AUTOMATA BASED MONITORING AND MINING OF EXECUTION TRACES

VOLUME I OF II

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2014

By
Giles Reger
School of Computer Science

Contents

Word Count: 81,789 (with Appendix: 94,543)

List of Tables

List of Figures

Abstract

Automata Based Monitoring and Mining of Execution Traces
Giles Reger
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2014

This thesis contributes work to the fields of runtime monitoring and specification mining. It develops a formalism for specifying patterns of behaviour in execution traces and defines techniques for checking these patterns in, and extracting patterns from, traces. These techniques represent an extension in the expressiveness of properties that can be efficiently and effectively monitored and mined.

The behaviour of a computer system is considered in terms of the actions it performs, captured in execution traces. Patterns of behaviour, formally defined in trace specifications, denote the traces that the system should (or should not) exhibit. The main task this work considers is that of checking that the system conforms to the specification i.e. is correct. Additionally, trace specifications can be used to document behaviour to aid maintenance and development. However, formal specifications are often missing or incomplete, hence the mining activity.

Previous work in the field of runtime monitoring (checking execution traces) has tended to either focus on *efficiency* or *expressiveness*, with different approaches making different trade-offs. This work considers both, achieving the expressiveness of the most expressive existing tools whilst remaining competitive with the most efficient. These elements of expressiveness and efficiency depend on the *specification formalism* used. Therefore, we introduce quantified event automata for describing patterns of behaviour in execution traces and then develop a range of efficient monitoring algorithms.

To monitor execution traces we need a formal description of expected behaviour. However, these are often difficult to write - especially as there is often a lack of understanding of *actual* behaviour. The field of specification mining aims to *explain* the behaviour present in execution traces by extracting specifications that conform to those traces. Previous work in this area has primarily been limited to simple specifications that do not consider data. By leveraging the quantified event automata formalism, and its efficient trace checking procedures, we introduce a generate-and-check style mining framework capable of accurately extracting complex specifications.

This thesis, therefore, makes separate significant contributions to the fields of runtime monitoring and specification mining. This work generalises and extends existing techniques in runtime monitoring, enabling future research to better understand the interaction between expressiveness and efficiency. This work combines and extends previous approaches to specification mining, increasing the expressiveness of specifications that can be mined.

Declaration

Copyright

Acknowledgements

Firstly I would like to thank my supervisors - Howard Barringer and David Rydeheard. I would like to thank Howard for introducing me to the field of runtime verification, his passion for the subject being evident. I would like to thank both Howard and David for their advice, conversation and support, without which I would not have completed this work.

I would also like to extend my thanks to Klaus Havelund and Yliès Falcone for the many discussions. Special thanks go to Klaus for always making me explain myself.

I would like to thank all of my family and friends who have patiently listened to me talk about my research. I'm afraid I am not planning on stopping.

Lastly I would like to thank my wife, Clare, for looking after me. I would never have finished without you telling me when to stop, and would not have eaten or slept properly without you making sure I did.

Chapter 1

Introduction

This thesis is about patterns of behaviour in computer systems; specifying, checking and extracting them. We present a new automata-based formalism for describing these patterns that is designed to be well-suited for the activities of monitoring and mining execution traces.

These patterns of behaviour capture one notion of system correctness, i.e. what the computer system does or is supposed to do, and can therefore be used to check correctness and document the system. These issues are important. There are numerous examples of cases where identifiable faults in computer systems have led to disastrous outcomes, even loss of life. We can attempt to identify and remove these faults by comparing formally specified, required or prohibited, patterns of behaviour against the actual behaviour of the system. Additionally in safety-critical systems it is common to *monitor* behaviour to ensure that unsafe actions are not taken. Formal descriptions of system behaviour can also aid other tasks, such as maintenance and development, by providing an understanding of what the system does. By describing *patterns* of behaviour we can capture complex relationships between elements of a system.

The notion of computer system behaviour can be described in various ways. One approach is to specify input-output functionality using logical expressions. Another is to build an abstract model relating program states. Here we consider a different notion of behaviour; the ordering of *events* that occur in a system. Instead of giving predicates on the contents of program variables, this approach abstracts a system as a set of event sequences, or execution traces, and specifies behaviour in terms of predicates on these. To define this abstraction it is necessary to relate these events to actions performed by the system.

A trace specification describes a pattern of behaviour and denotes a set of execution traces that contain this behaviour. There are several different methods that can be used to describe trace specifications. A natural choice is to use concepts from formal language theory where patterns of symbols are described using different forms of automata (or equivalently captured by formal grammars). Alternatively, we can consider temporal logics where temporal rules describe orderings between events. Both of these fields allow us to include symbols carrying data; in language theory, automata for data words have been suggested, and in logic we have the standard use of variables and quantification.

To inspect the traces of a system we can either extract these directly from the code (as done in symbolic execution), build a model of the system and explore this (as done in model checking) or run the system and record traces (as done in trace analysis). We take the approach of trace analysis. Whilst this approach can only sample a selection of possible executions it has the advantage of capturing the actual behaviour of the system, removing issues such as state explosion. Additionally, runtime traces record the actual data values used instead of taking an over-approximation. In trace analysis the abstraction activity is achieved through *instrumentation*; a mapping, from system activities to events, is used to instrument the system to output these events whenever the associated activity occurs.

Two interesting trace analysis activities are *monitoring* and *mining*. Both consider the satisfaction of a specification with respect to a set of traces but differ in their aim; monitoring performs deduction, checking whether a trace specification is satisfied, whereas mining performs induction, constructing a specification that is satisfied.

Trace monitoring is a verification activity that asks whether a trace is in the set of traces denoted by the given specification. However, as we are dealing with runs of real systems it would be preferable to check this whilst the system is running to detect errors as early as possible. This is the aim of *runtime verification* or *runtime monitoring*. The term monitoring refers to the fact that not only do we check correctness but we can also take actions to recover from failure. The activity of runtime verification is distinct from testing as it inspects patterns of behaviour of the running system using a formal notion of correctness. As this is a runtime activity we are concerned with *efficiency*, in terms of overhead and interference. Monitoring techniques also need to be *incremental* i.e. be able to make decisions as execution progresses.

Runtime monitoring can therefore complement other methods for checking the correctness of computer systems. As this process only considers the current trace, it is far more tractable than techniques such as model-checking, but it also gives stronger guarantees than testing as it is based on a formal notion of correctness.

The motivation behind mining traces is that computer systems are usually extensively, and sometimes systematically, tested to establish correctness, yet often lack a formal notion of this correctness. The general idea is to use traces extracted from a system to construct a specification, based on the assumption that the system is (mostly) correct. There exist several methods for mining or learning a model or specification from a set of traces. Some approaches allow interaction with an oracle that has knowledge of the true model, but this is outside the scope of passive trace analysis. Another approach from language theory uses the notion of congruence classes of strings to reduce a model accepting all traces to some minimal form. We take a different approach that uses a generate-and-check strategy where candidate specifications are constructed and filtered using the given traces. A key to this approach is that the checking process takes the form of runtime verification.

Specifications extracted from execution traces have been used for program comprehension, test generation, program verification and to support intrusion and malware detection.

This research considers a new formalism for specifying trace specifications suitable for the activities of monitoring and mining. Whilst there exist expressive trace specification languages (such as first-order linear temporal logic) these formalisms do not interact well with the requirements of monitoring or mining. Namely the need for an efficient, incremental, trace checking algorithm. The new formalism is *expressive*, combining the automata from language theory with the concept of quantification from logic, and admits a range of *efficient* monitoring algorithms.

In the rest of this chapter we present the contributions of this thesis. We begin with a brief synopsis of the work (Sec. 1.1) before discussing the contributions in context (Sec. 1.2) and finish with an overview of the thesis structure (Sec. 1.3).

1.1 The three Es

In discussion with Klaus Havelund [Hav13a], we have proposed three non-orthogonal dimensions for discussing formalisms for trace specifications. We consider the **E**xpressiveness of the formalism, the **E**fficiency of the trace-checking method, and the **E**legance of specifications written in that formalism.

We say that these dimensions are non-orthogonal as there are dependencies; more expressive systems will typically take more work to carry out trace checking and be more difficult to write concise specifications in. This space is not well understood. Previously the development of runtime monitoring systems has tended to either focus primarily on expressiveness or efficiency. As a consequence, the most expressive systems tend not be very efficient, and the most efficient systems have certain restrictions to their expressiveness. Mining techniques generally focus on simple specification formalisms and the generate-and-check approach used here depends on an efficient checking method. Elegance is an often overlooked, but important, property.

The main motivation of this work is therefore that, for both monitoring and mining, there are points in this space that are useful yet unexplored. This research presents quantified event automata as a new point in this space, placing itself between the most expressive and most efficient techniques. Here we discuss our work along these three dimensions.

1.1.1 Expressiveness

There are two aspects to the expressiveness dimension. On one hand we can ask the standard language theory questions about where in the Chomsky hierarchy we belong. We are also interested in the treatment of data in events, and can view this treatment as a measure of expressiveness.

It is natural to consider data values occurring in events. For example, an event might capture the fact that a file "out.txt" is opened and can therefore be written `open`("out"). In the field of logic we would usually describe languages that capture specification over such events as first-order, however in trace analysis the common term is *parametric* as events are viewed as taking data parameters. We can, of course, consider parametric events as propositional symbols and only consider the Chomsky hierarchy view (extended to infinite alphabets) but this is less informative than separating the two.

We distinguish between two different ways that data can occur:

- In a *quantified* manner. For example, the property that *every* file that is opened is eventually closed. If we had events `open`(f) and `close`(f) then f would be quantified and we only consider the ordering of `open` and `close` for the same values for f.

- In a *free* manner. For example, the property that a countdown latch (used in concurrency) must only decrease and waiting threads can only proceed when this count reaches zero treats the count value in a free manner. The value assigned to c in `change_count`(c) would be free to change and we would use the guard $c = 0$ to decide whether `proceed` could occur.

The difference being that in the first case we *fix* the data values and then consider behaviour, whilst in the second there is an interaction between data and behaviour. Quantified event automata combine logical quantifications with a form of extended finite state machine. Therefore, both universally and existentially quantified data is captured in the logical component whilst free data is captured using guards and assignments labeling an automaton's transitions.

We note that we only consider quantified data in our mining work, although our generate-and-check framework can be extended to include free data and we outline this extension in a discussion of future work at the end of the thesis. The treatment of data in specification mining has only begun to draw attention recently and whilst there are a few promising techniques [BJR06, HSJC12, AHK+12, BHLM13, WTD13] considering free data, only a few [LCH+09, LRRV12, LCR11] consider quantified data and none combine the two fully.

In the field of runtime monitoring a common trace specification formalism is linear temporal logic [Pnu77], which is usually defined over infinite traces. However, it is standard to define a finite-trace version [BLS10] as, in trace analysis, we are only concerned with *finite* traces (computer systems can only run for a finite time). We only consider the word problem of quantified event automata for finite traces. In general, this work is only concerned with finite things as these are the only things we can observe at runtime. As we do not concern ourselves with infinite traces we cannot address the notions of true liveness or fairness. This is because of our initial decision that we only consider the runtime actions of a system.

Therefore, quantified event automata capture both quantified and free forms of data but as they are based on automata the underlying language is regular. We make this distinction, even though the expressive power of quantified event automata is Turing complete due to the extended finite state machine component, as it has repercussions for *elegance* discussed below.

1.1.2 Efficiency

For a specification formalism to be useful for both monitoring and mining (specifically our generate-and-check approach to mining) we need an efficient trace checking method.

One of the main issues of efficiency comes from the appropriate handling of data; we do not need to consider ambiguous parsing or 'past time' formulas as in other languages. At a high level, whilst checking a trace we will construct and update a structure that encodes the current status of the trace with respect to the specification. Different parts of this structure will relate to different data values and when we process a parametric event we need to decide which part of the structure we need to update based on the data values. To be efficient we need to do this without searching the whole structure.

The two most efficient runtime verification systems (JavaMOP and TraceMatches, discussed later) both make use of a technique called *trace slicing*, which allows them to develop efficient indexing strategies. The general idea is to *slice* the trace with respect to values given to (in their case implicitly) quantified variables and then consider each slice as data-free.

Previous approaches employing trace slicing make certain design decisions that limit expressiveness. Quantified event automata takes a trace slicing approach, formalising the notion of quantification, and maintaining a higher level of expressiveness by including free variables among other things.

However, the notion of trace slicing is not inherently incremental (which is necessary for monitoring) as it requires us to know the domains of quantified variables up-front, which are typically extracted from the trace. We therefore have to develop an incremental semantics that can keep track of partial information about data values in the trace.

In the generate-and-check mining approach we do not require incremental checking but it is common to check many specification patterns at the same time. To do this efficiently we introduce a mechanism for combining these patterns together to check them all at once.

1.1.3 Elegance

The dimension of elegance is not as measurable as that of expressiveness or efficiency and treating the notion rigorously is beyond the scope of this work. Later (Sec. 2.2.4) we discuss how elegance might be measured but in our evaluation we will only briefly consider succinctness as one aspect of elegance. Here we informally argue for the elegance of our approach. By elegant we mean easy to understand and concise, properties that are often at odds.

We firstly note that the two components of quantified event automata are natural. Automata are a commonly used for capturing properties and are typically more concise than regular expressions for a useful class of properties. The notion of quantification in logic is standard.

There are two different styles of writing trace specifications: in the *validation* style we describe the minimal behaviour required for success and therefore want to ignore extra symbols, whereas in *violation* style we describe exactly the allowed behaviours and therefore want extra symbols to cause failure. Quantified event automata can capture both styles effectively using two different kinds of state.

Approaches that deal with quantified data typically implicitly quantify variables. However some approaches [Sto10, GDPT13] encode this information into their automata formalism. By separating quantification from the automata representation of the property and explicitly quantifying variables we clearly separate the concerns of quantified and free behaviour. Additionally, we include both universal and existential quantification.

However, we accept that elegance is not only subjective, it is also dependent on the domain of application. For example, there is a notion of *connectedness* that is used in other systems [MJG$^+$11] to restrict the data values of interest. We encode connectedness in quantified event automata using the notion of global guards. This is necessary as specifications that rely on this notion but are written without it can be complex and difficult to understand.

Finally, although most languages are translated into a form of automata for checking, other formalisms may offer more elegant methods for specifying some kinds of property. For example, although it is possible to encode context-free properties using quantified event automata these can be inelegant compared to other approaches that make use of context-free grammars. To

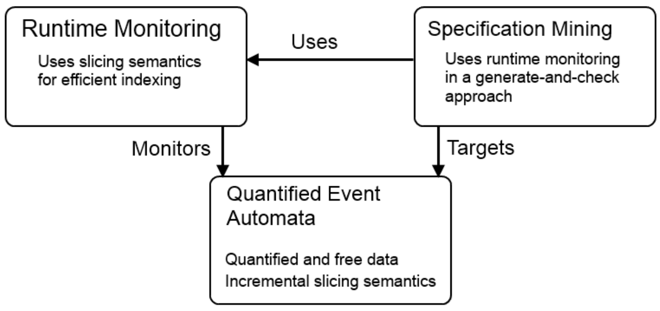

Runtime Monitoring
Uses slicing semantics for efficient indexing
Uses
Specification Mining
Uses runtime monitoring in a generate-and-check approach
Monitors
Targets
Quantified Event Automata
Quantified and free data
Incremental slicing semantics

Figure 1.1: The three components of this work and how they relate.

address this, we discuss possible extensions to context-free grammars in further work at the end of this thesis. We should also consider abstraction. Linear temporal logic (LTL) is a common specification formalism used in runtime monitoring and can concisely capture complex patterns of behaviour. As quantified event automata operate at the lowest level of abstraction, i.e. event transitions, some of these properties will require complex transition structures. However we can translate directly between LTL and quantified event automata (without free variables).

1.1.4 Summary

In summary, the aim of this work is to develop an expressive specification formalism that can describe trace specifications for use in runtime monitoring and specification mining. This requires us to explore a new point along the expressiveness-efficiency dimension. To be clear, we are aiming to develop a system that may not be more, or as, efficient as the most efficient techniques, but achieves greater expressiveness whilst remaining competitive.

Figure 1.1 illustrates how the different elements of this work are related. The runtime monitoring algorithm relies on the slicing-based semantics of quantified event automata to be efficient. The specification mining technique targets quantified event automata and makes use of the efficient runtime monitoring algorithm to check possible specifications.

1.2 Contributions

Here we consider the contributions made by this work and the potential impact they may have.

Quantified event automata (QEA)

We have developed quantified event automata as a new formalism for trace specifications that is suitable for both monitoring and mining.

Our first contribution in this space is a full account of the trace slicing approach that formalises the notions of quantification and acceptance. Previous accounts [AAC+05, MJG+11] do not consider how these two concepts interact. Our second contribution is an extension of trace slicing to allow free variables and existential quantification. Most runtime monitoring tools do not allow us to capture data in this way, with the notable exception of rule-based approaches [BRH08]. No mining approaches consider existential quantification. We also present a wide range of example quantified event automata taken from different domains.

Runtime monitoring

We develop and optimise an efficient monitoring algorithm for QEA. Firstly, this requires us to define a small-step (incremental) semantics for QEA. This development highlights the need for the *maximality* concept used elsewhere [MJG+11] and provides a new description that emphasizes the role of binding ordering more clearly than previous work. Additionally, we prove that this small-step semantics is equivalent to the non-incremental semantics.

We make use of four different verdicts to indicate the correctness of a trace with respect to a quantified event automaton, differentiating between verdicts that may change in the future and those that cannot. To do this we define a notion of strong states, from which either success or failure is unreachable. Distinguishing between these different verdicts is not novel [BRH08], but our method for automatically extracting this information from the specification is.

We present four alternative indexing techniques for quantified event automata. Two of these are natural exploitations of structural properties and the third is a straightforward adaptation of the approach taken by JavaMOP [MJG+11]. Our fourth is novel, adapting the symbol-based approach proposed by Dwyer et al. [PDE12] to allow for the notion of maximality.

Finally, we introduce the notion of algorithm selection based on structural properties of the specification. This general idea can be applied in any tool where complex mechanisms are required for dealing with certain parts of the language that are not always used in specifications. We found that this technique was essential for monitoring simple properties efficiently.

Specification mining

We introduce a new method for mining quantified parametric specifications i.e. QEA. We are the first to adapt the generate-and-check specification mining approach to this setting, which we achieve through the use of trace slicing.

We make general contributions to the generate-and-check approach which could also be applied in a propositional setting. Specification mining of this form usually has to check many patterns against a trace. To optimise this process we have introduced *pattern-checkers* which compile these patterns into a single structure, allowing us to process the trace once only. A common method used to refine the results of specification mining is to *combine* extracted patterns to form larger patterns. Previous work showed that using standard automata for mining means that extracted concrete patterns are imprecise leading to generalised combinations. To combat this we introduce *open automata* to soundly and precisely combine extracted patterns.

There has been previous work dealing with noise, or imperfections, in execution traces caused by bugs in the original system. However, these techniques either assume restrictive properties about the patterns used for mining or cannot be applied to the generate-and-check approach. We introduce a new technique for dealing with these so-called imperfect traces that measures the edit-distance between the trace and pattern. This technique is presented for the propositional case but could easily be lifted to the parametric one. This method can also suggest potential corrections as the edits relate to changes to the code. Therefore there are possible applications in the fields of bug finding and fixing.

Evaluation

This work has involved the development of two tools: one for runtime monitoring, the other for specification mining. To establish the utility of our techniques, and answer specific research questions about these tools, we developed extensive evaluation frameworks. To help this effort we have developed two sets of example specifications: one set describes the correct usage of parts of the Java standard library, and the other describes a hypothetical planetary rover case study inspired by discussions with Klaus Havelund from NASA's Jet Propulsion Laboratory.

1.2.1 Publications

The following describes publications derived from, or based on, work in this thesis.

A Tutorial on Runtime Verification [FHR13]. This invited book chapter accompanied a summer school course given at Marktoberdorf 2012 by Klaus Havelund. The chapter gives a thorough introduction to runtime verification and an overview of key tools in the area. Concepts from this chapter are used in Chapter 2 and examples are used throughout the thesis.

Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors [BFH$^+$12]. Presented at the 18th international symposium on formal methods (FM 2012), this paper introduces the quantified event automata specification formalism within the context of runtime verification. Chapters 3 and 5 build on the ideas in this paper.

A Pattern-Based Approach to Parametric Specification Mining [RBR13b]. Presented at the 28th international conference on automated software engineering (ASE 2013), this paper presents a new technique for extracting parametric specifications from program traces. This paper forms substantial parts of chapters 8 and 10.

Automata-based Pattern Mining from Imperfect Traces [RBR13a]. Presented at the second international workshop on software mining, this paper explores a novel idea for dealing with noise when mining specifications from so-called imperfect traces using automata-based pattern mining techniques. The paper is a condensed version of Chapter 11.

1.3 Structure

The work is split into Part I, developing quantified event automata as a specification formalism, Part II, developing a runtime monitoring approach for quantified event automata, and Part III, which looks at how we can mine quantified event automata from traces.

Part I

Chapter 2 discusses the notion of program specification. We explore what a specification is and how it is used in runtime monitoring and specification mining. This chapter therefore introduces the relevant background material for the rest of the thesis.

Chapter 3 introduces quantified event automata as a formalism for parametric trace specifications. We build up the formalism by motivating the different components separately and present a big-step style semantics.

Chapter 4 explores quantified event automata as a specification formalism, addressing the complexity of its trace checking process and discussing its expressiveness.

Part II

Chapter 5 develops a small-step semantics for quantified event automata, allowing a runtime trace to be processed an event at a time. This allows for online monitoring and we present a basic monitoring algorithm for this based on the small-step semantics.

Chapter 6 optimises the runtime monitoring process. We first extend the notion of acceptance to be more informative and then explore extensions to the monitoring algorithm that should improve performance.

Chapter 7 evaluates the quantified event automata monitoring approach. This uses a hypothetical case study to address a number of research questions and then the DaCapo benchmark suite to evaluate the applicability of the approach to real-world systems.

Part II

Chapter 8 introduces our pattern-mining technique for a form of quantified event automata that does not include free variables. The different stages of the mining process are explained and brought together in a worked example.

Chapter 9 explores what makes a good pattern library. Our specification mining approach makes use of a predefined pattern library that will determine what specifications can be returned. The library developed here is used for evaluation.

Chapter 10 evaluates the pattern-based specification mining approach. We measure the accuracy of the technique borrowing the notions of precision and recall from the field of information retrieval. We make use of traces generated from predefined ground truths and traces extracted from the DaCapo benchmark suite.

Chapter 11 tackles the problem of imperfect traces in the propositional setting.

Finally we conclude in **Chapter 12** and discuss possible future work.

Part I

Specifications

Chapter 2

On specifications and their uses in monitoring and mining

This research is about writing and using formal specifications of patterns of behaviour in computer systems so we begin with a discussion of specifications. In the next chapter we will be building a new specification formalism, which will be used for runtime monitoring and specification inference. Therefore, we review specifications within the context of these two activities.

This chapter therefore gives the background material for the rest of this work. The aim is to motivate the quantified event automata specification formalism whilst presenting existing work in the areas of runtime monitoring and specification inference.

Structure. We begin by introducing general terms and notation necessary for the rest of the thesis (Sec. 2.1). Next we present a high-level discussion of what it means to specify system behaviour (Sec. 2.2). Here we identify the kinds of specifications we are interested in in this work. We then introduce runtime monitoring (Sec. 2.3) and how the requirements of runtime monitoring effect the choice of specification language. We consider existing monitoring tools with respect to our developed technique in Section.4.5. As well as manually writing system specifications we can automatically extract them from 'correct' systems. We cover the basic ideas behind the disparate field of specification inference (Sec. 2.4), focusing again on the kinds of specification formalisms used. The specific approach used in this work is discussed in Chapter 8 and this chapter also discusses alternative existing approaches and contrasts them to the selected approach.

2.1 Preliminaries

We begin by introducing notation and terminology used throughout this thesis. At this point we introduce general concepts related to automata as they are a key component of this work.

2.1.1 Notation

In this thesis we use the following standard notation.

We use the concepts of sets, lists functions and maps (partial functions with finite domain) throughout . The set of sets of objects of kind T will be written 2^T or $\mathcal{P}(T)$, the set of lists of objects of kind T will be written T^* and the set of functions (maps) from objects of kind T to objects of kind S will be written $T \to S$ ($T \rightharpoonup S$).

We use curly brackets ($\{\}$) to denote a set, angled brackets ($\langle\rangle$) to denote a list (or tuple), and square brackets ($[]$) to denote a map. A map will typically be written as $[x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots]$. We will use standard set operators, list concatenation will be denoted using '.' and the empty list will be denoted by ϵ. We will use comprehensions to define sets, lists and maps. For example, $[x \mapsto v \mid \textit{condition on } x \textit{ and/or } v]$ and $[(x \mapsto v) \in \theta \mid \textit{condition on } x \textit{ and/or } v]$ where θ is a map. Formal notions of map override ($\dagger$) and least upper bound ($\sqcup$) are introduced later (Sections 3.1.2 and 5.2).

When we introduce a tuple $\Gamma = \langle A, B, C\rangle$ we will sometimes use standard dot notation to access an element of that tuple, i.e. $\Gamma.A$, which denotes the value of A in the tuple Γ.

2.1.2 Automata

Automata, or state machines, are popular for writing temporal specifications as they give an immediate incremental checking procedure using their transition function. Here we present the standard notion of automata.

