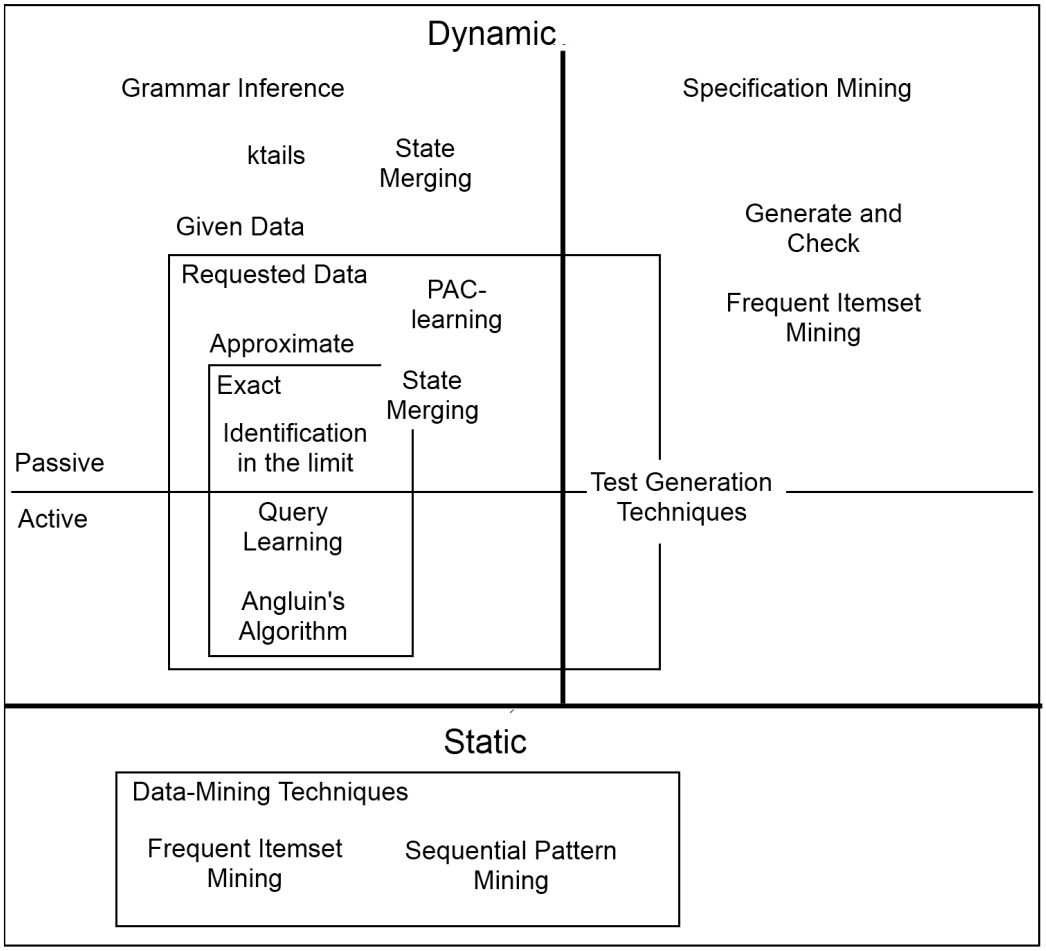

Figure 3.1: A map of specification inference techniques

Grammar Inference or (Dynamic or Static) Specification Mining

Grammar Inference is a machine-learning technique which attempts to construct a model or grammar to describe some language, from examples of words in (or not in) that language. Work in this area dates back to the 50s and 60s, where the focus was on learning natural languages, and since then the field has become very diverse - good overviews of this area can be found in [10, 113, 34, 88]. The term 'Specification Mining' has been attributed to Ammons et al. from a paper written in 2001. Some people [166] have applied the term only to *dynamic* analysis techniques, although some people [150] have applied the name to static techniques. I take the broad definition that Specification Mining is a collection of some techniques that attempt to infer or mine a specification of a program from some artefact of that program. There are generally two main program artefacts used in Specification Mining - *source code* and *execution traces* - the first being a *static* artefact and the second a *dynamic* artefact, we refer to these fields as dynamic and static specification mining respectively. Other artefacts have been used, for example Livshits and Zimmermann mine revision histories to find common error patterns [95]. Specification Mining is focussed on concrete systems whereas Grammar Inference is often applied in other areas, for example language acquisition. In this way, Grammar Inference techniques tend to be more theoretical, concentrating on the complexity of their algorithms and evaluating techniques with hypothetical languages and Specification Mining techniques tend to be more practical, concentrating on their application to real-world problems.

Given or Requested Data

An approach works with *given* data if it has access to a finite static amount of data and has no control over what is included in this data. An approach works with *requested data* if it can request which data it should be provided with, this is possibly infinite and is often viewed as being given incrementally. From a practical viewpoint given data relates to examining a set of log files or arbitrary executions at runtime, whereas given data relates to either manually or automatically ensuring that certain data is observed.

Approximate or Exact

An approach attempts to infer the *exact* specification or an *approximate* specification. It should be noted that distinction can be made for two different reasons - firstly, the approach may attempt to infer an approximate specification in an attempt to reduce the tractability of the problem, or secondly, the approach may only be able to infer an approximate specification as it is not provided with sufficient data to infer an exact specification. This later situation is the case in specification mining techniques, which attempt to generalise from some given data rather than infer the original specification.

Passive or Active

An approach is *passive* if it cannot guide which data is to be provided during the inference process and is *active* if it can. There is an overlap with given or requested data here - an active approach necessarily uses requested data but a passive approach may either work from given data or request all the data it is to use before the inference process. The distinction between the two forms of passive approach have both theoretical and pragmatic repercussions - mainly that some passive approaches can only guarantee exactness if the data has some certain properties.

Summary

The map does not contain every aspect of specification inference, instead it attempts to summarise the key dimensions. I shall add to and improve this map as my understanding of the area increases.

3.1.2 Formalising the Problem

In this section I discuss different approaches to formalising the problem of inferring a specification. In the case of inferring specifications from execution traces this is usually based on a sample of the form $S = S^+ \cup S^-$ where S^+ contain traces that are correct or positive and S^- contains traces which are incorrect or negative. We may not be able to assume that the positive and negative sample sets are disjoint - this seems like an obvious requirement but practically may not be the case. We call the case where a word exists in both the positive and negative sample set *noise* and. Noise may also refer to incorrectly labelled samples due to errors in the source of the samples or irrelevant details in the samples. Unless otherwise stated we assume that there is no noise in the samples.

Grammar Inference

The goal of Grammar Inference is to infer the language a set of words belongs to, possibly with the knowledge that an additional set of words does not belong to the language. The inferred language will be represented by some model, such as a grammar or automata - note that it is usually the language represented by the model, not the original model itself, that is being inferred, as one language can be represented by many different models. It is generally assumed that the alphabet is known previously or can be inferred from samples.

Here I outline a number of different attempts to phrase the grammar inference problem. The original grammar inference problem constrains the inferred model to one accepting the positive sample and not accepting the negative sample, a refinement of this demands that the inferred model is *minimal* in some sense. We always assume that $S^+ \neq \emptyset$ but in the case where $S^- = \emptyset$ a minimal model would be one that accepts everything and therefore this phrasing is only practically applicable to situations where some negative data is given.

Definition 11 (Basic Grammar Inference Problem)**.** *Given a sample $S = S^+ \cup S^-$ over some alphabet Σ find a model M such that $S^+ \subseteq L(M)$ and $S^- \cap L(M) = \emptyset$.*

Definition 12 (Minimal Grammar Inference Problem)**.** *Given a sample $S = S^+ \cup S^-$ over some alphabet Σ find a model M such that $S^+ \subseteq L(M)$ and $S^- \cap L(M) = \emptyset$ and there does not exist $M' < M$ such that $S^+ \subseteq L(M')$ and $S^- \cap L(M') = \emptyset$, for some definition of minimality $<$ on models.*

Alternatively the problem can be phrased in terms of the original hidden language that the sample has been generated from. These seems more appropriate, however also impossible in the general case of an arbitrary sample. Therefore, we introduce the concept of a sample satisfying some property necessary for inference to be decidable. Later we see that we can characterise this property for regular languages, however the most general such property would be that $S = \Sigma^*$ i.e. is complete. We can also add in a concept of approximation.

Definition 13 (Hidden Grammar Inference Problem)**.** *Given a hidden language $\mathcal{L}$ over some known alphabet Σ and sample $S = S^+ \cup S^-$, such that $S^+ \subseteq \mathcal{L}$, $S^- \cap \mathcal{L} = \emptyset$ and S satisfies some necessary property $\mathcal{P}$, find a model M such that $\mathcal{L}(M) = \mathcal{L}$.*

Definition 14 (Approximate Hidden Grammar Inference Problem)**.** *Given a hidden language $\mathcal{L}$ over some known alphabet Σ and sample $S = S^+ \cup S^-$, such that $S^+ \subseteq \mathcal{L}$, $S^- \cap \mathcal{L} = \emptyset$ and S satisfies some necessary property $\mathcal{P}$, find a model M such that $\mathcal{L}(M)$and $\mathcal{L}$ disagree on at most α words, for some finite α.*

The majority of work in this area has been restricted to the inference of *regular languages* - focussing on Deterministic Finite Automata (DFA) as a representation of these languages. The above phrasings can be used to describe regular inference by replacing the words model with DFA, however I rephrase the important problems here.

Definition 15 (Regular Inference Problem)**.** *Given a sample S^+ and S^- over some alphabet Σ find a DFA $\mathcal{A}$ such that $\forall s \in S^+ : \delta(q_0, s) \in F$ and $\forall s \in S^- : \delta(q_0, s) \notin F$.*

Definition 16 (Regular Inference Problem of minimum DFA)**.** *Given a sample S^+ and S^- over some alphabet Σ find a DFA $\mathcal{A}$ such that $\forall s \in S^+ : \delta(q_0, s) \in F$ and $\forall s \in S^- : \delta(q_0, s) \notin F$ such that no other DFA with fewer states than $\mathcal{A}$ also does this.*

Definition 17 (Regular Inference Problem of DFA n states)**.** *Given a sample S^+ and S^- over some alphabet Σ find a DFA $\mathcal{A}$ with n states such that $\forall s \in S^+ : \delta(q_0, s) \in F$ and $\forall s \in S^- : \delta(q_0, s) \notin F$.*

Definition 18 (Regular Inference Problem of Hidden DFA from Structurally Complete Sample)**.** *Given a hidden DFA $\mathcal{A}$ and a sample S^+ and S^- over some alphabet Σ such that $S^+ \subseteq L(\mathcal{A}$, $S^- \cap L(\mathcal{A}) = \emptyset$ and S is structurally complete with respect to $\mathcal{A}$, find a DFA $\mathcal{A}'$ such that $L(\mathcal{A}) = L(\mathcal{A}')$.*

A structurally complete sample is one which covers all transitions and includes a sample accepted by each final state - this is discussed further in Seciton 3.2.1. Note that only definitions 13, 14 and 18 relate to *exact* learning. The others represent some form of approximation through the generalisation of a sample. Approaches that use requested data or are active can be phrased as requesting subsets of the presentation of the language (where $S = \Sigma^*$).

Dynamic Specification Mining

This problem has been described in different ways in the literature. Ammons et al. were one of the first to pose this problem, they begin with a 'unsolvable' problem (Definition 19) concerning Application Programming Interfaces (API) and Abstract Data Types (ADT). The reason this problem is unsolvable is that the training set is unlabelled. They refine this definition by introducing the concept of interaction scenarios that are subtraces of the original interaction trace mentioning at most k objects. Motivated by ideas from PAC-learning (see Section 3.1.4) they make a final definition (Definition 20) of specification mining that infers a probabilistic finite state automata (PFSA)(see Section 2.2.3). Their use of interaction scenarios can be seen as a projection of a parametric trace based on parameters, the probably approximately part of the definition can deal with *noise* in the input traces.

Definition 19 (Ideal Specification Mining)**.** *Let I be the set of all traces of interaction with an API or ADT, and $C \subseteq I$ be the set of all correct such traces. Given an unlabelled training set $T \subseteq I$, find an automaton $\mathcal{A}$ that generates exactly those traces in C.*

Definition 20 (Specification Mining)**.** *Let I_S be the set of all interaction scenarios with an API or ADT that manipulate no more than k data objects. Let M be a target PFSA and P^M be the distribution over I_S that M generates. Given a confidence parameter $\delta > 0$ and an approximation parameter $\epsilon > 0$, find (in time polynomial in $1/\epsilon, 1/\delta, |Q|, |\Sigma|$) with probability at least $1 - \delta$ a PFSA $\widehat{M}$ such that $P^{\widehat{M}}$ is an ϵ-good approximation of P^M.*

Gabel and Su [54] give an alternative definition that also makes use of a projection over the alphabet (Definition 21). This directly relates to finding an instantiation of a template FSA $\mathcal{A}$ with events in a trace τ. Note that they refer to a specification being modelled by a projected trace, this is usual in this approach. This means that some intuitive ideas about 'next' events are lost - that is inferred models can only capture interactions between symbols of the projected alphabet. Additionally, the choice of a projecting (or local) alphabet may not be trivial.

Definition 21 (Generate and Check Specification Mining (SpecMine))**.** *Given a FSA $\mathcal{A}$ over an alphabet Σ_1 and execution trace $\tau \in \Sigma_2^*$ such that $\Sigma_1 \cap \Sigma_2 = \emptyset$. Does there exist a total injective function $\rho : \Sigma_1 \to \Sigma_2$ such that $\tau|_{co\text{-}Dom(\rho)} \models \rho(\mathcal{A})$ where $\rho(\mathcal{A})$ is equal to $\mathcal{A}$ with $\delta_{\rho(\mathcal{A})}(s, \rho(a)) = \delta_{\mathcal{A}}(s, a)$.*

Summary

Here I have presented different formalisations of the specification inference problem. If I were to give a more informal definition that may be useful to help us think about the problem being solved I would say that it is the problem of finding a model which describes the program that the programmer intended to create. This captures the fact that it is important to consider the fact that the samples will contain noise and that perhaps exact learning is not actually that desirable. Another way of approaching the problem is to consider the possible applications of the inferred specification and question what properties it should have - different applications may involve the solving of the problem phrased in different ways.

3.1.3 Evaluating Solutions

To compare and evaluate approaches it is necessary to have some method for measuring how alike two models are - the hidden model and the inferred model. For finite-state based formalism this is can be done through trace equivalence. Traditionally in Grammar Inference inferred models are evaluated by checking how many traces or strings in some test set are accepted by the inferred model. However, Specification Mining techniques have traditionally borrowed the evaluation techniques of *support* and *confidence* from

data mining. Recently, Walkinshaw et al. [167] have proposed that the dimensions of *precision* and *recall* from the field of information retrieval [163] should be used to evaluate inferred specifications. Their framework seems the most useful, and therefore I describe it here.

Precision captures a measure of exactness and recall a measure of completeness, they are used to measure the overlap between what is retrieved and what is relevant. If the hypothesis specification accepts a trace we say it has been 'retrieved' and if the actual specification accepts a trace we say it is 'relevant'. This intuition is biased towards positive behaviour and will not capture the situation where the hypothesis and actual specification both correctly reject a trace. Therefore, positive and negative versions are introduced as $Retrieve^+, Retrieve^-, Relevant^+$ and $Relevant^-$ and updated with respect to the test set as described in Table 3.1.

Hypothesis Specification	Actual Specification	$Retrieve^+$	$Relevant^+$	$Retrieve^-$	$Relevant^-$
accept	accept	add	add		
accept	reject	add			add
reject	accept		add	add	
reject	reject			add	add

Table 3.1: How to update Retrieved and Relevant sets

Precision can be given by

$$precision^+ = \frac{|Retrieved^+ \cap Relevant^+|}{|Retrieved^+|} \qquad precision^- = \frac{|Retrieved^- \cap Relevant^-|}{|Retrieved^-|}$$

Recall can be given by

$$recall^+ = \frac{|Retrieved^+ \cap Relevant^+|}{|Relevant^+|} \qquad recall^- = \frac{|Retrieved^- \cap Relevant^-|}{|Relevant^-|}$$

These can be interpreted in the following way

- High $precision^+$ means that the hypothesis is mostly correct for positive behaviour
- High $precision^-$ means that the hypothesis is mostly correct for negative behaviour
- High $recall^+$ means that the hypothesis is mostly complete for positive behaviour
- High $recall^-$ means that the hypothesis is mostly complete for negative behaviour

The model that accepts every trace will have total positive recall and the model that rejects ever trace will have total negative recall. A model which accepts only the training set will have perfect positive recall and precision on the training set.

Walkinshaw et al. note that if the test set is constructed randomly there may still be aspects of the DFA that are not explored as they have *low observability*, meaning that the probability of generating a string to identify that behaviour by random exploration is very low. If the samples are drawn from the inferred specification or from the actual specification in the same way as the training data there is also a danger that the test data will simply represent the training data. They suggest the use of *conformance testing* (Section 2.3) as a solution to this problem.

I also propose an approach for evaluating inferred specifications based on the concept that a trace can be scored based on how close it is to accepting a trace. The two metrics are

- *Distance into Trace* - A trace is scored based the longest prefix of the trace which is accepted

- *Distance from Trace* - A trace is scored on how many symbols most be ignored for it to be accepted

These metrics should be given proportional to the length of the trace they are measured on. We could also consider a static similarity measures on the inferred models - i.e how far δ and F differ.

Competitions

To encourage new pragmatic techniques for inferring finite state machines there have been number of competitions which I detail here

- *Abbadingo One* occurred in 1997 and was the first competition of its kind. It was developed to encourage the exploration of techniques for inferring large DFA from sparse data. Alphabets are set to 2 symbols. The results of the competition are described in [84] and the winning algorithm was called Evidence Drive State Merging (Section 3.2.2). The competition is described at `http://www-bcl.cs.may.ie/`.

- *Gowachin* started in 1998 and appears to be ongoing but unmaintained. The experimental setup from Abbadingo One was extended so that competitors could add their own problems. The competition is described at `http://www.irisa.fr/Gowachin/`.