Definition 1 (Finite State Automata)**.** *A finite state automaton (FSA) is a tuple* $\langle Q, \Sigma, q_0, \delta, F\rangle$ *where* Q *is a finite set of states,* Σ *is a finite alphabet of symbols,* $q_0 \in Q$ *is an initial state,* $\delta \in Q \times \Sigma \times Q$ *is a set of transitions, and* $F \subseteq Q$ *is a set of final states.*

FSA denote regular languages and are closed under union, intersection and complement. A FSA is *deterministic* if for every state $q \in Q$ and symbol $a \in \Sigma$ there is at most one transition $(q, a, q') \in \delta$. We sometimes refer to a deterministic FSA as a DFA.

We lift the transition function of an automata to words (finite sequences of symbols), as is standard. This is defined inductively for a set of states S:

$$\begin{array}{lcl} \delta(S, \epsilon) & = & S \\ \delta(S, \tau.a) & = & \bigcup_{s \in S} \delta(\{s' \mid (s, a, s') \in \delta\}, \tau) \end{array}$$

Therefore the states reachable by word τ are given by $\delta(\{q_0\}, \tau)$. We say a state s' is *reachable* from a state s if there exists a word τ such that $s' \in \delta(\{s\}, \tau)$. A FSA *accepts* a word if a final state is reachable from the initial state i.e. $\delta(\{q_0\}, \tau) \cap F \neq \emptyset$.

A FSA is *complete* if for every state $q \in Q$ and symbol $a \in \Sigma$ there exists at least one transition $(q, a, q') \in \delta$. If a FSA is complete then given a state and symbol we can always generate at least one next state. It is often laborious to write complete state machines and we therefore have a notion of 'completing' an FSA and there are two methods for this:

- *Skip style*: This introduces a self-looping transition for each missing symbol, therefore if a transition cannot be taken the next symbol is *skipped.*

- *Next style*: This introduces a transition to an implicit non-final state (with no outgoing transitions) for each missing symbol, therefore we will fail if we cannot take a transition on the *next* symbol.

We will refer to an automata-based language that assumes states are skip/next style as having a skip/next semantics. When giving graphical representations of automata we will use shaded states to denote final states, and draw states as circles if they have a skip semantics and as rectangles if they have a next semantics. Note that later we intermix these two kinds of state in the same machine.

Note. *We have introduced notions of completion rather than assuming an automata is complete as the efficiency of checking will depend on the manner of completion. For example, if a skip style is used then if there are no transitions then no work is required; however, if the automata has been completed we would need to process these self-looping transitions. Therefore, we do not assume complete automata for pragmatic reasons.*

2.2 Specifying system behaviour

In this section we discuss the concepts and terms used to describe the behaviour of computer systems, whether it is actual or intended behaviour. As mentioned previously, our notion of system behaviour is not the only one and there is a large range of specification techniques that we do not consider. We have chosen this view as it represents a useful class of specification.

2.2.1 Computer systems as state machines

At the lowest level a computer system consists of a list of operations that carry out some computation by manipulating memory. We typically work at a higher level of abstraction. Operations are grouped together as *transitions* between sets of memory configurations, labelled as *states*. Transitions may relate to observable inputs/outputs or internal actions of interest, let us call these *events*. Theoretically, computer systems can represent infinite state machines (although limitations on memory size and the domains of data structures restrict this in practice). In some cases we are interested in the contents of a state, i.e. the values assigned to variables. In this case we might treat these values as semantic objects, for example strings or lists, which allows us to describe their properties. In other cases we only consider the transitions allowable at each state.

2.2.2 Specifying properties of computer systems

The *behaviour* of a system is given by the states the system passes through. When considering this view, there are two types of behaviour we may wish to specify: allowed states and allowed sequences of states.

State specifications

A typical state specification might be that x is always positive. This predicate divides the memory configurations of the program into two states: one where x is positive and one where it is not. Function pre and post conditions are a form of state specifications, as are class invariants. These all place a predicate on the contents of memory in a particular context. State specification denote the set of allowable states. Checking state specifications dynamically is straightforward; we insert assertions wherever the relevant parts of memory are accessed or updated. Statically we can ask whether unsafe states are reachable.

Trace specifications

A sequence of states describes the behaviour of a system over time. Rather than the sequence of states we are often interested in the sequence of *events* that controlled that behaviour. We call a sequence of events a *trace*, hence trace specifications. A trace specification denotes the set of allowable traces.

Traces can be finite or infinite, and can be over discrete or continuous time. If we consider computer systems statically it is common to consider the possible infinite traces, whereas dynamic analysis can only deal with finite traces. Continuous time is only relevant if the system consists of multiple concurrent parts and we do not consider this situation further here.

2.2.3 Including data

So far we have noted that states abstract sets of memory configurations and that events label transitions between these. However, often the data being manipulated by the system is important and we need to make it apparent in our model.

A standard way of doing this is to allow events to carry data, for example `set_name`("Giles"). As it is usual to write the data values as parameters we call specifications over such events *parametric*. As data domains are often infinite, the use of such parametric events often leads to infinite alphabets. To deal with this it is usual to represent sets of parametric events using variables, for example, `set_name`(n).

We note here that time can be considered as data i.e. we can add a timestamp to an event to allow us to reason about the time at which the event occurred.

Specifications may deal with data parameters in two orthogonal ways: in a *quantified* manner or in a *free* manner. We discuss these two views in the following (a specification can mix the two views).

Quantified-view on data

In this view we are concerned with the behaviour of (or related to) specific data values. For example, the property that a file should only be read when open considers the behaviour related to each file and can be specified by saying that event `open`(f) occurs before `read`(f) for every file f. We call this the *quantified* view as when written in English this kind of specification will talk about 'for every' or 'for some'.

Typestate [SY86] (discussed later) is a popular specification approach that takes a quantified view. However, typestates are typically over a single type of value, whereas in general we could consider multiple values. For example, "every lock can only be taken by one thread". Here we implicitly quantify over one lock and two threads. Dealing with properties involving multiple quantifications is inherently more complicated.

Free-view on data

In this view we are concerned with how data values effect behaviour. For example, the property that the temperature should be kept above 16 degrees Celsius but below 30 using a heater and a fan can be specified by saying that when `temperature`(t) occurs then if $t < 16$ then `heater_on` should occur or if $t > 30$ then `fan_on` should occur. Behaviour is captured by states entered, so we can also use data values to restrict transitions taken. For example, we can specify that a counter is strictly increasing by stating that if `count`(x) follows `count`(y) then $x > y$. We call this the *free* view as the values assigned to variables are free to change over time.

The most common specification formalism that takes a free-view on data is the *extended finite state machine*. Here transitions are labelled with enabling and update functions and there is a notion of assigning values to variables. There are other state machine extensions that manipulate data in this way [NSV04].

2.2.4 An elegant specification language

Previously (Sec. 1.1) we mentioned that *elegance* is a desirable trait in a specification language. Here we briefly discuss what elegance might mean, and how it could be measured. However, this work is not primarily concerned with elegance and later we only discuss it briefly. The notion of an elegant specification language could be considered in many ways, including:

- **Succinctness.** Intuitively a shorter specification is more elegant. If it takes fewer symbols to express an equivalent concept then there are fewer components to understand, hence it should be easier to understand. However, this is a rather crude view and the opposite may be the case. For example, we could encode all specifications in a concise integer form; this would be succinct but not elegant. Measuring succinctness is straightforward; we can count the number of symbols required to express a property.

- **Expressive power.** If a specification language contains symbols able to express a large amount then specifications written in that language should be succinct. Especially if the language provides capacity for *abstraction*, allowing complex concepts to be condensed for ease of understanding. We have formal notions of expressive power, allowing us to measure this. However, these do not necessarily capture notions such as abstraction.

- **Readability.** Separate from the notion of succinctness and expressive power is readability, i.e. how easy is it to discern the intended meaning from the specification. This can be dependent on both the complexity of the specification language, and the way in which the specification is presented. For example, one might expect small specifications to be easy to read in a graphical language (such as automata) but that this would fail to scale to larger specifications. There exist certain metrics [O11, BW08] for measuring readability, often based on user-studies. These will typically take a set of features, such as number of lines or identifier lengths, and compute a readability score.

- **Cognitive complexity.** It is possible to take a more structured approach. In the study of cognitive complexity [Str92, HBPS11, Rau96, MA08, CJH13] a complexity model of a system or process (in our case specification) is built and analysed. There are notions of *weak* and *strong* requirements, reflecting to what extent the modelled system captures human cognition. In the field of specification a model might consider the number of symbols, perhaps giving weights based on arity, kind, or placement within the specification. This approach is based on principles from psychology and requires careful application.
- **Usability.** A specification language should be accompanied by tools - both theoretic and pragmatic. From a theoretical viewpoint we should be able to manipulate specifications, perhaps transforming or combining them. From a pragmatic viewpoint we want to be able to easily create, edit and visualize specifications. Usability is often an objective quality and a common method for measuring this dimension is through user studies. For example, one might ask students to perform the same tasks with different specification languages and measure how quickly and effectively they perform these tasks.

In summary, there are many aspects to elegance and a good specification language should consider these at some level.

2.3 An introduction to runtime monitoring

Runtime monitoring, sometimes referred to as runtime verification, is the process of checking a property against the runtime behaviour of a system. Let us begin with a brief overview of the stages involved in monitoring a system, captured in Figure 2.1:

1. *Monitor creation*: A monitor is created from a formal property.
2. *Instrumentation*: The system is instrumented to generate events; discrete objects that contain some information about the system.
3. *Execution*: The system is executed, generating events for the monitor.
4. *Responses*: The monitor produces a *verdict* for each event giving the current status of the property. It also sends *feedback* to the system that may give

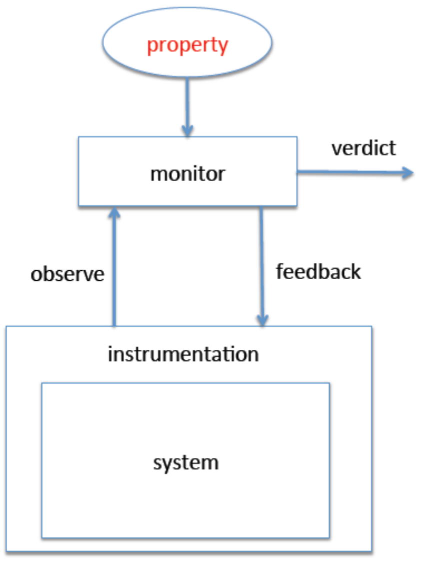


Figure 2.1: Runtime monitoring.

further information so that more specific corrective actions can be taken.

Therefore, the inputs to the process are a formal description of a property and a finite trace of events, and the output is a finite sequence of pairs of verdicts and feedback (corresponding to each event).

Runtime verification has a number of advantages over static formal program verification: Firstly, there is more information available at runtime than from static analysis of the code; Secondly, rather than checking a model or abstraction of the code the actual code is being verified; Lastly, action can be taken immediately if an execution is seen to violate a specification. Additionally, as mentioned previously, by considering only the current trace we do not suffer from the state explosion issues seen in model checking. Runtime monitoring extends the notion of testing by including a formal notion of behaviour.

In the rest of this section we consider some key concepts in runtime monitoring.

2.3.1 Instrumentation

Events and responses need to be communicated from and to the monitored system. In the runtime verification process this is handled via *instrumentation*. It is a matter of discussion whether this instrumentation forms part of the runtime verification system itself or not. Some tools intermix scripts defining instrumentation and properties, whereas others keep them firmly separate. From a theoretical point of view it is reasonable to separate the two, and we will not explicitly consider instrumentation in our work. However instrumentation is a key part of the runtime verification process and, at the very least, an *interface* should be defined between a runtime verification tool and monitored system.

The instrumentation approach used depends on the system being instrumented. Whilst instrumentation can require manual effort to insert assertions into code there has been much progress in automating the task. A key development is that of AOP (Aspect Oriented Programming) [EFB01] where specified code representing cross-cutting concerns is *weaved* into the program at specified points. The majority of the tools that target `Java` programs use `AspectJ` [Lad03], an AOP language for `Java`.

2.3.2 Responses

One advantage of verifying a system at runtime is that the system can take corrective action if a property is violated, using the results of verification to steer itself towards more desirable behaviours. To achieve this, monitors communicate with the system through *verdicts* and *feedback*. Verdicts give the status of the monitored system with respect to a property and feedback provides additional information about what actions should be taken, if any.

A runtime verification system will return a verdict from some verdict domain $\mathbb{D}$ after processing each event, or the whole trace.

In the most simple case this verdict domain would be $\mathbb{B}$, either true or false. However, many runtime verification systems use verdict domains containing three or more values to give a more fine-grained result as illustrated in Figure 2.2 and explored in detail by Bauer et al.

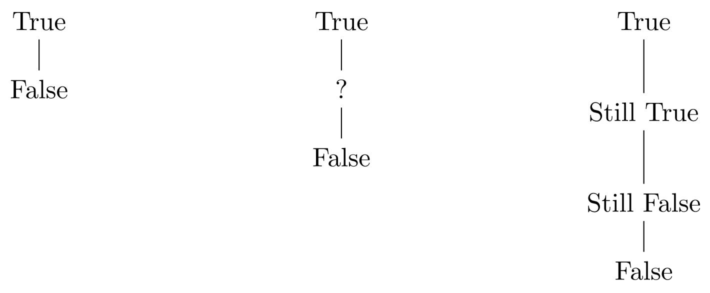


Figure 2.2: Possible verdict domains.

[BLS07, BLS10]. The first step is to introduce a third value to indicate that the system has not succeeded or failed yet. An alternative is to extend $\mathbb{B}$ with two new values, which we call *weak success* and *weak failure* here. These indicate that the monitored system is currently succeeding/failing but its status may change in the future. This linear order of four verdicts has been explored in detail in previous work [BLS10].

The area of feedback has not been widely explored within the context of runtime verification. There are relations to automatic program repair, program steering, fault protection, self-healing systems, planning, and runtime enforcement to mention just a few topics. However, we will not discuss it further in this work.

2.3.3 A matter of efficiency

As runtime monitoring may have to run alongside existing code our interest in efficiency is not restricted to how fast monitoring algorithms can run. We are concerned with the following forms of efficiency:

- **Overhead.** This is usually taken as the extra time required to execute the system with monitoring. This will consist of time spent monitoring as well as interference.
- **Interference.** The extra time taken to execute the original code after interference from monitoring code. This is most apparent in multi-threaded code where monitoring code takes locks, creating a bottleneck. Interference can also have a positive effect, where monitored code causes optimisations to be triggered making the original code run faster.
- **Throughput.** The number of events processed in a given time. This may be in terms of overall time or time spent inside the monitor.

The specification formalism and the checking mechanism are linked as one may place restrictions on the other. Therefore, the checking mechanism should be considered when designing a specification formalism. Furthermore, as is often the case, the way a specification is written can also have a large impact on efficiency.

2.3.4 Classifying approaches

Many different runtime monitoring approaches have been suggested. Here we briefly consider how these may be compared. In 2004 Delgado et al. [DGR04] suggest classifying runtime monitoring systems using the dimensions of Specification Language, Monitor, Event Handler, and Operational Issues. We discuss these dimensions shortly. What Delgado et al. mention but do not include in their taxonomy or synopsis of tools is the subject of usability, such as visualisation, ease of forming specifications, or automated tools for translating specifications.

As discussed previously, we would like to suggest the three (non-orthogonal) dimensions of Expressiveness, Efficiency and Elegance that can be considered at a somewhat higher level. Expressiveness is the range of different properties that a system can express. Efficiency is the speed at which a system can check a property against a trace, discussed earlier. Elegance, or usability, is how easily specifications can be written and understood by a human.

Specification language

A runtime verification system will typically provide a *Domain-Specific Language* (DSL) for describing properties. The different approaches for defining a DSL may be categorized as follows [BH11a]:

1. *External.* The DSL is a stand-alone language.
    (a) *Compilation.* A property is parsed and translated into a program, representing a monitor for that property, which is then *executed.*
    (b) *Interpretation.* A property is parsed and translated into a data structure, representing a monitor for that property, which is then *interpreted.*
2. *Internal.* The DSL is embedded in an existing General-Purpose Language (GPL) and is therefore directly executable.
    (a) *Shallow.* A property can make use of features in the GPL.
    (b) *Deep.* A property is represented as a data structure in the GPL.

Many runtime verification approaches use their own specification languages to describe properties that they can monitor. Different specification languages will offer different levels of expressiveness, therefore restricting what types of properties can be expressed. Languages may also operate at different levels of abstraction, although a common approach is to define *events* in terms of language features and then specify properties over these events.

Monitor

Here we are concerned with *when* and *where* monitoring occurs. This can either be *online*, monitoring occurs whilst the system is running, or *offline*, monitoring occurs after the system has run, applied to a log file. If running online, the monitor can either be *inline*, the monitor is included in the code of the system, or *outline*, the monitor exists as an external entity. Note that offline implies outline. In the online case a key consideration is whether the monitor will share the resources of the application.

Another concern is whether instrumentation is part of the monitoring system (and if so, if it is automated). As mentioned previously, a common approach is to use AOP. Some systems automatically produce `AspectJ` files (i.e. JavaMOP), or extend `AspectJ` directly (i.e. TraceMatches), whilst some outline approaches use `AspectJ` to call the monitor (i.e. RuleR).

Event handler

Here we consider how events, and the verdicts returned, are handled. The first consideration is whether an approach detects *violation*, when the monitored property becomes false, or *validation*, when the monitored property becomes true, or either. Furthermore, the approach may be able to differentiate between different kinds of violation.

Another consideration is whether the monitor can affect the application's behaviour e.g. perform actions in response to a certain verdict. A common approach is to attach arbitrary pieces of code to be executed when a certain verdict is returned.

Operational issues

The main differentiating factor here is which systems the approach can be applied to i.e only `Java` programs. Whilst most outlined approaches can be applied generally they will often have been developed with a particular implementation language in mind.

Delgado et al. also consider whether the tool is dependent on certain hardware or software, as well as the maturity of the tool.

2.3.5 Existing work

Here we give a brief overview of the existing work in this field.

The programming approach

The most primitive approach to runtime monitoring is to write a custom monitor for each property we wish to check. This is generally possible in most programming languages, but is made far easier through the use of AOP. To monitor a system with AOP we simply capture the events we are interested in and then write some logic to check the relevant property. In `AspectJ` a *pointcut* syntactical defines program points that should be instrumented (later some tools use these in specifications). The drawback here is that the language has no abstraction to separate description of the property from the mechanics of checking it, and there is little scope for reuse of monitors.

Generally, to capture temporal behaviour we need to remember what method calls have occurred previously. The standard way of doing this is through the use of an automaton, or state machine.

Design by Contract

Design by Contract is a term coined by Bertrand Meyer [Mey92a] in the late 80s. The general idea is that every piece of code (method) comes with an associated *contract* that states what must hold before the code (precondition) and gives guarantees of what will hold after it (postcondition). These can be combined to give an *invariant* i.e. something that can be assumed before and after the code. These contracts can be checked to ensure that different pieces of code use each other correctly. These contracts are typically non-temporal.

There are a number of well-known tools for writing and checking contracts. The key example being the Eiffel language [Mey92b] produced by Meyer. For `Java` we have the Java Modelling Language (JML) [LBR98], jContractor [KHB99] and Jass (`Java` with assertions) [BFMW01]. Jass is interesting as, on top of the standard notion of contract, it provides trace-assertions for defining the internal temporal behaviour of a method. These are based on the process algebra CSP (Communicating Sequential Processes), where a trace is defined by beginnings and ends of method invocations.

We are interested in approaches that capture trace specifications over many method calls.

Propositional approaches

The first runtime verification approaches operated on events without data values, we call these *propositional*.

Java-MaC [LBaK$^+$98] is based on the Monitoring and Checking (MaC) framework. The MaC framework uses two specification languages, MEDL and PEDL. MEDL (Meta-Event Definition Language) is similar to PT-LTL (Past-Time Linear Temporal Logic) with timing operators and is used to specify properties to monitor. PEDL (Primitive Event Definition Language) is used for instrumentation. PEDL scripts define the MEDL events and conditions in terms of system objects, and this mapping is used to generate an event recogniser and observation filter. The DMaC [ZSLL09] tool, standing for Distributed Monitoring and Checking, builds on the MaC framework to provide a tool for specifying and verifying distributed network protocols.

TemporalRover (TR) [Dru00] is a commercial runtime verification tool, first appearing in 2000, developed by Doron Drusinsky. It can be used to verify applications written in C, C++, Java, Verilog and VHDL, using specifications written in LTL or MTL (Metric Temporal Logic) augmented with additional operators. TR assertions are written as comments in the application, which are expanded into source code by the TR parser. The ATG Rover, a related tool, can be used to generate test sequences from the specifications, a form of Model Based Testing. More recently Drusinsky has focused on using UML as a specification language.

JavaPathExplorer [HR01] (JPaX) has been developed by NASA to verify code related to their Mars mission. The tool performs both logic based monitoring and error pattern analysis for common types of errors. Logics are expressed in Maude [Cla96](a term rewriting logic). An instrumentation script specifies how the `Java` bytecode is to be instrumented by Jtrek, a `Java` bytecode engineering tool. JavaPathExplorer is related to the JPF (JavaPathFinder) tool.

Parametric approaches

More recently runtime monitoring approaches have focussed on parametric monitoring, which considers events carrying data. Here we describe the broad areas that these tools can be divided into. These are discussed further (in relation to QEA) in Sec. 4.5.

- **Slicing.** This approach is most prominent in the JavaMOP [MJG$^+$11, CR09] tool, but is used elsewhere. The general idea is to 'slice' a trace into a set of propositional values; we discuss this further below.

- **Rule-based.** A parametric rule system is defined where rules *rewrite* a set of facts about the monitored system. Key examples of this approach are the RULER [BHRG09] and TRACECONTRACT [BH11b] tools.
- **Extending LTL.** Different extensions of LTL have been considered. Generally these add some form of restricted quantification but do not consider data in a free sense.
- **Query languages.** These approaches treat a trace as a set of records and phrase monitoring in terms of database-like queries.

The main concern with parametric monitoring is maintaining efficient processing whilst allowing for expressive statements about the use of data.

More on the slicing approach

As the slicing approach is the approach taken by QEA we expand on this notion here, but all of the key concepts will be introduced for QEA in the next Chapter.

The slicing approach deals with quantified data by taking each set of data values (those bound to the quantified variables) and considers only those events *relevant* to those values. Note that this is not slicing in the same way as we have with, say, vegetables as an event can belong to more than one slice if it contains values from more than one set.

Checking is performed by *slicing* the trace into multiple smaller traces relevant to each binding of quantified variables. Each of these traces can then be treated as propositional. These approaches tend not to treat data in a free way, or if they do, only in a limited sense. Therefore, the general notion is to transform a parametric case into multiple propositional ones, to be checked by a propositional checker.

For example, take the trace $\texttt{f}(1).\texttt{f}(2).\texttt{g}(3)$. We would create two slices, both consisting of the propositional trace $\texttt{f.g}$; one for the values $\{1,3\}$ and one for the values $\{2,3\}$.

The sets of data values used are typically constructed as followed. If we mention variables x and y then we construct the sets of values X and Y, giving all possible values for x and y, and slice the trace for each pair in $X \times Y$. All current approaches consider universal quantification only i.e. the property must hold for all trace slices. There is a notion of *connectedness*[1] that restricts the set of bindings used to ones transitively connected by events in the trace i.e. we only consider $(1,2) \in X \times Y$ if event $\texttt{g}(1,2)$ appears in the trace. This makes sense when we consider data values being created from other data values, for example one collection being constructed from another, as data values not connected in this way have no semantic relation.

This slicing approach is popular as it allows for efficient checking and a separation of concerns. We can separately optimise propositional checking and the slicing activity, and slicing allows for efficient indexing, discussed further in Sec. 6.5.

Reducing overhead

Finally, some tools exist explicitly to tackle the monitoring overhead problem.

[1]Only named explicitly by JAVAMOP

For example, Clara [BLH10] attempts to statically evaluate runtime monitors ahead of time to reduce monitoring overhead. The idea of *state estimation* [SBS+12] allows monitors to sample the program (build snapshots of the trace) and *estimate* the missing behaviour, thus reducing overhead. The field of predictive analysis [CSR08] attempts to *predict* errors that could occur (but might not occur in the monitored trace) by considering alternative orderings of events in the trace (that could be caused by different thread interleavings). This reduces overhead by reducing the number of runs that must be inspected.

2.4 Inferring specifications

Deciding whether a program satisfies a specification is a matter 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 not. This is what we did in runtime verification.

We now consider the 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*. This is the aim of specification inference.

In the rest of this section we will cover the fundamental ideas behind the (disparate) field of specification inference. It is worth noting that many techniques and theoretical results are heavily tied up in a particular domain and that we can draw a distinction between *grammar inference* and *specification mining*.

Grammar inference.

This 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. Because of the focus on language (rather than programs) these techniques do not generally consider models over symbols parameterised with data values. There exist some good reviews of the field [AS83, Mur96, dlH05, dlH10, Leu07]. The grammar inference problem is usually described in terms of uncovering a hidden language. Conceptually, the process is often modelled as a student attempting to learn a language. The most popular problem is that of *regular inference*; constructing a model for a regular language.

Specification mining.

The term 'Specification Mining' has been attributed to Ammons et al. from a paper written in 2002 [ABL02]. Some writers [WBHS08] have applied the term only to techniques that analyse execution traces, although others [SYFP07] have applied the name to techniques that analyse source code. We take the broad definition that Specification Mining is a collection of techniques that attempt to mine a specification of a program from some artefact of that program. The specification mining problem is usually described in terms of describing a set of observations in some minimal way, however some approaches take the hidden language view. Interest in this area has grown recently, leading to a book [LKHL11] and review paper [RBK+13].

Our contribution belongs to the field of specification mining, however we also discuss the field of grammar inference here as it is relevant. Additionally, many recent advances in specification mining draw on established ideas from the field of grammar inference.

2.4.1 The ins and outs

As a process, specification inference is very general. We put some observations about a system in and get a description of the system out. There are many choices of the kinds of observations we can make about the behaviour of a system and many ways in which we can describe a computer system, as previously discussed.

Making observations

There are three kinds of artefacts that can describe the behaviour of a system:

1. **Source code.** This completely describes a system's behaviour. However, it does not reflect the common, or actual, behaviour exhibited by the program. Additionally we do not capture any input-specific or timing behaviour that is available at runtime. These techniques are called *static*.

2. **Execution traces.** These describe the actions the system takes when run. However, we must run the system to capture traces and therefore require a range of inputs to exercise the system. Thankfully, many systems already have large test suites that do this, and there exist techniques for automatically generating tests with certain coverage properties. One disadvantage is that we can lose structure obvious in the source code i.e. if one method is always called by another then their temporal relationship is uninteresting. These techniques are called *dynamic*.

3. **Development documents.** Design documents, source control logs and bug reports can all reflect the behaviour, intended or actual, of a computer system. Some work [LZ05b] has mined these artefacts, but it still remains relatively unexplored due to the largely unstructured quality of this data.

We are interested in execution traces, and therefore we must decide how we collect these traces. We can *passively* observe the traces or *actively* ask for them. To be more precise, 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. We decide to take a passive approach as it is more widely applicable as it does not require a *teacher* to answer queries about the system.

There is also the notion of observed traces being *positive* or *negative* i.e. belonging to the behaviour of the system or not. Later (Sec. 2.4.4) we see that most cases of exact learning can only be achieved using negative traces. The significance of negative traces is more prominent in the grammar inference community where there is a more obvious source for them - they are rarely used in specification mining as it is difficult to record what a program does not do.

There is an underlying assumption that the observations we make are correct and are of a correct system. If this is not the case then the observations will contain *noise*. There has been an increased interest in the study of systems that can deal with noise in recent years. In the student/teacher description this has been phrased in terms of having an unreliable teacher [Leu07]. In terms of computer systems this noise is usually framed as programming bugs.

Describing the system

The output will be a temporal specification, and the exact form that might take will be discussed later. However, it will either *exactly* describe the system or *approximately* describe it.

There are two reasons an inferred specification may be approximate. 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. The general passive approach we have chosen is necessarily approximate. If we can make no guarantees about the coverage of the original system then it is possible that some behaviours have not been observed and consequently cannot be captured in the output.

Additionally, some approaches attempt to describe the target system using a single specification, whereas others generate many small specifications describing the relationship between different parts of the system.

2.4.2 Target specification formalisms

There have been a range of specification formalisms that have been targeted for specification inference.

Propositional regular

The most popular target for specification inference is a regular language captured by a finite state machine - although some approaches have considered regular grammars.

Some techniques target only a set of predefined regular languages, for example specifications of the form 'if `a` happens then later `b` happens' or '`a` and `b` alternate'. Some methods choose to extract many small automata or rules and some focus on extracting a single automaton representation.

Beyond regular... but still propositional

There have been a few approaches that attempt to infer specifications that are more expressive than regular. Firstly, there has been work reducing the problem of inferring an *even linear context-free language* to that of inferring a regular language [M9̈6, Tak88, KMT95]. Others infer context-free grammars given information about grammatical structure [SM00, SK99]. Work has also been done on inferring probabilistic context-free grammars [TH08]. There has also been some work applying genetic algorithms to evolve pushdown automata [Lan95, NP07] and context-free grammars [Wya93, Pan10, CK10]. Genetic algorithms have also been used to infer stochastic regular grammar [SO95] and stochastic context-free grammars [KL97].

Quantified parametric

Three techniques have targeted languages that have a general approach that treats data in a quantified way. JMiner [LCR11] targets quantified state machines whose semantics are determined using slicing in connected mode i.e. the state machine describes a propositional language and slicing uses connected bindings to translate a parametric trace into many propositional ones. Pradel and Gross [PG09] take a similar approach by identifying (connected) object collaboration sets and using them to slice traces, building up an automaton using a non-standard technique. Both approaches use heuristics to identify possible alphabets. Tark [LCH+09, LRRV12] targets quantified binary temporal rules with equality constraints (QBEC). For example, the forward-eventual rule is of the form $\forall \overline{x}.\mathtt{a}(\overline{x}_1) \rightarrow \mathtt{b}(\overline{x}_2)$ where $\overline{x}_1$ and $\overline{x}_2$ are sublists of $\overline{x}$. A trace satisfies this rule if, for any binding to the variables $\overline{x}$, if $\mathtt{a}(\overline{x}_1)$ occurs, then at some point later $\mathtt{b}(\overline{x}_2)$ occurs. Tark also captures some limited information about free variables as it can infer that the value of a free variable is constant (and that constant value).

Some techniques take a limited approach that considers typestate only (i.e. a single quantified variable). Xiao et al. infer typestates by separating the behaviours related to each instance of an object. Alternatively, some propositional techniques [YEB+06] use a context-sensitive approach to deal with data values - i.e. slicing on callee object identity.

Free parametric

The target of specification inference techniques that treat data in a free way is always some form of extended finite state machine. However, the exact form often varies.

In the passive setting, Lorenzoli [LMP08] extracts state machines with transitions labelled with guards and Mariani [MP08] extracts data recurrence patterns that capture the reuse of values through equality constraints. Lo et al. [LM12] mine live sequence charts enriched with invariants.

In the active setting, approaches [BJR06, AHK+12] extend Angluin's original L* algorithm [Ang87] to augment state machines with transition constraints. There has also been a recent interest [HSJC12, BHLM13] in Register Automata where data values can be saved and compared. Xiao et al. [XSL+13] infer typestates with guarded transitions.

Difference between quantified and free parametric

There is a fundamental difference between techniques that focus on free, rather than quantified, uses of parameters. The free variable approach augments a specification mined by a propositional technique, whereas the quantified variable approach uses quantifications to identify the propositional parts. Applying a propositional technique to the trace $\mathtt{open}(1).\mathtt{open}(2).\mathtt{open}(3).\mathtt{close}(1).\mathtt{close}(3).\mathtt{close}(2)$ would not produce the specification that correctly abstracts the data, i.e., $\forall f.\mathtt{open}(f).\mathtt{close}(f)$.

An empirical study [LMS12] examined whether the techniques for mining with free variables of Lorenzoli [LMP08] and Mariani [MP08] necessarily achieve specifications that better describe the underlying software than their propositional counterparts. They conclude that neither significantly increase accuracy. However, these results cannot be applied to work for quantified variables, or other techniques for free variables.

2.4.3 Using specification inference

The fields of grammar inference and specification mining both began with the same goal - to explain and understand. However, there are other uses for specification inference, which we discuss here. We note that our principal interest in this work remains the comprehension of program behaviour.

Program Comprehension

The majority of techniques aim 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. There is also a need for understanding *legacy systems* that need to be updated or maintained. Inferred specifications can be used to form design documentation or used in combination with testing or verification to ensure that certain previous behaviours have not been altered after change. Many of the techniques reviewed in a recent review paper [RBK+13] aim to understand the usage of APIs.

Testing

Inferred specifications have been used to generate tests. This is called model-based testing and there has been work integrating the specification inference approach with model-based testing for black-box systems [RSM08, PVY01]. A related area is that of automatically generating tests to improve specification inference - for example, the TAUTOKO tool [DKM+10] explores undefined transitions in a mined specification by mutating an existing test suite. Shabaz et al. [SLG07b, SLG07a] use inferred models of program components to generate integration tests. Pradel et al. [PG12] use specification mining to drive test generation for bug finding.

Verification

Inferred specifications can be used for runtime verification. A key application of this is to infer specifications for a library from one system and check them against another system. There has also been some work on both inferring and checking specifications at runtime [GS10]. Additionally, Puasuareanu et al. [PGB+08] use an active technique for assume-guarantee reasoning - assumptions as labelled transition systems are extracted with the help of a model checker.

Security

Shu et al. infer certain security protocols. Firstly, they describe how specification inference techniques can be used to test if security protocols obey an important security property [SL07]

- message confidentiality under the general Dolev-Yao attacker model [DY83]. Later [SHL08] they suggest an approach for combining protocol inference with fuzz testing [Oeh05] to detect security (and reliability) problems in communication protocols.

Based on an approach by Forrest et al. [HFS98] 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. [SBDB01] use an ad-hoc technique for inferring state machines to describe normal behaviour. Gosh et al. [GMS00] describe three approaches to learning normal program behaviour that learn three different models - Elman recurrent neural networks, string transducers and finite state machines. Goa et al. [GRS04] extract an execution graph during a training period and continue to update this during monitoring. Ingham et al. [ISBF07] learn DFA representations of the HTTP protocol to protect web applications.

Specification inference has also been used for malware detection. Christodorescu et al. [CJK07] 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. Their approach created a lot of false positives but was able to identify programs as malicious.

2.4.4 The limitations

The field of grammar inference tells us a lot about what is possible and feasible. The main result is that regular inference is NP-complete and this partly motivates heuristic specification mining techniques that tackle this intractability by recovering approximate specifications.

The field of*computational learning theory* [Ang92] 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 [Sak97, dlH05] - *identification in the limit* [Gol67], *query learning* [Ang88] and *PAC learning* [Val84]. Pit [Pit89] 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.

Identification in the limit [Gol67]. 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 the language 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 [CL11] outline Gold's results as follows - The class of finite languages and any finite class of recursive languages is identifiable in the limit on the basis of positive data only; A super-finite[2] class of languages is not identifiable in the limit on the basis of positive evidence only; The class of recursive languages is identifiable in the limit on the basis of negative and positive data. Note that given a complete labelled enumeration of all words we can generate a minimal model for a language - called *identification by enumeration.*

We can consider complexity in the length of the input traces. It was shown by Gold that the problem of identifying a minimum DFA from a given finite set of examples is NP-complete

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

[Gol78]. The regular inference problem has been compared to breaking the RSA cryptosystem [KV94]. 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.

Query Learning [Ang88]. To tackle the NP-hardness of the language learnability problem Angluin [Ang88] 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. It has been shown that the class of regular languages is polynomially identifiable using both equivalence and membership queries, but not only using either on its own.

Probably Approximately Correct (PAC) learning [Val84]. In this learning model, given a set of samples, a learner must select a hypothesis *generalisation* function from 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. It has been shown that regular languages are PAC-learnable.

2.4.5 Evaluating solutions

For exact techniques the inferred specification is guaranteed to be correct. In this case the evaluation considers the efficiency of the approach. For approximate techniques we find two approaches to evaluation in the literature, which we describe below.

Explorative

In some cases [ABL02, GS08b, GS12, LCR11, YEB^{+}06] evaluation is carried out by selecting one or more real-world application and applying the developed technique, recording running times and inspecting the 'quality' of the mined specifications - sometimes manual effort is expended to identify 'true' and 'false' mined specifications. This method is often used to justify the utility of the technique by demonstrating that known or interesting specifications can be extracted. Additionally, techniques that work in the presence of noise are sometimes used to identify bugs in software where a specification is extracted and found not to hold in some places.

Measuring accuracy

A more thorough approach [LK06a, LRRV12, WB08] measures the *accuracy* of the inferred specification. The general approach is to have a set of training traces and a set of test traces, apply the specification inference technique to the training set and then measure accuracy on the test set. At a basic level accuracy can be given by the percentage of correctly classified traces in the test set. The notion of accuracy can be refined further by borrowing the dimensions of *precision* and *recall* from the field of information retrieval [vR79]. Precision captures a measure of exactness and recall a measure of completeness. Walkinshaw et al. [WBJ08] have proposed a more informative approach that extracts precision and recall separately for positive and negative behaviour. This is motivated by the observation that precision/recall is biased towards positive behaviour and will not capture the situation where the inferred and actual specification both correctly reject a trace. Pradel et al. [PBG10] also consider techniques for measuring the *structural similarity* between ground truths and extracted models.

The ground truths can be formed in two different ways. Firstly, a system that generates the traces can be manually inspected to identify the intended behaviour. Secondly, representative specifications are used to generate the traces and then used themselves as ground truth.

2.5 Summary

We have reviewed methods of temporal specification within the context of runtime verification and specification inference, paying particular attention to first-order, or parametric, provisions.

On one hand we have efficient runtime monitoring techniques such as JavaMOP [MJG+11, CR09] and TraceMatches [AAC+05] that use a slicing approach, and on the other hand we have very expressive techniques such as RuleR [BGHS04, BRH08, BHRG09] who's focus on expressiveness leads to a lack of efficiency. The slicing techniques focus on the quantified view of data and do not allow for existential quantification.

Tark [LCH+09, LRRV12], JMiner [LCR11], Pradel and Gross [PG09] and TzuYu [XSL+13] learn forms of quantified parametric specification. GkTail [LMP08], KLFA [MP08], a new method using data classifiers [WTD13] and many extensions of L^* [BJR06, HSJC12, AHK+12, BHLM13] learn forms of free parametric specification.

In the next few chapters we introduce quantified event automata, a specification formalism that captures both a free and quantified view on data, including existential quantification. This automata formalism is designed to admit a slicing semantics.

Chapter 3

Quantified Event Automata

This chapter presents quantified event automata (QEA) as an expressive formalism for writing parametric specifications. We separate the part of the specification concerned with orderings of events and free variables from the part concerned with quantification. The first is captured by event automata (EA) and consists of a form of extended finite sate machine. We use automata as they will give us a straight-forward incremental monitoring algorithm later. We extend EA with logical quantification to define QEA.

We can see this separation in the specification process. An event automaton is written to describe the behaviour of a particular set of values. For example, how a given user object should interact with a given file object. A quantified event automaton then extends this notion to multiple sets of of values by replacing the values with quantified variables, for example, for every user and file object.

Structure. We begin with our basic definitions (Sec. 3.1) before introducing simple event automata (Sec. 3.2) as simple state machines. We then add a simple notion of quantification to define simply quantified simple event automata (Sec. 3.3) giving a form of QEA without free variables. Free variables are then introduced to give a (Turing-complete) version of event automata (Sec. 3.4). Finally, we extend our notion of quantification to give the quantified event automata (Sec. 3.5) that we will use later.

3.1 Definitions

In this section we introduce the building blocks of this runtime monitoring approach. We met some of these notions informally in Section 2.3.

3.1.1 Events and traces

A symbol is either a variable or a value. We ignore the notion of type here as it does not provide any distinctions that would significantly alter the development.

Definition 2 (Symbols)**.** *Let* $Sym = Val \cup Var$ *be the set of all symbols where* Val *is a countably infinite set of variables and* Val *is a countably infinite set of values disjoint from* Var*.*

We use s to refer to symbols and x, y to refer to variables. Values can represent anything i.e. integers, strings or objects from an Object Oriented programming language. The sets *Val* and *Var* are countably infinite but a finite trace can only contain a finite number of values and a QEA can only contain a finite number of variables. Therefore, when considering whether a trace satisfies a specification we will only consider a finite subset of each.

Events and traces are built from symbols and event names.

Definition 3 (Events and Traces). *An event is a pair $\langle e, \overline{s}\rangle \in \Sigma \times Sym^*$, written $e(\overline{s})$. An event $e(\overline{s})$ is ground if $\overline{s} \in Val^*$. Let Event and GEvent be the sets of all events and ground events respectively. A* trace *is a finite sequence of ground events. Let $Trace = GEvent^*$ be the set of all traces and ϵ be the empty trace.*

We use $\mathbf{a}, \mathbf{b}$ to refer to events and σ, τ to refer to traces. Recall that we are only interested in finite traces (Sec. 1.1.1).

3.1.2 Bindings

Bindings are maps (partial functions with *finite* domain) from variables to values, i.e, elements of $Bind = Var \rightharpoonup Val$. Let $\mathsf{dom}(\theta)$ be the domain of binding θ. If binding θ maps x to v then we write $\theta(x) = v$. We overload this notation to allow us to apply bindings to symbols in general. Given a symbol s we let $\theta(s)$ equal v if θ maps x to v and equal s otherwise i.e. we lift θ to a total function on symbols.

Given two maps A and B, the map override operator is defined as:

$$(A\dagger B)(x) = \begin{cases} B(x) & \text{if } x \in \mathsf{dom}(B), \\ A(x) & \text{if } x \notin \mathsf{dom}(B) \text{ and } x \in \mathsf{dom}(A), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

There exists a (well-known[1]) partial order $\sqsubseteq$ on bindings such that $\theta_1 \sqsubseteq \theta_2$ iff θ_1 is a submap of θ_2 i.e.,

$$\theta_1 \sqsubseteq \theta_2 \text{ iff } \forall x \in \mathsf{dom}(\theta_1) : x \in \mathsf{dom}(\theta_2) \land \theta_2(x) = \theta_1(x)$$

3.1.3 Creating bindings from events

We can apply a binding to an event, replacing any variables defined in the binding.

Definition 4 (Substitution). *The binding θ can be applied to an event $e(\overline{s})$ as follows:*

$$\theta(e\langle s_0, \ldots, s_n\rangle) = e\langle \theta(s_0), \ldots, \theta(s_n)\rangle$$

Recall that we have lifted bindings to total functions on symbols.

This can be used to give a definition of a ground event and an event *matching* to give a binding.

[1]This ordering has been called *informativeness* in other work [CR09, RC12] as a larger binding contains more information.

Definition 5 (Matching)**.** *Given a ground event* $\mathbf{a}$ *and an event* $\mathbf{b}$*, let* $\texttt{matches}(\mathbf{a},\mathbf{b})$ *hold iff there exists a binding* θ *s.t.* $\theta(\mathbf{b}) = \mathbf{a}$*. Let* $\texttt{match}(\mathbf{a},\mathbf{b})$ *denote the smallest such binding w.r.t* $\sqsubseteq$ *(the submap relation) if it exists (and is undefined otherwise).*

For example, let us attempt to match the event $\mathbf{b} = \texttt{e}(x, y, 2, x)$ with different ground events.

ground event $\mathbf{a}$	matches	match	reason
$\texttt{f}(2)$	*false*	–	event names different
$\texttt{e}(1,2,3,1)$	*false*	–	$3 \neq 2$ in third position
$\texttt{e}(1,1,2,2)$	*false*	–	x cannot equal 1 and 2
$\texttt{e}(1,1,2,1)$	*true*	$[x \mapsto 1, y \mapsto 1]$	
$\texttt{e}(1,2,2,1)$	*true*	$[x \mapsto 1, y \mapsto 2]$	

3.1.4 Projecting traces with sets of events

A trace can be *projected* with respect to a set of ground events to remove events not in the given set.

Definition 6 (Projection)**.** *The projection of* $\tau \in$ *Trace w.r.t. a set of ground events* $\mathcal{A}$ *is defined as:*

$$\epsilon \downarrow_{\mathcal{A}} = \epsilon \qquad \tau\mathbf{a} \downarrow_{\mathcal{A}} = \begin{cases} (\tau \downarrow_{\mathcal{A}}).\mathbf{a} & \text{if } \mathbf{a} \in \mathcal{A} \\ (\tau \downarrow_{\mathcal{A}}) & \text{otherwise} \end{cases}$$

For example, given $\mathcal{A} = \{\texttt{e}(1), \texttt{e}(2)\}$ if $\tau = \texttt{e}(1).\texttt{e}(5).\texttt{e}(2).\texttt{e}(6)$ then $\tau \downarrow_{\mathcal{A}} = \texttt{e}(1).\texttt{e}(2)$.

3.1.5 Guards and assignments

We consider two kinds of functions on bindings: guards and assignments. We will be using bindings to represent state (i.e. stored values) and these functions will allow us to manipulate this state. For example, a binding $[count \mapsto 1]$ may be used to store the fact that the variable count has value 1 and the guard $count > 1$ (denoting $\lambda\theta.\theta(count) > 1$) would be false for this binding and the assignment $count{+}{+}$ (denoting $\lambda\theta.\theta\dagger[count \mapsto \theta(count) + 1]$) would return the binding $[count \mapsto 2]$.

Definition 7 (Guards and Assignments)**.** *A guard* $g \in Guard = Bind \rightarrow \mathbb{B}$ *is a predicate on bindings i.e. a computable function from bindings to the boolean domain* $\mathbb{B} = \{\top, \bot\}$*. An assignment* $\gamma \in Assign = Bind \rightarrow Bind$ *is a total computable function from bindings to bindings.*

We do not specify a particular language for guards and assignments here - we assume they are computable but make no further assumptions. The definition of QEA is therefore parameterised by this choice of language for guards and assignments. This is an important observation that will have implications when discussing the complexity of trace checking and the expressiveness of the formalism. In examples we will use standard programming language notation for guards and assignments. The majority of examples use integers and standard operations on them but some also make use of data structures such as sets. We will refer to the trivially true guard as *true* and the trivial identity assignment as *id*. We assume that an assignment language would deal with the so-called *frame problem* by ensuring that assignments maintain values they do not explicitly update i.e. the assignment $x = y + 1$ would maintain the value of y.

3.1.6 A note on types

We have chosen not to include types in this framework as it is not necessary for the theoretical developments and would only be necessary for two tasks: firstly ensuring that the input trace matches with what is expected in the specification, and secondly type-checking guards and assignments. As we do not have a notion of types we assume that the guard or assignment languages deal with type-safety appropriately e.g. only allowing integer operations to be performed on integers.

To include a notion of type we would add the concept of event *signatures* defining the allowed types of event parameters and then use this signatures to type-check traces, guards and assignments and to ensure that specifications were type-consistent. This would be relatively straightforward so is not discussed further here.

3.2 Simple Event Automata

Simple event automata (SEA) describe a pattern of behaviour for a (typically small) set of ground events. These ground events may contain data values and we could think of the ordering constraints on ground events as a description of how these values should behave. We explore this notion further before defining SEA.

3.2.1 Dealing with symbols not in the alphabet

Earlier we stated that state machines are often used to describe event orderings. We revisit this idea here. As an example consider a simple file handling system with open and close operations. We can specify the property that these operations must alternate with the following automaton (recall our notation in Sec. 2.1.2 that square states have a next-style semantics and shaded states are final).

$$\begin{tikzcd} {} \arrow[r] & \boxed{1} \arrow[r, bend left, "\texttt{open}"] & \boxed{2} \arrow[l, bend left, "\texttt{close}"] \end{tikzcd}$$

The previously discussed standard acceptance condition for automata can then be used to check that the word

`open.close.open.close`

satisfies our specification. But what do we do when faced with the word

`open.use.close.open.use.close`

where the `use` event is of no importance to the property being verified? Our specification does not mention the symbol `use`, and we need to define how to deal with this symbol. A natural step to take is to define acceptance over a projected trace, filtering out any symbols not in the automaton's alphabet. Therefore, an automaton accepts a word iff the projected subword is accepted.

Projection is not the only approach that could be used here, we could add additional transitions to the automaton to deal with this unknown symbol i.e. use a skip semantics (see Sec. 2.1.2). So what is the difference between a skip-semantics and projection? Projection makes a distinction between symbols that are *relevant* to the automaton and those that are not. In a skip-semantics some symbols might be relevant at some states but not others. Projection allows us to tell whether an event is relevant to an automaton *without inspecting that automaton's current state*. This will be *important* later when we consider efficient monitoring, where we need to identify one or two automata relevant to an event among tens of thousands.

In most real world applications, events are more complicated than simple event names such as `open` and `close`, they are parametrised with data values. For example, $\texttt{open}(\mathit{file\_name})$ or $\texttt{send}(\mathit{user}, \mathit{msg})$. We can label the transitions of our automaton with these parametrised events, defined earlier as an event name and list of symbols. The following automaton specifies that opening and closing the file manual.pdf must alternate.

$$\begin{tikzcd} {} \arrow[r] & \boxed{1} \arrow[r, bend left, "\texttt{open}(\text{manual.pdf})"] & \boxed{2} \arrow[l, bend left, "\texttt{close}(\text{manual.pdf})"] \end{tikzcd}$$

We can still use our natural intuition of projection to deal with traces such as

$$\texttt{open}(\text{manual.pdf}).\texttt{open}(\text{readme.txt}).\texttt{close}(\text{readme.txt}).\texttt{close}(\text{manual.pdf})$$

by filtering out events that are not in the alphabet of the automaton, giving us the projected trace

$$\texttt{open}(\text{manual.pdf}).\texttt{close}(\text{manual.pdf}).$$

We can also write the following automaton to specify that opening and closing the file readme.txt must alternate.

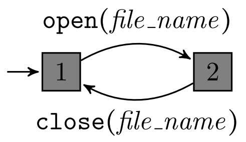
$$\begin{tikzcd} {} \arrow[r] & \boxed{1} \arrow[r, bend left, "\texttt{open}(\text{readme.txt})"] & \boxed{2} \arrow[l, bend left, "\texttt{close}(\text{readme.txt})"] \end{tikzcd}$$

Projecting the same trace with respect to the alphabet of this automaton gives us the projected trace

$$\texttt{open}(\text{readme.txt}).\texttt{close}(\text{readme.txt}).$$

This leads to redundancy; we have written two automaton that only differ in the file they are specifying the property for. We therefore take the natural step of replacing values with variables, producing a schema[2] that can be instantiated with a binding to give an automaton for the binding's value. For example, the following schema states that $\texttt{open}(\mathit{file\_name})$ and $\texttt{close}(\mathit{file\_name})$ must alternate for a given $\mathit{file\_name}$.

[2]Later we will introduce simple event automata to describe these schema.