- *GECCO* occurred in 2004 and was focussed on inferring reasonably small DFA (10 to 50 states) from noisy data (10%). The winner [62] was a genetic algorithm which evolved the transition matrix and used an optimal state labelling technique to infer the acceptance set. The competition is described at `http://cswww.essex.ac.uk/staff/sml/gecco/NoisyDFA.html`.

- *Omphalos* occurred in 2004 and was focussed on inferring context-free grammars. The winning algorithm does not appear to be available. The competition is described at `http://www.irisa.fr/Omphalos/` and the competition server is still live.

- *Stamina* occurred in 2010 and focussed on inferring DFA. The two parameters to the competition were alphabet size and sparsity of training data. The entries were scored using positive and negative precision and recall as described above. The winner was the DFASAT algorithm, which will be published at a later date. The competition is described at `http://stamina.chefbe.net/`.

These competitions have helped to develop some of the best techniques in current use and hopefully they shall continue to do so.

QUARK

QUARK [97] has been presented as a framework for assessing specification inference tools, but I have not seen this used outside of the research group in which it was developed. Their approach measures accuracy in terms of precision and recall, robustness in terms of accuracy in the presence of errors and scalability in terms of the size of the actual specification. They include an error injection module to measure robustness and allow either FSA or PFSA to be inferred.

3.1.4 Learnability

The field of*computational learning theory* [9] studies the feasibility of learning, where a computation is feasible if it can be carried out in polynomial time. There are three main models for learning [139, 34] - *identification in the limit* [59], *query learning* [8] and *PAC learning* [162].

Identification in the limit [59]

Gold was the first to formalise the problem of grammar inference and has produced a number of results. In his *identification in the limit* model a learner is presented with each example in turn and hypothesises a model for $\mathcal{L}$ after each example. Identification occurs when, after seeing a significantly large number of examples, the learner produces the same, correct (with respect to the seen examples), hypothesis for two consecutive examples. Clark and Lappin [28] outline Gold's results as

1. The class of finite languages is identifiable in the limit on the basis of positive data only
2. A finite class of recursive languages is identifiable in the limit on the basis of positive data only
3. A super-finite[1] class of languages is not identifiable in the limit on the basis of positive evidence only
4. The class of recursive languages is identifiable in the limit on the basis of negative and positive data

Note that given a complete enumeration of all words in Σ^* labelled with whether they are in $\mathcal{L}$ or not we can generate a minimal model for $\mathcal{L}$ - this is called *Identification by enumeration.*

We can consider complexity in the length of the input traces. It was shown by Gold that the problem identifying a minimum DFA from a given finite set of examples is NP-complete [60]. The regular inference problem has been compared to breaking the RSA cryptosystem [77]. However, by assuming additional information (the samples given are structurally complete) it is possible to construct a polynomial algorithm within this learning model that can exactly identify a regular language (see Section 3.2).

Query Learning [8]

To tackle the NP-hardness of the language learnability problem Angluin [8] presented a framework in which a learner can ask questions of a *teacher* or *oracle*. A membership query asks whether a word belongs to the hidden language and an equivalence query asks whether a hypothesised language is equivalent to the hidden language. I describe Angluin's algorithm in Section 3.2.3. It has been shown that the class of regular languages is polynomially identifiable using equivalence and membership queries, but not only using equivalence queries.

Probably Approximately Correct (PAC) learning [162].

In this learning model given a set of samples a learner must select a hypothesis *generalisation* function form a set of possible functions or concepts such that the hypothesis is *probably approximately correct* - that is with high probability the hypothesis will have a low generalisation error, or approximate the actual distribution of the samples. Importantly it has been shown that regular languages are PAC-learnable.

Complexity Results for Dynamic Specification Mining

Gabel and Su [54] give a polynomial reduction from their definition of the specification mining problem (Definition 21) to the well known NP-complete HamPath problem [75].

Definition 22 (HamPath)**.** *Given a directed graph $G = (V, E)$, does G have a path that visits each vertex $v \in V$ exactly once?*

[1] Containing all finite languages and at least one infinite language

The reduction goes as follows - Given a graph G construct a NDFA $\mathcal{A}$ such that $\Sigma = V$, $S = \{s_0, s^*\} \cup \bigcup_{v \in V} \{s_v\}$, $q_0 = s_0$, $\forall (u, v) \in E : \delta(s_u, u) = s_v$, $\forall u \in V : \delta(s_0, \epsilon) = s_u \wedge \delta(s_u, u) = s^*$, $F = \{s^*\}$. Given a trace $\tau = a_0 \dots a_n$ over Σ' such that $\forall 0 < i \neq j \leq n : a_i \neq a_j$ and $|\Sigma'| = |V| = n$, graph G has a Hamiltonian path if and only if $SpecMine(\mathcal{A}, \tau)$. The proof in the $\Rightarrow$ direction assumes that G has a Hamiltonian path p and constructs a mapping $\rho : \Sigma \to \Sigma'$ as $\rho(p_i) = a_i$ where p_i is the ith vertex in the path p. Therefore, τ is accepted by $\mathcal{A}$ using the path $s_0, s_{p_1}, \dots, s_{p_n}$ as $\epsilon.\rho(p_1) \dots \rho(p_n) = a_1 \dots a_n = \tau$. The $\Leftarrow$ direction assumes $SpecMine(\mathcal{A}, \tau)$ and therefore the existence of some mapping $\rho : \Sigma \to \Sigma'$ such that $\mathcal{A}$ has a path p that accepts τ of the form $s_0, s_{\rho^{-1}\tau_1}, \dots, s_{\rho^{-1}\tau_n}$. Note that ρ^{-1} exists as ρ is total injective. As the path visits every state in the automaton once then $\rho^{-1}(\tau_1), \dots, \rho^{-1}(\tau_n)$ is a Hamiltonian path of G.

Summary

Pit [128] gives a thorough overview of the complexity results for Grammar Inference. It has been noted that although the Grammar Inference problem is intractable in the worst case (without access to an oracle) it is reasonable in the average case.

3.1.5 Summary

In this section I have given an overview of the problem of inferring specifications. The problem is, in the general case, NP-hard, however there exist learning models in which polynomial time algorithms exist. In the next few sections I discuss concrete techniques for inferring different forms of specifications.

3.2 Grammar (or Model) Inference Techniques

In this subsection I review existing techniques for grammar inference, concentrating on regular inference. I begin by describing the regular inference search space - the DFAs consistent with given a sample. I then split examine passive and active regular inference techniques separately before considering approaches for inferring context-free grammars and existing tools in this area. Note that passive techniques operate within the complexity constraints of the *identification in the limit* learning model described above and active techniques operate within the complexity constraints of the *query learning* model.

3.2.1 The Regular Inference Search Space

The regular languages have received a lot of attention in the literature as they are the simplest language in the Chomsky hierarchy and therefore may present easier problems and have solutions that could be generalised to more complex classes of languages. Here I discuss some results, mainly from [41], related to how the search for a regular language to describe a sample of strings can be organised.

Nerode's Right Congruence

Nerode's right congruence, also known as the Myhill - Nerode Theorem describes how the set Σ^* can be partitioned in such a way that each partition represents a different state in a state machine. *Nerode's right congruence* $\equiv_{\mathcal{L}}$ is a congruence on the set of words Σ^* for some regular language $\mathcal{L}$, which splits the set Σ^* into a number of equivalence classes, such that no two words in an equivalence class can be differentiated by any suffix. A language is regular if there are a finite number of equivalence classes, as there will then be a finite number of states. Nerode's right congruence $\equiv_{\mathcal{L}}$ is given by the following for $u, v \in \Sigma^*$

$$u \equiv_{\mathcal{L}} v \text{ iff } \forall w \in \Sigma^* : uw \in \mathcal{L} \Leftrightarrow vw \in \mathcal{L} \tag{3.1}$$

This has a number of consequences, the main one being that two states accepting the same sets of words can be merged. Although, in practical examples it is not generally possible to tell if two words cannot be differentiated by any suffix, as this may require an infinite amount of data.

We can use $\equiv_{\mathcal{L}}$ to give a canonical minimal DFA accepting $\mathcal{L}$. Let $\mathbf{u}_{\mathcal{L}}$ represent the equivalence class of u with respect to $\equiv_{\mathcal{L}}$ then define $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ such that Σ is given, $Q = \{\mathbf{u}_{\mathcal{L}} \mid u \in \Sigma^*\}$, $q_0 = \epsilon_{\mathcal{L}}$, $F = \{\mathbf{u}_{\mathcal{L}} \mid u \in \mathcal{L}\}$ and for all $a \in \Sigma$ we have $\delta(\mathbf{u}_{\mathcal{L}}, a) = \mathbf{ua}_{\mathcal{L}}$.

This gives an alternative way of showing that a language is non-regular besides the usual *pumping lemma* for regular languages. The pumping lemma states that if $\mathcal{L}$ is regular there exists a constant c such that for all $w \in \mathcal{L}$ with $|w| \geq c$ we can write $w = xyz$ where $y \neq \epsilon$, $|xy| < c$ and $\forall k \geq 0 : xy^k z \in \mathcal{L}$. The pumping lemma gives a *necessary* condition for regularity, whilst Nerode's right congruence gives a *necessary* and *sufficient* condition.

Capturing a Regular Language

Here I give a brief overview of different properties of *sample sets* taken from regular languages. A *positive sample set* for a language is a set of samples from that language and a *negative sample set* for a language is a set of samples not from that language. A *presentation* of a language is a complete labelling of all words in Σ^* as to whether they are in the language or not, and a *text* is a set of all words in the language. Note that, unless the language is finite, a presentation or text for a language will be infinite. I make the following, more pragmatic, assertions about sample sets.

Definition 23 (Complete, Characteristic and Structurally Complete Samples)**.** *A sample $S = S^+ \cup S^-$ for a DFA $\mathcal{A}$ is a set of words from Σ^* such that $S^+ \in L(A)$ and $S^- \notin L(A)$. A sample is (in increasing size)*

- *(structurally) complete (with respect to $\mathcal{A}$) iff S^+ covers every transition in $\mathcal{A}$ and for every state in F contains a string accepted by that state*
- *characteristic iff given $\mathcal{L} = L(\mathcal{A})$*
  1. $\forall w \in N(\mathcal{L}) : (w \in \mathcal{L} \Rightarrow w \in S^+) \wedge (w \notin \mathcal{L} \Rightarrow \exists u \in \Sigma^* : wu \in S^+)$
  2. $\forall w \in S_p(\mathcal{L}), \forall u \in N(\mathcal{L}) : L_w \neq L_u \Rightarrow \exists v \in \Sigma^* : (wv \in S^+ \wedge uv \in S^-) \vee (uv \in S^+ \wedge wv \in S^+)$

  Where L_w denotes all the suffixes of w in $\mathcal{L}$.
- *complete with (with respect to $\mathcal{A}$) if $S^+ = L(\mathcal{A})$, this impractical for a non-finite language*

Recall the definitions given on page 7. This definition of a (structurally) complete sample has only been used more recently, traditionally a structurally complete sample set only covered every transition in the automata. Recall that the kernel of a language $N(\mathcal{L})$ represents the transitions of the canonical automata accepting $\mathcal{L}$ and the short prefixes $S_P(\mathcal{L})$ represent its states. Therefore a sample which is characteristic is also structurally complete. However, a characteristic set is stronger as the second condition ensures that for every two states that should not be merged there exists a suffix that tells them apart in the sample. Note that the size of a characteristic set is $O(|Q|^2|\Sigma|)$ as the kernel has at most $1 + |Q||\Sigma|$ elements and there are $|Q|$ short prefixes.

Generally we assume $S^+ \cap S^- = \emptyset$, however a case where $S^+ \cap S^- \neq \emptyset$ would indicate *noise* and it is interesting to consider how well learning approaches perform in the presence of such noise.

Derived Automata

We can construct or derive automata from a sample set and then we can derive generalisations of these automata by merging states. I describe these automata and their properties in this section. Given a sample set $S = S^+ \cup S^-$ we can define the maximal canonical automata which accepts the positive samples.

Definition 24 (Maximal Canonical Automata (MCA)). *Given a positive sample S^+, the DFA $MCA(S^+)$ is a maximal canonical automata such that*

- Σ *is the set of symbols in* S^+, $q_0 = \epsilon$, $F = S^+$,
- $Q = \{v_{i,j} \mid i \leq |S^+| \wedge S_i^+ = u \wedge j \leq |u| \wedge v_{i,j} = u_1 \dots u_j\} \cup \{\epsilon\}$,
- $\delta(\epsilon, a) = \{v \mid v = a = a_{i,1} \wedge i \leq M\}$ *and* $\delta(v_{i,j}, a) = \{v_{i,j+1} \mid v_{i,j+1} = v_{i,j}a_{i,j+1} \wedge a = a_{i,j+1} \wedge i \leq M \wedge j \leq |S_i^+| - 1\}$

Where a state $v_{i,j}$ represents having seen the first j symbols of the ith word in S^+ and S_i^+ gives the ith string in S^+.

Two states may represent the same string but are still separate states, meaning that MCA are usually non-deterministic. The transitions take a state representing a string u to all states that represent the string ua. Conceptually a MCA has a non-branching path from the initial state for each string in S^+. Note that $L(MCA(S^+)) = S^+$ and $MCA(S^+)$ is the automata with the largest number of *useful* states with respect to which S^+ is structurally complete, where a state is *useful* if there is a string in the language of that automata that must pass through that state to be accepted. There is a lot of redundancy in a MCA - we can remove this redundancy by merging states with identical prefixes to create a prefix tree acceptor. We can additionally augment this prefix tree acceptor with the information from the negative sample S^+.

Definition 25 (Prefix Tree Acceptor (PTA)). *Given a positive sample S^+, the DFA $\mathcal{PTA}(S^+)$ is an augmented prefix tree acceptor for a sample S^+ if and only if for every string in S^+ $\mathcal{PTA}(S^+)$ contains a path from the initial state to an accepting state, modulo common prefixes.*

Definition 26 (Augmented Prefix Tree Acceptor (APTA)). *Given a sample $S = S^+ \cup S^-$, the DFA $\mathcal{APTA}(S)$ is an augmented prefix tree acceptor for a sample S if and only if for every string in S^+ $\mathcal{APTA}(S)$ contains a path from the initial state to an accepting state, and for every string in S^- $\mathcal{APTA}(S)$ contains a path from the initial state to a non-accepting state, all modulo common prefixes.*

We can capture an intuition about merging states by a partition π which groups states that are to be merged - a partition π is a set of pairwise disjoint nonempty subsets of a set of states Q. The most specific partition without any merges is $\{\{q_1\} \dots \{q_n\}\}$ and the most general partition is $\{Q\}$. We call each element of a partition a *block* and assume a function $B(p, \pi)$ that gives all states in the same block as p in the partition π. Given an automata $\mathcal{A}$ and a partition π we can define a *quotient automata* of $\mathcal{A}$ which represents merging the states of $\mathcal{A}$ with respect to π.

Definition 27 (Quotient Automata (QA)). *Given a DFA $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ and a partition π a quotient automata is a DFA $\mathcal{A}_\pi = (Q_\pi, \Sigma, \delta_\pi : Q_\pi \times \Sigma \to 2^{Q_\pi}, B(q_0, \pi), F_\pi)$ where*

- $Q_\pi = \{B(p, \pi) \mid p \in Q\}$, $F_\pi = \{B(p, \pi) \mid p \in F\}$
- $\delta_p i$ *satisfies* $\forall b, b' \in Q_\pi, \forall a \in \Sigma : b' = \delta_\pi(b, a) \Leftrightarrow q_j = \delta(q_i, a)$ *where* $b = B(q_i, \pi)$ *and* $b' = B(q_j, \pi)$

We can define a prefix tree acceptor $PTA(S^+)$ as the quotient automata $MCA(S^+)_{\pi_{S^+}}$ where π_{S^+} is defined by $B(q, \pi_{S^+}) = B(p, \pi_{S^+})$ iff $Pr(q) = Pr(q')$ where $Pr(s)$ gives the prefixes of state s.

Finally we note that there exists a universal automata that accepts all strings over some alphabet. This is the most general automata and can be maintained from any automata $\mathcal{A}$ by the partition $\pi = \{Q\}$ that places all states in a single block.

Definition 28 (Universal Automata)**.** *Let $\mathcal{U}$ be the universal automata of that accepts all words in Σ^* for some Σ. Given an automata $\mathcal{A}$ over Σ we can define its universal automata as $\mathcal{U}_\mathcal{A} = \langle \{q\}, \Sigma, \delta, q, \{q\} \rangle$ where $\forall q \in Q, \forall a \in \Sigma : \delta(q, a) = q$.*

The Search Space

We can define a partial ordering between partitions such that π_i *refines* π_j if every block of π_j is a union of one or more blocks of π_j. Formally, $\pi_j \preceq \pi_i$ iff $\forall b \in \pi_j \exists b_0 \dots b_k \in \pi_i : b = b_0 \cup \dots \cup b_k$. Let $\ll$ denote the transitive closure of $\preceq$. $\ll$ is as a partial order on the set of all partitions for a given automata $\mathcal{A}$ - this set of partitions describes all automata more general than $\mathcal{A}$. If $\pi_j \ll \pi_i$ then $\mathcal{A}_{\pi_j}$ is more general than $\mathcal{A}_{\pi_i}$, which we write $\mathcal{A}_{\pi_j} \ll \mathcal{A}_{\pi_i}$. This can be linked to language subsumption, as $\mathcal{A}_{\pi_j} \ll \mathcal{A}_{\pi_i}$ if $L(\mathcal{A}_{\pi_j}) \subseteq L(\mathcal{A}_{\pi_i})$.

Given an automata $\mathcal{A}$ we can construct a lattice by partially ordering the quotient automata of $\mathcal{A}$ using $\preceq$, we define this construction as $Lat(\mathcal{A})$. $\mathcal{A}$ is the universal element in $Lat(\mathcal{A})$ and $\mathcal{U}$ is the null element. The depth of an automata $\mathcal{A}_\pi$ in $Lat(\mathcal{A})$ can be given by $|Q| - |\pi|$ - the depth of $\mathcal{A}$ is 0 and the depth of $\mathcal{U}$ is $|Q| - 1$. The lattice is illustrated in Figure 3.2. This shows how refining partitions and merging states correspond to generalising the represented language.

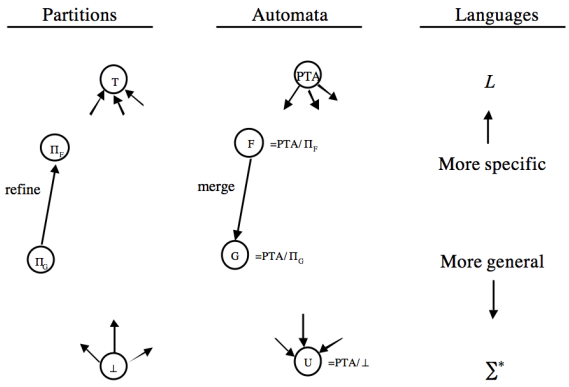

Figure 3.2: Demonstrating the search space - replicated from [67]

Given a sample $S = S^+ \cup S^-$ the search space of regular inference for a minimal model (in the sense of Definition 16) is $Lat(MCA(S^+))$ as any automata not in this lattice will not accept S^+. The goal of regular inference is then to generalise from $MCA(S^+)$ to the automata $\mathcal{A}$ used to generate S. Dupont [41] shows many useful properties of this search space - firstly that $\mathcal{A}$ belongs to $Lat(MCA(S^+))$ if S^+ is structurally complete with respect to $\mathcal{A}$ - this is one of the most important results from the field of regular inference.

The next result states that the canonical automata accepting $L(\mathcal{A})$ belongs to $Lat(PTA(S^+))$ if S^+ is structurally complete with respect to $\mathcal{A}$ - hence, most approaches work with the PTA rather than the MCA as it is more compact. However, Dupont shows that $Lat(PTA(S^+))$ is properly included in $Lat(MCA(S^+))$ and that there are S^+ such that some languages are only represented by non-deterministic finite automata in $Lat(MCA(S^+))$ and therefore that there exist S^+ such that the languages that can be identified from $Lat(PTA(S^+))$ are properly included in those that can be identified from $Lat(MCA(S^+))$. Recall that all automata in $Lat(PTA(S^+))$ are deterministic.

The automata we are searching for is the deepest automata in $Lat(MCA(S^+))$ such that it is consistent with S^-. As the lattice is only partially ordered there may be many such automata - Dupont calls this set of automata the *border set*. We can define this border set as $B(S) = \{\mathcal{A} \mid \mathcal{A} \in Lat(MCA(S^+)) \wedge \nexists \pi : L(\mathcal{A}_\pi) \cap S^- = \emptyset\}$. A similar definition can be made using PTA. It should be noted that attempting to identify $B(S)$ by enumeration under the control of S^- is not tractable.

3.2.2 Passive Techniques for Regular Inference

The section describes different methods for passively learning DFA. These are all state-merging techniques attempting to generalise the automata described by some positive samples. Note that if the set of possible samples is not structurally complete with respect to the original automata the inference process can only approximate this automata.

Trakhtenbrot and Barzdin's algorithm [159] One of the first passive algorithms is due to Trakhtenbrot and Barzdin [159] who wrote a book titled 'Finite automata : behaviour and synthesis', which I have not obtained. Walkinshaw et al. [166] describe their algorithm as constructing the *augmented prefix tree acceptor* for the samples and then merging all states with identical suffix-trees. This requires a *complete* sample set to exactly identify the automata, which is impossible for any non-finite language. Trakhtenbrot and Barzdin show that a complete set up to some given length is sufficient - I have not been able to find information about how they compute this length, but it has been reported that this is generally too large to be of practical use [34].

RPNI - Regular Positive Negative Inference algorithm [121, 120, 83] This refinement of Trakhtenbrot and Barzdin's algorithm requires the sample sets to be characteristic - recall that this means that all transitions are covered (structural completeness) and for every two non-equivalent states there exist strings that can separate them. Dupont [39] describes the RPNI algorithm as conducting an ordered search of the lattice $Lat(PTA(S^+))$. The algorithm is given in Figure 1 and begins by constructing the prefix tree acceptor for the positive sample set S^+ and initialising the partition π to the most general one. It then attempts to merge each pair of states by constructing the merged partition π' and the (determinised) quotient automata $M_{\pi'}$. If $M_{\pi'}$ is consistent with S^- it is kept and as the ith state has been merged with something the loop is exited. The quotient automata is determinised by merging all states that lead to non-determinism - if two states can be reached from another state using the same symbol they are merged.

Data: Sample $S = S^+ \cup S^-$
Result: A DFA compatible with S
1 *Initialization*
2 $M_\pi = PTA(S^+)$
3 $\pi = \pi_0 = \{\{0\}, \ldots, \{|M_\pi.Q| - 1\}\}$
4 *State Merging*
5 **for** $i = 1$ **to** $|M_\pi.Q| - 1$ **do**
6 **for** $j = 0$ **to** $i - 1$ **do**
7 $\pi' = (\pi \backslash \{B(i,\pi), B(j,\pi)\}) \cup \{B(i,\pi) \cup B(j,\pi)\}$
8 $M_{\pi'} = deterministic\_merge(derive(M_\pi, \pi'))$
9 **if** $consistent(M_{\pi'}, S^-)$ **then**
10 $M_\pi = M_{\pi'}$ and $\pi = \pi'$
11 **break** (out of j loop)
12 **return** M_π

Algorithm 1: The RPNI algorithm (reproduced from [123])

If we let $N = \sum_{u \in S^+} |u|$ and $M = \sum_{u \in S^-} |u|$ then the RPNI algorithm takes at most $N - 1$ steps round the i-loop and $i - 1$ steps round the j-loop and therefore in the order of $O(N^2)$ loops round the j-loop. The time complexity of determinising the automata is $O(N)$ and checking the consistency is $O(M)$.

Therefore the overall complexity is $O((N+M).N^2)$. Dupont presents an incremental version of RPNI, called RPNI2 [39], which can receive data incrementally to allow a trade off between space complexity and convergence rate.

The RPNI algorithm works within the learnability model of identification in the limit and the reason it can infer an exact DFA in polynomial time is that it the sample must be characteristic. Parekh and Honavar [123] show how it can be used within the PAC-learning model to probably infer some approximation of the DFA. Their algorithm is shown in Algorithm 2. They begin by showing that the Kolmogrov complexity of a DFA with N states is in the order of $O(logN)$ and use this to show that there exists a characteristic set of *simple* examples such that the length of each string is at most $2N-1$. The algorithm uses two polynomials p and q where p is defined such that a sample S of size $p(N, \frac{1}{\delta})$ contains the characteristic set of *simple examples* with probability greater than $1-\delta$ and $q(i, \frac{1}{\epsilon}, \frac{1}{\delta}) = \frac{1}{\epsilon}[2ln(i+1)+ln(\frac{1}{\delta})]$. On each loop of the algorithm a number of states is guessed (i is doubled each time) and the set of examples extended so that it will contain a characteristic set of *simple* examples with probability $1-\delta$. This is used to run RPNI to generate M, and q is used to construct a test set. If M is consistent with the test set then it ϵ-approximates the unknown DFA. Let us consider the probability that the algorithm halts at some step i and returns a DFA with an error *greater* than ϵ.

$$\begin{aligned}
Pr(\text{M and A are consistent on same } \alpha) &\leq 1-\epsilon \\
Pr(\text{M and A are consistent on all } \alpha \in T) &\leq (1-\epsilon)^{|T|} \\
&\leq (1-\epsilon)^{\frac{1}{\epsilon}[2ln(i+1)+ln(\frac{1}{\delta})]} \\
&\leq e^{-[2ln(i+1)+ln(\frac{1}{\delta})]} \text{ since } 1-x \leq e^{-x} \\
&\leq \frac{\delta}{(i+1)^2}
\end{aligned}$$

Therefore the probability he algorithm halts at some step i and returns a DFA with an error *greater* than ϵ is less than $\sum_{i=1}^{\infty} \frac{\delta}{(i+1)^2}$, which is less than δ. Therefore the algorithm returns a DFA with error at most ϵ with probability greater than $1-\delta$.

Data: approximate bound ϵ, probability bound δ
Result: A DFA M
Initialisation
$i = 1, EX = \emptyset, p(0, 1/\delta) = 0$
Main
repeat
  Draw $p(i, 1/\delta) - p(i-1, 1/\delta)$ examples according to $\mathbf{m}_r$
  Add drawn examples to EX and let $S = \{u \in EX \mid |u| \leq 2i = 1\}$
  $M = RPNI(S)$
  Draw $q(i, 1/\epsilon, 1/\delta)$ examples according to $\mathbf{m}_r$ and call this set T
  if *consistent(M,T)* **then output** M and **halt else** i = i*2
until *eternity*;

Algorithm 2: Algorithm $\mathcal{A}_3$ taken from [123]

Abbadingo One The Abbadingo One competition [84] (Section 3.1.3) spawned some useful refinements of RPNI which remain at the forefront of state-merging regular inference algorithm research. The Blue-Fringe and EDSM algorithms were competitors in the competition and the active QSM algorithm combined the two approaches after the competition. I first describe the key approaches of the Blue-Fringe and EDSM algorithms and a simple combination of these approaches is presented in Algorithm 3.

The Blue-Fringe algorithm contains an approach for selecting potential pairs of states to merge. The intuition is that we can partition the states into two sets - red nodes which are considered mutually

umergable, a 'blue fringe' of states adjacent to red nodes which are candidates for merging and all the other white nodes. Note that red nodes represent members of the set of short prefixes and blue nodes represent members of the kernel.

The (Evidence Drive State Merging) EDSM [27] algorithm contains an approach which can be used to score a particular merge pair. Given two states their score is computed by comparing the state transitions that happen after each state - importantly, if a transition from one state leads to an accepting state but leads to a non-accepting state for the other state then this pair is given a negative score.

```
1 Colour all nodes in the APTA white, mark root node as red
2 while there exists a non-red node do
3     Mark all nodes adjacent to red nodes as blue
4     max = 0; candidate = null;
5     foreach blue node b do
6         foreach red node r adjacent to b that can be merged with b do
7             score = compute_compatibility(b, r)
8             if score > max then candidate = r and max = score
9         if candidate = null then mark b as red else merge b with candidate
```

Algorithm 3: The Blue-Fringe ESM approach (reproduced from[166])

Algeria [20] Carrasco and Oncina use a different means of determining of two states should be merged to infer *stochastic* deterministic regular languages in the form of deterministic stochastic DFA. The algorithm is given in Algorithm 4 - this is very similar to RPNI but note that compatibility is computed very differently. Two states are compatible if they have similar outgoing transition probabilities within a given confidence bound. Two states q_1 and q_2 are said to be different for symbol a if given n_1 and n_2 incoming transitions for q_1 and q_2 respectively and outgoing transitions f_1 and f_2 outgoing transitions for symbol a for q_1 and q_2 respectively, we have $|\frac{f}{n} - \frac{f'}{n'}| > \sqrt{\frac{1}{2} log \frac{2}{\alpha}} \left(\frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n'}} \right)$ (this is the Hoeffding bound [68]). Two states are then compatible if they are not different on any symbol and all states reached by following those transitions are compatible.

```
Data: Sample set S and 1-confidence level α
Result: A stochastic DFA M
1 M = stochastic PTA from S
2 for j = successor(firstnode(M)) to lastnode(M) do
3     for i = firstnode(M) to j do
4         if compatible(i, j, α) then
5             M = merge(A, i, j)
6             M = determinise(A)
7             break
8 return M
```

Algorithm 4: The Algeria algorithm (reproduced from[20])

Biermann's Algorithm or ktails [15] . In 1972 Biermann suggested an approach which merges two states if they have the same k-tails - that is they have identical suffixes of length k. Leucker notes that this algorithm can be reduced to the problem to a constraint satisfaction problem over the natural numbers

and as the basic algorithm is reasonably straightforward I present this variant here. Given a positive sample $S = S^+ \cup S^-$ define the partial function $O : \Sigma^* \rightharpoonup \{1, 0, ?\}$ as

$$O(w) = \begin{cases} 1 & w \in S^+ \\ 0 & w \in S^- \\ ? & \text{otherwise} \end{cases}$$

Let $\texttt{agree} : \Sigma^* \times \Sigma^* \to \mathbb{B}$ be defined such that $\texttt{agree}(u, v) \equiv (O(u) \neq O(v) \neq ?)$. Let S_u represent the state reached by string u, if we let S_u be a variable ranging over the number of states then we can define two sets of constraints

$$\begin{array}{lcl} C1(n, O) & = & \{S_u \neq S_v \mid \neg\texttt{agree}(u, v) \wedge u < n \wedge v < n\} \\ C2(n, O) & = & \{S_u = S_v \Rightarrow S_{ua} = S_{va} \mid a \in \Sigma, ua, va, a \in Dom(O) \wedge u < n \wedge v < n\} \end{array}$$

C1 test u and u' on the empty suffix and C2 guarantees right congruence. Define the set of constraints $\mathcal{C}_n(O) = C1(n, O) \cup C2(n, O)$ with $n \in \mathbb{N}_0$. A DFA with n states exists conforming to the set of examples S iff $\mathcal{C}_n$ is solvable over the first n+1 integers. Let $\mathcal{D}(\mathcal{C}_n(O)$ be the set of variables used in constraints in $\mathcal{C}_n(O)$. The DFA can be built from $\mathcal{D}(\mathcal{C}_n(O)$ as follows:

- $|Q| = n$, $q_0 = S_\epsilon$
- δ satisfies $\delta(n, a) = n'$ if there is $S_u, S_{ua} \in \mathcal{D}(\mathcal{C}_n(O)$ with $S_u = n, S_{ua} = n'$
- F satisfifes $\forall S_u \in \mathcal{D}(\mathcal{C}_n(O) : O(u) = + \Rightarrow n \in F \wedge O(u) = - \Rightarrow n \notin F$ where $S_u = n$.

There also exists an efficient encoding of this constraint problem as a SAT problem. The k-tails algorithm has been extended in a number of ways

- Probabilistic Finite State Automata (PFSA) can be inferred using an algorithm called **sk-strings** [135]. This an extension of the ktails algorithm - a canonical DFA is built from the input examples and equivalent states are merged. States are equivalent ($q \equiv_s p$) if they agree on the top s percent of their most probable k-strings :

$$\begin{array}{lcl} k\text{-}strings(q) & = & \{u \mid u \in \Sigma^* \wedge ((|u| = k \wedge \delta(q, u) \subset Q) \vee (|u| < k \wedge \delta(q, u) \cap F \neq \emptyset))\} \\ choose(q, s, k) & = & \{u \mid S = k\text{-}strings(q) \wedge u \in S \wedge R(u, S) > \frac{|S| * s}{100}\} \\ p \equiv_s kq & \text{iff} & choose(p, s, k) \sim choose(q, s, k) \end{array}$$

  Where $R(u, S)$ gives a rank of the string u in S according to its probability and $\sim$ can stand for a number of set relations, the choice of which dictates the strictness of the merging.

- Lee et al. [86] employ a technique called *parametric trace slicing* to reduce the problem to that of inferring a normal DFA (their JMiner tool infers a PFSA using sk-strings and then refines this). The parametric trace slicing approach is discussed further in Chapter **??**.

- Extended Finite State Machines (EFSM) can be inferred using the *GK-tails* algorithm [103, 104], this work makes use of the axiomatic invariant detection tool *Daikon* [46](see Section 3.3.2) to generate predicates. The algorithm extends ktails so that it reasons about predicates on states and events. Their approach first merges *input-equivalent* traces - that is traces which are the same up to consistent renaming of parameters. These traces are then annotated with Daikon so that each element of the trace is labelled with a set of predicates relating to parameters encountered so far. They then create a canonical DFA for the set of labelled traces and merge equivalent states. States are equivalent if they share the same k-futures - where a k-future can be modelled by equivalence, weak subsumption or strong subsumption of predicates.

Using Genetic Algorithms There exist a number of approaches to regular inference that employ genetic algorithms - these are described in detail in Chapter 4, which looks more closely at the use of genetic algorithms in specification inference. There are generally two approaches - either to evolve a DFA directly or two evolve a partitioning (or set of state merges).

Summary - Passive techniques for regular inference all construct a DFA from the samples and then generalise this by merging states. A complete generalisation would merge all states, however this would be a trivial specification. Therefore, some additional information is required to prevent undesired merges - this has been given by negative information, a statistical similarity measure and a heuristic measure. If the sample set is structurally complete with respect to some DFA $\mathcal{A}$ then it is possible to exactly infer $\mathcal{A}$, otherwise any generalisation is an approximation.

3.2.3 Active Techniques for Regular Inference

An active learning approach to Grammar Inference was first taken by Dana Angluin [7], she introduced the seminal L^* algorithm which has been studied, extended and applied in many places in the literature. In this section I begin by discussing the practical implications of learning with an Oracle and then go on to describe Angluin's algorithm (L^*) and its extensions. I finish by describing QSM, an active approach based on the state-merging approach referred to previously.

Learning with an Oracle

The query learning framework assumes an *Oracle* which can be asked a combination of *membership* and *equivalence* queries. A membership query asks whether a word in Σ^* is in $\mathcal{L}$ and an equivalence query asks whether a hypothesized DFA is correct - if it is not correct the *Oracle* returns a counter-example. The assumption of such an oracle can be problematic practically. Answering membership queries is not difficult in practice but can be expensive. However, it is often impractical to assume an entity that can answer equivalence queries, although this problem has been partly addressed by an area called *Conformance Testing*(Section 2.3) [25] which can give high confidence that a specification is equivalent to a program. As membership queries are generally a lot cheaper than equivalence queries so approaches tend to aim to carry out more membership than equivalence queries. However, Angluin showed that equivalence queries are necessary to identify a DFA.