We can construct the previous automaton by instantiating this schema with the bindings [*file_name* $\mapsto$ manual.pdf] or [*file_name* $\mapsto$ readme.txt]. We do not have a notion of a trace being accepted by a schema; it needs to be instantiated to be used.

3.2.2 The basic structure

A simple event automaton (SEA) is finite-state automaton with events labelling transitions.

Definition 8 (Simple Event Automaton). *A SEA $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ is a tuple where Q is a finite set of states, $\mathcal{A} \subseteq Event$ is a finite alphabet, $\delta \subseteq (Q \times \mathcal{A} \times Q)$ is a finite transition set, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The variables of $\mathcal{E}$ are those that appear in its alphabet:*

$$\mathsf{vars}(\mathcal{E}) = \{x \mid \exists e(\overline{s}) \in \mathcal{A}.x \in \overline{s} \ \wedge \ x \in Var\}.$$

$\mathcal{E}$ is ground *if* $\mathsf{vars}(\mathcal{E})$ *is empty.*

A non-ground SEA is a schema and therefore we will assume ground SEA in the following notion of acceptance and then introduce an alternative form of acceptance that involves constructing a ground SEA from a schema using a binding.

3.2.3 Defining acceptance

Here we introduce a notion of acceptance for ground SEA by defining a ground SEA's language. To do this we introduce a lifted transition relation that closes the set of transitions δ to give a skip semantics. As the SEA is ground there are no variables in events in δ.

Definition 9 (Transition Relation). *Consider a ground SEA $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$. Let $q \xrightarrow{\mathbf{a}}_{\mathcal{E}} q'$ hold if*

$$(q, \mathbf{a}, q') \in \delta \cup \{(q, \mathbf{b}, q) \mid \neg\exists q' : (q, \mathbf{b}, q') \in \delta\}$$

We lift this to traces as in Sec. 2.1.2 i.e. $q \xrightarrow{\epsilon}_{\mathcal{E}} q$ holds and $q \xrightarrow{\tau.a}_{\mathcal{E}} q'$ holds iff there exists q'' such that $q \xrightarrow{\tau}_{\mathcal{E}} q''$ and $q'' \xrightarrow{\mathbf{a}}_{\mathcal{E}} q'$. Note that SEA are, in general, non-deterministic. For ground SEA determinism is defined as usual i.e. events labelling transitions out of a state are unique.

The language of a ground SEA is the set of all traces over its (ground) alphabet that reach an accepting state using its transition relation.

Definition 10 (Simple Event Automaton Language). *The language of the ground SEA $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ is defined as*

$$\mathcal{L}(\mathcal{E}) = \{\tau \in \mathcal{A}^* \mid \exists q \in Q : q_0 \xrightarrow{\tau}_{\mathcal{E}} q \wedge q \in F\}$$

Note. *We have chosen a skip semantics (see Sec. 2.1.2) instead of a next semantics as it is often more straightforward to define the behaviour that causes failure than the behaviour that avoids it. To define failing behaviour we want to ignore events that maintain the status, hence a skip semantics. To define accepting behaviour we want to implicitly fail whenever we violate the given behaviour, hence a next semantics. We revisit the idea of specifying for violation or validation later. As it will be useful for specification we will use states with a next semantics when defining properties (recall these states are differentiated by their shape) which can be replaced via a simple translation that adds in the implicit failure state* $q_\perp$ *and transitions to it.*

3.2.4 Acceptance with respect to a binding

The previous notion of acceptance was for ground SEA only. Here we introduce SEA instantiation that allows us to define the language of a non-ground SEA for a given binding.

Definition 11 (Simple Event Automaton Instantiation)**.** *Given a simple event automaton* $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ *and a binding* θ *such that* $\mathsf{dom}(\theta) = \mathsf{vars}(\mathcal{E})$*, let* $\mathcal{E}(\theta) = \langle Q,\ \theta(\mathcal{A}), \theta(\delta), q_0, F\rangle$ *be the* θ*-instantiation of* $\mathcal{E}$ *where*

$$\begin{array}{rcl}\theta(\mathcal{A}) & = & \{\theta(\mathbf{a}) \mid \mathbf{a} \in \mathcal{A}\}\\ \theta(\delta) & = & \{(q, \theta(\mathbf{a}), q') \mid (q, \mathbf{a}, q') \in \delta\}\end{array}$$

Note that $\mathcal{E}(\theta)$ is necessarily ground as $\mathsf{dom}(\theta) = \mathsf{vars}(\mathcal{E})$. The language of a non-ground SEA $\mathcal{E}$ given a binding of its variables θ is $\mathcal{L}(\mathcal{E}(\theta))$.

3.2.5 Acceptance for general traces

In our motivating discussion, we were given traces over symbols not in a automaton's alphabet but our notions of language so far have been over traces of symbols in the automaton's alphabet. We extend this notion of acceptance to traces over any alphabet through the use of projection (defined previously in Def. 6).

Definition 12 (Simple Event Automaton General Language)**.** *The general language of the ground SEA* $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ *given alphabet* $\mathcal{B}$ *such that* $\mathcal{A} \subseteq \mathcal{B}$ *is noted and defined as*

$$\mathcal{L}_G(\mathcal{E}, \mathcal{B}) = \{\tau \in \mathcal{B}^* \mid \tau \downarrow_{\mathcal{A}} \in \mathcal{L}(\mathcal{E})\}$$

we omit $\mathcal{B}$ *when it is clear from context e.g. the ground events appearing in some trace.*

As $\mathcal{L}(\mathcal{E}) \subseteq \mathcal{L}_G(\mathcal{E}, \mathcal{B})$ we will use $\mathcal{L}(\mathcal{E})$ to refer to the *general* language of $\mathcal{E}$. Whenever $\mathcal{B}$ is not clear from context we will take it as the set of ground events *GEvent*.

3.2.6 Example

Let us complete our example of the file usage property by framing it formally and considering a trace and two different bindings. The schema for this property is

$$\mathcal{E} = \left\langle \{0,1,2\}, \{\texttt{open}(f), \texttt{close}(f)\}, \left\{ \begin{array}{c} (1, \texttt{open}(f), 2) \\ (1, \texttt{close}(f), 0) \\ (2, \texttt{open}(f), 0) \\ (2, \texttt{close}(f), 1) \end{array} \right\}, 1, \{1,2\} \right\rangle$$

as described graphically earlier, note that we have implicitly added the state 0 as states with a next-style semantics are used. We now consider the trace

$$\tau = \texttt{open}(\text{manual.pdf}).\texttt{open}(\text{readme.txt}).\texttt{open}(\text{readme.txt}).\texttt{close}(\text{manual.pdf})$$

and whether it is accepted with respect to the bindings $[f \mapsto \text{manual.pdf}]$ and $[f \mapsto \text{readme.txt}]$.

Our first question is $\tau \in \mathcal{L}(\mathcal{E}([f \mapsto \text{manual.pdf}])$? The first step is to construct $\mathcal{E}([f \mapsto \text{manual.pdf}])$ as

$$\left\langle \{0,1,2\}, \left\{ \begin{array}{c} \texttt{open}(\text{manual.pdf}), \\ \texttt{close}(\text{manual.pdf}) \end{array} \right\}, \left\{ \begin{array}{c} (1, \texttt{open}(\text{manual.pdf}), 2) \\ (1, \texttt{close}(\text{manual.pdf}), 0) \\ (2, \texttt{open}(\text{manual.pdf}), 0) \\ (2, \texttt{close}(\text{manual.pdf}), 1) \end{array} \right\}, 1, \{1,2\} \right\rangle.$$

Next, we project τ with respect to the alphabet $\{\texttt{open}(\text{manual.pdf}), \texttt{close}(\text{manual.pdf})\}$.

$$\tau \downarrow_{\{\texttt{open}(\text{manual.pdf}), \texttt{close}(\text{manual.pdf})\}} = \texttt{open}(\text{manual.pdf}).\texttt{close}(\text{manual.pdf}).$$

We then ask if there is a sequence of transitions for this projected trace ending in a final state, and as the sequence of transitions

$$1 \xrightarrow{\texttt{open}(\text{manual.pdf})} 2 \xrightarrow{\texttt{close}(\text{manual.pdf})} 1$$

ends in a final state the trace τ is accepted by $\mathcal{E}([f \mapsto \text{manual.pdf}])$. Our next question is $\tau \in \mathcal{L}(\mathcal{E}([f \mapsto \text{readme.txt}])$? We go through the same steps of instantiation and projection as before, which leads us to ask whether there is a sequence of transitions for $\texttt{open}(\text{readme.txt}).\texttt{open}(\text{readme.txt})$ ending in a final state. But as the only sequence of transitions is

$$1 \xrightarrow{\texttt{open}(\text{readme.txt})} 2 \xrightarrow{\texttt{open}(\text{readme.txt})} 0$$

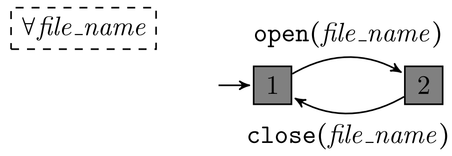
the trace τ is not accepted by $\mathcal{E}([f \mapsto \text{readme.txt}])$.

3.3 Simply Quantified Simple Event Automata

We now generalise the notion of SEA schema by quantifying over their variables to give a family of ground SEA.

3.3.1 Quantifying over SEA

Let us continue our example from the previous section involving files. To say that for all files *file_name* the operations open and close on that file alternate we will quantify over the variables in a SEA to construct a set of bindings, which are used to construct a set of ground instances of the SEA. Our property involving files is therefore specified by the following non-ground SEA combined with a quantification $\forall$*file_name*.



Let us assume that the *domain* of $file\_name$ consists of the two values readme.txt and manual.pdf. We then construct two bindings, $[file\_name \mapsto \text{readme.txt}]$ and $[file\_name \mapsto \text{manual.pdf}]$, and instantiate the SEA to give the two ground SEA we saw previously. A trace is then accepted if it is accepted by *both* of these ground SEA, remembering that we will ignore (project out) symbols not in their alphabets. As before, to decide whether the QEA accepts the trace

$$\texttt{open}(\text{manual.pdf}).\texttt{open}(\text{readme.txt}).\texttt{close}(\text{readme.txt}).\texttt{close}(\text{manual.pdf})$$

we construct the two ground SEA and check if they accept the trace, using projection.

The language of this quantified SEA is the set of traces which interleave a valid trace of one ground SEA with a valid trace of the other i.e.:

$$\left\{\begin{array}{l}
\epsilon,\\
\texttt{open}(\text{readme.txt}).\texttt{close}(\text{readme.txt}),\\
\texttt{open}(\text{manual.pdf}).\texttt{close}(\text{manual.pdf}),\\
\texttt{open}(\text{readme.txt}).\texttt{close}(\text{readme.txt}).\texttt{open}(\text{manual.pdf}).\texttt{close}(\text{manual.pdf}),\\
\texttt{open}(\text{readme.txt}).\texttt{open}(\text{manual.pdf}).\texttt{close}(\text{readme.txt}).\texttt{close}(\text{manual.pdf}),\\
\texttt{open}(\text{readme.txt}).\texttt{open}(\text{manual.pdf}).\texttt{close}(\text{manual.pdf}).\texttt{close}(\text{readme.txt}),\\
\ldots
\end{array}\right\}$$

It is possible to construct a ground SEA that captures exactly this language, given in Fig. 3.1. However, note that this is only possible when we know the values that $file\_name$ can take and that it will often lead to a state explosion that will, in almost all occasions, be impractical. It will always be possible to construct such a ground SEA as the construction consists of standard union and intersection operations on automata.

This interleaving looks similar to the *shuffle product* on formal languages [BBC$^+$10]. The (ordinary) shuffle of two languages (over the same alphabet) is in arbitrary interleaving of

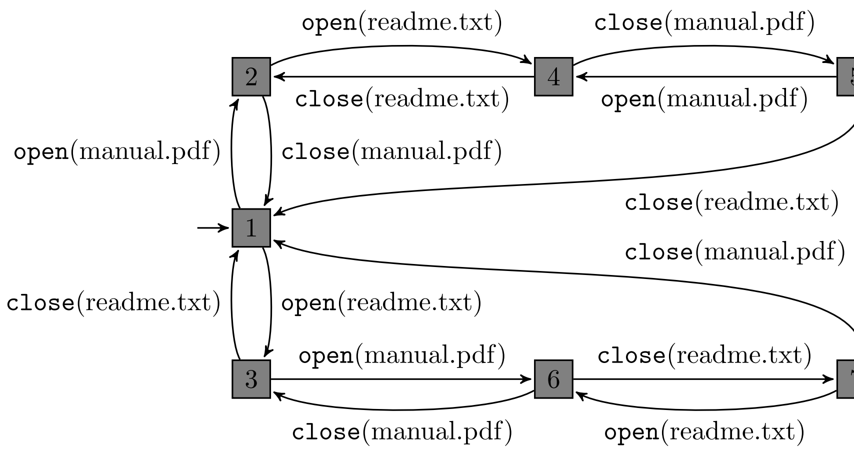


Figure 3.1: A ground SEA capturing the language interleaving of two other ground SEAs.

symbols from each language. Therefore, if a set of ground SEA's alphabets are disjoint then an accepted trace can be given as a shuffle of traces from these ground SEA. However, if one or more ground alphabets intersect then this is no longer the case as shared ground events would only occur once and the ground traces would need to 'synchronize' on their ordering. This might be the case if we have both $\texttt{f}(x)$ and $\texttt{f}(y)$ in the alphabet of the SEA. The event $\texttt{f}(1)$ would belong to the traces of the SEA instantiated with $[x \mapsto 1, y \mapsto 2]$ and $[x \mapsto 2, y \mapsto 1]$.

In the previous discussion we assumed a domain for the variable $file\_name$. To interpret quantifications we must define these domains i.e. what the variables are quantifying over. When we read the quantified SEA above we intuitively think of *file_name* as ranging over all files in the universe. However, this does not make sense as it is not possible to view all possible files in the universe within a finite trace. Instead, we are actually concerned with all files in a given trace, or to be more precise, all files which are opened or closed in the trace. Therefore, the domain of *file_name* should be all values that could replace *file_name* when matching any event in the trace with $\texttt{open}(\textit{file\_name})$ or $\texttt{close}(\textit{file\_name})$. This means that the domain of any variable is given by the trace and the SEA's alphabet. As the trace is finite, the domain of each quantified variable will also be finite and the generated set of bindings used to instantiate the SEA will be finite.

3.3.2 The structure

A simply quantified simple event automaton is a pair consisting of a set of universally quantified variables and a simple event automaton.

Definition 13 (Simply Quantified)**.** *A SQSEA $\mathcal{Q} = \langle X, \mathcal{E} \rangle$ is a pair of a set of quantified variables X and a simple event automaton $\mathcal{E}$ such that $X = \mathsf{vars}(\mathcal{E})$.*

Note that X can be empty and in this case our SQSEA is equivalent to a ground SEA. We introduce the concept of a binding being *total* with respect to the quantified variables of a SQSEA.

Definition 14 (Total binding). *Given a SQSEA with quantified variables X, a binding θ is* total *iff it binds exactly those variables in X*

$$\mathsf{total}(\theta, X) \text{ \textit{iff} } \mathsf{dom}(\theta) = X$$

We omit reference to X if it is obvious from context.

3.3.3 Defining acceptance

A trace τ is accepted by a SQSEA $\langle X, \mathcal{E}\rangle$ if τ is accepted by $\mathcal{E}(\theta)$ for every total binding θ of variables X. Before defining the language of a SQSEA we must first define the set of bindings that can be constructed from a trace.

We first derive the *domain* of each quantified variable from the trace by matching events in the SEA's alphabet against events in the trace.

Definition 15 (Derived Domain). *The derived domain of a trace τ for a SEA $\mathcal{E}$ with alphabet $\mathcal{A}$ is a map from variables to sets of values:*

$$\mathsf{Dom}^{\mathcal{E}}(\tau)(x) = \{\mathtt{match}(\mathbf{a}, \mathbf{b})(x) \mid \mathbf{b} = e(\ldots, x, \ldots) \in \mathcal{A} \wedge \mathbf{a} \in \tau \wedge \mathtt{matches}(\mathbf{a}, \mathbf{b})\}.$$

The SEA $\mathcal{E}$ is omitted if it is clear from context.

Next we construct all possible bindings from the derived domain and then select the relevant bindings. A binding can be constructed from $\mathsf{Dom}(\tau)$ if its domain is a subset of the domain of $\mathsf{Dom}(\tau)$ and it is consistent with $\mathsf{Dom}(\tau)$ i.e. only maps variables to values given by $\mathsf{Dom}(\tau)$. The relevant bindings are those that give a unique ground SEA after instantiation i.e. total bindings constructed from $\mathsf{Dom}(\tau)$

Definition 16 (Constructed Bindings). *Let us define the constructed and relevant bindings as follows:*

$$\begin{array}{lcl} \mathsf{construct}(\tau) & = & \{\theta \in \mathit{Bind} \mid \forall (x \mapsto v) \in \theta : v \in \mathsf{Dom}(\tau)(x)\} \\ \mathsf{relevant}(\tau, X) & = & \{\theta \in \mathsf{construct}(\tau) \mid \mathsf{total}(\theta, X)\} \end{array}$$

A trace is therefore accepted by a SQSEA iff for every relevant binding derived from the trace, the trace is in the (general) language of the instantiated SEA.

Definition 17 (Acceptance). *SQSEA $\langle X, \mathcal{E}\rangle$ accepts a ground trace τ iff*

$$\forall \theta \in \mathsf{relevant}(\tau, X) : \tau \in \mathcal{L}(\mathcal{E}(\theta))$$

Note that this requires us to have the full trace as $\mathsf{Dom}(\tau)$ is required to generate the relevant bindings. If this were not the case then we could easily transform this approach into one that takes one event at a time. Finally, we define the language of a SQSEA as the set of traces it accepts.

Definition 18 (SQSEA Language). *A trace τ is in the language of a SQSEA $\mathcal{Q}$, noted $\mathcal{L}(\mathcal{Q})$, iff it is accepted by $\mathcal{Q}$, in which case we say that τ satisfies $\mathcal{Q}$.*

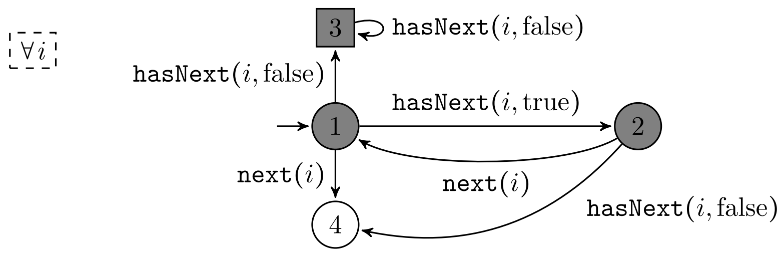


Figure 3.2: SQSEA for proper usage of `Java` iterators (HasNext)

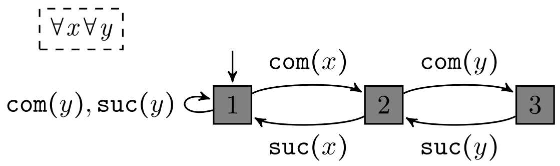


Figure 3.3: SQSEA for nested commands

3.3.4 Examples

We consider two examples of simply quantified simple event automata. Appendix A.1.1 gives a more detailed worked example. Recall the notation of automata introduced in Section 2.1.2.

HasNext

We first consider the proper usage of `java.util.Iterator` i.e. every call to `next` is preceded directly by a call to `hasNext` returning true, ensuring that we never call `next` on an empty iterator. The SQSEA for this property is given in Fig. 3.2. Here the values true and false are used directly in events labelling transitions, an alternative would be to add them to the event name. We use state 4 as a failure state; if we reach it we can never achieve acceptance. We could have used next states instead, as we do in state 3 where we capture the incorrect behaviour of calling `next` after `hasNext` returns false.

The rover system

The next property we consider will be related to an example scenario we will be using frequently in the rest of this thesis. We therefore take some space to describe this scenario here, but it is described in greater detail in Appendix A.4.

Our scenario consists of a number of inter-communicating autonomous rovers operating on a remote planet surface (for example, Mars) and a number of satellites orbiting the planet used to relay messages from a base station on the home planet, Earth. The rovers receive commands from the base station and send replies via the satellites. A command consists of a unique identifier, a command name and some payload data and all commands must be acknowledged using the commands identifier.

Nested Commands

The property considered here is that if a command with identifier y is issued after a command with identifier x is issued (but before it succeeds) then command with identifier y should succeed before command with identifier x i.e. commands nest. The SQSEA for this is given in Fig 3.3. Looping transitions on the initial state are required for the case where command y occurs before command x. This case will happen for half of the quantified variables as each value will appear in both the x and y position.

Consider the following trace:

$$\tau = \texttt{com}(1).\texttt{com}(2).\texttt{suc}(2).\texttt{com}(2).\texttt{suc}(1)$$

This gives us $\mathsf{Dom}(\tau) = [x \mapsto \{1,2\}, y \mapsto \{1,2\}]$, resulting in four relevant bindings. Let us just consider the binding $[x \mapsto 1, y \mapsto 1]$ and observe what happens in the presence of the non-determinism that this introduces. The projected trace is

$$\tau \downarrow_{\mathcal{E}([x\mapsto 1, y\mapsto 1])} \quad = \quad \texttt{com}(1).\texttt{suc}(1)$$

and as state 2 is a next state we have at least two possible sequences of transitions:

$$\begin{array}{c} 1 \xrightarrow{\texttt{com}(1)}_{\mathcal{E}([x\mapsto 1,y\mapsto 1])} 2 \xrightarrow{\texttt{suc}(1)}_{\mathcal{E}([x\mapsto 1,y\mapsto 1])} 1 \\ 1 \xrightarrow{\texttt{com}(1)}_{\mathcal{E}([x\mapsto 1,y\mapsto 1])} 2 \xrightarrow{\texttt{suc}(1)}_{\mathcal{E}([x\mapsto 1,y\mapsto 1])} q_\perp \end{array}$$

i.e. we can match $\texttt{suc}(1)$ with $\texttt{suc}(x)$ or $\texttt{suc}(y)$. As at least one of these ends in a final state we have $\tau \downarrow_{[x\mapsto 1, y\mapsto 1]} \in \mathcal{L}(\mathcal{E}([x \mapsto 1, y \mapsto 1])$. Note that τ does not satisfy the property as $\texttt{com}(2)$ is issued for a second time and does not succeed before $\texttt{com}(1)$ is issued, this error occurs for $[x \mapsto 1, y \mapsto 2]$.

3.4 Event Automata

We add free variables to simple event automata, along with guards and assignments to manipulate them, to define event automata. For SEA the notion of acceptance only applied to ground SEA and we directly matched events labelling transitions with ground events in a trace. Adding free variables means that our notion of instantiation can now be *partial* i.e. we can supply values for only *some* of the variables in the schema. We will differentiate between *free* variables and *quantified* variables. Free variables will take different values from matching events as we evaluate the trace and quantified variables will be initialised beforehand as before.

3.4.1 The need for local state

We begin by giving an example of using free variables in schema, before formally defining these new structures. We revisit our rover example (page 61), although this property could apply to any general setting where we have nodes (physical or virtual) sending messages to each other that must be acknowledged. We consider the property that a sent message must be

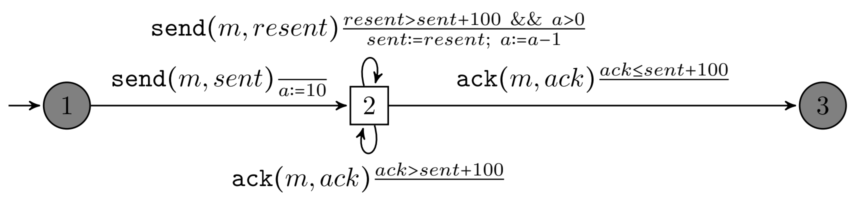

$\texttt{send}(m, resent)\,\frac{resent>sent+100\ \&\&\ a>0}{sent:=resent;\ a:=a-1}$
$\texttt{send}(m, sent)\,\frac{}{a:=10}$
$\texttt{ack}(m, ack)\,\frac{ack\leq sent+100}{}$
1
2
3
$\texttt{ack}(m, ack)\,\frac{ack>sent+100}{}$

Figure 3.4: A schema for describing how messages are sent and acknowledged.

Table 3.1: Evaluating a trace for the schema in Fig. 3.4 and a version instantiated with $[m \mapsto \mathrm{A}]$. Note the use of the implicit 'fail' state $q_\perp$.

τ	Uninstantiated					Instantiated with $[m \mapsto \mathrm{A}]$				
event	state	binding				state	binding			
		m	$sent$	$sent$	a		$sent$	$sent$	ack	a
	1	-	-	-	-	1	-	-	-	-
`send`(A,0)	2	A	0	-	10	2	0	-	-	10
`send`(A,110)	2	A	110	110	9	2	110	110	-	9
`send`(B,120)	$q_\perp$	B	120	110	9	*ignored*				
`ack`(A,220)	$q_\perp$	B	120	110	9	1	110	110	220	9
`send`(A,250)	$q_\perp$	B	120	110	9	2	250	110	220	8
`ack`(A,275)	$q_\perp$	B	120	110	9	3	250	110	275	8

acknowledged within 100 units of time and can be resent up to 10 times, but is only resent after the 100 units of time have elapsed. This is captured by the schema in Fig.3.4. The variable *sent* stores the last time the message was sent and a records the number of resend attempts, *resent* and *ack* are only used temporarily. Note that the alphabet contains the distinct events `send`$(m, sent)$ and `send`$(m, resent)$ that share the same event name.