Some people have assumed the setting where *maybe* can be returned by the *Oracle*. These are called an *inexperienced teacher* in [88]) and *partially ignorant teachers* in [51]. Finally, Parekh and Honavar [123] discuss what restrictions must be placed on an oracle to prevent potential *collusion* between the oracle and learner. This generally consists of a third-party being able to remove or re-order queries and replies.

Angluin's Algorithm

I begin by giving an overview of the algorithm before discussing its extensions. Berg et al. [13] give a good introduction to the algorithm that is more approachable than the original paper.

We begin with a *Learner* that attempts to build a unique minimal automaton to describe an unknown language. The algorithm uses a prefix-closed set $U \subseteq \Sigma^*$ to identify states, and a suffix-closed set $V \subseteq \Sigma^*$ to distinguish states - this is similar to the constraints used in k-tails as both are based on Nerode's right congruence. The *Learner* builds a table $T : U \to (V \to \mathbb{B})$ with $|U|$ rows and $|V|$ columns. The table is

said to be *closed* and *consistent* as follows

$$\forall u \in U, a \in \Sigma, \exists u' \in U : T(ua) = T(u') \quad \text{(closed)}$$
$$\forall u, u' \in U, a \in \Sigma : (T(u) = T(u')) \Rightarrow (T(ua) = T(u'a)) \quad \text{(consistent)}$$

If the table is closed, the set U can be split into $U = U_S \cup U_S.\Sigma$ where U_S is a set of *short prefixes* differentiating different states. The table begins with $U = V = \{\epsilon\}$. A membership query is asked for each row in the table and then whilst the table is not closed or consistent rows are added to fix this and the table completed again. Once the table is closed and consistent a hypothesized DFA can then be constructed from T so that Q is the set of distinct rows (U_S), q_0 is the row $T(\epsilon)$, δ is defined by $\delta(T(u), a) = T(ua)$ and the accepting states are those rows where $T(u)(\epsilon) = true$. If the *Oracle* accepts this hypothesis a unique minimum DFA has been found, otherwise all prefixes of the counterexample are added to U and the process continued.

This is guaranteed to find a canonical minimal DFA but may make a very large number of queries. We can put a bound on the maximum number of membership queries that can be asked - the maximum size of the table. For a DFA of n states and alphabet Σ and where counterexamples are of length at most m we have that $|U_S| \leq n + m(n-1)$ and $|V| = n$ as U_S is expanded only when T is not closed and when a counter-example is received and V is expanded when T is not consistent. The size of the table is then given by $(|U_S| + |U_S.\Sigma|) * |V| = ((n + m(n-1)) + |\Sigma|(n + m(n-1)))(n) = O(|\Sigma|mn^2)$. Therefore the algorithm is bounded by a polynomial in n and m.

Berg et al. [13] noted that If for the prefix-closed subclass of regular languages if there exists an entry in T for ua we do not have to make membership queries for u. The prefix-closed subclass of regular languages is a useful subclass as it represents automata which model systems that always stay in some 'good' state.

Extensions to Angluin's Algorithm

There exist a number of extensions of L^* as described below.

Dealing with Inexperienced or Ignorant Teachers An approach by Leucker [88] leads to an extension called ABL* where notions of *weak* closure and consistency are defined in the presence of *maybe*. Due to this weakness a DFA cannot be taken directly from the table and a version ktails is used. Additionally techniques for dealing with *partially ignorant teachers* have been explored [51].

Infinite Languages. The ultimately periodic subset of ω-regular languages, equivalent to deterministic weak Büchi Automaton, can be inferred in polynomial time by an algorithm called L^ω [110]. The ω-regular languages are the infinitary counterpart to regular languages and the ultimately periodic subset are those languages that contain only ultimately periodic words. An ultimately periodic word is an infinite word that can be factored into the form uv^ω. The main problem tackled by this approach is that of identifying the accepting states, which must be visited infinitely often.

Mealy Machines. Mealy Machines can be inferred by a simple extension to L*, defined as L$^*_M$ in [147]. The L$^*_M$ algorithm works by making the observation table map to output strings, the same rules for consistency and completeness can apply by using equivalence on this set. This work also uses a more efficient approach to process counter-examples that keeps T consistent by separating an existing prefix and distinguishing suffix, in an algorithm they name L^+_M, experimental results showed this to reduce the number of equivalence queries by an average factor of 1.32 - this is significant considering the expense of equivalence queries.

Timed Automata. Timed Automata described by Deterministic Event-Recording Automaton (DERA) are infreed in three different extensions to the L^* algorithm called TL^*_{sg}, TL^*_{nsg} and TL^*_s [65]. As it is not obvious how to generalise Nerode's right congruence to timed languages (which is necessary to extend the table) they use an abstraction of timed automata and introduce an *assistant* to translate queries.

Angluin's Algorithm Applied to Parameterised Models

I am particularly interested in grammar inference methods for parametric specifications. Shahbaz et al. [149, 148, 89], Berg et al. [14] and Aarts et al. [2] all extend the table used in the L^* algorithm to capture parameters, as well as considering constraints on parameters.

Shahbaz et al. [149, 148, 89] infer Parameterised Finite State Machines, however the description of their technique in their published work lacks clarity.

Definition 29 (Parameterised Finite State Machine from [149, 148])**.** *A Parameterised Finite State Machine is a 7-tuple $\langle Q, I, O, D_I, D_O, T, q_0\rangle$ where Q is a set of states, q_0 is an initial state, I and O are input and output alphabets, D_I and D_O are sets of input and output parameter values and $T \subseteq (Q \times I \times O \times Q \times (2^D_I \Rightarrow \mathbb{B}) \times (2^{D_I} \to D_O))$ is the transition relation.*

Note that transitions are labelled with source and target states, input and output symbols, a predicate on input parameters and a function taking a set of input parameters to an output parameter. They extend the table to include this information and introduce concepts of the table being *balanced* and *dispute-free* to replace the original concept of the table being *consistent*. Their algorithm is difficult to read and it is difficult to see how concrete values are abstracted consistently - the main point of interest. A lack of experimental results make it difficult to measure how effective their technique is. However, this approach is also detailed in a PhD thesis [146], which I have not yet studied in detail. Their approach has been implemented in the RALT tool mentioned later.

The following description has been added later and should be incorporated into the main description. This comes from reading the relevant part of [146]. The observation table is extended so that it consists of (S, R, E, T) where $S, R \subseteq \mathcal{U} \cup I^*$ ($\mathcal{U}$ is the set of all parameterised strings), $E \subseteq I^+$ and $T : (S \cup R) \to (E \to Set[D^+_I, (O^+, D^+_O)])$. S is prefix-closed and E is suffix-closed. The table T therefore has cells that consist of sets of pairs of input-parameter strings and parameterised output strings. Such a set can be split into *distinguishing subsets* such that each subset has a different output string. If a row has a number of distinguishing subsets it is called *disputed*. A disputed row is *treated* or resolved if for each distinguishing subset there is a string $t \in S \cup R$ such that t uniquely identifies each element of that distinguishing subset (the string for that row s followed by a parameterised event). Two rows are balanced if they are defined for the same input parameter values for all $e \in E$ (all columns) and the table is balanced if all rows are balanced. An equivalence relation on rows can be applied to a balanced table to identify and distinguish states. This all seems odd - *there is no abstraction.*

Berg et al. [14] allow transitions to be labelled with a parametrised action and a guard where parameters are restricted to be booleans and the guard is a conjunction of either the positive or negative occurrence of each of the transitions parameters. The form of parameterised systems they infer are given as follows.

Definition 30 (Parameterised System from [14])**.** *A Parameterised System is a 4-tuple $\mathcal{P} = \langle Q, \to, q_0, F\rangle$ where Q is a set of states, q_0 is an initial state, F is a set of accepting states and $\to \subseteq (Q \times Act \times G \times Q)$ is a transition relation where Act is a set of actions of the form $a(p_1 \dots p_n)$ for $p_i \in \mathbb{B}$ and G is a set of guards of the form $\bigwedge l_i$ where $l_i = p_i$ or $\neg p_i$.*

They constrain their parameterised systems to be deterministic and write abstract actions as $\alpha(\overline{p})$ and concrete actions as $\alpha(\overline{d})$. They assume the guards of outgoing transitions from each state with the same

action name form a partition of the parameter space. They make two extensions to Angluin's algorithm. Firstly, instead of requiring the table to be complete they label each *short prefix* u in the table with a set of *representative continuations* $u\alpha(\overline{d})$, attempting to capture the partition of outgoing transitions from the state represented by u. Secondly, to construct hypothesis parameterised systems they label each entry in the table with a guard consistent with that label, which are then refined based on the results of queries. They use a measure of 'parameter complexity' to describe their results - for state q and action name α this is defined as the total number of different parameters used in guards on transitions from q for actions α. An artificial benchmark showed that they needed in the order of 1,000,000 membership queries and 1,000 equivalence queries to infer a parameterised system with 50 states and an average parameter complexity of 5. A real-world benchmark with 4 states and average parameter complexity of 0.5 required 21 membership queries and 4 equivalence queries. Their approach has been implemented in the LearnLib tool mentioned later.

Aarts et al. [2] adapt ideas from predicate abstraction to infer Symbolic Mealy Machines. They define concrete actions to be of the form $\alpha(d_1, \ldots, d_n)$ where α has an arity defined by a tuple of domains $D_1 \times \ldots \times D_n$ giving the concrete values permissible in each place in that action - $(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n$. For each action α there is a set of symbolic counterparts of the form $\alpha(p_1, \ldots, p_n)$ where p_i is some abstract parameter. The transitions of a Symbolic Mealy Machine match a symbolic action against a concrete action and use the bindings and a set of state variables to check a guard. If this guard matches a set of assignments to state variables are executed and an output action computed. They assume that symbolic mealy machines are complete and deterministic.

Definition 31 (Symbolic Mealy Machine (SMM) from [2])**.** *A Symbolic Mealy Machine is a 6-tuple* $\langle I, O, L, l_0, V, \rightarrow \rangle$ *where* I *and* O *are disjoint finite sets of input and output actions,* L *is a finite set of locations.* $l_0 \in L$ *is an initial location,* V *is a finite set of state variables and* $\rightarrow$ *is a finite set of symbolic transitions of the form* $\alpha(p_1, \ldots p_n)\mathbf{if} g/v_1, \ldots v_n := e_1, \ldots e_n; \beta(e_1^{out}, \ldots, e_n^{out})$ *where* g *is a guard (boolean expression over* $p_1, \ldots p_n$*) and* e_i *is an expression.*

They give a denotation of a Symbolic Mealy Machine in the form of an infinite state Mealy Machine $\langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \lambda \rangle$ where Σ_I and Σ_O are the sets of all concrete input and output actions from I and O, Q is the set of all pairs (l, σ) (where σ is a valuation of state variables) and δ and λ are defined such that for any symbolic transition from l to l', any valuation σ and data values $\overline{d}$ such that $\sigma(g[\overline{d}/\overline{v}])$ is true we have

$$\delta((l, \sigma), \alpha(\overline{d})) = (l', \sigma') \text{ iff } \forall 1 \leq i \leq k : \sigma'(v_i) = \sigma(e_i[\overline{d}/\overline{v}]) \text{ and } \forall v \notin (v_1, \ldots v_k) : \sigma'(v) = \sigma(v)$$

and $\lambda((l, \sigma), \alpha(\overline{v})) = \beta(e_1^{out}, \ldots, e_n^{out})$. This is a straightforward substitution of concrete values in for abstract parameters, which means that constraints and state variables can be directly encoded into the machine.

To apply regular inference they must define an abstraction from the possibly infinite sets of Σ_I and Σ_O to small finite sets of abstract input and output symbols. They define an abstraction entity to do this. In the L^* algorithm the abstraction entity can be used to synthesize an assistant which translates abstract queries into concrete queries They note that an abstraction entity must keep track of relevant history information to ensure that abstractions are consistent.

Definition 32 (Abstraction from [2])**.** *Given* I *and* O *disjoint finite sets of input and output actions an* $(I, O)-$*abstraction is a 7-tuple* $\langle \Sigma_I^A, \Sigma_O^A, r_0, abstr_I, abstr_O, \delta^R \rangle$ *where* Σ_I^A *and* Σ_O^A *are finite sets of input and output symbols,* R *is a possibly infinite set of local states,* r_0 *is an initial local state,* $abstr_I : R \times \Sigma_I \rightarrow \Sigma_I^A$ *maps input symbols to abstract ones and* $abstr_O$ *does the same for output symbols and* $\delta^R : R \times (\Sigma_I \cup \Sigma_O) \rightarrow R$ *updates the local state.*

They define some properties of abstractions that make the regular inference problem decidable. This reduces the problem of inferring an infinite-state parametric machine to that of constructing a suitable abstraction. They describe a method for systematically constructing abstractions which makes a lot of assumptions about available information and conditions on the inferred machine.

Query-Driven State Merging (QSM) [40]

This is an active approach that extends the state merging approach taken by RPNI extended with the Blue Fringe and EDSM approaches. The active part of this approach consists of asking the user queries, who can be modelled as an oracle. The algorithm is given in Figure 5. The different functions used are as follows

- `ChooseStatePairs` uses the BlueFringe and EDSM combined approach from above to select a pair of states to merge. The first state will belong to the mutually umergable 'red' states and the second will be an adjacent state with a high compatibility score (with respect to transitions). If there is more than one such pair a random one can be selected. If no such pair exists the assignment is considered to be false.
- `Merge` constructs the deterministic version of A with q and q' merged.
- `compatible` checks if the given automata is compatible with the sample set.
- `GenerateQuery` generates queries to pose to the user based on the current solution, the two states to be merged and the resultant quotient automata. Given the current solution A let L be the language of A, let $x \in Sp(L)$(short prefixes) and $y \in N(L)$(kernel) be the short prefixes of q and q' respectively. Let $vw \in L/y$ be a suffic of q in A. A generated sample is some string $xvw \in L\ L(A_{new})$ such that $xv \in L$. Note that xv is an accepted string and w a continuation to be checked.
- `CheckWithOracle` submits the generated query to the oracle

This extension attempts to make the set of samples characteristic - the generated samples are taken from those that would be in a characteristic sample. Therefore, this is a polynomial time algorithm which can infer an *exact* automata if we assume we can check queries with an oracle in constant time - it is this assumption which the query learning framework uses to break the NP-hardness of the grammar inference problem.

Summary Active approaches have the ability to infer more complete DFA but have practical limitations - for example the problem of answering equivalence queries. Using a user as an oracle or an assistant is an interesting approach that may work, but such approaches would need to limit the number of queries sent to the user to make this approach practical.

3.2.4 Beyond Regular Inference

When going beyond regular languages we only currently concern ourselves with context-free languages. The first thing to note is that the class of context-free languages can be identified in the limit. As far as I can tell, there does not exist a polynomial algorithm for identifying a context-free grammar through the use of queries. To tackle this computational hardness researches explored restrictions on the class of languages. An example of a subclass of context-free-grammars that has yielded positive results is the subclass of *even linear* languages - where each production contains only terminals or has a single non-terminal with an equal number of non-terminals on either side. This has been tackled by reducing the problem of inferring an even linear context-free language to that of inferring a regular language

```
Data: A sample S = S+ ∪ S−
Result: A DFA consistent with S and the answers to queries
A = APTA(S)
while (q, q′) = ChooseStatePairs(A) do
    A_new = Merge(A, q, q′)
    if compatible(A_new, S) then
        while Query = GenerateQuery(A, A_new) do
            if CheckWithOracle(Query) then
                S+ = S+ ∪ Query
            else
                S− = S− ∪ Query
                return QSM(S)
        A = A_new
return A, S
```

Algorithm 5: The QSM algorithm

[108, 157, 80]. Another direction taken by Sakakibara et al. [141, 140] is based on assuming additional information is given. They note that the problem of inductively learning context-free grammars can be split into that of determining the grammatical structure or topology of the grammar and identifying the terminals and non-terminals. Therefore if some *structural* information is given identifying the topology the search space of the problem is greatly reduced. Tu and Honavar [160] present an approach that infers a probabilistic context-free grammar in chomsky normal form by carrying out biclustering on a table that enumerates all symbol pairs in a set of positive samples. Their approach assumes the sample set is independent and identically distributed. There also exist approaches using genetic algorithms to infer representations of context-free grammars directly - see Chapter 4 for further details.

3.2.5 Tools

There exist a number of tools which have attempted to bring collections of grammar inference algorithms and techniques together.

`libalf` **[18, 161]** This is a tool that has been proposed in the last few years and incorporates the basic L^* algorithm, k-tails and RPNI, as well as a few new variants. The tool consists of a number of components allowing for algorithms to share common code and different optimisation filters (active assistants) to be introduced. The tool is written in C++ (a JNI interface exists for interfacing with Java) and is publicly available.

RALT [146] The Rich Automata Learning and Testing library implements the L^* algorithm and some variants for inferring Mealy machines. This tool is written in Java and does not appear to be publicly available.

The LearnLib tool. Researchers at the University of Dortmund have constructed a tool for experimenting with different regular inference techniques and optimisations - the tool implements a number of extensions to the L* algorithm [132, 131]. There has been a number of papers looking at how this tool can be used to explore further extensions [133, 130, 4, 155]. Equivalence queries are *approximated* using conformance testing techniques. This tool is written in C++ and is publicly available.

3.2.6 Summary

In this section I have introduced a number of existing grammar inference techniques. The state of the art is in regular inference - particularly variants of L^* and the QSM algorithm. There exist learning algorithms for more expressive languages but this area has not been explored as thoroughly. Of particular interest to me are those approaches that infer specifications with parameters - notably the GK-tails approach which extends k-tails and the three extensions of L^*. The most complete method for inferring parametric specifications appears to be that described by Aarts et al. [2](page 32). This is not a complete account of the area, but is reasonably comprehensive. In particular, I am missing accounts of earlier approaches from the pattern recognition community and approaches for context-free grammar inference from the natural language processing community.