We might ask why it matters *when* we bind variables; beforehand by initializing the automaton with a binding or whilst processing the trace. If we initialise the automaton with a binding we replace the variable with a value, effectively fixing it, whereas whilst processing the trace we can rebind it whenever we encounter it. For example, we can obtain another automaton by instantiating the schema in Fig. 3.4 with the binding $[m \mapsto \mathrm{A}]$. Table 3.1 describes how the state and binding is transformed for each automaton whilst stepping through a trace. Note that τ is projected on the alphabet of the automaton, and therefore the third event is not considered by the uninitialised schema. On the third message the schema rebinds the message m when it matches `send`(B,120) with `send`$(m, sent)$ and fails as it cannot take a transition. The trace is accepted by the instantiated automaton as it only considers those events where m is fixed as A. This tells us that it is necessary to instantiate schema and have two different kinds of variables: those that are replaced and those that are not.

3.4.2 Adding free variables

An event automaton extends the notion of simple event automaton by adding guards and assignments to transitions.

Definition 19 (Event Automaton)**.** *An Event Automaton $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ is a tuple where Q is a finite set of states, $\mathcal{A} \subseteq Event$ is a finite alphabet, $\delta \subseteq (Q \times \mathcal{A} \times Guard \times Assign \times Q)$ is a finite transition set, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states. As before, the variables of $\mathcal{E}$ are those that appear in its alphabet:*

$$\mathsf{vars}(\mathcal{E}) = \{x \mid \exists e(\overline{s}) \in \mathcal{A} : x \in \overline{s} \ \wedge \ x \in Var\}.$$

Note that we do not differentiate between quantified and free variables; we are allowed to replace any variables using instantiation and leave any variables free to be updated. Quantified event automata introduced in the next section will use quantifications to separate these properly.

3.4.3 Extending the notion of instantiation

The main difference in EA instantiation is that we remove references to quantified variables from guards and assignments by partially evaluating them.

Definition 20 (Event Automaton Instantiation)**.** *Given a binding θ, let $\mathcal{E}(\theta) = \langle Q, \mathcal{A}(\theta), \delta(\theta), q_0, F\rangle$ be the θ-instantiation of EA $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ where*

$$\begin{array}{rcll} \mathcal{A}(\theta) & = & \{\mathbf{b}(\theta) \mid \mathbf{b} \in \mathcal{A}\} & \\ (q, \mathbf{b}(\theta), g', \gamma', q') \in \delta(\theta) & \textit{iff} & (q, \mathbf{b}, g, \gamma, q') \in \delta & \textit{and } g'(\varphi) = g(\theta \ \dagger \varphi) \\ & & & \textit{and } \gamma'(\varphi) = \gamma(\theta \ \dagger \varphi). \end{array}$$

Acceptance with respect to a binding is then taken as before i.e. $\tau \in \mathcal{L}(\mathcal{E}(\theta))$.

3.4.4 Extending previous constructions to EA

Finally, we update the notion of language acceptance to account for free variables, assuming that quantified variables have been removed. Firstly, we define the ground alphabet of an EA as the set of all ground events that could match an event in the EA's alphabet.

Definition 21 (Ground Alphabet)**.** *The ground alphabet of EA $\mathcal{E}$ with alphabet $\mathcal{A}$ is*

$$\mathtt{ground}(\mathcal{E}) = \{\mathbf{a} \in GEvent \mid \exists \mathbf{b} \in \mathcal{A} : \mathtt{matches}(\mathbf{a}, \mathbf{b})\}$$

Now we introduce the transition relation for EA, this is where transition guards and assignments are dealt with. Previously this relation was only required to 'close' δ but here it also keeps track of bindings of free variables. It is important to note that the values associated with free variables will be updated whenever they match with a new event.

Definition 22 (Configurations and Transition Relation)**.** *Let $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ be our EA. Let $\langle q, \theta\rangle \in Config = Q \times Bind$ be an $\mathcal{E}$-*configuration*. Let $\langle q, \varphi\rangle \xrightarrow{\mathbf{a}} \langle q', \varphi'\rangle$ hold if*

$$\begin{array}{l} \exists \mathbf{b} \in \mathcal{A}, \exists g \in Guard, \exists \gamma \in Assign : (q, \mathbf{b}, g, \gamma, q') \in \delta \ \wedge \\ \qquad\qquad \mathtt{matches}(\mathbf{a}, \mathbf{b}) \wedge g(\varphi \dagger \mathtt{match}(\mathbf{a}, \mathbf{b})) \wedge \varphi' = \gamma(\varphi \dagger \mathtt{match}(\mathbf{a}, \mathbf{b})). \end{array}$$

Let $c \xrightarrow{\mathbf{a}}_{\mathcal{E}} c'$ hold if $c \xrightarrow{\mathbf{a}} c'$ holds. Let $c \xrightarrow{\mathbf{a}}_{\mathcal{E}} c$ hold if there is no c' such that $c \xrightarrow{\mathbf{a}} c'$ holds.

Again, we lift this to traces in the standard way. The $\rightarrow$ relation relates configurations $\langle q, \varphi \rangle$ and $\langle q', \varphi' \rangle$ by ground event **a** if there exists a transition in δ starting in q, s.t. the events match, the guard is satisfied, and the new configuration contains state q' and the binding given by the assignment. This lifts δ to configurations but may be undefined on some configuration/ground event pairs as δ is not necessarily complete. The $\rightarrow_{\mathcal{E}}$ completes this relation, and therefore can be written as a function $\mathit{Config} \times \mathit{GEvent} \rightarrow 2^{\mathit{Config}}$. This completion follows a skip semantics as before but note that it does not include matching so the binding in a configuration remains unchanged when taking an *implicit* self-loop but might be updated taking the same *explicit* self-loop.

Like SEA, EA are non-deterministic. This not only comes from the fact that δ may define two different transitions with the same event, but also because two different events can match the same ground event if they have the same name. We will see an example of this later when we discuss non-determinism for QEA.

Note. *With the introduction of guards and assignments the next-state translation becomes slightly more complicated as the complement of outgoing transitions must take the complement of guards. We assume that the free variables and assignments used in implicit failure transitions are such that the bindings remain unchanged.*

The language of an EA is then the set of all traces over the ground alphabet that reach an accepting state using the EA's transition relation.

Definition 23 (Event Automaton Language)**.** *The language of the EA $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F \rangle$ is defined as*

$$\mathcal{L}(\mathcal{E}) = \{\tau \in \mathtt{ground}(\mathcal{E})^* \mid \exists \langle q, \varphi \rangle \in \mathit{Config} : \langle q_0, [\ ] \rangle \xrightarrow{\tau}_{\mathcal{E}} \langle q, \varphi \rangle \wedge q \in F\}$$

As before we define a general language using projection.

Definition 24 (Event Automaton General Language)**.** *The general language of an EA $\mathcal{E}$ given alphabet $\mathcal{B}$ such that* $\mathsf{ground}(\mathcal{E}) \subseteq \mathcal{B}$ *is defined as*

$$\mathcal{L}_G(\mathcal{E}, \mathcal{B}) = \{\tau \in \mathcal{B}^* \mid \tau \downarrow_{\mathsf{ground}(\mathcal{E})} \in \mathcal{L}(\mathcal{E})\}$$

we omit $\mathcal{B}$ when it is clear from context.

3.4.5 Example

Let us consider an auction bidding site where items are posted and bid on. In this example we are only concerned with bidding events and assume that every time a bid is made on the site we record an event `bid`$(\mathit{item}, \mathit{amount})$ that records a unique identifier for the item and the amount bid. The property we want to capture is that bids on an item are strictly increasing.

The variable for the item will be quantified, as we are considering the behaviour of a single item we will fix it before analysing a trace. However, the value for amount should change during the trace. Therefore, if a transition is labelled with the event `bid`(1,*amount*) and we receive the event `bid`(1,10) we bind *amount* to 10 no matter what previous value *amount* had. So that

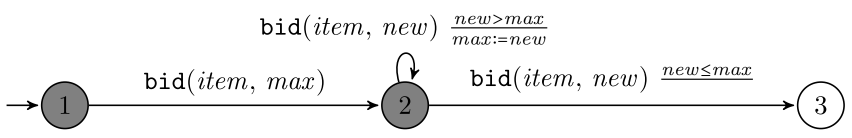

bid(*item*, *new*) $\frac{new>max}{max:=new}$
bid(*item*, *max*)
bid(*item*, *new*) $\frac{new \leq max}{}$
1
2
3

Figure 3.5: A schema for describing how items are bid on in an auction.

we can check that *amount* is greater than the previous bid amount we keep track of a current binding of free variables as well as a current state in *configurations*.

Fig. 3.5 gives an EA for this example. To use this the variable *item* should be instantiated to a particular item. The variables *max* and *new* are then updated as the trace is processed i.e., consider the following transition sequence for a trace dealing with a 'hat' item.

$$\langle 1, [\ ] \rangle \xrightarrow{\texttt{bid}(`hat',4)} \langle 2, [max \mapsto 4] \rangle \xrightarrow{\texttt{bid}(`hat',5)} \langle 2, [max \mapsto 5, new \mapsto 5] \rangle \xrightarrow{\texttt{bid}(`hat',20)}$$
$$\langle 2, [max \mapsto 20, new \mapsto 20] \rangle \xrightarrow{\texttt{bid}(`hat',18)} \langle 3, [max \mapsto 20, new \mapsto 18] \rangle$$

Note how guards are used to decide which state can be reached and an assignment is used to update the *max* bid value.

An important observation is that the values bound to the free variables *max* and *new* change during the processing of the trace. The QEA specification does not make sense without this behaviour.

We give more detailed examples in Appendix A.1.2, including a demonstration of how EA can be used to monitor and detect SQL injection. We also discuss free variables and their relation to concepts in other specification languages in Sec. 3.5.10.

3.5 Quantified Event Automata

In this section we introduce quantified event automata as a combination of quantifications with event automaton. We introduce existential quantification, given domains, type variables and global guards as advanced quantification concepts. In SQSEA we universally quantified all variables of a SEA but here we are adding quantifications to EA which have the notion of free and bound variables. In this case we will only quantify some (or none) of an EA's variables, making an EA a special case of a QEA. As well as the increased expressiveness of EA we will also extend what we are allowed to do in quantifications, as motivated in the next few sections.

3.5.1 The need for existential quantification

It seems natural to universally quantify variables as restrictions usually apply to all instances of a certain object. However, we may also want to existentially quantify variables. For example, if we consider the communication between rovers in our rover example (see page 61) we may want to specify that we find one rover that has successfully sent a message to all other rovers. A QEA for this property is shown in Fig. 3.6. This contains two new constructs: existential quantification and global guards. The global guard *control* ≠ *rover* is used to exclude any

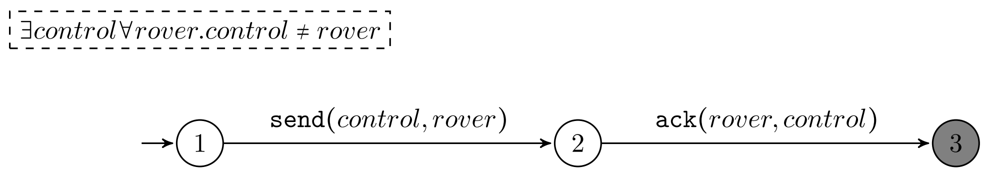


Figure 3.6: There exists a controller rover who has sent a message to all rovers (that have been sent messages).

bindings where *control* and *rover* are the same. The trace

$$\texttt{send}(A,B).\texttt{send}(B,C).\texttt{ack}(C,B).\texttt{send}(C,A).\texttt{send}(A,C).\texttt{ack}(B,A).\texttt{ack}(C,A)$$

satisfies this property as the rover A sends a message to all other rovers (B and C) and has that message acknowledged. Neither B nor C achieve this and note that A does not send a message to itself.

3.5.2 Joining the domains of quantified variables

The previous example has a possibly problematic quality; controllers are not considered rovers. Take, for example, the trace $\tau = \texttt{send}(A,B).\texttt{send}(C,B).\texttt{ack}(B,A)$. The derived domain in this case is $\mathsf{Dom}(\tau) = [control \mapsto \{A,C\}, rover \mapsto \{A,B\}]$. The trace is accepted even though A does not send a message to C as there is a lack of symmetry in the events $\texttt{send}(control, rover)$ and $\texttt{ack}(rover, control)$. If we want rovers to be all things that send and receive messages we could add the event $\texttt{send}(rover, control)$ to the alphabet, but this seems redundant as we never want to use this event.

Instead, we will introduce the concept of *type variables* that indicate that two or more quantified variables should share a domain. Therefore, the quantification would be $\exists control : R \forall rover : R.control \neq rover$. With this quantification the QEA does not accept the above trace as C is included in the domain of *rover*.

3.5.3 When might we not want to derive the domain

Previously we derived the domains of quantified variables directly from the trace. However, there may be circumstances where it is necessary to 'override' this domain when there is prior knowledge of some objects that should (or should not) occur in the trace and the specification checks that they do (or do not). For example, consider a modification to the above property. Instead of combining the domains of rover and control perhaps instead we want the controlling rover to be taken from a predefined set of known 'good' controllers. We could achieve this by overriding the type variable given to *control* so that this is used when evaluating traces in place of the derived one, meaning that only prescribed rovers could satisfy the existence criteria.

3.5.4 Formalising our new notion of quantification

We now introduce quantified event automata (QEA) in their final form. These consist of an EA with some or none of its variables quantified by ∀ or ∃ with a, possibly overridden, type variable and global guard. *Type variables* are symbols used to denote sets of values, let $\mathcal{T}$ be the set of all type variables. We use X, Y to denote type variables. The domain of a variable will be given by its type variable and may either be explicitly given or derived from the trace.

Definition 25 (Quantified Event Automaton). *A QEA is a tuple $\langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ where $\mathcal{E}$ is a EA, $\Lambda \in (\{\forall, \exists\} \times \mathsf{vars}(\mathcal{E}) \times \mathcal{T} \times \mathit{Guard})^*$ is a list of quantified, typed, variables with guards and $\mathcal{D} \in \mathcal{T} \rightharpoonup 2^{\mathit{Var}}$ is a (partial) type domain map. The variables quantified by a QEA are given by*

$$\mathsf{vars}(\Lambda) = \{x \in \mathit{Var} \mid (\_, x, \_, \_) \in \Lambda\}$$

The partial type domain $\mathcal{D}$ map can give explicit domains to type variables. If a type variable is not defined in $\mathcal{D}$ then its domain will be derived in a similar way to what happened for SQSEA in Def. 15. A QEA is well-formed if each variable in $\mathsf{vars}(\mathcal{E})$ appears at most once in Λ. Our previous definition of total bindings (Def. 14) can be lifted to this new form of quantifier list by using $\mathsf{vars}(\Lambda)$ in place of X.

3.5.5 Defining acceptance

We introduce a new form of acceptance based on this updated form of quantification. First we update our concept of a derived domain to account for type variables.

Definition 26 (Derived Domain). *Let $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ be a QEA. Define* varsOf *and* typeOf *for converting between type variables and quantified variables as follows:*

$$\begin{array}{rcl} \mathsf{varsOf}(X) & = & \{x \in \mathit{Var} \mid (\_, x, X, \_) \in \Lambda\} \\ \mathsf{typeOf}(x) & = & X \in \mathcal{T} \text{ if } x \in \mathsf{varsOf}(X) \end{array}$$

Let $\mathsf{Dom}^{\mathcal{E}}(\tau)$ map type variables to their domains as follows:

$$\begin{array}{rl} \mathsf{Dom}^{\mathcal{E}}(\tau)(X) = \{\mathtt{match}(\mathbf{a}, \mathbf{b})(x) \mid & x \in \mathsf{varsOf}(X) \wedge \mathbf{b} = e(\ldots, x, \ldots) \in \mathcal{A} \wedge \\ & \qquad \mathbf{a} \in \tau \wedge \mathtt{matches}(\mathbf{a}, \mathbf{b})\}. \end{array}$$

We omit $\mathcal{E}$ if it is clear from context.

Note that typeOf is well-defined if $\mathcal{Q}$ is well-formed. We are in a position to define acceptance using the quantification list and the constructions previously mentioned. To do this we walk down the quantification list building up a total binding and then use this binding to check whether an instantiated EA is satisfied i.e. the trace is in its general language.

Definition 27 (Acceptance). *QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ accepts a ground trace τ iff $\tau \vDash_{[\,]}^{\mathcal{E}, \mathsf{Dom}(\tau) \dagger \mathcal{D}} \Lambda$ where $\vDash_{\theta}^{\mathcal{E},\mathcal{D}}$ is defined as*

$$\begin{array}{lll} \tau \vDash_{\theta}^{\mathcal{E},\mathcal{D}} (\forall, x, X, g).\Lambda' & \textit{iff} & \textit{for all } d \textit{ in } \mathcal{D}(X) \textit{ if } g(\theta \dagger [x \mapsto d]) \textit{ then } \tau \vDash_{\theta \dagger [x \mapsto d]}^{\mathcal{E},\mathcal{D}} \Lambda' \\ \tau \vDash_{\theta}^{\mathcal{E},\mathcal{D}} (\exists, x, X, g).\Lambda' & \textit{iff} & \textit{for some } d \textit{ in } \mathcal{D}(X), g(\theta \dagger [x \mapsto d]) \textit{ and } \tau \vDash_{\theta \dagger [x \mapsto d]}^{\mathcal{E},\mathcal{D}} \Lambda' \\ \tau \vDash_{\theta}^{\mathcal{E},\mathcal{D}} \epsilon & \textit{iff} & \tau \in \mathcal{L}(\mathcal{E}(\theta)) \end{array}$$

We omit $\mathcal{E}$ and $\mathcal{D}$ when they are clear from context and write $\vDash$ for $\vDash_{[\,]}$.

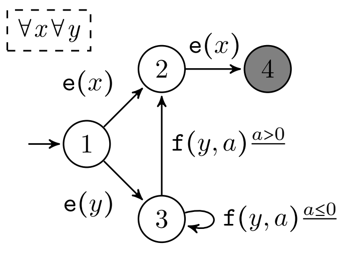
Finally, the language of a QEA is the set of traces it accepts.

Definition 28 (QEA Language). *The language of a QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ is noted and defined as*

$$\mathcal{L}(\mathcal{Q}) = \{\tau \mid \tau \models \Lambda\}$$

If $\tau \in \mathcal{L}(\mathcal{Q})$ we say that τ satisfies $\mathcal{Q}$.

3.5.6 Demonstrating non-determinism



Non-deterministic QEA

We briefly consider the different ways in which a QEA can be non-deterministic. Let the QEA on the right be $\mathcal{Q} = \langle \forall x \forall y, \mathcal{E}, [\,] \rangle$ and consider the trace

$$\tau = \mathtt{e}(1).\mathtt{e}(2).\mathtt{f}(2,0)$$

By computing $\mathsf{Dom}(\tau) = [x \mapsto \{1,2\}, y \mapsto \{1,2\}]$ we find that there are four bindings which should be used to instantiate $\mathcal{E}$ and and therefore project τ:

$$\begin{array}{lcl@{\qquad\qquad}lcl} \tau\downarrow_{[x\mapsto 1, y\mapsto 1]} & = & \mathtt{e}(1) & \tau\downarrow_{[x\mapsto 2, y\mapsto 1]} & = & \mathtt{e}(1).\mathtt{e}(2).\mathtt{f}(2,0) \\ \tau\downarrow_{[x\mapsto 1, y\mapsto 2]} & = & \mathtt{e}(1).\mathtt{e}(2).\mathtt{f}(2,0) & \tau\downarrow_{[x\mapsto 2, y\mapsto 2]} & = & \mathtt{e}(2).\mathtt{f}(2,0) \end{array}$$

The configurations reached by the first and last of these projections are

$$\langle 1, [\,] \rangle \xrightarrow{\mathtt{e}(1)}_{\mathcal{E}([x\mapsto 1, y\mapsto 1])} \langle 2, [\,] \rangle \quad \text{and} \quad \langle 1, [\,] \rangle \xrightarrow{\mathtt{e}(1)}_{\mathcal{E}([x\mapsto 1, y\mapsto 1])} \langle 3, [\,] \rangle$$

as both $\mathtt{e}(x)$ and $\mathtt{e}(y)$ match with $\mathtt{e}(1)$, and

$$\begin{array}{c} \langle 1, [\,] \rangle \xrightarrow{\mathtt{e}(2)}_{\mathcal{E}([x\mapsto 2, y\mapsto 2])} \langle 3, [\,] \rangle \xrightarrow{\mathtt{f}(2,0)}_{\mathcal{E}([x\mapsto 2, y\mapsto 2])} \langle 3, [a \mapsto 0] \rangle \\ \text{and} \\ \langle 1, [\,] \rangle \xrightarrow{\mathtt{e}(2)}_{\mathcal{E}([x\mapsto 2, y\mapsto 2])} \langle 3, [\,] \rangle \xrightarrow{\mathtt{f}(2,0)}_{\mathcal{E}([x\mapsto 2, y\mapsto 2])} \langle 2, [a \mapsto 0] \rangle \end{array}$$

as $a > 0$ and $a \leq 0$ are both true for $a = 0$. Demonstrating two forms of non-determinism. We can say that EA $\mathcal{E}$ is deterministic if at most one configuration is related to two configurations by an event i.e.

$$\forall c_1, c_2, c_3 \in \mathit{Config}, \forall \mathbf{a} \in \mathsf{ground}(\mathcal{E}) : c_1 \xrightarrow{\mathbf{a}}_{\mathcal{E}} c_2 \wedge c_1 \xrightarrow{\mathbf{a}}_{\mathcal{E}} c_3 \Rightarrow c_2 = c_3$$

Note that non-determinism is also effected by the quantification list. If we exchange $\forall x \forall y$ for $\forall x \forall y : x \neq y$ then we eliminate the first form of non-determinism above.

3.5.7 Building relevant bindings

Let us update our definitions of constructed and relevant bindings (previously Def. 16) within the context of type variables. A binding can be constructed from $\mathsf{Dom}(\tau)$ if its domain is a subset of the domain of $\mathsf{Dom}(\tau)$ and it is consistent with $\mathsf{Dom}(\tau)$ i.e. only maps variables to values given by $\mathsf{Dom}(\tau)$.

Definition 29 (Constructed Bindings with Type Variables)**.** *Let* construct *be updated as follows:*

$$\mathsf{construct}(\tau) = \{\theta \in \mathit{Bind} \mid \forall (x \mapsto v) \in \theta : v \in \mathsf{Dom}(\tau)(\mathsf{typeOf(x)})\}$$

The notion of relevant bindings should now consider global guards. The relevant bindings are those that give a unique ground EA after instantiation, i.e. total bindings constructed from $\mathsf{Dom}(\tau)$, and where bindings satisfy global guards.

Definition 30 (Relevant Bindings with Global Guards)**.** *Let* relevant *be updated as follows:*

$$\begin{array}{lcl} \mathsf{relevant}(\tau,\epsilon) & = & \{\theta \in \mathsf{construct}(\tau) \mid \mathsf{total}(\theta, \mathsf{vars}(\Lambda))\} \\ \mathsf{relevant}(\tau,(\_,\_,g,\_).\Lambda') & = & \{\theta \in \mathsf{relevant}(\tau,\Lambda') \mid g(\theta)\} \end{array}$$

3.5.8 Connectedness as a special global guard

We introduce a concept called *connectedness*[3] and show how to construct a special global guard that filters out bindings that are not connected, we will define what this means shortly. This approach could also be used to introduce other special global guards that filter out sets of bindings with certain properties. These global guards are special as they access information about events in the trace, which seems reasonable as the domains of the quantified variables are also reliant on this.

We begin by introducing the concept of a binding being connected for a given trace. A binding is connected if there exist events in the trace that 'connect' the variables it binds. For example, if the alphabet consisted of events $\mathtt{a}(x,y)$ and $\mathtt{b}(y,z)$ and the trace was $\mathtt{a}(1,2).\mathtt{b}(2,3).\mathtt{b}(3,4)$ then the binding $[x \mapsto 1, y \mapsto 2, z \mapsto 3]$ would be connected as $\mathtt{a}(1,2)$ connects $[x \mapsto 1]$ with $[y \mapsto 2]$ and $\mathtt{b}(2,3)$ connects $[y \mapsto 2]$ with $[z \mapsto 4]$. But $[x \mapsto 1, y \mapsto 3, z \mapsto 4]$ would not be connected as there is no event connecting $[x \mapsto 1]$ with $[y \mapsto 3]$ or $[z \mapsto 4]$.

Definition 31 (Connected Binding)**.** *Given a trace τ and an QEA $\langle \Lambda, \mathcal{E}, \mathcal{D}\rangle$ with alphabet $\mathcal{A}$, let the set of τ-connected bindings be the smallest set $C(\tau)$ such that:*

$$\begin{array}{lll} \varphi & \in C(\tau) \textit{ iff} & \exists \mathbf{a} \in \tau, \exists \mathbf{b} \in \mathcal{A}. \varphi \sqsubseteq \mathsf{match}(\mathbf{a},\mathbf{b}) \wedge \mathsf{dom}(\varphi) \in \mathsf{vars}(\Lambda) \\ \varphi_1 \sqcup \varphi_2 & \in C(\tau) \textit{ iff} & \varphi_1, \varphi_2 \in C(\tau) \wedge \varphi_1 \cap \varphi_2 \neq \emptyset \end{array}$$

where $\sqcup$ is the least upper bound operator on maps defined as the smallest (wrt $\sqsubseteq$) binding containing φ_1 and φ_2.

We use this notion of connectedness to introduce a special global guard that accepts a binding for a set of variables X iff the submap defined for X is connected.