3.3 Dynamic Specification Mining Techniques

Dynamic specification mining attempts to mine a specification from a set of execution traces. This is generally carried out offline, although there does exist an online approach. In this section I begin by giving an overview of the different approaches from the literature and then expand on particular techniques for one of these - generate-and-check.

3.3.1 An Overview of Approaches

The following are different approaches that have been taken to mine specifications from execution traces.

Generate and Check : Pattern Checking

This approach uses a small predefined set of patterns to generate hypothesis specifications and check these against examples. This is motivated by the idea that most specifications use only a few small patterns - in 1999 Dwyer et al. carried out a study [42] which identified many commonly used patterns, such as the alternating pattern $(ab)^*$. Taking either a known or inferred alphabet an instance of each pattern for each combination of symbols is generated and checked. For example, the pattern $(ab)^*$ over alphabet $\{x, y, z\}$ generates $(xy)^*, (yx)^*, (xz)^*, (zx)^*, (yz)^*, (zy)^*$ to be checked. These patterns capture *desired* rather than *undesired* behaviour.

Frequent Itemset Mining.

Lo et al. [99, 98, 100, 101, 96] have carried out extensive work using the data-mining technique of frequent itemset mining to mine specifications. This approach bears many similarities to the pattern-checking approach. The group has taken an incremental approach to publishing - over ten publications in the last five years describe similar approaches and results. Their general approach is to prune the traces with respect to support before generating patterns they are confident in to check. This basic approach (two event patterns) has complexity $O(n + (a \times b))$ where n is the cumulative length of all traces, a is the total number of frequent events and b is the maximum length of a trace.Earlier work only produced sets of closely occurring events but later work introduces orderings between these events. One interesting approach [102] uses the inferred temporal patterns to steer the previously described ktails algorithm for regular inference.

Moderated Regular Extrapolation

Hungar et al. [71] have developed a rather ad-hoc method they call *moderated regular extrapolation* that generates a model of a system using both automatic and manual techniques. Experts supply constraints on

the generated model in the form of LTL trace specifications and independence relations (between events). Traces are collected and abstracted in some problem-specific way before being compacted into a prefix acceptor. Information from traces are used to generate a 'model', which is not formally defined. Test suite generation techniques are used to validate and enhance the model, event independence relations are used to generalise the model (by capturing allowable reorderings) and a model checker is used to check the model conforms to the LTL specifications. Manual intervention is required if the model becomes inconsistent. Their approach is applied to a telephony system. The approach is very interesting, but not particularly rigorous and the manual aspect slightly deterring.

Summary

It is reasonably difficult to categorise approaches as this area is relatively new and solutions have been developed concurrently in different communities. Additionally, because of the wide spread of solutions from different areas and the general applicability of the approach I am not confident that I have captured all of the different approaches existing in the literature.

3.3.2 Generate And Check for Invariants

An early generate and check approach for state specifications, or state invariants, is the Diakon tool [45, 46, 117]. The approach examines an execution trace which records the value of variables at different points in the program and postulates a number of invariants which it then checks these for other runs of the program. The invariants Diakon can infer are

- Invariants over a variable - such as being constant or coming from a small set
- Numerical invariants for single variables - being in a range, being no-zero, being equal to some constant modulo some other constant
- Numerical invariants for two variables - linear relationships, ordering relationships and functional equivalence (being equivalent under some function or the function of the two variables satisfying some invariant)
- Numerical invariants for three variables - linear relationships
- Single sequence invariants - minimum and maximum conditions, (lexicographical) orderings, invariants holding over the whole sequence
- Two sequence invariants - elementwise linear relationship, comparison, subsequence relationships
- Numeric and sequence invariant - membership

From this (non-comprehensive) list it can be seen that the invariants that can be inferred are very expressive. To infer invariants they use a library of template invariants instantiated with variables observed at runtime. The set of inferred invariants is then filtered for redundant invariants before being checked.

3.3.3 Generate And Check for Trace Specifications

Here I describe methods for specification mining that utilise the pattern checking technique. I split this section by discussing the patterns used, how patterns to check are generated, how they are checked and how the results are processed - including combining patterns, pruning results and defining success.

Peracotta	Javert		OCD	SAM
[175, 176, 177]	[54]	[53]	[55]	[90]
$S^*(PP^*SS^*)^*$	$(ab^+)^*$	$(ab)^*$	ab	$(ab)^*$
$(PS)^*$	$(b^+c)^*$	$(ab^*c)^*$	ab^+	$\mathbf{G}(a \rightarrow \mathbf{X}(a\mathbf{U}b))$
$(PSS^*)^*$		$(ac)^*$	a^+b	$\mathbf{G}(a \rightarrow \mathbf{X}b)$
$(PP^*S)^*$			$ab?$	$\mathbf{G}(a \rightarrow \mathbf{XF}b)$
$S^*(PS)*$			$a?b$	
$(PP^*SS^*)^*$			a^+b^*	
$S^*(PSS^*)^*$			a^*b^+	
$S^*(PP^*)^*$			$(ab\|ba)$	
			$(a^+b^+)\|(b^+a^+)$	

Figure 3.3: Example Patterns (columns are unrelated)

Patterns Used

Patterns used in some different approaches have been outlined in Figure 3.3. One of the earliest (2001) [44] approaches using this technique employed ad-hoc templates in a framework that would be difficult to extend, for example the template *do not dereference null pointer* $< p >$, although they do consider the two templates $a(\neg b)^*$ and ab. The next (2004) set of approaches using this technique, Perracotta [175, 176, 177], used small (2 or 3 element) regular expression patterns based on the *Response* pattern [42] that says whenever P happens, S must also eventually happen - the alternating pattern given above is an example of this. Later work [54, 53] combines patterns to create final specifications and the patterns chosen reflect this, the patterns given in [55] relate to patterns composable from the two patterns in [54]. Li et al[90](2010) focus on mining temporal properties for hardware design and build a binary pattern language over temporal and timing operators, for example the pattern $\mathbf{G}(a \rightarrow \mathbf{X}(a\ \mathbf{U}\ b))$. In contrast, Weimer and Necula [171](2005) only use the alternating pattern $(ab)^*$. Few approaches consider an events contextual information, for example parameter or return values, although Yang et al. [177] perform a 'context-sensitive' slicing of traces and Gabel and Su [55] relate patterns to single objects.

Generating Patterns.

Once a pattern language has been identified a tool must generate all the patterns it wants to check against the sample set. The first thing that it requires is an alphabet of symbols. Most approaches considered here scan the sample traces to collect an alphabet. OCD [55] generates templates on the fly, as does work by Engler et al. [44]. The number of patterns checked will be a product of the set of templates used and the size of the alphabet. Let a template t_n be a regular expression over n distinct variables. Let T_n be a set of templates such that all templates in T_n range over n or fewer variables. Let $A = |\Sigma|$ be the size of the alphabet. The number of patterns generated is bounded by $\binom{n}{A} = \frac{n!}{A!(n-A)!}$.

Efficiently Checking Patterns

After generating patterns these must be checked, below I detail some of the approaches that have been taken to check patterns:

- Peracotta [177] introduce a simple matrix approach for checking binary patterns - for each pattern and an alphabet of n symbols an $n \times n$ matrix is constructed with each cell representing a FSA for that pattern. Then, for each observed event in the trace and for each pattern being checked the rows and columns relating to the observed event are iterated over, with the contained FSAs being updated accordingly.

- Javert [54, 53] employ a symbolic algorithm using Binary Decision Diagrams (BDD) to represent the current state of each of the instantiated templates being checked - each of which is encoded using $(|\Sigma| * \lceil \texttt{log}_2(|\Sigma'|)\rceil) + \lceil \texttt{log}_2(|Q|)\rceil$ boolean variables. This set of all automaton configurations can be represented efficiently as a BDD and the algorithm for updating the BDD on the observation of an event is then given in terms of BDD operations.
- In OCD Gabel and Su [55] base their checking approach on the assumption that properties occur within a small finite window. They use a sliding window over each trace to generate and check new and check previously seen patterns on the fly.
- In SAM Li et al. [90] extends the matrix approach. As their framework allows multiple events to occur at the same time (this makes sense in their setting of hardware systems) they extend the approach to ensure that no two FSAs are updated more than once on a single cycle.

Note that all approaches are very sensitive to the size of the alphabet - the matrix approach uses $O(A^{n-1}l)$ time for an alphabet of A symbols, a pattern with n symbols and a trace of length l.

Combining Patterns

Yang et al. [177] proposed a method for *chaining* inferred alternating patterns using the rule

$$\frac{A \to B \qquad B \to C}{A \to C}$$

Gabel et al. have created two tools Javert [53] and `OCD` [55] which use more complex inference rules to combine patterns - furthermore they argue that the chaining rule used by Yang et al. is not statistically sound. The two inference rules used are

$$\frac{(a\mathcal{L}_1^* b)^* \qquad (a\mathcal{L}_2^* b)^*}{(a(\mathcal{L}_1|\mathcal{L}_2)^* b)^*} \quad \text{(Branching)}$$

$$\frac{(a\mathcal{L}_1 b)^* \qquad (b\mathcal{L}_2 c)^* \qquad (ac)^*}{(a\mathcal{L}_1 b\mathcal{L}_2 c)^*} \quad \text{(Sequencing)}$$

More recently (2010) Li et al.'s work [90] with more complex temporal patterns also introduced similar inference rules over their more complex patterns.

$$\frac{(ab)^* \qquad (bc)^* \qquad (ac)^*}{(abc)^*} \quad \text{(Alternating Pattern Chaining)}$$

$$\frac{\mathbf{G}(a \to \mathbf{XF}b) \qquad \mathbf{G}(b \to \mathbf{XF}c)}{\mathbf{G}(a \to \mathbf{XFG}(b \to \mathbf{XF}c))} \quad \text{(Eventual Pattern Chaining)}$$

$$\frac{\mathbf{G}(a \to \mathbf{X}(a\mathbf{U}b)) \qquad \mathbf{G}(b \to \mathbf{X}(b\mathbf{U}c))}{\mathbf{G}(a \to \mathbf{X}(a\mathbf{U}(b\mathbf{U}c)))} \quad \text{(Until Pattern Chaining)}$$

Improving Specifications with Test Generation

To check and improve the mined specifications some approaches use test generation techniques to produce a set of tests that should pass based on the specification. Dallmeier et al. [31] present the Tautoko tool which mines specifications and then generates tests which explore undefined transitions by mutating an existing test suite. Elsewhere, Xie and Notkin [174] have combined a specification miner for algebraic specifications with a test generation technique.

Pruning Results

This approach will produce a vast number of specifications, many of which are highly irrelevant and inaccurate. All of the above approaches implement some sort of ranking based on the *support* and *confidence* of specifications. The support of a specification is the number of times it occurred positively in the sample set and the confidence of a specification is the number of times it appeared positively over the number of times it appeared both positively and negatively. Additionally heuristics are given for pruning results - for example Peracottta [177] uses a reachability heuristic and a name similarity heuristic. OCD [55] aggressively prunes specifications during the mining process.

Defining Success

Support and confidence can be used to give the **accuracy** of an approach in terms of a **false-positive** rate - a false-positive occurs when the mining algorithm reports an incorrect specification. Along with accuracy another measure of success is **coverage** or **completeness** - the former can be coached in terms of false-negatives and describes how well the exact behaviour is covered, the latter is a slightly weaker concept stating that the sample state must be covered by the inferred properties. Obviously achieving better coverage than completeness involves highly unsound extrapolation.

Summary

The generate-and-check concept is very simple - use a set of templates to generate a set of properties to check over the sample execution traces, possibly allowing for some noise. The tricky part is ensuring that the resulting specifications are *accurate* and *relevant* as well as making sure that the approach is *efficient* enough to scale to real-world problems.

3.4 Static Specification Mining Techniques

Before discussing the static specification mining techniques I briefly compare this approach with the dynamic specification mining approach. I then discuss the different static techniques found in the literature. **This section is included for completeness and the reader should not expend too much effort attempting to understand its content.**

The majority of these employ data-mining techniques on information extracted from the source code. There do exist other techniques [150, 81, 105, 172, 33, 32] but for time and space reasons I do not discuss these here. The explanations of the approaches are long and reasonably detailed, this is partly due to the fact that it is difficult to condense these approaches into a general approach in the same way as it is for dynamic generate and check approaches.

3.4.1 Comparing the Static and Dynamic Approaches

Static specification mining differs from dynamic specification mining in the artefact that it mines. As artefacts to mine specifications from execution traces and source code each have their advantages and disadvantages - the source code describes the exact behaviour of the program, but does not include runtime information or indicate the most common behaviours. Additionally mining from source code may lead to specifications of implementation decisions not design decisions. For this, and other, reasons mining from execution traces seems preferable. The main advantage of using execution traces is the availability of runtime information - it is possible to see which parts of the program are exercised the most and typical inputs to the program. When wanting to learn what the program is *supposed to do* this is important as the source code may contain many code paths that are never exercised, and the frequency of specifications

occurring in execution traces can then be used to reason about their relevance. Additionally the source code may not be available - this 'black-box' situation is common in the field of reverse engineering and often motivates the use of dynamic program analysis.

3.4.2 Data Mining Techniques

Frequent itemset mining [64] is a technique from data-mining which takes a set of itemsets, which is just a set of arbitrary items, and a support threshold *min_support* and returns a set of patterns that occur in at least *min_support* itemsets. A mined pattern is closed if it is not subsumed by any other mined pattern. Sequential pattern mining [111, 6] is also a data-mining approach similar to frequent itemset mining but is applied to temporally ordered data and finds frequently occurring subsequences.

PR-Miner

Li et al. [91] describe the tool PR-MINER that attempts to find associates among elements by identifying elements which are frequently used together in source code, with no explicit concept of order. The source code is parsed into a database by first hashing each program element into a number and then mapping function definitions to a set of these numbers to be written as a row in the database. A row in the database is an itemset and the frequent itemset mining algorithm finds the most frequent itemsets, that is the sets of program elements which are frequently used together. They call these itemsets *programming patterns* (although they have no structure) and infer association rules of the form $set_1 \Leftarrow set_2$ from them by selecting a non-empty subset as the lefthandside and the non-empty remainder as the righthandside. In this way each programming pattern gives $2^n - 2$ association rules. The confidence of each association rules is calculated as $\frac{\#set_1}{\#(set_1 \text{ and } set_2)}$ where $\#set$ gives the number of times that set appears in the database. They can then rank their violating rules in terms of confidence and support. They attempt to prune false positives caused by rules spanning functions by an intra-procedural analysis. I note a number of limitations to this technique - firstly, rules can only occur within the scope of a single function, secondly, local variables are hashed to their data type meaning local variables of the same type are considered the same, and finally, that their technique does not capture the ordering between program elements. Their evaluation showed a large false-positive rate - 25, 75 and 85 percent for the three reported evaluations.

Jadet

Wasylkowski et al. [170] describe the tool JADET that creates per-object usage models describing the 'normal' orderings of calls on that object. Their approach works on a set of Java classes and they create usage models for all objects that are created using `new`, appear as method parameters, are returned from methods or are used as exceptions. They begin by constructing a *method-model* for each method, similar to a call graph, where states are locations in the code and transitions are instructions. A special end state is connected to all states after a `return` instruction with an ϵ-transition and exceptions are handled in a more complicated way, introducing additional states. To create a set of *object-usage-models* from a method model for each object they project out all transitions using that object. To deal with aliasing they use a data-flow analysis and introduce a special case for objects that are *cast* to a particular type. For each method they construct a set of *control flow relations* of the form $R(Obj) = \{(m, n) \mid$ n can be called after m$\}$ by examining each objects object-usage-model. The union of all sets of control flow relations for objects for a method is taken as an itemset, i.e. $R(M) = \{(m, n) \mid (m, n) \in R(O)$ and O used in $M\}$. Frequent itemset mining is then applied to the set of all $R(M)$ for all methods M in the given Java classes. They filter out redundant patterns from the mined patterns and use the resulting patterns to detect and rank anomalies in the code. It should be noted that the patterns here are different from those used by Li et al. [91] as here a pattern is a set

of ordered pairs. Their experiments seem promising, although false-positive rates are still an issue. They usefully note a number of limitations to their approach which include the fact that they do not support the whole Java language (for example multithreading), they abstract away from structural information in the code and do not capture negative information in the form of things that cannot happen.

Kagdi et al.

Kagdi et al. [74] compare a sequential pattern mining approach to a frequent itemset mining approach. Their itemset mining approach is similar to that of PR-Miner described above. To carry out sequential pattern mining they have to construct a temporally ordered dataset - this is difficult as sometimes the ordering between calls is undefined and compilers order them differently. Their approach is not described in much detail but it is indicated that it follows a similar line to that of PR-Miner. It is noted that the search space of frequent itemset mining is 2^n for n call-usages but for sequential pattern mining is at most 2^{mk} for m partially ordered call-usages with an average of k calls. They note that the later approach took four times longer (242 minutes) to run. Their experimental results show that sequential pattern mining produces more patterns but fewer variants and violations then frequent itemset mining and they argue this means that it is less likely to produce false-negatives.

Ramanathan et al.

Ramanathan et al. [136] introduce a technique called *predicate mining* that identifies the preconditions that must hold whenever a procedure is called. They infer both *data-flow* predicates that capture values held by variables at callpoints and *control-flow* predicates that capture orderings between procedure calls. A flow-analysis is used to build these predicates by associating predicates with statements, a predicate specification language is presented and a set of inference rules given which relate the results of flow analysis to the allowed predicates. One approach would be to define the predicates for a procedure are the intersection of all predicates at all call points to that procedure however this is often over-conservative. Instead they utilise frequent itemset mining is to infer data-flow predicates, where an itemset represents the predicates at a call point, and sequential pattern mining is used to infer control-flow predicates. Some work is done take account of structural equivalence of predicates by examining type and positional parameter information. Their evaluation infers many preconditions for procedures, although many of these consist of few (< 2) predicates. They note that they could improve their approach by utilising theorem proving to infer more involved preconditions.

Acharya et al.