Definition 32 (Connected Guard)**.** *Given a trace τ and set of variables X, define the guard* $\mathsf{connected}(X)$ *as follows*

$$\mathsf{connected}(X)(\theta) \textit{ iff } X \subseteq \mathsf{dom}(\theta) \textit{ and } [(x \mapsto \theta(x)) \mid x \in X] \in C(\tau)$$

[3]This concept is only explicitly discussed in work on JavaMOP [MJG+11] who have a special 'connectedness mode' that alters the acceptance semantics.

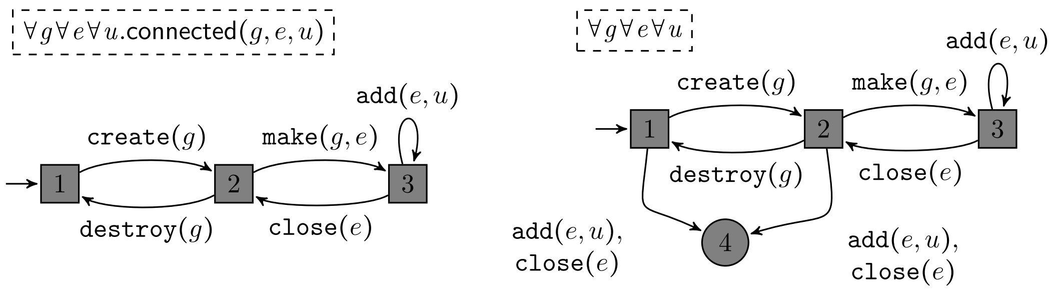


Figure 3.7: Two QEA specifying the (same) behaviour of an event planning system where users (u) are added to an event (e) created in a group (g)

An example

Consider an online event planning system (or a social network where users can organise their own events) which has groups, users and events. Groups can be created and destroyed, events can be created in, and removed from, groups and users can be added to events. The property that we consider is that an event can only be created for a group that exists and users can only be invited to an event that is open.

The two QEA in Figure 3.7 specify the desired behaviour, let us refer to these as $\mathcal{J}$ and $\mathcal{K}$. Note how connectedness makes $\mathcal{J}$ more concise than $\mathcal{K}$ as we do not need to include transitions for the case where the given event is not made in the given group. To better understand how this works consider the following trace.

$$\tau = \texttt{create}(\mathrm{A}).\texttt{create}(\mathrm{B}).\texttt{make}(\mathrm{A},\mathrm{M}).\texttt{add}(\mathrm{M},1)$$

We have $\mathsf{Dom}(\tau) = [g \mapsto \{A, B\}, e \mapsto \{M\}, u \mapsto \{1\}]$ so there are two total bindings: $\theta_1 = [g \mapsto \mathrm{A}, e \mapsto \mathrm{M}, u \mapsto 1]$ and $\theta_2 = [g \mapsto \mathrm{B}, e \mapsto \mathrm{X}, u \mapsto 1]$. The projections are

$$\tau\downarrow_{\theta_1} = \texttt{create}(\mathrm{A}).\texttt{make}(\mathrm{A},\mathrm{X}).\texttt{add}(\mathrm{M},1) \qquad \tau\downarrow_{\theta_2} = \texttt{create}(\mathrm{B}).\texttt{add}(\mathrm{X},1)$$

and therefore the states reached are as follows.

$$1 \xrightarrow{\tau\downarrow_{\theta_1}}_{\mathcal{J}.\mathcal{E}(\theta_1)} 3 \text{ and } 1 \xrightarrow{\tau\downarrow_{\theta_2}}_{\mathcal{J}.\mathcal{E}(\theta_2)} q_\perp \quad , \quad 1 \xrightarrow{\tau\downarrow_{\theta_1}}_{\mathcal{K}.\mathcal{E}(\theta_1)} 3 \text{ and } 1 \xrightarrow{\tau\downarrow_{\theta_2}}_{\mathcal{K}.\mathcal{E}(\theta_2)} 4$$

To decide acceptance for $\mathcal{J}$ we must construct $C(\tau)$ as follows

$$C(\tau) = \left\{ \begin{array}{c} [g \mapsto A], [g \mapsto B], [e \mapsto M], [u \mapsto 1], [g \mapsto A, e \mapsto M], \\ [e \mapsto M, u \mapsto 1], [g \mapsto A, e \mapsto M, u \mapsto 1] \end{array} \right\}$$

Both QEA accept τ, as when deciding if τ satisfies $\mathcal{J}$ the non-connected binding θ_2 is not considered. State 4 is required in $\mathcal{K}$ as without it θ_2 would lead to failure.

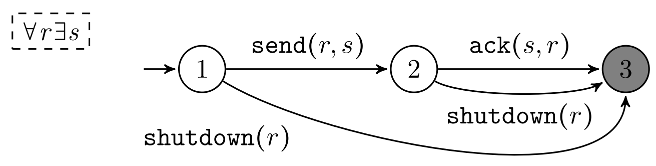


Figure 3.8: Every rover must send a message that is acknowledged by a satellite, unless the rover is shutdown.

3.5.9 The partial quantifier 'trick'

We discuss a potential problem with quantification and propose a solution in the form of an alternative kind of *partial* quantifier that is defined in terms of existing constructs.

As motivation consider a property (related to the acknowledgements property discussed earlier) that states that every rover must send a message that is acknowledged by a satellite *unless the rover is shut down*. The QEA in Fig. 3.8 attempts to capture this property but the trace consisting of a single event $\texttt{shutdown}(A)$ will not satisfy the QEA.If a quantified variable's domain is empty then the standard rules for universal or existential quantification over an empty domain will apply. In this case, the problem is that the domain of s is empty and an empty existential quantification is false.

The 'trick' is to introduce a dummy value into a domain if it is empty. In the previous example if we added 1 to the domain of s then the QEA would accept the trace. The introduction of the dummy value can be achieved by making the definition of a type variable's domain dependent on the trace. For example,

$$\mathcal{D}(X) = \begin{cases} \{dummy\} & \text{if } \mathsf{Dom}(\tau)(X) = \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Let us use the notation of $\hat{\forall}$ and $\hat{\exists}$ for this partial quantifier trick. We would therefore replace the quantifications in Fig. 3.8 with $\forall r, \hat{\exists} s$. Appendix A.1.3 gives another example.

3.5.10 A discussion of free variables

Finally, we return to the concept of free variables, which we can now discuss within the context of QEA. As explained in Sec. 3.4, the values associated with free variables may be updated during the processing of a trace. This might seem non-standard from a logic perspective but is common in automata theory, although they are often called registers [NSV04].

From a temporal logic viewpoint we might consider these as constants, rather than variables, which are interpreted at each time point. However, this view struggles to account for assignments updating and introducing new free variables.

Free variables can be viewed as existentially quantifying the variable over a limited scope. Note that an assignment updating the variable, or another occurrence of the variable would end this scope. To explore this notion consider the following first-order LTL specification:

$$\forall x. \square\, (\texttt{f}(x) \rightarrow \exists y. \lozenge (\texttt{g}(x, y) \wedge y > x))$$

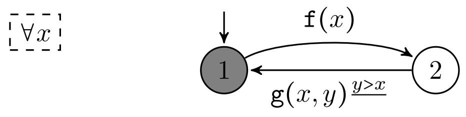


Figure 3.9: A QEA for the first-order LTL property $\forall x. \Box(\texttt{f}(x) \rightarrow \exists y. \Diamond(\texttt{g}(x,y) \wedge y > x))$.

This property is captured by the QEA in Fig. 3.9, which uses a free variable for y. This accepts the trace

$$\texttt{f}(1).\texttt{f}(2).\texttt{g}(1,3).\texttt{g}(2,1).\texttt{f}(1).\texttt{g}(1,5).\texttt{g}(2,5)$$

as for each trace projection (for $x = 1, 2$) the variable y is assigned to the relevant value. For $x = 1$ we have the trace projection $\texttt{f}(1).\texttt{g}(1,3).\texttt{f}(1).\texttt{g}(1,5)$ and y is bound to 3 first and then 5 later. For $x = 2$ the trace projection is $\texttt{f}(2).\texttt{g}(2,1).\texttt{g}(2,5)$ and we first attempt to take the transition from state 2 to 1 with $\texttt{g}(2,1)$ but this fails to satisfy the guard $y > x$, which is later satisfied by $\texttt{g}(2,5)$.

3.6 Summary

In this chapter we have defined QEA by building up the notions of EA and different forms of quantification. It should be noted that the QEA presented here are different from those described in the published paper [BFH+12]. The differences are that the QEA introduced in the paper did not have a notion of external domain or type variables. These were omitted for space reasons.

There are further worked examples of QEA in Appendix A. These may be helpful in exemplifying how the different concepts in QEA can be used to define different kinds of properties.

Chapter 4

Properties of QEA

This chapter explores the QEA formalism. We discuss styles of writing QEA and its limitations as well as the complexity of the trace checking process and general expressiveness.

Structure. We begin (Sec. 4.1) by discussing the stylistic choices when writing QEA and how this relates to common notions of temporal properties, followed by a discussion of the issues involved in specifying for different application domains (Sec. 4.1.1). We then (Sec. 4.2) use example QEAs to discuss the limitations our approach. We follow this by discussing the complexity of the trace checking process (Sec. 4.3) and the expressiveness of our formalism (Sec. 4.4). Finally we review other parametric runtime monitoring approaches and how they compare to QEA (Sec. 4.5).

4.1 Exploring specification style

In this section we explore the different styles of writing specifications.

4.1.1 Specifying for different domains

To help us understand the differences in specifying properties in different domains we have chosen two different domains and written a range of properties in them. These can be found in Appendix A and will be used during our evaluation in Chapter 7. Here we discuss the specifications and what we found when writing them.

The `Java` API

Our first domain is that of the `Java` standard library, this has often been used in previous runtime verification work. We do not attempt to be comprehensive, as in other work [LJMR12] but select interesting properties. Appendix A.3 discusses specifications from three categories. The *communication* mainly considers resource usage in terms of readers, writers and sockets. The *collections* category considers the safe usage of iterators in different contexts and the persistence of hashes used hashing structures. The *concurrency* category considers asynchronous access to synchronous collections and lock properties. The HasNext property on page 61 is part of this domain, as is the UnsafeIter property used as an example in the next section.

To keep specifications concise we mapped sets of methods to events, selecting which parameters and return values should appear in the event, this process is part of the instrumentation effort. All of these properties use only universal quantification and are safety specifications (discussed below). Free variables were only used to implement the context-free property CloseFiles on page 342. Specifications are generally quite small, mentioning only a few events and at most two quantified variables.

Our planetary rover case study

This is the running example first mentioned in Sec. 3.3.4. The general setting and a range of specifications are given in Appendix A.4. The general idea is that we have a number of entities that can communicate using messages that are either commands or acknowledgements. We also consider the internal behaviour of the planetary rovers. These have a number of resources that should be managed and can run tasks, which execute the received commands.

We split our properties into those about *external* behaviour, i.e. communication, and *internal* behaviour. We have already discussed some external behaviour, i.e. NestedCommands on page 62, AcknowledgeCommands on page 62 and ExistsLeader on page 66. The internal specifications were first presented in our published book chapter [FHR13] and were inspired by a co-author's experience at NASA JPL.

Some of these specifications are more complex than those used in the `Java` API domain. They make extensive use of free variables, use existential quantification and in some cases have larger alphabets. One major difference is that we do not have the notion of objects being inherently *connected* to each other (a style discussed below).

4.1.2 Validation versus violation

There are generally two styles for describing trace properties. In the violation style we describe the set of traces that we should belong to, whereas in validation style we describe the set of traces we should not belong to. The terminology may seem backwards, but it describes when we have detected an error i.e. have we violated our description or validated it.

To demonstrate the two different styles consider the UnsafeIter property introduced in Appendix A.3.2. This states that an iterator created from a collection should not be used after the collection is updated. Fig. 4.1 gives two QEAs for this property. The first uses a violation style, describing all of the safe behaviours, and the second uses a validation style, describing the unsafe behaviour. The first returns false for an incorrect trace but the second returns true.

The style is separate from the verdict we return, although the two are related. If we negated the second QEA we get the QEA defined in Fig. A.16 in Appendix A.3.2 where we describe the failing behaviour but still produce a false verdict for an incorrect trace. Some systems, such as TraceMatches only detect validations of a property and therefore the property must evaluate to true on an incorrect trace.

We will generally write properties in either violation or validation style but return false on an incorrect trace. However, as we separate monitoring from instrumentation a user can

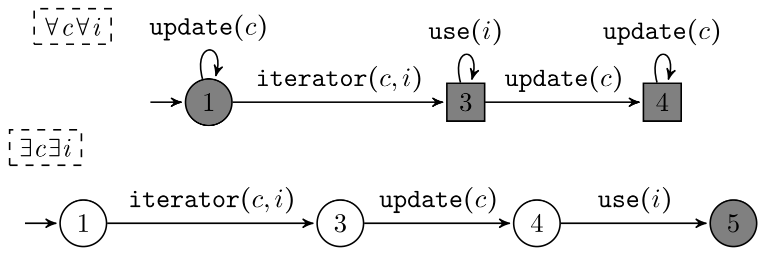


Figure 4.1: Two QEAs for the safe usage of iterators created from collections.

decide what action to take on each verdict. The distinction is important when the specification language is not closed under negation or, for example, in the case of TraceMatches where the formalism (regular expressions) are inelegant for describing one kind of behaviour (validation).

For a discussion of QEA negation see Sec. **??**.

4.1.3 Safety and Co-Safety

Here we discuss the relationship between styles of QEA and the kinds of temporal properties that are commonly used i.e. the notions of safety and co-safety. These concepts are usually phrased in terms of possible extensions of the trace we have seen so far. We revisit this topic in section 6.1 where we introduce new verdict categories.

We begin by introducing the notions of good and bad prefixes. Given a language L, a word w is a bad (good) prefix for L if for every u we have $w.u \notin L$ ($w.u \in L$). These concepts usually apply to *infinite* traces, but in this work we consider finite traces so we take the common approach [HR04] of assuming the status on the final state is stationary i.e. continued infinitely.

Safety. A property is a safety property if it can be violated by a finite trace i.e. every trace not in its language has a finite bad prefix. Conceptually these properties capture the idea that nothing bad ever happens. This is a very common kind of property and relate to a QEA with pure universal quantification and all accepting next states. The pure universal quantification is important, as we show in Sec. 6.1 that a property can only be violated by a finite trace if all quantifications are universal.

Co-Safety. A property is a co-safety property if it can be satisfied with a finite trace i.e. every trace in its language has a finite good prefix. These represent *reachability* properties. Co-safety properties typically relate to a QEA with pure existential quantification and a sequence of skip states with the last state being final. Again, the existential quantification is important.

Not all properties are safety or co-safety properties as they can be a mixture of the two. The general notion here is that we get 'safety-like' behaviour from universal quantification and next states, and 'co-safety-like' behaviour from existential quantification and skip states.

4.1.4 Special cases

We consider two special cases of behaviour involving multiple quantified variables.

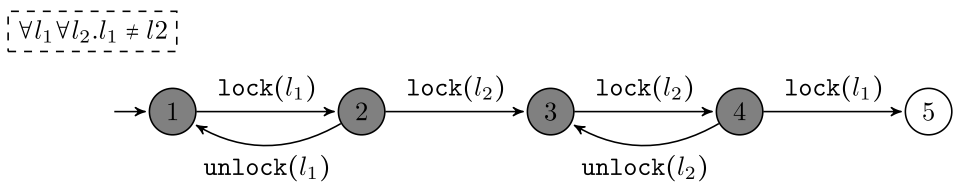


Figure 4.2: A QEA specifying that locks should be taken in a consistent order.

Connected behaviour

There is a class of properties that concerns so-called *connected* behaviour where the objects in the domains of two variables have a semantic relationship. A key example of this is where, in Object-Oriented programming, we have one object is created from another. For example, in the UnsafeIter example above. In this case each iterator will be connected to at most one collection. This connectedness generally goes against our notion of quantification i.e. $\forall x \forall y$ usually means every pair of values for x and y. We introduced the connectedness global guard to capture this relationship.

Symmetry and global guards

In contrast to connected behaviour we sometimes have symmetric behaviour where we talk about two instances of the same object. Here we will have two instances of the same event name i.e. `lock`(l_1) and `lock`(l_2). In this case we will have two instantiations of the QEA for every pair of values v_1 and v_2 i.e. $[l_1 \mapsto v_1, l_2 \mapsto v_2]$ and $[l_1 \mapsto v_2, l_2 \mapsto v_1]$. This may effect the way that the specification is written and will effect efficiency as we will be checking twice as many instantiations as we need to.

For example, consider the lock ordering property introduced in Appendix A.3.3 on page 347. This is a property giving a guarantee of deadlock freeness that states that locks are always taken in a consistent order, Fig. 4.2 gives the QEA. For each pair of locks each instantiation captures the different behaviours of either lock being taken first. Note that we exclude the case where the two locks are the same. Later (Sec. 6.3.2) we show that introducing a global guard that selects only half the of the behaviours can considerably increase efficiency but requires us to specify the behaviours for either lock being taken first.

4.2 Limitations

In this section we discuss the limitations of QEA as a specification language. We show that there are some properties that QEA is not an appropriate formalism for. This is mainly because the specification must rely too heavily on the guard and assignment language.

4.2.1 Using a stack

Here we consider a context-free property; the correct usage of a stack. This can generally be captured by stating that ever `pop` has a corresponding `push` with the same object i.e. if we `push`

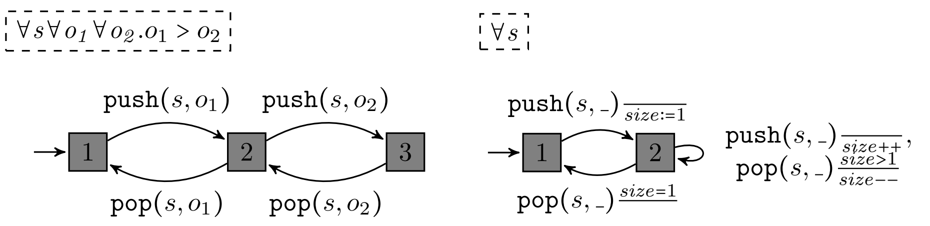
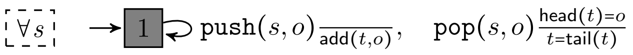


Figure 4.3: Correct usage of a (limited) stack.



$\forall s$
1
push(s, o) add(t, o),
pop(s, o) head(t)=o / t=tail(t)

Figure 4.4: Correct usage of a stack, using a stack.

A then B and then call `pop` twice we should get B then A. This also implies that we cannot `pop` from an empty stack.

We can describe the limited stack property (where each object can appear at most once) using QEA, this is captured by the QEAs in Fig 4.3. The QEA on the left says that given a stack s and *any* two objects o_1 and o_2 if o_1 is pushed onto s and then o_2 is pushed onto s then o_2 must be popped from s before o_1 is. This QEA does not allow an object to be pushed onto a stack twice, hence the limited stack assumption. Also, this QEA prevents us from calling `pop` on an empty stack as `pop` is not allowed from the initial state. However, our instrumentation would require us to construct this event *after* the `pop` call was made as a popped object is needed. The QEA on the right of Fig. 4.3 deals with this issue by checking the size of the stack before calling the `pop` function without considering the return value.

We have described this example as a limitation because in order to specify the complete property we must use a stack in assignments, as shown in Fig. 4.4. We could have ensured the correct nesting using counters, but not that the objects pushed and popped agreed. This demonstrates that in some cases the use of automata for the event language restricts what we can do without resorting to using arbitrary code in assignments. However, as discussed later, we could extend our approach by replacing event automata with event context-free grammars.

4.2.2 Sorting a list

Next, we consider a non-temporal property that is often used as a basic example in program verification; sorting a list. As this is a non-temporal property we just encode the condition in a function placed on a single transition in an EA i.e. whenever we see the given event we should check the condition. Therefore, a QEA for checking that the `sort` function sorts lists correctly is given in Fig. 4.5. This assumes a function is_sorted that can check if one list is a sorted version of another. QEA specify patterns of behaviour, so when the property is static they are not appropriate.

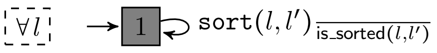


Figure 4.5: A QEA for the non-temporal property of list sorting.

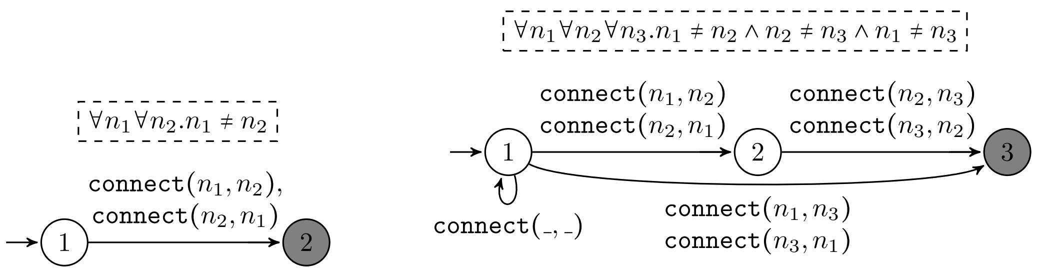
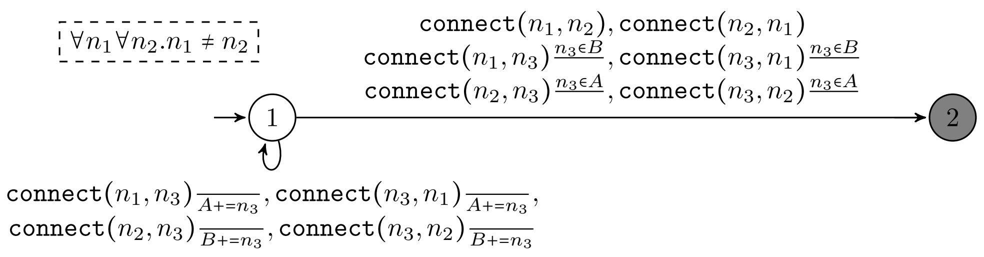


Figure 4.6: Two QEAs for 1-hop and 2-hop graph connectedness.



$\forall n_1 \forall n_2 . n_1 \neq n_2$
connect(n_1, n_2), connect(n_2, n_1)
connect(n_1, n_3) $\frac{n_3 \in B}{}$, connect(n_3, n_1) $\frac{n_3 \in B}{}$
connect(n_2, n_3) $\frac{n_3 \in A}{}$, connect(n_3, n_2) $\frac{n_3 \in A}{}$
connect(n_1, n_3) $\frac{}{A += n_3}$, connect(n_3, n_1) $\frac{}{A += n_3}$,
connect(n_2, n_3) $\frac{}{B += n_3}$, connect(n_3, n_2) $\frac{}{B += n_3}$

Figure 4.7: A QEA for full graph connectedness.

4.2.3 Graph connectedness

Finally, consider the graph-connectedness property. As a concrete example, imagine that we had a network of inter-communicating sensor nodes and we wanted to establish that eventually there exists a path between every pair of nodes (assuming that, once established, paths exist forever). If we wanted to capture the property that every pair of nodes was connected we would state that there is a `connect` event for every pair events, as seen in the QEA on the left of Fig. 4.6. We could then extend this to the property that every pair of nodes was connected by at most one other node by quantifying over three nodes, as seen in the QEA on the right of Fig. 4.6. However, QEA do not allow us to quantify over an arbitrary set of variables or have an arbitrary number of transitions. This is one possible extension further work could consider.

We can specify the property by resorting to guards and assignments again. To do this we can store the set of connected nodes per node and then transition to an accepting state if these two sets is intersecting. This QEA is given in Fig. 4.7 but it is not elegant.

4.3 The complexity of trace checking

We consider the theoretical upper bound on the time complexity of the trace checking process described in Def. 27 by placing bounds on the number of generated bindings and work needed to process each binding. We also discuss how the structure of the specification effects these bounds.

4.3.1 Building a complexity model

In this section we consider the complexity checking acceptance (Def. 27) given a QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ with $\mathcal{E}{=}\langle Q, \mathcal{A}, \delta, q_0, F \rangle$ and a trace τ. Deciding acceptance involves constructing the bindings and then checking acceptance for each binding.

Number of bindings.

The maximum set of bindings we consider is given by $\mathsf{relevant}(\tau, \Lambda)$ (Def. 29). We say maximum as we would stop checking if failure success was found, depending on quantification. Let us consider the size of this set. Let $n = |\Lambda|$ be the number of quantified variables. Let $m_x = |\mathsf{Dom}(\tau)(x)|$ be the number of values in the domain of x for each $x \in \mathsf{vars}(\Lambda)$ and let m be the maximum such value. We place an upper bound on the number of bindings generated:

$$|\mathsf{relevant}(\tau, \Lambda)| \quad \leq \quad \prod_{x \in \mathsf{vars}(\Lambda)} m_x \quad \leq \quad m^n$$

This means that the number of bindings generated is at most exponential in the number of quantified variables. It should be noted that we expect this to be small, between 1 and 3 and unlikely to be above 5 (see the examples in Appendix A). However, we expect that the number of values per variable, m, could become very large, in the order of millions. For illustration purposes imagine we have a QEA with $n = 2$ and a trace with $m = 10M$. The number of relevant bindings constructed would be bounded by $100M$.