Acharya et al. [5] describe an approach that takes as input some source files and descriptions of the APIs of interest. Their approach consists of four parts

1. They adapt a model checker to generate interprocedural context-flow-sensitive static traces related to the APIs of interest. They employ *push-down model checking* [47] by modelling the program as a push-down automata and for each exit point of the program constructing a *trigger* FSA that accepts all strings beginning at the entry to the program and ending at that exit point. For each trigger FSA they use a model checker to compute the set of paths through the program from which they can project calls to the APIs of interest, this is an over approximation due to data-flow insensitivity.

2. They extract usage scenarios from these static traces by identifying calls from the different APIs and extracting a partial order of API calls for each API and a partial order between APIs.

3. They mine frequent closed partial orders from these usage scenarios. A frequent closed partial order if it is frequent and closed in the itemset mining sense. They use an existing tool [126] to do this, which employs a data-mining technique to a database of usage scenarios.

4. They extract specifications that are universally true for a set of programs by mining patterns from one half of the programs and checking them on the other. This seeks to prune false-positives.

Their evaluation seems promising, however there appears to be a reasonably amount of manual work required to make this approach work. Given the size of the programs being mined - a total of 200k lines across 72 programs it is reasonably impressive that their use of model-checking to produce the traces worked, however I wonder if this will scale well for larger programs.

Summary

There have been many attempts at using data-mining techniques to infer specifications from source code. Here I have described the most recent approaches in the literature. The unifying theme is to develop a technique for extracting some data from the source code and then applying the data-mining techniques to this data. This approach can be split into three parts - extracting data, mining patterns and pruning patterns. The majority of approaches could make use of techniques used in other approaches for the last two parts.

3.5 Applications

In this section I describe some applications of specification mining in the literature as well as some possible applications I have considered myself. I began my search by examining the background section in the PhD thesis of Shahbaz [146].

3.5.1 Program Comprehension

The majority of approaches can be thought of as approaches to aid program understanding. By inferring a specification of normal program behaviour a programmer can better understand how to use a third-party piece of code or check to see if their understanding of what the code does is correct. Here I describe a few instances of where specification inference techniques have specifically been used to aid program understanding.

Inferring UML diagrams

Makinen and Systa [109] present the Minimally Adequate Synthesizer (MAS) interactive algorithm which synthesizes UML statechart diagrams *from sequence diagrams*. Traces are constructed for each object in the system and consist of pairs (s, r) where r is a message received by the object and s is the message previously sent by the object (or null if no message has been sent since the last message was received). The language of the state chart can then be inferred from the set of such traces constructed from the sequence diagrams - note that this is prefix-closed. Angluin's algorithm (L^*) is applied to the sample set where the user is used as an oracle and certain optimisations reduce the number of queries asked. Note that this approach does not infer a statechart from an execution trace or code but from a different form of specification.

Legacy Systems

Hungar et al. [71] present an approach for learning a specification of a legacy system to support testing. Angluin's algorithm (L^*) is used with a testing framework used to answer membership queries (they do not cover equivalence queries). They key difficulty in the approach was that L^* produces propositional strings as membership queries and these need to be translated into test cases involve non-propositional symbols. They reverse the abstraction introduced by L^* heuristically using a number of previously developed techniques. They note that inferred models could be used for regression testing and test case analysis, but do not implement these techniques. During evaluation the number of membership queries was polynomial in the number of states (78 states led to 132,300 membership queries) but an optimisation based on the prefix-closedness of the language bounded this to quadratic.

Inferring a Specification for a Biometric Passport

Aarts et al. [3] the extension of L^* described in Section 3.2.3 (and coded in the LearnLib tool) for inferring Symbolic Mealy Machines to construct a description of the protocol used in a biometric passport. They had to construct an abstraction mechanism by hand to allow them to use their technique. Inference took less than 60 minutes and created a model with 5 locations and 19 transitions. They compared the inferred model to a hand-coded model and the inferred model was found to be more general than but consistent with the hand-coded model as this model was not complete.

Understanding Change

With the complexity of large systems and the inevitable coupling of program components, one problem in the area of software development is understanding how an update to a system has changed its functionality. Once a program has entered the maintenance phase this problem can be compounded by decreasing documentation and losing those involved in the original project. Currently *regression testing* is used to check that updates to a system have not introduced a new bug. But one possible application of specification inference would be to track changes to the programs inferred specification.

3.5.2 Testing

The next main use of specifications is in the area of ensuring program correctness through testing. Here specifications are often informal, however a number of approaches have shown how testing techniques can be improved when a formal specification is available.

Integration Testing

Shahbaz et al. [149, 148] infer parameterised models using an extension of L^* (the first L^* extension for dealing with models with parameters). Given a set of components with known interfaces (consisting of input and output symbols with parameter signatures) and a communication architecture between these components. They use this to construct a composed system such that two components can be pairwise connected if they have corresponding input and output symbols in their interfaces. Any non-connected interfaces are considered external interfaces to the composed system. For each component they manual construct a mapping from interfaces and parameter domains to the associated parts of the parameterised model. A model for each component can then be inferred in separation (using unit testing and conformance testing techniques to answer queries). These models are then composed in the same manner as the composed system and used to generate integration tests.

Model Based Testing meets Black Box Testing

Raffelt et al. [133] use the LearnLib tool to carry out black-box testing of a web-based bug tracking system using tests generated from an inferred Mealy Machine. In the inferred model they note that it is only possible to reach a useful subgraph by taking a login transition. More interestingly, Peled et al. [127] present a formal treatment of how black-box testing can be carried out using L^*.

Bug Location

It should be noted that as a specification has been derived from the program itself it should be easier to locate the bug by creating a link between the points in the program that led to the failing part of the specification being created. This can be seen more obviously in the static specification mining techniques described above but one could imagine a similar approach applied to dynamic specification mining where the source is available.

3.5.3 Verification

The more rigorous approach to ensuring program correctness is formal verification, here formal specifications are required to carry out techniques such as model checking and runtime verification.

Automating Assume-Guarantee Reasoning

Puasuareanu et al. [129] use Angluin's algorithm to compute assumptions to carry out assume-guarantee reasoning which takes a divide-and-conquer approach to verifying large systems. A component in a large system is verified within the context of a number of assumptions. Assumptions are given as deterministic labelled transition systems (a finite state machine where all states are accepting) and a notion of weakest assumptions are introduced. The L^* algorithm is used to infer assumptions using the LTSA model checker as an oracle. Different setups are used for different assume guarantee rules, here I describe the setup for the asymmetric rule

$$\frac{\langle A\rangle M_1\langle P\rangle \qquad \langle true\rangle M_2\langle A\rangle}{\langle true\rangle M_1||M_2\langle P\rangle}$$

Where M_1 and M_2 are components, P is the property being checked, A is an assumption, $||$ is parallel composition and $\langle A\rangle M\langle P\rangle$ is true if component M guarantees P assuming A. Membership queries are given by checking $\langle t\rangle M_1\langle P\rangle$ for query trace t. Equivalence queries are given by first checking $\langle A\rangle M_1\langle P\rangle$, returning a counter example if it is false, and then secondly checking $\langle true\rangle M_2\langle A\rangle$, if this is true then a weakest assumption has been found, otherwise the counterexample is checked to see if either A is not strong enough or no assumption exists.

3.5.4 Security

A recent approach to online security that aims to detect abnormal behaviour has yielded some interesting approaches that make use of formal specifications. Additionally, in the same way that inferring a specification can help you verify a program is correct it is shown that inferring a specification for a communication protocol can help show that the protocol is secure.

Security in Communication Protocols

Shu et al. [152] describe how specification inference techniques can be used to test if security protocols obey an important security property - message confidentiality under the general Dolev-Yao attacker model

[36]. The structure of protocols is learnt as a Symbolic-Parameterised Extended Finite State Machine (SP-EFSM) of the form

Definition 33 (Symbolic-Parameterised Extended Finite State Machine (SP-EFSM))**.** *An SP-EFSM is a 7-tuple* $\langle Q, q_0, A, \Sigma, \Lambda, \Gamma, \delta \rangle$ *where* Q *is a set of states,* q_0 *is the initial state,* $A = \langle K, N \rangle$ *is a set of atoms or parameter values where* K *is a finite set of symbolic private/public key pairs and* N *is a finite set of nonces,* Σ *and* Λ *are finite sets of input and output symbols respectively with typed single parameter signatures taken from* $(Int, Key, Nonce)$*,* Γ *is a finite set of typed state variables with initial values, and* δ *is a set of transitions of the form* $t = (s, s', i, o, p, a)$ *with start and end states* s *and* s'*, input and output symbols* i *and* o*, a predicate* p *on state variables and input parameters and an action* a *on state variables and input/output parameters.*

As this machine has finite domains for its parameters it can be translated into an equivalent finite state machine - this construction is referred to as the *reachability graph* of the SP-EFSM. In their setup they do not assume that the learner knows the alphabet beforehand, instead they add an additional query to the oracle which can return a set of input symbols not contained in some given query. They adapt the L^* algorithm to learn SP-EFSM reachability graphs by allowing a row to record output strings - a reachability graph is just a Mealy Machine. They can then check the inferred model against the general Dolev-Yao attacker model.

Shu et al. [151] also suggest an approach for combining protocol inference with fuzz testing to detect security (and reliability) problems in communication protocols. Fuzz testing [119] is a technique that mutates the test data to attempt to uncover undesired behaviour. Shu et al. infer protocols for two MSN instant messaging protocols and generate a fuzzed test suite from this, they use this to find a number of non-trivial bugs in the protocols. The inferred state machine has 50 states and 70 transitions and they refer to both Angluin's algorithm and Bierman's algorithm but it is unclear which they use to achieve their results, however, they note that Angluin's algorithm is very costly to run to completion but can be used to give an approximate model.

Intrusion Detection

Based on an approach by Forrest et al. [69] that identifies anomalous sequences of system calls to detect intrusions a number of attempts have been made to infer specifications for intrusion detection.

Sekar et al. [145] develop their own approach for inferring finite state automata to describe normal behaviour. They tag traces with the program point from which each system call is made (given by the program counter), these are used as states in the inferred FSA. This works for statically linked programs but not for dynamically linked programs. Additionally, if system calls are made by library functions the structure of the program is lost. Therefore, they employ a technique called *system call tracing* to keep track of where in the program system calls originated from. Gosh et al. [58] describe three approaches to learning normal program behaviour that learn three different models - Elman recurrent neural networks, string transducers and finite state machines. Their learning algorithms are heuristically based and draw loosely from speech recognition techniques. They found that the neural network performed the best as they took the least time to train and gave fewer false alarms. Goa et al. [57] extract an execution graph during a training period and continue to update this during monitoring. The execution graph captures function calls and returns and is built by encoding observed traces. Ingham et al. [72] learn DFA representations of the HTTP protocol to protect web applications. They first introduce a scheme for tokenizing HTTP request to produce a trace alphabet and then describe their own induction algorithm, which they call the Burge DFA induction algorithm. The approach is to statically initialise the DFA with a state for each token in the trace and then use the traces to build transitions between token states reflecting the order of tokens in the trace.

Malware Detection

Christodorescu et al. [26] use a dynamic specification mining technique to infer specifications from a known malicious program and a set of benign programs to identify the parts of the specification that identify it as being malicious. They infer what they refer to as *dependence graphs* from execution traces - which at a high level capture typestate properties but they restrict the form they are interested in. A graph is created with system calls as nodes and vertices indicated that two system calls used the same object. They compute *contrast subgraphs* (an approach taken from data-mining) of the dependence graph taken from the malicious code using the graphs taken from the benign code and a number of other heuristics. Their approach created a lot of false positives but was able to identify programs as malicious. Sathyanarayan et al. [143] use a static analysis to extract critical API calls from known malicious programs to construct signatures for entire classes of malware. The signature for a malicious program is modelled by the frequency of certain API calls, although not their ordering. Beaucamps et al. [12](presented at RV10) extract trace abstractions from execution traces generated by malicious programs by rewriting given subtraces into abstract symbols representing their functionality. A behaviour pattern is used to detect behaviours and is a regular language. A string rewriting system is generated from a set of behaviour patterns to rewrite the concrete trace into some abstract trace to be compared against a malware signature. This approach does not construct behaviour patterns (a form of specification inference) but would benefit from an approach that did.

3.5.5 Controlling Programs

A specification of what a program is supposed to do can potentially be used to ensure that the program operates correctly.

Runtime Enforcement

A runtime enforcer [48, 92] sits between a program and its environment and ensures that the programs observable behaviour conforms to some specification. To understand where this might be useful consider two programs that control a robotic arm - the first has been thoroughly tested but is inefficient, whereas the second is very fast but relatively untested. One could mine the behaviour of the first and use the second, forcing it to only exhibit the safe behaviour. Specification inference might be required as describing exactly the behaviours required might be difficult. This has not appeared in the literature.

Multi-Mode Program Steering

Program steering is the process of controlling a programs behaviours to better achieve a goal. Ernst and Lin [93, 94] showed how a multi-mode program could be steered using inferred state specifications (using the Diakon tool). I discuss this further in Chapter **??**.

3.5.6 Music Recognition

Alcazar et al. [29] use grammar inference techniques to identify music styles. The idea is that their technique can be used to implement or improve content based retrieval in multimedia databases. They develop an encoding for music, which captures the pitch and duration of notes, and use this as the alphabet. They use a grammar inference tool which implements approaches from the natural language community which I have not yet covered- Error Correcting Grammatical Inference (ECGI) [142].

3.5.7 Summary

There have been many useful and interesting applications of grammar inference techniques. Apart from the work on detecting malicious programs by Christodorescu et al. [26], there have not been any applications of specification mining outside of program testing - and this has been reasonably qualitative so far, without any large scale tests or case studies. Additionally, the application of grammar inference techniques to the area of inferring program specifications has only become a hot topic in recent years.

3.6 Summary

In this chapter I have discussed the specification inference problem and a number of specification inference techniques. I have also extended my motivation for this domain by presenting existing and potential applications of specification inference techniques. I note that there exist (at least) five techniques for inferring parametric specifications, which are:

1. GK-tails uses the Diakon invariant specification mining tool to augmented a trace with predicates. The k-tails approach is then adapted to work over these augmented traces.

2. JMiner uses *parametric trace slicing* to abstract away parameters and then employs a standard regular inference tool to build a parametric specification. However, the approach makes the large assumption that events can only have one specification of formal parameters. Additionally there specifications do not capture conditions or local state.

3. Work by Shahbaz et al. [149, 148, 89] extends L^* to infer an expressive form of machine but the description of their technique is unclear.

4. Work by Berg et al. [14] extends L^* to infer a limited form of parameterised machines which only allow boolean parameters and guards over those parameters.

5. Work by Aarts et al. [2] uses predicate abstraction to reduce the problem of inferring parameterised machines to that of inferring Mealy machines. However, an abstraction technique must be given by the user unless a number of assumptions are made.

In the remainder of this section I describe some key points to take away from this chapter.

Known and Unknown Alphabets

It is usually assumed that the alphabet is known beforehand - in Grammar Inference this is given and in Dynamic Specification Mining it is inferred from the trace. In a passive approach, where all samples are given up front, it is only possible to build a model using symbols in those samples. However, in an active approach it is conceivable that the alphabet is discovered by querying the oracle. It should be noted that in some cases, such as inferring specifications for libraries or API's, the alphabet will definitely be known beforehand.

Imperfect Traces

If we assume that samples are correctly labelled then it is possible that we may miss some desired behaviour as bugs may exist in the code such that an execution trace labelled correct may still exhibit incorrect behaviour. Unless a small number of violations are allowed important specifications may not be inferred - these important specifications can be thought of as being *statistically significant*.

Noise

An execution trace may contain coincidental or uninteresting relations between events, this noise will create many uninteresting specifications. We want to avoid this as inferred specifications may be presented to the user. One example of noise would be where one method always calls another method, these method calls will alternate but this relation is uninteresting. It is possible to use a heuristically guided examination of the source code to identify noisy specifications.

Collecting Training Sets.

When using a passive learning approach we need to generate a representative set of execution traces, to do so it is necessary to have a representative set of inputs to the program that exercises the program adequately. Thankfully, these often already exist in the form of test suites. However, note that test suites often focus on *positive* behaviour only.

Concurrent Programs.

When attempting to infer specifications for a concurrent programs it is possible to either consider the global trace or per-thread traces. If we consider the global trace then specifications relating to thread-private objects may not be inferred, however if we only consider per-thread traces then intra-thread specifications may not be inferred. Generally it is best to consider both the global trace and per-thread traces, however this is more expensive.

Negative versus Positive Specifications.

We can specify program correctness in terms of behaviours that are allowed or in terms of those that are not - it is obvious that we can exactly divide behaviours in this way. However, expressing a specification that represents this divide may be difficult. Different uses of a specification will require it to be expressed either positively or negatively, so ideally one would be able to switch between the two. If the class of specification languages is *closed under negation* then the negation of a specification is guaranteed to be in that class of languages. This discussion of negative and positive specifications becomes more relevant when one is approximating a specification as it effects the choice between *over-approximation* and *under-approximation*.

Chapter 4

Genetic Algorithms and Specification Mining

In this s I briefly outline what genetic algorithms are (Section 4.1) and then discuss different approaches taken in the literature (Section 4.2) for inferring specifications using genetic algorithms.

4.1 Genetic Algorithms

A genetic algorithm [61] carries out a parallel search for an optimal solution to a problem. Genetic algorithms are an evolutionary strategy inspired by human evolution and take many of their terms and ideas from this area. A *population* of potential solutions (or *individuals*) is maintained and incrementally updated by applying *genetic operators* to individuals selected by a *selection mechanism*. Selection is based on the *fitness* of an individual, calculated by a *fitness function*.

Fitness Functions

A fitness function (or objective function) takes an individual and returns a value (usually between 0 and 1) which indicates how good a solution to the current problem it is. The genetic algorithm is searching for a maximal solution and if an individual has n varying elements the search space can be viewed as an n-dimensional plane given by the fitness function. Along the fitness axis this plane will have a number of maxima - and possibly a unique greatest maxima. The goal is to find this unique greatest maxima or the highest point on this plane possible. However, a big problem in genetic algorithms is getting stuck in local maxima. Techniques such as random jumps and multiple separate populations can be used to maintain individual diversity and thus avoid getting stuck in a local maxima.