We can also place bounds on m by relating it to the size of the trace τ. Let $k_{\mathbf{a}} = i$ be the arity of event $\mathbf{a} = \mathsf{e}(x_1, \ldots, x_i) \in \mathcal{A}$ and let k be the maximum such value. Note that we expect k to be small, again in the order of 1-5. The worst case is that every event introduces a new value and every event in the trace has the maximum number of parameters:

$$m \quad \leq \quad k|\tau| \quad \Rightarrow \quad |\mathsf{relevant}(\tau, \Lambda)| \quad \leq \quad (k|\tau|)^n$$

Note that this is a bound on m, not m^n as each of the n variables may be drawn from the same domain. Recall that we expect k and n to be small; n and k are often 1 in previous examples, which would give us an upper bound on m of $|\tau|$. The example in Appendix A with the largest k and n has $k = 2$ and $n = 3$, giving an upper bound on m of $6|\tau|^3$, but again this assumes that every event introduces new values.

Next we consider a more realistic example where values are reused. Let us assume that the domains of the quantified variables are disjoint i.e. each event contributes an average of $\frac{k}{n}$ values to the domain of x. Next, we introduce the concept of value reuse. Let r_x be the average number of times a value in the domain of $x \in \mathsf{vars}(\Lambda)$ is reused and let r be the minimum such value. Therefore, the maximum number of objects introduced per event is given by $\frac{k}{rn}$. Note that we would typically expect $r > k$ and therefore $\frac{k}{r} < 1$ if $n \geq 1$. Making these assumptions we get:

$$m \quad \leq \quad \frac{k|\tau|}{rn} \quad \leq \quad \frac{|\tau|}{n} \quad \Rightarrow \quad |\mathsf{relevant}(\tau, \Lambda)| \quad \leq \quad \left(\frac{k|\tau|}{rn}\right)^n \quad \leq \quad \left(\frac{|\tau|}{n}\right)^n$$

This captures the intuition that if we have more quantified variables then a trace of the same length will yield fewer bindings. Here we assumed that $\frac{k}{r}$ was less than one and therefore could be ignored. However, it might be significant with respect to $|\tau|$, and it is often the case that $r >> k$. In fact, this relationship can quickly dominate. For example if $n = 2$, $k = 2$, $r = 10k$ and $|\tau| = 1M$ then $|\mathsf{relevant}(\tau, \Lambda)| \leq \left(\frac{2\times 1M}{2\times 10k}\right)^2 = 1k$.

Work per binding.

Now we have thoroughly discussed the number of bindings generated we can briefly go through the work required to process each binding. For a binding θ we are asking

$$\tau \downarrow_{\mathcal{E}(\theta)} \in \mathcal{L}(\mathcal{E}(\theta))$$

This will involve constructing $\mathcal{E}(\theta)$, which will be bounded by $|\delta|$ as its main work is an iteration of this set, constructing $\tau \downarrow_{\mathcal{E}(\theta)}$ and checking if $\tau \downarrow_{\mathcal{E}(\theta))} \in \mathcal{L}(\mathcal{E}(\theta))$. If $\mathcal{Q}$ is non-deterministic this last part will involve keeping track of a set of configurations. The second two points can be executed at the same time and, assuming that matching and guard/assignment evaluation is constant time, will be bounded by $|\delta| \times |\tau|$, which gives an upper bound on the work required per binding.

Note. *The parameterisation of QEA with a guard and assignment language means that we must make an assumption about the complexity of evaluating these guards and assignments. Here we assume this is constant. This is clearly not generally true as we could encode the entire monitoring algorithm as a single guard in a QEA and wrap a trace in an event. More concretely, whilst most guards and assignments used in this work consider constant operations using integers, some make use of data structures such as sets whose implementation would not be constant in time. The additional complexity would be included here.*

Summary.

Putting this together we must first construct $\mathsf{Dom}(\tau)$, which will be $\Theta(|\tau| \times |\mathcal{A}|)$. The overall complexity is therefore in the order of

$$\Theta(|\tau| \times |\mathcal{A}|) + O\left(\left(\frac{k|\tau|}{rn}\right)^n |\delta| \times |\tau|\right)$$

For big $|\tau|$ this becomes $O(\frac{|\tau|^{n+1}}{n})$ unless r becomes significant i.e. if we reuse values frequently. This shows that the *structure* of the trace is important in deciding efficiency. A similar conclusion is reached by Lee et al. [LCR11], although they do not discuss reuse rate.

4.3.2 The importance of choosing the right specification

It is clear that the number of quantified variables of a QEA significantly effects efficiency. For example, Fig. 4.8 gives two equivalent QEA that, due to their differing number of quantified variables, have very different complexity bounds. The QEA on the left states that for every message there exists a start and end time, with the end time being less than 100 units of time after the start time, such that the message is sent at the start time and acknowledged at the end

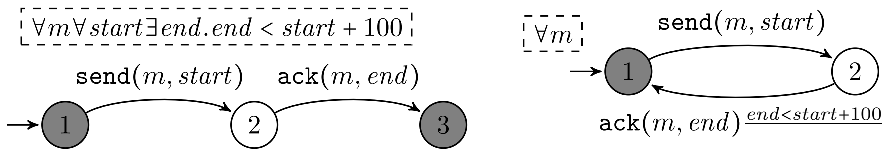


Figure 4.8: Two equivalent QEA that have very different complexity bounds.

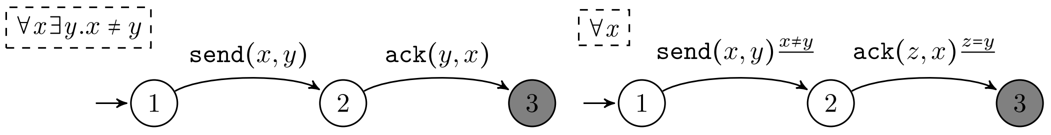


Figure 4.9: A simple example of quantifier stripping

time. The QEA on the right states that every message that is sent at time *start* is acknowledged within 100 units of time, and can then be sent again. For large traces, the first QEA has a complexity bound of $O(|\tau|^4)$, whereas the second has a bound of $O(|\tau|^2)$.

As the number of quantifiers is key to complexity we consider a method that can be used to remove quantifiers. The QEA on the left of Fig. 4.9 captures the property that every node has another node that it communicates with. We can safely remove the existential quantification and move the global guard to a transition guard. Another example is given in Appendix A.1.4.

4.4 On the expressiveness of QEA

In this section we discuss the expressiveness of the QEA language.

4.4.1 Choice of guard and assignment language

By parameterising the definition of QEA with a guard and assignment language we have made it trivially Turing complete as we could select some Turing complete language and encode anything we wanted into the guards and assignments. Note that this would violate our assumption in the previous section that the evaluation of guards and assignments is constant time.

It is likely that we do not need such an expressive guard and assignment language and that choosing a less expressive language would give better complexity guarantees whilst potentially decreasing expressiveness. We briefly consider the effects of choosing particular guard and assignment languages.

- **Equality guards and storing assignments.** If we only allow guards to check for *equality* between free variables and stored variables and assignments to store the values of free variables then EA become register automata [NSV04].
- **Counting.** An interesting small guard and assignment language is one that only allows guards to compare integers and assignments to store, increment and decrement integers. This allows us to model some context-free properties as we seen in the CloseFiles example in Appendix A on page 342.

- **Embedded DSL.** If we embed the guard and assignment language in a programming language, for example QEA is written as an embedded DSL, we have access to the full functionality of the programming language.

4.4.2 Separation

The formulation of QEA only requires that EA provide a predicate function on traces. Therefore, there is a real separation between EA and QEA as the way we define this predicate function does not effect how QEA operates. This means that we could also consider quantified regular expressions or grammars and the sections discussing QEA would not change substantially. JavaMOP uses this approach i.e. allows the use of different propositional languages. A formalisation of this framework remains future work.

We note that it would be relatively straightforward to define event regular expressions, i.e. regular expressions over events, and also to include guards and assignments by extending the definition of an event to include these. This would not give us an added expressiveness but would give us an alternative, sometimes more concise method of presenting the same specifications. The natural extension for context-free grammars where we allow events, guards and assignment triples as terminal symbols is straightforward. However, we might consider allowing variables to label non-terminal symbols, similar to the rules of RuleR.

Finally, we might question why we would want to introduce more expressive 'event languages' when EA are trivially Turing-complete. The answer is that we introduce a specification language to allow us to write elegant and analysable specifications but by coding a specification mainly in the assignments of an EA we depart from this.

4.4.3 The effects of quantification

As mentioned at the beginning of this chapter, we can construct a ground SEA from an SEA given the domains of variables. This is true of QEA also, given the (finite) domains of all quantified and free variables we can construct a ground EA that accepts the same traces as the QEA (with an enormous state explosion). Therefore, it might be argued that quantification adds no real expressive power. However, this considers QEA given the domains of all of its variables, perhaps within the context of a trace. A QEA therefore represents an infinite family of sets of ground EA. For example, consider the following restricted bracketing problem. Given a string of different coloured brackets, where brackets of the same colour cannot nest, do all brackets match? This can be captured in a QEA (i.e. NestedCommands on page 62) but not in a ground EA without specifying the number of different coloured brackets beforehand.

4.5 Comparing with other specification languages

We now compare QEA with other specification languages, focusing mainly on those used in runtime verification tools. We limit our discussion to expressiveness (and in some cases elegance) and leave the issue of efficiency to later. Some of these were discussed briefly in Sec. 2.3.5.

4.5.1 A note on embedded DSLs

Some specification languages are embedded in another language, or produce code that is, and can therefore make use of this language to extend their specifications. The question is then whether we consider this language as part of the specification language.

For example, TRACEMATCHES is an extension of `AspectJ`. The language definition has no support for free variables, but we can define variables within the aspect file and refer to them within the code to execute on a match and can therefore simulate some notion of free variables. Furthermore, JAVAMOP allows us to execute arbitrary code whenever an event is created. Some languages, such as TRACECONTRACT are designed as internal DSLs, in this case the design of the specification language uses the original language to capture certain features. We consider the specification language, and not what can be achieved through additional programming.

4.5.2 Dealing with parameters using slicing

QEA is closely related to techniques that take a slicing view. However, all of the existing techniques make use of universal quantification only and do not capture free-variables in their specification language

Typestate

Introduced in 1986, typestate [SY86] is a refinement of the concept of type. An objects type determines what operations it can participate in. Typestate introduces the notion of an object being in different states (contexts) and therefore having a restricted set of operations it can participate in. Original examples focus on the issue of initialization where an object is either in an initialized state or not and can only be used when initialised.

Typestate checking can be presented with a slicing view. To check that each object obeys its typestate specification we only consider events relevant to that single object. This is straightforward as typestates are about single types, not collections of types.

Larva

Larva [CPS09] allows one to specify real-time propositional properties using the Dynamic Automata with Events and Timers (DATE) specification formalism. There is a focus on real-time properties, hence the inclusion of timers. We place Larva here, rather than in the propositional section, as it allows propositional monitors to be replicated for each object of a given type. This is the idea behind slicing viewed from the opposite direction. Furthermore, this replication mechanism can be nested, introducing a connectedness semantics for multiple quantifications. This can be captured in QEA using the connected global guard.

LarvaStat [CGP10] is an extension to Larva that allows the collection of trace statistics, for example the frequency an event occurs between two other events.

Tracematches

TRACEMATCHES [AAC+05, BHL+07] is an extension of the `AspectJ` language, implemented in the `abc` compiler [TMs]. Properties are written as regular expressions over pointcuts with variables, which are implicitly universally quantified. Like in AspectJ, we can attach advice to tracematches to be executed on a match.

Again, the semantics are specified using slicing i.e. propositional trace is generated for each binding of quantified variables. Here, the regular expression matches if any *suffix* of the trace matches the expression, we say TRACEMATCHES is *suffix-matching*. TRACEMATCHES therefore catches *validations* and specifications describe incorrect behaviour. This is the reason for suffix-matching, it is often easier to describe failure in terms of the few error causing events. For multithreaded program we can either consider the trace of each single thread separately, or the interleaved trace of all threads.

In QEA the suffix-matching semantics can be simulated using non-determinism.

JavaMOP

JAVAMOP [MJG⁺11, CR09] is a tool and an associated specification language from the *Monitoring-Oriented Programming* (MOP) approach, an attempt to formalise the process of monitoring programs as a programming methodology. JAVAMOP is packaged as a stand-alone tool that compiles specifications into AspectJ aspects. Aspects can then be directly weaved into the monitored system. JAVAMOP allows a user to embed code (actions) into the specification, making feedback part of the system. JAVAMOP is therefore mainly an inline tool, but could be used outline given additional effort. The RV System [MR10] is a commercial extension of JAVAMOP.

JAVAMOP separates the monitoring and specification activities using a framework that utilizes multiple *logic plugins*. All of these logics are propositional and parametric monitoring is achieved through slicing. Logics currently implemented include finite state machines, extended regular expressions, context free grammars, linear temporal logic (future and past) and string rewriting systems. Quantified variables are declared upfront, events are defined using AspectJ pointcuts referring to these variables, and the property is then declared using these events. Like TRACEMATCHES, JAVAMOP can also be run in suffix-matching and per-thread modes.

JAVAMOP is often shown to be the most efficient existing parametric runtime verification tool. This is largely due to extensive engineering effort and complex indexing mechanisms. We mention these later where relevant.

JAVAMOP assumes that for every parametric event $\mathtt{e}(\overline{x})$ we can construct a pair $\langle \mathtt{e}, \theta \rangle$ i.e. every event has an implicit list of variable parameters. This means that we cannot have an event name occurring in a specification more than once i.e. we cannot have both $\mathtt{lock}(l_1)$ and $\mathtt{lock}(l_2)$. Therefore, if we consider only the regular propositional plugins for JAVAMOP then it is less expressive than TRACEMATCHES. JAVAMOP has non-regular propositional plugins for context-free grammars and string rewriting. Both can be simulated using guards and assignments, although elegance would significantly deteriorate in some places. As discussed earlier, we could also use non-regular event languages. JAVAMOP also has two modes that allow us to match on any partial binding, or any partial but maximal binding. These modes can be simulated by a conjunction of QEAs i.e. producing a QEA for each subset of quantified variables, removing transitions with removed variables.

Difference with other parametric approaches

Trace slicing is only one way in which data parameters are dealt with in runtime monitoring. Other approaches typically only consider the whole specification on each event (unlike the local view of trace slicing), we call these approaches *global*.

The difference between global and slicing approaches can be seen in their treatment of the notion of *next*. For example, if we consider the statement 'For all x if $\texttt{f}(x)$ then on the next step $\texttt{g}(x)$' a slicing interpretation would allow the trace $\texttt{f}(1).\texttt{f}(2).\texttt{g}(1).\texttt{g}(2)$, whereas a global approach would not.

Summary

Both Larva and TraceMatches have a regular propositional language, therefore they are equivalent in expressive power to QEA without free variables. Also note that whilst TraceMatches and JavaMOP do not define free variables in their specification languages they can make use of additional programming to capture some free behaviour.

In conclusion, QEA is more expressive than Larva, JavaMOP and TraceMatches, although some of their modes may lead to more elegant specifications.

4.5.3 Rule-based approaches

Here we consider approaches that specify properties using rewrite rules.

Eagle and RuleR

EAGLE [BGHS04] [BRH08] is a rule-based framework for defining and implementing finite trace monitoring logics. The EAGLE logic is a restricted first order, fixed-point, linear-time temporal logic with chop over finite traces. The monitoring algorithm operates on a state-by-state basis, as opposed to storing the execution trace, avoiding the need for backtracking. Hawk [dH05] is a monitoring framework for EAGLE.

EAGLE is a very expressive and powerful logic, however it is not necessarily that efficient. The RuleR [BGHS04, BRH08, BHRG09] tool has been been developed as a practically useful and more efficiently executable subset of EAGLE. RuleR specifications are written using parameterised conditional rules. The general semantics can be summarised as updating a set of *states*, each containing a set of *rule activations*, which themselves consist of a rule name and a binding for that rule's parameters. Each state is updated using the next event by attempting to fire each of its rule activations. A rule definition takes the form

$$\text{modifier name}(x_1 : \tau_1, x_2 : \tau_2 ...) : \text{antecedent} \rightarrow \text{consequent};$$

The modifier determines the rule persistence i.e. whether it stays in the state if it has not fired. The variables are simply typed and can be used in the antecedent and consequent. A rule is fired if its antecedent evaluates to true and firing a rule consists of applying its consequent, which might create new states, or add or remove rule activations from the current state. Antecedents can contain free variables to be matched against the rest of the state (i.e. we ask if another rule activation exists with a suitable binding) and computations such as comparing and adding integers. Notably, rule activations can be given as parameters to other rules, allowing us to save some state of execution.

The multiple states represent a breadth-first non-deterministic exploration of all possible states the rule system could be in given the trace. A specification will give acceptance or failure conditions. RuleR uses a five-valued verdict domain (true, false, still true, still false and unknown) and acceptance conditions might, for example, forbid a particular rule activation leading to still false if that rule activation is in all states.

It should be noted that RuleR struggles with efficiency in some cases due to its expressiveness. The ability of rule antecedents to refer to other rule activations using free variables means that a general indexing strategy is difficult to define.

Rule systems that reflect universally quantified state machines can be straightforwardly encoded in QEA and vice versa. However, there are properties in both languages that are difficult to specify in others. Rule systems that make use of global facts about a system require a QEA to store these facts in assignments. QEA that rely heavily on the notion of maximality (i.e. have many quantified variables bound at separate points) require rule systems that keep track of the current maximal binding, which can become complicated.

Due to its ability to parameterise rules with rule activations, RuleR is Turing-complete.

Logscope

LogScope [BGHS10] is a restriction of RuleR to automata i.e. a rule's antecedent is an event. It also introduces a pattern language. LogScope was developed in response to a need at NASA's Jet Propulsion Laboratory, so many of the decisions were based on usability considerations. LogScope [BGHS10]'s parametrised automata can be simulated in EA using non-determinism.

Tracecontract

TraceContract [BH11b] is an internal DSL (an API in `Scala`) for writing monitors. It offers an experimental combination of parameterized state machines with anonymous states, referred to as state logic, future time linear temporal logic, and rule-based programming; as well as free combinations of these forms. Logfire [Hav13b] is a recent development that is based on the Rete algorithm [For82] borrowed from the field of Artificial Intelligence. Logfire attempts to achieve an efficient checking mechanism for the rule-based approach. As these are internal DSLs they are technically Turing-complete.

Summary

The main difference between these techniques and QEA is the matter of elegance. There are properties that are straightforward to capture in these rule-based systems which are awkward to express with QEA, and vice versa.

4.5.4 Temporal logic

We consider extensions of LTL within the context of runtime verification and first-order LTL in general.

Extending LTL

There have been efforts that extend LTL, taking a finite path semantics.

J-Lo [Bod05, SB06] uses a parameterised LTL with `AspectJ` pointcuts as propositions. These pointcuts may define and access variables, which are implicitly universally quantified. Checking is performed by rewriting the formula for each new event by splitting it into a now and next part. The domains of quantified variables are captured by the rewritten formula and are therefore collected over the trace.

Stolz introduced temporal assertions with parametrized propositions [Sto10], which adds parametric events to next-free LTL. The domains of quantified variables are taken from the current state and quantifiers must be associated with a constructing event, i.e. one that introduces the binding. Therefore, unlike J-LO, quantifiers can be nested. Parametrized LTL formulas are checked at runtime by translation to a form of alternating automaton that constructs and passes bindings through the transition function.

J-LO's quantifications are global (like QEA), whereas Volker's work considers nested quantifications where the domain of quantification is taken at the current state. As both approaches include a global notion of next the translation into QEA would not be completely straightforward, but would be possible.

First Order LTL

There are different formulations of first order LTL and in runtime verification we would usually consider a restricted form where, in each frame, only one predicate is true (representing the event occurring on that step) and give a finite-path semantics. We do not yet fully understand the relationship between first order LTL and QEA, but in the following we discuss the relationship.

We can automatically translate certain forms of FO-LTL formulas into QEA. Any FO-LTL $Q.F$ where Q is a list of quantifiers and F is a quantifier-free next-free[1] formula can be translated into QEA by applying the standard automata translation to F. The next-free requirement is due to the local-only interpretation of next in QEA i.e. we cannot assert anything about the next event in the trace, only in the trace slice. We note that not every FO-LTL formula can be put into prenex normal form.

However, the next-free restriction is not universal. For example, the FO-LTL formula $\forall x : \Box(\texttt{f}(x) \rightarrow \bigcirc\texttt{g}(x) \land \texttt{g}(x) \rightarrow \bigcirc\texttt{f}(x))$ can be represented as a QEA by observing that any trace satisfying the formula is an alternation of `f` and `g`. We can also translate QEA using free-variables into FO-LTL, although we do not have a general procedure. For example the QEA that asserts that c must always be incremented by one can be captured by the FO-LTL formula $\forall c : \texttt{count}(c) \rightarrow \bigcirc\texttt{count}(c+1)$. However some QEA are less straightforward to phrase as an FO-LTL formula due to the interaction between quantified and free variables.

[1]We state this requirement informally. A formula is next-free if it never places any requirements on what must come next e.g. does not use the next operator and only uses Until with negation on the left.

4.5.5 Other approaches

We consider approaches that do not belong to the previous categorisations.

UML state charts

The state chart part of the Unified Modeling Language (UML) [Obj09] is a form of extended finite state machine that includes parametric events, state variables, transition guards and actions. They, therefore, closely resemble our Event Automata. There are a number of differences, for example nested states, orthogonal regions, internal transitions, but due to the expressiveness of UML statecharts and EA it would be straightforward to directly move between the two. As QEA are strictly more expressive than EA (before fixing domains), QEA are more expressive than UML state charts.

Query languages

Program Query Language (PQL) [MLL05] captures properties about sequences of events associated with a set of related objects. Queries are written in a context free language and can refer to field accesses as well as method calls. Queries can be statically evaluated to reduce the number of dynamic checks. Queries are checked at runtime via translation to a state machine.

Program Trace Query Language (PTQL) [GOA05] poses SQL-like relational queries over program traces, which are viewed as sets of timestamped records where events are modeled as relations. Event parameters are captured as fields and can be used arbitrarily in queries.

This approach takes a different view from QEA and has shown to be less efficient for monitoring at runtime.

Using Register Automata

A recent technique [GDPT13] makes use of Register Automata [NSV04] for parametric runtime verification. Specifications are given in the Temporal Object Property Language (TOPL) [GPD11], which allows the user to describe an automata view of their property. The underlying idea of this approach is that values can be saved to, and compared with, registers. Quantification is achieved by non-determinism i.e. by staying in the same state (without creating the new binding) and transitioning to the new state (creating the new binding). As acceptance is determined by not reaching an error state quantification is implicitly universal.

Register Automata can be captured by EA so this approach is strictly less expressive than QEA. To simulate TOPL we would replicate the idea of capturing quantification using non-determinism.

Specification mining

Let us compare QEA to the languages targeted by the specification mining techniques we identified in Chapter 2. In terms of quantifications, the specifications mined by JMiner [LCR11] are closed to QEA as they employ a (connected) slicing semantics. The binary rules of Tark [LCH$^+$09, LRRV12] represent very simple (two-state) automata that can easily be represented by QEA. Neither approach considers existential quantification. TzuYu [XSL+13] extracts stateful typestates i.e. singularly quantified state machines with free variables. All of the active techniques extract some form of extended finite state machine that could be captured by EA. However, in this work we will be focusing on mining QEA without free variables.

4.6 Summary

In this chapter we have discussed different styles that can be used when specifying QEA and have discussed the complexity of the trace checking process and the expressiveness of QEA in relation to alternative specification languages.

4.6.1 The missing pieces

Now that we have introduced QEA and explored their properties we consider some final linguistic concerns required to make QEA a well-rounded specification language. There are some questions we have not asked including:

- Is the problem of deciding whether the language of a QEA is empty decidable? If so, how does one compute this?
- Are QEA closed under negation? If so, how do you compute the negation?
- Are QEA closed under union or intersection? If so, how do you compute these?

In each case we can consider these questions for QEA generally and for syntactic restrictions of QEA. For example, considering the class of QEA without free variables would simplify the exploration. Where free variables are included the answers to these questions will be dependent on the chosen guard/assignment language.

We do not attempt to address these questions here. However, we note that that our method for defining the domain of quantification in terms of the trace will complicate the issue. We would only be able to take the intersection and union of QEA that have the same domains for quantified variables. Having the same alphabet and quantification list (up to variable renaming) would be sufficient for this, however not necessary.

Whilst these issues are of some interest they are not directly concerned with the runtime verification or specification mining applications explored in this work, which are only concerned with the word problem i.e. is a given trace in the language of a given QEA.

Part II

Runtime Monitoring

Chapter 5

Incremental Monitoring

In the previous chapter we introduced QEA with a big-step semantics i.e. the acceptance of a trace was based on analysing the whole trace. This approach is unsuitable for monitoring at *runtime* as events are received one at a time, therefore this chapter introduces a small-step semantics that processes traces incrementally, giving the status of the monitored QEA after processing each event. This small-step semantics will be equivalent to the big-step semantics for each prefix of the trace.

Outline. We will begin in Section 5.1 by exploring what makes the incremental monitoring process difficult and outline the work that needs to be done. Based on some constructions related to bindings (Sec 5.2) we then build a small-step semantics (Sec. 5.3) and discuss its complexity (Sec. 5.4). Next we prove that this new semantics is equivalent with that given previously (Sec. 5.5). Finally, we present an algorithm for carrying out monitoring and show that this implements the small-step semantics (Sec. 5.6).

5.1 The problem of monitoring

We use the UnsafeIter example, given in Fig. 5.1, to demonstrate the challenges involved in incremental monitoring and illustrate how we will solve them.