Genetic Operators

Genetic operators are used to move around the search space - they take a subsection of the population and move it within this search space, hopefully towards a maxima. Generally genetic operators fall into two categories:

1. Mutation - takes a single individual and makes a small change to it
2. Crossover - takes two individuals and combines some information from each to make a new individual

The standard crossover technique is to generate two new individuals by taking the first half of the first and second half of the second as one new individual and the second half of the first and first half of the second as another individual - where it is assumed we can halve our individuals.

Selection Mechanism

On each generation of the genetic algorithm some part of the populated is selected for some genetic change. There are a number of popular selection mechanisms, some of each can be combined.

1. Elitism - Selects a top percentage of the fittest individuals
2. Roulette - Performs a random selection with repetitions, the probability of an individual being selected is given by the ratio between its fitness and the average fitness.
3. Tournament

Out of those selected some will be mutated and some subject to crossover - these rates will either be given by a quota or a probability.

Representing Individuals

How individuals are represented is very important. It will effect how expensive genetic operations are and, very importantly, how expensive it is to measure fitness. Following genetic terminology the representation of an individual is often called a *chromosome* and is the individual's *genotype*, whereas the interpretation of this coding is the individuals *phenotype*. It is often useful to code individuals as a fixed length bit string - this makes mutations and crossovers very easy. Finding a compact and efficient representation for an individual can be very difficult.

Summary

Here I given a very brief overview of genetic algorithms. Many of the papers in the next section give more detailed introductions and explain terms in more depth. The field of genetic algorithms and genetic programming is a well-established

4.2 Previous Attempts

The earliest attempt at grammar inference using genetic algorithms was by Fogel et al. [49] in 1966, who attempted to evolve DFAs for regular languages. Thing have moved on quite a bit since then. Here I outline some of the more recent approaches to evolving models for regular and context-free languages, as well as other more expressive variants.

4.2.1 For Regular Languages

The majority of uses of genetic algorithms have been to infer regular languages - it should be noted that other reasonably efficient techniques exist to solve this problem although these depend on the samples satisfying certain properties. In an early work (1982) Tomita [158] suggested a set of regular languages to use for evaluation, these are given in 4.1 and are used extensively in the literature to evaluate genetic algorithm approaches to regular inference.

Evolving a Partitioning

Approaches that attempt to evolve a partitioning of states are motivated by the regular inference search space as described in Section 3.2.1. A prefix tree acceptor is constructed from the sample sets and a genetic algorithm searches for an optimal partitioning of states.

	Description	RE	# states in FSA
L1	All a's	a^*	1
L2	a and b repeated	$(ab)^*$	2
L3	Not having odd number of b's then odd number of a's		4
L4	No more than 2 consecutive a's		3
L5	Even number of a's and even number of b's		4
L6	Number of a's and b's congruent modulo 3		3
L7		$a^*b^*a^*b^*$	4
L8		a^*b	2
L9		$(a^*.c^*)b$	4
L10		$(aa)^*(bbb)^*$	5
L11	Even number of a's and odd number of b's		4
L12		$a(aa)^*b$	3
L13	Even number of a's		2
L14		$(aa)^*ba^*$	3
L15		$(bc^*b.ac^*a$	4

Table 4.1: Tomita Regular Languages

Dupont presents the GIG method [38]. Partitions are represented by a *left-to-right canonical group-number encoding* - a group number encoding records the group for each state and the left-to-right canonical encoding orders groups by their minimal rank. The partition $\{\{1,3,5\},\{2\},\{4\}\}$ is given a group number encoding of $(2,1,2,3,2)$ which has a left-to-right canonical group-number encoding of $(1,2,1,3,1)$ - this makes all equivalent partitions equivalent by renaming. A structural mutation is used which randomly reassigns a state to a block - this can be the same, a different or a new block. A structural crossover is used which merges two blocks from two different partitions - $\{\{1,3,5\},\{2\},\{4\}\}$ and $\{\{1,2\},\{3,4\},\{5\}\}$ may produce $\{\{1,3\},\{2\},\{4,5\}\}$ and $\{\{1,2\},\{3\},\{4,5\}\}$ by merging the third block from each partition. The GIG method worked well on the Tomita languages and reasonably compared to the RPNI algorithm. The experimental setup randomly generates enough samples for the probability of the sample size to be characteristic, when this is just enough RPNI beats GIG but when they multiply this by three GIG bits RPNI - showing that more samples increases the accuracy of the approach. An incremental version of the evolution approach is used, this begins with a smaller set of the positive samples and finds an optimal solution in this search space before incrementally expanding the positive set. Dupont allows his algorithm to run for 2000 steps and uses a population size of 100, a crossover rate of 0.2 and mutation rate of 0.01. The population is 50% randomly generated and 50% drawn from the set of partitions presenting a single merge from the partition given by the PTA.

Pawar and Nagaraja [125] build on Dupont's approach, although very incrementally. They use four different forms of structural mutations, with the first two relating to Dupont's one

1. Randomly select a state and place it in a different block
2. Randomly select a state and place it in a new block
3. Randomly select a block and randomly choose a new block for each state in that block
4. Replace the entire partition with a randomly generated partition

They give further experimental results which showed that the third mutation operator improves helps minimise the number of partitions.

Hingston [67] also attempts to evolve a partitioning. This approach differs in that it uses the concept of Minimum Message Length [168] to measure fitness - this builds on previous work using the idea of minimising *entropy* [56, 124, 134, 156]. In fact Hingston seeks find an FSA F which, for data set D, minimises the quantity

$$DescriptionLength(F) + DescriptionLength(D \mid F)$$

Hingston represents partitions by a set of pairs of states to be merged - or a boolean mask over the set of all state pairs. His approach works with a positive sample S^+ only. Given the prefix tree acceptor PTA of the positive sample the fitness of an FSA F is given by

$$exp\left(-ln(2)\left(\frac{F.MML}{PTA.MML}\right)^2\right)$$

A mutation either sets a bit in the boolean mask to zero or one. The standard one-point crossover is used (taking the first half of the first and second half of the second). The approach taken is comparable to other approaches, but not significantly better. They advantage of this approach is that it does not require any negative data.

Evolving an Automata

Lucas and Reynolds [107, 106] evolve the transition function for a DFA. They note that to construct a DFA they must generate the number of states, transition function and state label vector - they use a state label vector (or output function $\Gamma : State \rightarrow Class$) to be more general than a simple acceptance class, but only consider two classes so this is equivalent. They note that they can either fix the number of states using prior knowledge or repeat the process with an increasing number of states. The transition function is represented by a $|\Sigma| \times |Q|$ matrix making the search space $|Q|^{|\Sigma||Q|}$, however they only consider *binary alphabets* so their search space is $|Q|^{2|Q|}$. To generate a state label vector they develop a technique called *Smart State Labelling* based on a proposed transition function and a training set. This takes an array $h[s][c]$ which gives for state s and class c the number of times in the training set for class c a transition function ends in state s and lets the class of state s be the class with the maximum number of entries. Their fitness function computes the proportion of the sample set which is correctly classified. They do not use a crossover operator but compare three different mutation operators which are

1. *Standard* - Changes one entry in the transition matrix randomly
2. *Sampled* - For each transition t a count is kept of the correct $c(t)$ and incorrectly $i(t)$ classified words it is involved in processing. The probability of mutating a transition is given by $p(t) = \frac{i(t)}{c(t)+i(t)}$ where $p(t) = 0$ if $i(t) = 0$. This additionally means that there is a zero probability of mutating an unreachable transition.
3. *Quick-Samples* - Instead of calculating $c(t)$ and $i(t)$, which involves iterating over the sample set again, each state is labelled with the strings it should and should not accept (as a final state) during fitness evaluation. This can be used to carry out state smart labelling. The number of errors at a state is given by the size of its minority set and a state is chosen with a likeliness proportionate to its number of errors. One of the erroneous strings from the chosen state is selected and the transition to mutate is chosen at random from the transitions that string passes through. This punishes a transition for being involved in an incorrect classification and does not reward it for being involved in a correct classification.

Their experimental setup generates random DFA of size n between 8 and 32 and depth $(2logn) - 2$ (in the same way as the Abbadingo One competition). They manage to correctly identify a DFA with 32 states using 10,620 steps (with sampled mutation). They compare their approach to EDSM, which they beat most of the time - especially on large DFA with sparse data. Additionally they carry out some tests were they artificially add noise to the training samples and in this case their approach vastly outperforms EDSM.

Niparnan and Chongstitvatana [118] evolve finite state machines in the form of Mealy Machines. They evolve the transition function and infer the output function from the training set. They first note that measuring the fitness of an inferred output function by comparing output strings leads to inaccuracies when there are only small differences. The transition matrix is evolved as above and once this has been evolved the output function is inferred from the training set. A $|Q| \times |\Sigma| \times |\Gamma|$ matrix OC is maintained (Σ input alphabet and Γ output alphabet) such that $OC[q][a][b]$ gives the number of times the machine transited from state q using input symbol a producing output symbol b - note that the machine is assumed to be deterministic so a destination state is not required. They then give the output function by

$$\gamma(q,a) = \begin{cases} anything & \forall x \in \Gamma : OC[q][a][x] = 0 \\ b & OC[q][a][b] > 0 \wedge \forall x \in \Gamma : x \neq b \Rightarrow OC[q][a][x] < OC[q][a][b] \end{cases}$$

Their fitness function for δ is then given by

$$\sum_{i=1}^{|Q|} \sum_{j=1}^{|\Sigma|} max(\{OC[i,j,x] \mid x \in \Gamma\})$$

Therefore a transition with no conflict in OC will be rewarded by how frequently that transition is used and a transition with some conflict will 'lose out' on the conflicting outputs and therefore be punished. In experiments they allowed their algorithm to run for 10,000 steps. They managed to infer Mealy machines with an input alphabet of 1 bit (2 symbols), an output alphabet of 3 bits (8 symbols) and 10 states consistently and up to 20 states in some cases.

Finally Bongard and Lipson [19] have developed an *active* approach which they describe as *coevolutionary*. The idea is to evolve some possible models and then evolve some membership queries which cause maximal disagreement among these models. The membership queries are used to augment the available information and the process repeats. For models they use two subpopulations to maintain diversity in their models and the following fitness function for i training sentences labelled by m_i where t_i is the inferred labelling.

$$1 - \frac{\sum_{j=1}^{i} |t_j - m_j|}{i}$$

Standard mutation and crossover operations were used. For the membership queries they use the following fitness function for k candidate models where $m_j(s)$ is the classification of s by the jth model.

$$f(s) = 1 - 2|0.5 - \frac{\sum_{j=1}^{k} m_j(s)}{k}|$$

On unbalanced DFAs their approach beat the EDSM algorithm on small DFA and performed as well as it for large DFA. Their approach also outperforms the approach suggested by Lucas and Reynolds (see above).

4.2.2 For Context-Free Languages

As there have been fewer successes in constructive approaches to context-free grammar inference, there have been a number of approaches attempting to utilise genetic algorithms to evolve representations of

context-free languages. People have either attempted to evolve a pushdown automata or a grammar directly.

Evolving a Pushdown Automata

Lankhorst [85] represents PDAs by a list of variable length transitions. Mutation randomly changes a part of a randomly selected transition to a value in the range of that position. The standard crossover operator is used. He uses a complicated fitness function based on

1. The number of correctly accepted or rejected sentences
2. The relative length of the prefix which is correctly accepted or rejected
3. The number of symbols left on the stack (acceptance is by empty stack)

A number of reasonably complicated context-free languages are inferred within 1000 generations.

Naidoo and Pillay [114] encoded PDAs directly as a transition graphs. Mutation is by replacing a randomly selected subgraph and crossover is by swapping randomly selected subgraphs. Using a population size of 2000 and a maximum number of generations of 50 all the languages in the following table were successfully inferred.

Language	Description
L1	$a^n b^n$ for $n > 0$
L2	$a^n c b^n$ for $n > 0$
L3	all odd length palindromes on $\{a, b\}$
L4	$ss^r \in \{a, b\}^*$ where s^r is the reverse of s
L5	$scs^r \in \{a, b\}^*$ where s^r is the reverse of s
L6	$s \in \{a, b\}^*$ where number of a's equal to the number of b's
L7	$a^n b^{2n}$ for $n \geq 0$
L8	aa^*ba^* (regular)
L9	$a^n b^{2n}$ for $n > 0$
L10	All strings with balanced brackets over $\{(,)\}$

However, the description of the experimental setup does not indicate how the test or training data was generated - specifically whether this was random or had to meet certain criteria. They did place a restriction on the maximum number of nodes as 5 and the maximum number of transitions per node as between 4 and 6 (depending on the language!).

Evolving a Grammar

In 1993 Wyard [173] suggested an approach which evolves context-free grammars represented as lists of production rules. Mutation randomly changes a symbol and crossover can occur at any point on the two lists of production rules such that it does not break a production. Wyard noted that getting stuck in local maxima was a big problem. For the correctly bracketed language only two out of five attempts converged at a correct solution (in 16 and 3 generations respectively).

More recently Pandey [122] (coming from a natural language processing background) has introduced a library for inferring context-free grammars using genetic algorithms. Choubey and Kharat [24, 23] have also recently tackled this problem and have achieved reasonable results, but the description of their methods are unreadable.

4.2.3 Other

There have also been some interesting approaches to evolving other forms of language models.

Stochastic regular grammar - Schwehm and Ost [144] present a genetic programming technique for evolving a stochastic regular grammar. A stochastic regular grammar is encoded as a bitstring. The objective function (fitness function) is then given by a weighted combination of the complexity of the hypothesis grammar and a χ^2-measure of how well the grammar fits the sample.

Stochastic Context-Free Grammars - Keller and Lutz [78] describe an approach for evolving stochastic context-free grammars (SCFG) using the minimum description length (MDL) principle as a fitness function. A SCFG is a variant of a context-free grammar with each production associated with a probability. The MDL principle is transferred into Bayesian terms so that it is phrased as maximising $P(G|C)$ where G represents a candidate grammar and C represents the sample set of traces (or Corpus in this work). An initial grammar is constructed and evolved using standard mutator and crossover operators.

Turing Machines - Naidoo and Pillay [115] present a genetic programming approach to inferring Turing machines represented directly as transition graphs. Fitness is given by the number of correctly accepted and rejected strings. Mutation is by random subgraph replacement and crossover by random subgraph switching. Their approach appears to work well, but their description of the evaluation does not include all the necessary details - for example how training and test sets were generated.

4.2.4 Summary

There have been a number of successful attempts to infer models for unknown languages using genetic algorithms. I note the following

- None of these approaches have been explored within the context of inferring trace specifications for programs
- Genetic algorithms work particularly well for more expressive formalisms (as the search space is a lot larger)
- I have not found any approaches attempting to infer parametric languages/specifications with free variables

Chapter 5

Conclusion

This report has attempted to summarise the field of specification inference. There will be omissions - if you read this and can think of any, and are feeling kind, then please contact me. The same applies to mistakes. If nothing else, I hope I have gathered together an interesting set of references.

Challenges

At some point I hope to use this space to give a concise overview of what I see as the outstanding challenges in the field.

Bibliography

[1] `http://www.eclipse.org/aspectj/`.

[2] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In *Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'10, pages 188–204, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. Inference and abstraction of the biometric passport. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I*, ISoLA'10, pages 673–686, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Fides Aarts and Frits Vaandrager. Learning i/o automata. In *Proceedings of the 21st international conference on Concurrency theory*, CONCUR'10, pages 71–85, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34, New York, NY, USA, 2007. ACM.

[6] Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[8] Dana Angluin. Queries and concept learning. *Mach. Learn.*, 2:319–342, April 1988.

[9] Dana Angluin. Computational learning theory: survey and selected bibliography. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, STOC '92, pages 351–369, New York, NY, USA, 1992. ACM.

[10] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15:237–269, September 1983.

[11] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Michael R. Lowry, Corina S. Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Richard Washington. Combining test case generation and runtime verification. *Theor. Comput. Sci.*, 336(2-3):209–234, 2005.

[12] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior abstraction in malware analysis. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 168–182, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. Technical report, In International Workshop on Software Verification and Validation (SVV, 2003.

[14] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, pages 107–121, 2006.

[15] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21:592–597, June 1972.

[16] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, PPPJ '07, pages 135–144, New York, NY, USA, 2007. ACM.

[17] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, and S. Vanak. Testing methods for x-machines: a review. *Form. Asp. Comput.*, 18:3–30, March 2006.

[18] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In *Computer Aided Verification*, pages 360–364, 2010.

[19] Josh Bongard and Hod Lipson. Active coevolutionary learning of deterministic finite automata. *J. Mach. Learn. Res.*, 6:1651–1678, December 2005.

[20] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 139–152, London, UK, 1994. Springer-Verlag.

[21] Gregory J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of The ACM*, 13:547–569, 1966.

[22] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113 –124, september 1956.

[23] N.S. Choubey and M.U. Kharat. Pda simulator for cfg induction using genetic algorithm. In *Computer Modelling and Simulation (UKSim), 2010 12th International Conference on*, pages 92 –97, march 2010.

[24] N.S. Choubey and M.U. Kharat. Stochastic mutation approach for grammar induction using genetic algorithm. In *Electronic Computer Technology (ICECT), 2010 International Conference on*, pages 142 –146, may 2010.

[25] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4:178–187, May 1978.

[26] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 5–14, New York, NY, USA, 2007. ACM.

[27] Orlando Cicchello and Stefan C. Kremer. Beyond edsm. In *Proceedings of the 6th International Colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '02, pages 37–48, London, UK, UK, 2002. Springer-Verlag.

[28] Alexander Clark and Shalom Lappin. Computational learning theory and language acquisition. In R. Kempson, N. Asher, and T. Fernando, editors, *Handbook of Philosophy of Linguistics*. Elsevier, 2011. forthcoming.

[29] P.P. Cruz-Alcázar, E. Vidal-Ruiz, and J.C. Pérez-Cortés. Musical style identification using grammatical inference: The encoding problems. In Alberto Sanfeliu and José Ruiz-Shulcloper, editors, *Progress in Pattern Recognition, Speech and Image Analysis*, volume 2905 of *Lecture Notes in Computer Science*, pages 375–382. Springer Berlin / Heidelberg, 2003.

[30] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[31] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96, New York, NY, USA, 2010. ACM.

[32] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24, New York, NY, USA, 2006. ACM.

[33] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In Andrew Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 733–733. Springer Berlin / Heidelberg, 2005.

[34] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recogn.*, 38:1332–1348, September 2005.

[35] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17:900–910, September 1991.

[36] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198 – 208, mar 1983.

[37] P. Dupont, F. Denis, and Y. Esposito. Links between probabilistic automata and hidden markov models: probability distributions, learning models and induction algorithms. *Pattern Recogn.*, 38:1349–1371, September 2005.

[38] Pierre Dupont. Regular grammatical inference from positive and negative samples by genetic search: the gig method. In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 236–245, London, UK, 1994. Springer-Verlag.

[39] Pierre Dupont. Incremental regular inference. In *Proceedings of the 3rd International Colloquium on Grammatical Inference: Learning Syntax from Sentences*, pages 222–237, London, UK, 1996. Springer-Verlag.

[40] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The qsm algorithm and its application to software behaviour model induction. *Appl. Artif. Intell.*, 22:77–115, January 2008.

[41] Pierre Dupont, Laurent Miclet, and Enrique Vidal. What is the search space of the regular inference? In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 25–37, London, UK, 1994. Springer-Verlag.

[42] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 411–420, New York, NY, USA, 1999. ACM.

[43] T. Elrad, R.E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[44] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.

[45] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27:99–123, February 2001.

[46] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 449–458, New York, NY, USA, 2000. ACM.

[47] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 232–247, London, UK, 2000. Springer-Verlag.

[48] Yli Falcone, Jean-Claude Fernandez, and Laurent Mounier. Enforcement monitoring wrt. the safety-progress classification of properties. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing - Software Verification and Testing Track*, 2009.

[49] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, Chichester, WS, UK, 1966.

[50] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19:215–261, September 2009.

[51] Michael Frazier, Sally Goldman, Nina Mishra, and Leonard Pitt. Learning from a consistently ignorant teacher. In *Proceedings of the seventh annual conference on Computational learning theory*, COLT '94, pages 328–339, New York, NY, USA, 1994. ACM.

[52] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17:591–603, June 1991.

[53] Mark Gabel and Zhendong Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 339–349, New York, NY, USA, 2008. ACM.

[54] Mark Gabel and Zhendong Su. Symbolic mining of temporal specifications. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 51–60, New York, NY, USA, 2008. ACM.

[55] Mark Gabel and Zhendong Su. Online inference and enforcement of temporal properties. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 15–24, New York, NY, USA, 2010. ACM.

[56] B.R. Gaines. Behaviour/structure transformations under uncertainty. *International Journal of Man-Machine Studies*, 8(3):337 – 365, 1976.

[57] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 318–329, New York, NY, USA, 2004. ACM.

[58] Anup K. Ghosh, Christoph Michael, and Michael Schatz. A real-time intrusion detection system based on learning program behavior. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, RAID '00, pages 93–109, London, UK, 2000. Springer-Verlag.

[59] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[60] E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978.

[61] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[62] J. Gomez. Self adaptation of operator rates in evolutionary algorithms. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, 2004.

[63] G. Gonenc. A method for the design of fault detection experiments. *IEEE Trans. Comput.*, 19:551–558, June 1970.

[64] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[65] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theor. Comput. Sci.*, 411:4029–4054, October 2010.

[66] 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 Comput. Surv.*, 41:9:1–9:76, February 2009.

[67] Philip Hingston. A genetic algorithm for regular inference. In *Genetic and Evolutionary Computation Conference*, 2001.

[68] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):pp. 13–30, 1963.

[69] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6:151–180, August 1998.

[70] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[71] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. In *IEEE International Test Conference (ITC), Charlotte, NC, September 30 - October*, page 2003. IEEE Computer Society, 2003.

[72] Kenneth L. Ingham, Anil Somayaji, John Burge, and Stephanie Forrest. Learning dfa representations of http for protecting web applications. *Comput. Netw.*, 51:1239–1255, April 2007.

[73] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. Technical report.

[74] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. Comparing approaches to mining source code for call-usage patterns. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 20, Washington, DC, USA, 2007. IEEE Computer Society.

[75] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[76] Charles W. Kastner. A hierarchy of languages, logics, and mathematical theories, March 2003.

[77] Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM*, 41:67–95, January 1994.

[78] Bill Keller and Rüdi Lutz. Evolving Stochastic Context-Free Grammars from Examples Using a Minimum Description Length Principle. In *Paper presented at the Workshop on Automata Induction Grammatical Inference and Language Acquisition, ICML-97*, pages 09–7, 1997.

[79] A. N. Kolmogorov. Three approaches to the quantitative definition of information. 1965.

[80] Takeshi Koshiba, Erkki Mäkinen, and Yuji Takada. Learning strongly deterministic even linear languages from positive examples. In *Proceedings of the 6th International Conference on Algorithmic Learning Theory*, ALT '95, pages 41–54, London, UK, 1995. Springer-Verlag.

[81] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: inferring the specification within. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[82] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[83] Kevin Lang. Random dfa's can be approximately learned from sparse uniform examples. In *In Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pages 45–52. ACM Press, 1992.

[84] 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 *Proceedings of the 4th International Colloquium on Grammatical Inference*, pages 1–12, London, UK, 1998. Springer-Verlag.

[85] M.M. Lankhorst. A genetic algorithm for the induction of pushdown automata. In *Evolutionary Computation, 1995., IEEE International Conference on*, volume 2, pages 741 –746 vol.2, nov-1 dec 1995.

[86] Choonghwan Lee, Feng Chen, and Grigore Roşu. Mining parametric specifications. In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, 2011. to appear.

[87] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[88] Martin Leucker. Learning meets verification. In *Proceedings of the 5th international conference on Formal methods for components and objects*, FMCO'06, pages 127–151, Berlin, Heidelberg, 2007. Springer-Verlag.

[89] Keqin Li and Muzammil Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *In FORTE, volume 4229 of LNCS*, pages 436–450. Springer, 2006.

[90] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. In *DAC '10: Proceedings of the 47th Design Automation Conference*, pages 755–760, New York, NY, USA, 2010. ACM.

[91] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, 30(5):306–315, 2005.

[92] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12:19:1–19:41, January 2009.

[93] Lee Lin and Michael D. Ernst. Improving the adaptability of multi-mode systems via program steering. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 206–216, Boston, MA, USA, July 12–14, 2004.

[94] Lee Chuan Lin. Program steering : Improving adaptability and mode selection via dynamic analysis. Master's thesis, Massachusetts Institute of Technology, 2004.

[95] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM.

[96] D. Lo, G. Ramalingam, V.P. Ranganath, and K. Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 62 –71, October 2009.

[97] David Lo and Siau-Cheng Khoo. Quark: Empirical assessment of automaton-based specification miners. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 51–60, Washington, DC, USA, 2006. IEEE Computer Society.

[98] David Lo and Siau-Cheng Khoo. Mining patterns and rules for software specification discovery. *Proc. VLDB Endow.*, 1(2):1609–1616, 2008.

[99] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469, New York, NY, USA, 2007. ACM.

[100] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining past-time temporal rules from execution traces. In *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*, pages 50–56, New York, NY, USA, 2008. ACM.

[101] David Lo, Siau-Cheng Khoo, and Chao Liu. Mining temporal rules for software maintenance. *J. Softw. Maint. Evol.*, 20(4):227–247, 2008.

[102] David Lo, Leonardo Mariani, and Mauro Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 345–354, New York, NY, USA, 2009. ACM.

[103] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Inferring state-based behavior models. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 25–32, New York, NY, USA, 2006. ACM.

[104] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, New York, NY, USA, 2008. ACM.

[105] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *SIGOPS Oper. Syst. Rev.*, 41:103–116, October 2007.

[106] Simon M. Lucas and T. Jeff Reynolds. Learning dfa: evolution versus evidence driven state merging. In *IEEE Congress on Evolutionary Computation (1)*, pages 351–358, 2003.

[107] Simon M. Lucas and T. Jeff Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27:1063–1074, July 2005.

[108] Erkki Mäkinen. A note on the grammatical inference problem for even linear languages. *Fundam. Inf.*, 25:175–181, February 1996.

[109] Erkki Mäkinen and Tarja Systä. MAS - an interactive synthesizer to support behavioral modelling in uml. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.

[110] Oded Maler and Amir Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118:316–326, May 1995.

[111] Florent Masseglia, Maguelonne Teisseire, and Pascal Poncelet. Sequential pattern mining: A survey on issues and approaches. *Encyclopedia of Data Warehousing and Mining*, 2005.

[112] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42:46–55, September 2009.

[113] Kevin Murphy. Passively learning finite automata. Technical report, 1996.

[114] Amashini Naidoo and Nelishia Pillay. Evolving pushdown automata. In *Proceedings of the 2007 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, SAICSIT '07, pages 83–90, New York, NY, USA, 2007. ACM.

[115] Amashini Naidoo and Nelishia Pillay. Using genetic programming for turing machine induction. In *Proceedings of the 11th European conference on Genetic programming*, EuroGP'08, pages 350–361, Berlin, Heidelberg, 2008. Springer-Verlag.

[116] S. Naito and M. Tsunoyama. Fault detection for sequential machinesby transition-tours. *Proc. FTCS (Fault Tolerant Comput. Syst.)*, pages 238–243, 1981.

[117] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. *SIGSOFT Softw. Eng. Notes*, 27:229–239, July 2002.

[118] Nattee Niparnan and Prabhas Chongstitvatana. An improved genetic algorithm for the inference of finite state machine. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 189–, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[119] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3:58–62, March 2005.

[120] J. Oncina, P. Garca, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *Ieee Transactions on Pattern Analysis and Machine Intelligence*, pages 448–458, 1993.

[121] J Oncina and P. Garcia. *Inferring regular languages in polynomial update time*, chapter -. World Scientific Publishing, 1991.

[122] H.M. Pandey. Context free grammar induction library using genetic algorithms. In *Computer and Communication Technology (ICCCT), 2010 International Conference on*, pages 752 –758, sept. 2010.

[123] Rajesh Parekh and Vasant G. Honavar. Learning dfa from simple examples. *Mach. Learn.*, 44:9–35, July 2001.

[124] J. D. Patrick and K.E. Chong. Real-time inductive inference for analysing human behaviour. *International Joint Conference on AI (IJCAI'91), Workshop number 6 on Integrating AI into Databases, Sydney*, 1991.

[125] Pravin Pawar and G. Nagaraja. Regular grammatical inference: A genetic algorithm approach. In *Proceedings of the 2002 AFSS International Conference on Fuzzy Systems. Calcutta: Advances in Soft Computing*, AFSS '02, pages 429–435, London, UK, 2002. Springer-Verlag.

[126] J. Pei, H. Wang, J. Liu, K. Wang, Jianyong Wang, and P.S. Yu. Discovering frequent closed partial orders from strings. *Knowledge and Data Engineering, IEEE Transactions on*, 18(11):1467 –1481, nov. 2006.

[127] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7:225–246, November 2001.

[128] Leonard Pitt. Inductive inference, dfas, and computational complexity. In Klaus Jantke, editor, *Analogical and Inductive Inference*, volume 397 of *Lecture Notes in Computer Science*, pages 18–44. Springer Berlin / Heidelberg, 1989.

[129] Corina S. Păsăreanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Form. Methods Syst. Des.*, 32:175–205, June 2008.

[130] Harald Raffelt, Tiziana Margaria, Bernhard Steffen, and Maik Merten. Hybrid test of web applications with webtest. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, TAV-WEB '08, pages 1–7, New York, NY, USA, 2008. ACM.

[131] Harald Raffelt and Bernhard Steffen. Learnlib: A library for automata learning and experimentation. In Luciano Baresi and Reiko Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 377–380. Springer Berlin / Heidelberg, 2006.

[132] Harald Raffelt, Bernhard Steffen, and Therese Berg. Learnlib: a library for automata learning and experimentation. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 62–71, New York, NY, USA, 2005. ACM.

[133] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. In *Proceedings of the 3rd international Haifa verification conference on Hardware and software: verification and testing*, HVC'07, pages 136–152, Berlin, Heidelberg, 2008. Springer-Verlag.

[134] A. Raman, Peter Andreae, and J. Patrick. A beam search algorithm for pfsa inference. *Pattern Anal. Appl.*, 1(2):121–129, 1998.

[135] A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In *International Conference on Machine Learning*, 1997.

[136] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 123–134, New York, NY, USA, 2007. ACM.

[137] John Rushby. Verified software: Theories, tools, experiments. chapter Automated Test Generation and Verified Software, pages 161–172. Springer-Verlag, Berlin, Heidelberg, 2008.

[138] K. K. Sabnani and A. T. Dahbura. A protocol testing procedure. *Computer Networks and Isdn Systems*, 1988.

[139] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185:15–45, October 1997.

[140] Yasubumi Sakakibara and Mitsuhiro Kondo. Ga-based learning of context-free grammars using tabular representations. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pages 354–360, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[141] Yasubumi Sakakibara and Hidenori Muramatsu. Learning context-free grammars from partially structured examples. In *Proceedings of the 5th International Colloquium on Grammatical Inference: Algorithms and Applications*, pages 229–240, London, UK, 2000. Springer-Verlag.

[142] Juan Sánchez and José Benedí. Statistical inductive learning of regular formal languages. In Rafael Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Computer Science*, pages 130–138. Springer Berlin / Heidelberg, 1994.

[143] V. Sai Sathyanarayan, Pankaj Kohli, and Bezawada Bruhadeshwar. Signature generation and detection of malware families. In *Proceedings of the 13th Australasian conference on Information Security and Privacy*, ACISP '08, pages 336–349, Berlin, Heidelberg, 2008. Springer-Verlag.

[144] Markus Schwehm and Alexander Ost. Inference of stochastic regular grammars by massively parallel genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 520–527, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[145] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–, Washington, DC, USA, 2001. IEEE Computer Society.

[146] Muzammil Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Software Components to support Integration Testing.* PhD thesis, Institut Polytechnique de Grenoble, Grenoble, France, December 2008.

[147] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 207–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[148] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and integration of parameterized components through testing. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer Berlin / Heidelberg, 2007.

[149] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning parameterized state machine model for integration testing. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 755–760, Washington, DC, USA, 2007. IEEE Computer Society.

[150] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 174–184, New York, NY, USA, 2007. ACM.

[151] Guoqiang Shu, Yating Hsu, and David Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *Proceedings of the 28th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, FORTE '08, pages 299–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[152] Guoqiang Shu and D. Lee. Testing security properties of protocol implementations - a machine learning based approach. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on*, page 25, june 2007.

[153] Ray J. Solomonoff. A formal theory of inductive inference. part i. *Information and Computation/information and Control*, 7:1–22, 1964.

[154] Ray J. Solomonofi. A formal theory of inductive inference. part ii. 1964.

[155] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-driven development with the jabc. In *Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*, HVC'06, pages 92–108, Berlin, Heidelberg, 2007. Springer-Verlag.

[156] Andreas Stolcke and Stephen M. Omohundro. Best-first model merging for hidden markov model induction. Technical report, 1994.

[157] Yuji Takada. Grammatical inference for even linear languages based on control sets. *Information Processing Letters*, 28(4):193 – 199, 1988.

[158] M. Tomita. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pages 105–108, Ann Arbor, Michigan, 1982.

[159] B. A. Trakhtenbrot and IA. M. Barzdin. *Finite automata : behavior and synthesis [by] B. A. Trakhtenbrot and Ya. M. Barzdin. Translated from the Russian by D. Louvish. English translation edited by E. Shamir and L. H. Landweber.* North-Holland Pub. Co.; American Elsevier, Amsterdam, New York,, 1973.

[160] Kewei Tu and Vasant Honavar. Unsupervised learning of probabilistic context-free grammar using iterative biclustering. In *Proceedings of the 9th international colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '08, pages 224–237, Berlin, Heidelberg, 2008. Springer-Verlag.

[161] `http://libalf.informatik.rwth-aachen.de/`.

[162] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27:1134–1142, November 1984.

[163] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, London, 2 edition, 1979.

[164] M. P. Vasilevskii. Failure diagnosis of automata. *Kybernetika*, 1973.

[165] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. *SIGSOFT Softw. Eng. Notes*, 30:273–282, September 2005.

[166] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Improving dynamic software analysis by applying grammar inference principles. *J. Softw. Maint. Evol.*, 20:269–290, July 2008.

[167] Neil Walkinshaw, Kirill Bogdanov, and Ken Johnson. Evaluation and comparison of inferred regular grammars. In *Proceedings of the 9th international colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '08, pages 252–265, Berlin, Heidelberg, 2008. Springer-Verlag.

[168] C. S. Wallace and D. M. Boulton. An Information Measure for Classification. *Computer Journal*, 11(2):185–194, August 1968.

[169] C. S. Wallace and D. L. Dowe. Minimum message length and kolmogorov complexity. *Computer Journal*, 42:270–283, 1999.

[170] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44, New York, NY, USA, 2007. ACM.

[171] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 461–476. Springer Berlin / Heidelberg, 2005.

[172] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis, 2002.*, 2002.

[173] P. Wyard. Context free grammar induction using genetic algorithms. In *Grammatical Inference: Theory, Applications and Alternatives, IEE Colloquium on*, pages P11/1 –P11/5, apr 1993.

[174] Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 1100–1101. Springer Berlin / Heidelberg, 2004.

[175] Jinlin Yang and David Evans. Automatically inferring temporal properties for program evolution. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 340–351, Washington, DC, USA, 2004. IEEE Computer Society.

[176] Jinlin Yang and David Evans. Dynamically inferring temporal properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, pages 23–28, New York, NY, USA, 2004. ACM.

[177] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 282–291, New York, NY, USA, 2006. ACM.