Imagine we begin monitoring and receive the event $\texttt{create}(\mathrm{A})$. We need to record the information this contains; that c maps to A and the trace slice for this binding. We then receive the event $\texttt{create}(\mathrm{B})$. We have two values for c and two trace slices, so we construct the map

$$\begin{array}{lcl} [c \mapsto \mathrm{A}] & \mapsto & \texttt{create}(\mathrm{A}) \\ [c \mapsto \mathrm{B}] & \mapsto & \texttt{create}(\mathrm{B}) \end{array}$$

However, we note that storing these trace slices directly will soon become inefficient. Instead, we can replace the trace slice with the configurations reached in the EA. So our map becomes

$$\begin{array}{lcl} [c \mapsto \mathrm{A}] & \mapsto & \{\langle 2, [\ ]\rangle\} \\ [c \mapsto \mathrm{B}] & \mapsto & \{\langle 2, [\ ]\rangle\} \end{array}$$

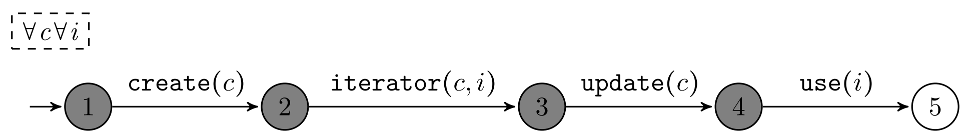


Figure 5.1: A version of the UnsafeIter property from Appendix A.3.2.

This gives the domain of c and the status of the associated (partially) instantiated EA.

At this point the domain of i is empty and the verdict is true as universal quantification over an empty domain is always true. We now receive the event $\texttt{iterator}(A, 1)$, which introduces an iterator object, leading to the new binding $[c \mapsto \text{A}, i \mapsto 1]$.

When adding this new entry we have to associate it with a set of configurations. From our knowledge of the big-step semantics we can see that the projected trace so far for this binding is $\texttt{create}(A).\texttt{iterator}(A, 1)$ and therefore the configuration should be $\langle 3, [\ ]\rangle$. So the question is how do we get to this based only on the information available i.e. our map from bindings to configurations. There are three bindings in the domain of our map in the following (partial) order:

$$\begin{tikzcd}
{[c \mapsto \text{A}]} \arrow[dr] & & {[c \mapsto \text{B}]} \arrow[dl] \\
& {[\ ]} &
\end{tikzcd}$$

Two bindings are consistent with the binding we want to add, $[c \mapsto \text{A}, i \mapsto 1]$, but the biggest of these, $[c \mapsto \text{A}]$, stores the most information about relevant events. In fact, if $[c \mapsto \text{A}, i \mapsto 1]$, is not in our domain then the biggest consistent binding stores *all* of the information about the relevant events seen so far. Therefore, when adding $[c \mapsto \text{A}, i \mapsto 1]$ we look for transitions out of the configuration associated with $[c \mapsto \text{A}]$. In this case $2 \xrightarrow{\texttt{iterator}(i,c)} 3$ matches. We will call the biggest consistent binding the *maximal* binding. For completeness we must also add the binding $[c \mapsto \text{B}, i \mapsto 1]$ using $[\ ]$ as it is the maximal binding.

If we then receive $\texttt{iterator}(\text{B},2)$ we would carry out the same process, leaving us with the following map:

$$\left[\begin{array}{lcl}
[c \mapsto \text{A}] & \mapsto & \{\langle 2, [\ ]\rangle\} \\
[c \mapsto \text{B}] & \mapsto & \{\langle 2, [\ ]\rangle\} \\
[c \mapsto \text{A}, i \mapsto 1] & \mapsto & \{\langle 3, [\ ]\rangle\} \\
[c \mapsto \text{B}, i \mapsto 1] & \mapsto & \{\langle 1, [\ ]\rangle\} \\
[c \mapsto \text{A}, i \mapsto 2] & \mapsto & \{\langle 1, [\ ]\rangle\} \\
[c \mapsto \text{B}, i \mapsto 2] & \mapsto & \{\langle 3, [\ ]\rangle\}
\end{array}\right]$$

If we then receive $\texttt{update}(A)$ we generate the binding $[c \mapsto \text{A}]$. This binding already exists in our map, so we do not create it. The binding is also *relevant* to $[c \mapsto \text{A}, i \mapsto 1]$ and $[c \mapsto \text{A}, i \mapsto 2]$ as it is a submap of both bindings. This means that the event will belong to the trace slice of each binding. In the case of $[c \mapsto \text{A}, i \mapsto 1]$ this updates the configuration to state 4.

Finally, if we observe $\texttt{use}(1)$ we will create the binding $[i \mapsto 1]$. This is a new binding, so we can add it to our map using the maximal binding $[\ ]$, so it remains in the initial trace. This binding is also relevant to $[c \mapsto \text{A}, i \mapsto 1]$ and $[c \mapsto \text{B}, i \mapsto 1]$. In the case of $[c \mapsto \text{A}, i \mapsto 2]$ this means that the state is update to the non-final state 5. The trace slice for this binding is

$$\texttt{create}(\mathrm{A}).\texttt{iterator}(\mathrm{A},1).\texttt{update}(\mathrm{A}).\texttt{use}(1)$$

As one entry *for a total binding* is now in a non-accepting state the false verdict is returned for the first time, at the point where the error occurs.

Let us now summarise what we have learned. To avoid having to iterate over the trace twice we must build the derived domain and keep track of the status of each instantiated EA on the fly. We can capture the derived domain in the bindings that can be built from it, instead of storing this separately, and capture the status of each instantiated EA by the reachable configurations. Most importantly, when introducing a new binding we need to use the configurations associated with *maximal* existing binding as the start set of configurations as these contain all information pertaining to the trace slice for the new binding.

5.2 Bindings

In this section we discuss bindings and their properties relevant to this work. Some of these definitions and properties have been discussed elsewhere [RC12] in relation to runtime monitoring, and this other work gives a more thorough discussion. We introduced the concept of bindings in Sec. 3.1.2.

Firstly, to help us separate quantified and free variables in bindings we introduce the function

$$\mathsf{quantified}^{\mathcal{Q}}(\theta) = [(x \mapsto v) \in \theta \mid x \in \mathsf{vars}(\mathcal{Q}.\Lambda)]$$

Where $\mathcal{Q}$ is omitted if clear from context.

5.2.1 Least upper bounds

Two bindings are *compatible* iff they define the same value for every variable they share. A set of bindings is (pairwise) consistent if each pair is compatible:

$$\begin{array}{lcl}\mathsf{compatible}(\theta_1,\theta_2) & = & \forall x \in \mathsf{dom}(\theta_1) \cap \mathsf{dom}(\theta_2) : \theta_1(x) = \theta_2(x) \\ \mathsf{consistent}(\Theta) & = & \forall \theta_1,\theta_2 \in \Theta : \mathsf{compatible}(\theta_1,\theta_2)\end{array}$$

The least upper bound (lub) of two compatible bindings θ_1 and θ_2, written $\theta_1 \sqcup \theta_2$, is the smallest binding that contains θ_1 and θ_2 i.e. satisfying

$$\mathsf{dom}(\theta_1 \sqcup \theta_2) = \mathsf{dom}(\theta_1) \cup \mathsf{dom}(\theta_2) \qquad \text{and} \qquad (\theta_1 \sqcup \theta_2)(x) = \theta_1(x) \text{ or } \theta_2(x).$$

For a set of bindings Θ we define upper bounds, least upper bounds and maximums:

- Binding θ' is an *upper bound* of Θ iff $\forall \theta \in \Theta : \theta \sqsubseteq \theta'$. Note that, by definition of $\sqsubseteq$, Θ only has upper bounds if it is consistent.
- Binding θ' is a *least upper bound (lub)* of Θ iff it is an upper bound and $\theta' \sqsubseteq \theta''$ for any upper bound θ'' of Θ. Again, Θ only has lubs if it is consistent, furthermore it has at

most one lub $\bigsqcup\Theta$ as if there existed another lub θ_2 we would necessarily have $\bigsqcup\Theta \sqsubseteq \theta_2$ and $\theta_2 \sqsubseteq \bigsqcup\Theta$ and both lubs would be equal. For consistent set of bindings $\Theta = \{\theta_1, \ldots \theta_n\}$ it is the case that the lub of Θ is the smallest binding that contains all bindings in Θ i.e.,

$$\bigsqcup\Theta = \theta_1 \sqcup \ldots \sqcup \theta_n$$

- Binding θ' is a *maximum* of Θ iff it is in Θ and is a lub of Θ. As lubs are unique, maximums are unique and again only exist if Θ is consistent. Let us write the maximum of Θ as $\mathsf{max}\Theta$.

A set of bindings Θ is lub-closed iff

$$\forall \Theta' \subseteq \Theta : \mathsf{consistent}(\Theta') \Rightarrow \bigsqcup\Theta' \in \Theta$$

We write $\mathtt{close}_{\sqcup}(\Theta)$ for the lub-closure[1] of Θ i.e., the smallest lub-closed superset of Θ. Note that this must exist as the set of all bindings is lub-closed. Given a set of bindings Θ and a binding θ let $\mathsf{below}(\Theta, \theta)$ be the set of bindings in Θ 'below' θ i.e.,

$$\mathsf{below}(\Theta, \theta) = \{\theta' \in \Theta \mid \theta' \sqsubseteq \theta\}$$

Note that θ is not necessarily in Θ.

We now define an important property about lub-closed sets of bindings:

Lemma 1. *If Θ is lub-closed then* $\mathsf{below}(\Theta, \theta)$ *has a maximum element.*

Proof. As $\mathsf{below}(\Theta, \theta) \sqsubseteq \Theta$ and Θ is lub-closed we know that $\bigsqcup\mathsf{below}(\Theta, \theta) \in \Theta$. If $\theta \in \Theta$ then $\theta = \bigsqcup\mathsf{below}(\Theta, \theta)$ as θ is larger than all bindings in $\mathsf{below}(\Theta, \theta)$ by definition, and therefore $\theta = \mathsf{max}\ \mathsf{below}(\Theta, \theta)$. If $\theta \notin \Theta$ then $\bigsqcup\mathsf{below}(\Theta, \theta) \sqsubseteq \theta$ and therefore, by definition of $\mathsf{below}(\Theta, \theta)$, we know that $\bigsqcup\mathsf{below}(\Theta, \theta) \in \mathsf{below}(\Theta, \theta)$, and $\bigsqcup\mathsf{below}(\Theta, \theta) = \mathsf{max}\ \mathsf{below}(\Theta, \theta)$. ☐

We saw earlier that when adding a new binding we want to extend the most informative (maximal) binding. Therefore, when adding a binding θ we will be taking the lub-closed set of existing bindings Θ and want to find the maximum element of $\mathsf{below}(\Theta, \theta)$ i.e.,

$$\mathsf{max}\ \mathsf{below}(\Theta, \theta)$$

From the above proof we pick out another important property which we will also use later.

Lemma 2. *If $\theta \in \Theta$ and Θ is lub-closed then* $\mathsf{max}\ \mathsf{below}(\Theta, \theta) = \theta$.

The proof of this property is given in the proof above. To summarise, for all sets of bindings Θ and bindings θ:

1. $\mathsf{max}\ \mathsf{below}(\mathtt{close}_{\sqcup}(\Theta), \theta)$ always exists
2. $\theta \in \Theta \Rightarrow \theta = \mathsf{max}\ \mathsf{below}(\mathtt{close}_{\sqcup}(\Theta), \theta)$

[1]Note that we do not use the ambiguous syntax employed in previous work [RC12] to write $\sqcup\Theta$ for the lub-closure of Θ as well as the lub of Θ.

5.2.2 Dealing with partial bindings

As we only have partial information about the trace we will be creating *partial* bindings i.e., bindings that do not capture all quantified variables. To deal with partial bindings we introduce the concept of event relevance, which defines when a ground event is relevant to a binding within the context of a QEA.

Definition 33 (Event Relevance)**.** *For a QEA with a quantification list* Λ *and EA with alphabet* $\mathcal{A}$*, a ground event* $\mathbf{a}$ *is* relevant *to a binding* θ *if it matches with an event in* $\mathcal{A}$ *without capturing new quantified variables:*

$$\mathsf{relevant}(\mathbf{a},\theta,\mathcal{A}) \textit{ iff } \exists \mathbf{b} \in \mathcal{A}(\theta) : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b}),\Lambda) = [\ ]$$

The first part of the intuition behind this definition is straightforward; an event is relevant to a binding if it it matches with an event in the alphabet. The second part is important as we are dealing with *partial* bindings; if the resulting (quantified) binding is not included in the original binding then it is relevant to a more informative binding i.e. one that binds more quantified variables.

Previously, we checked whether an event $\mathbf{a}$ was relevant to a binding θ by checking if $\mathbf{a} \in \mathsf{ground}(\mathcal{E}(\theta))$. When θ is total this is equivalent to event relevance.

Lemma 3. *For any EA* $\mathcal{E}$ *with alphabet* $\mathcal{A}$*, list of quantifications* Λ*, and event* $\mathbf{a}$

$$\forall \theta \in \mathit{Bind} : \mathtt{total}(\theta,\Lambda) \Rightarrow (\mathtt{relevant}(\mathbf{a},\theta,\mathcal{E}.\mathcal{A}) \Leftrightarrow \mathbf{a} \in \mathtt{ground}(\mathcal{E}(\theta)))$$

Proof. As θ is total, by definition there are no quantified variables in $\mathcal{A}(\theta)$ and therefore for all $\mathbf{b} \in \mathcal{A}(\theta)$ we have that $\mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\ ]$. This means that event relevance reduces to the following for total θ:

$$\mathtt{relevant}(\mathbf{a},\theta,\mathcal{A}) \text{ iff } \exists \mathbf{b} \in \mathcal{A}(\theta) : \mathtt{matches}(\mathbf{a},\mathbf{b})$$

Which is equivalent to the definition of $\mathtt{ground}(\mathcal{E}(\theta))$ in Def. 21. □

5.2.3 Extending bindings

The next concept we introduce is that of binding *extensions*, which construct the set of bindings that extend a given binding using a given event. It is important that we capture all information that can be extracted from an event, which requires a number of steps.

The extensions of binding θ based on the event $\mathbf{a}$ are the consistent combinations of the bindings we get when we match $\mathbf{a}$ with any binding in alphabet $\mathcal{A}$. To define this set we must first introduce $\mathsf{from}(\theta,\mathbf{a},\mathcal{A})$, the set of bindings that can be built from $\mathbf{a}$, and $\mathsf{all\_from}(\theta,\mathbf{a},\mathcal{A})$, the lub-closure of these bindings and their submaps.

Definition 34 (Extending a Binding)**.** *Let the bindings extending* θ *for an event* $\mathbf{a}$ *be defined as follows*

$$\mathsf{extensions}(\theta,\mathbf{a},\mathcal{A}) = \{\theta \dagger \theta' \mid \theta' \in \mathsf{all\_from}(\theta,\mathbf{a},\mathcal{A}) \wedge \theta' \neq [\ ]\}$$

where

$$\begin{array}{lcl}\mathsf{from}(\theta,\mathbf{a},\mathcal{A}) & = & \{\texttt{quantified}(\texttt{match}(\mathbf{a},\mathbf{b})) \mid \mathbf{b}\in\mathcal{A}(\theta)\wedge \mathsf{matches}(\mathbf{a},\mathbf{b})\}\\ \mathsf{all\_from}(\theta,\mathbf{a},\mathcal{A}) & = & \mathsf{close}_{\sqcup}\ \{\theta' \mid \exists\theta''\in\mathsf{from}(\theta,\mathbf{a}).\theta'\sqsubseteq\theta''\}\end{array}$$

The set $\mathcal{A}$ is omitted if clear from context.

Therefore, the set of binding extensions is given by adding to θ any (non-empty) binding that can be built from the incoming event (i.e. is in $\mathsf{all\_from}$).

Let us use the lock ordering property, discussed in Sec. 4.1.4 and defined in Appendix A.3.3 on page 347, to show why we must close the set of bindings we extract from the event. This specification has events in the alphabet of the EA that lead to multiple bindings of quantified variables that interact. Consider the event $\texttt{lock}(1)$. The big step semantics tells us that we must consider the total binding $[l_1\mapsto 1, l_2\mapsto 1]$. To create this we need to match $\texttt{lock}(1)$ with $\texttt{lock}(l_1)$ and $\texttt{lock}(l_2)$ to construct the bindings $[l_1\mapsto 1]$ and $[l_2\mapsto 1]$ and then combine these, hence taking the least upper bound closure. Appendix A.2.1 gives a further example motivating the need for this definition of binding extensions.

A note on the relation between event relevance and binding extensions

Our initial intuition might be that if an event $\mathbf{a}$ is relevant to a binding θ then there are no extensions, as we can see that

$$\mathsf{relevant}(\mathbf{a},\theta,\mathcal{A})\Rightarrow\mathsf{from}(\theta,\mathbf{a})\subseteq\{[\ ]\}$$

as $\mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\ ]$ in the definition of relevance. Recall that event relevance is based on the idea that θ is the *most* informative binding that is consistent with the event.

However, note that both definitions rely on the existence of an event in the alphabet and it is possible that there may be different events that make an event relevant and populate $\mathsf{from}(\theta,\mathbf{a})$. If they were the same event we would have a contradiction as the definition of relevance says that matching with $\mathbf{b}$ binds no new quantified variables, whereas $\mathsf{from}(\theta,\mathbf{a})$ selects *only* new quantified variables.

Event relevance and binding extensions will serve two different purposes; event relevance will be used to decide whether an event should be used to update a binding's configurations, and binding extensions will be used to construct new bindings.

5.3 A small step semantics for QEA

In this section we introduce an alternative semantics for QEA that updates the information it holds about the trace seen so far with a new event to produce a new verdict. As mentioned previously, this semantics will be equivalent to the big-step semantics for each trace prefix, we prove this later. Appendix A.2.2 gives a thorough worked example of this semantics.

5.3.1 Semantics with projections

Firstly, we show how bindings of quantified variables can be created on-the-fly and associated with their relevant projections. To do this we construct a *monitoring state* that associates bindings with projected traces.

$$\mathit{MonitoringState} = \mathit{Binding} \rightharpoonup \mathit{Trace}$$

To update a monitoring state we use the binding extensions introduced in the previous section. When adding a binding extension we must also associate it with a projection; whether the new event is associated with a binding extension is decided by event relevance.

Given a binding θ associated with trace σ we construct a monitoring state mapping extensions of θ to appropriate extensions of σ.

Definition 35 (Extending Projections)**.** *Let the projection extensions be defined as:*

$$\mathsf{extend_p}(\mathbf{a}, \theta, \sigma) \quad = \quad \left[\theta' \mapsto \sigma' \mid \theta' \in \mathsf{extensions}(\theta, \mathbf{a}) \;\wedge\; \sigma' {=} \begin{cases} \sigma.\mathbf{a} & \textit{if } \mathsf{relevant}(\mathbf{a}, \theta') \\ \sigma & \textit{otherwise} \end{cases} \right]$$

As outlined earlier, it is important that new bindings are added using the *maximal* existing binding. The following construction achieves this by iterating through existing bindings from largest to smallest (with respect to $\sqsubseteq$) and only adding bindings if they do not already exist. Therefore we will only be able to add a new binding if it is extending the maximal existing binding.

Definition 36 (Single Step Monitoring Construction)**.** *Given ground event* $\mathbf{a}$ *and monitoring state M. Let $\theta_1, \ldots, \theta_m$ be a linearisation of the domain of M i.e. if $\theta_j \sqsubset \theta_k$ then $j > k$ and every element in the domain of M is present once in the sequence, hence $m = |M|$. We define* $(\mathbf{a} * M) = N_m \in \mathit{MonitoringState}$ *where N_m is iteratively defined as follows for $i \in [1, m]$.*

$$\begin{aligned} N_0 &= [\;] \\ N_i &= N_{i-1} \dagger \mathsf{Add}_i \dagger \begin{cases} [\theta_i \mapsto M(\theta_i).\mathbf{a}] & \textit{if } \mathbf{a} \textit{ is relevant to } \theta_i \\ [\theta_i \mapsto M(\theta_i)] & \textit{otherwise} \end{cases} \end{aligned}$$

where $\mathsf{Add}_i = [(\theta' \mapsto \sigma') \in \mathsf{extend_p}(\mathbf{a}, \theta_i, M(\theta_i)) \mid \theta' \notin \mathsf{dom}(N_{i-1})]$

This builds up a new monitoring state from an old monitoring state, adding any binding extensions and appending the event to a projected trace if it is relevant. It is the linearisation of the domain of M with respect to $\sqsubseteq$ that ensures maximality. Note that, trivially, the linearisation used does not matter; if two bindings are not related by $\sqsubseteq$ they are inconsistent and cannot have the same extending binding.

This single step construction can then be used to give a monitoring state for a trace, typically the trace seen so far.

Definition 37 (Stepwise Monitoring)**.** *For a trace $\tau = \mathbf{a}_0.\mathbf{a}_1 \ldots \mathbf{a}_n$ we define the final monitoring state M_τ as* $\mathbf{a}_n * (\ldots * (\mathbf{a}_0 * [\;[\;] \mapsto \epsilon\;]) \ldots)$.

As is noted in Section 5.1 storing projections directly is inefficient and cumbersome. Instead we want to store the set of configurations that those projections reach. Next we extend the notions of extension to deal with configurations instead of projections.

5.3.2 Semantics with configurations

To deal with configurations instead of projections we replace monitoring states with monitor lookups, which map bindings to sets of configurations

$$\mathit{MonitorLookup} = \mathit{Binding} \rightharpoonup \mathcal{P}(\mathit{Config})$$

a set of configurations is necessary as a QEA can be non-deterministic.

To update a monitor lookup we update the set of configurations associated with a binding. We build up the next configurations by first constructing those configurations that are the result of taking a transition and then those that remain as no transition can be taken (recall we use a skip-style semantics).

Definition 38 (Next)**.** *For EA $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$ define the next configurations for binding θ, configurations C and a ground event $\mathbf{a}$ as follows:*

$$\mathsf{next}(\theta, \mathbf{a}, C) = \mathsf{move}(\theta, \mathbf{a}, C) \cup \mathsf{stay}(\theta, \mathbf{a}, C)$$

where the sets move *and* stay *are defined as*

$$\langle q_2, \gamma(\varphi \dagger \mathsf{match}(\mathbf{a}, \mathbf{b}(\theta)))\rangle \in \mathsf{move}(\theta, \mathbf{a}, C) \;\; \mathit{if} \;\; \begin{array}{l} \exists q_1 \in Q, \exists g \in \mathit{Guard} : \\ \langle q_1, \varphi\rangle \in C \land (q_1, \mathbf{b}, g, \gamma, q_2) \in \delta \land \\ \mathsf{okay}(\mathbf{a}, \mathbf{b}, \theta, \varphi, g) \end{array}$$

$$\langle q_1, \varphi\rangle \in \mathsf{stay}(\theta, \mathbf{a}, C) \;\; \mathit{if} \;\; \begin{array}{l} \nexists\, \mathbf{b} \in \mathcal{A} : \exists q_2 \in Q, \exists g \in \mathit{Guard}, \exists \gamma \in \mathit{Assign} : \\ \langle q_1, \varphi\rangle \in C \land (q_1, \mathbf{b}, g, \gamma, q_2) \in \mathsf{E}.\delta \land \mathsf{okay}(\mathbf{a}, \mathbf{b}, \theta, \varphi, g) \end{array}$$

in terms of the okay *function, defined as*

$$\begin{array}{rl} \mathsf{okay}(\mathbf{a}, \mathbf{b}, \theta, \varphi, g) = & \mathsf{matches}(\mathbf{a}, \mathbf{b}(\theta)) \land \mathsf{quantified}(\mathsf{match}(\mathbf{a}, \mathbf{b}(\theta))) = [\ ] \\ & \land g(\varphi \dagger \mathtt{match}(\mathbf{a}, \mathbf{b}(\theta)) \end{array}$$

A configuration $\langle q_2, \varphi'\rangle$ is the result of a move if there is a configuration $\langle q_1, \varphi\rangle$ in C such that a transition starts in q_1 and ends in q_2, the guard is satisfied and φ' is the result of applying the assignment appropriately. Note that we make use of a function okay to check matching, the guard and, importantly, relevance. A configuration is said to stay if it was not used to construct a configuration that is the result of a move i.e. there is no appropriate transition in δ.

This declarative presentation of the `next` function may not be completely transparent, so for added clarity it is also captured as the Next function in Algorithm 1. To see that these are equivalent note that the algorithm considers each possible transition and adds a new configuration for each transition that can be made (relating to the move set) and retains the configuration if no transitions can be made (relating to the stay set). The checks and operations carried out for each transition can be directly mapped to those used in the definition of next.

Algorithm 1 Finding the next configurations when adding an event to a projection.

```
function NEXT($\theta$ : Binding, $\mathbf{a}$ : GEvent, C : Set[Config]) :Set[Config]
    next ← $\varnothing$
    for $\langle q, \varphi \rangle$ in C do
        for $(q_1, \mathbf{b}, g, \gamma, q_2) \in \mathsf{E}.\delta$ do
            if $q_1 = q \wedge \mathtt{matches}(\mathbf{a}, \mathbf{b}(\theta))$ then
                $\varphi' \leftarrow \varphi \dagger \mathtt{match}(\mathbf{a}, \mathbf{b}(\theta))$
                if $\mathtt{quantified}(\varphi')$=[ ] and $g(\varphi')$ then
                    next ← next + $\langle q_2, \gamma(\varphi') \rangle$
        if no transitions are taken then
            next ← next $\cup \langle q, \varphi \rangle$
    return next
```

An important property of next is that if an event is not relevant to a binding then its next configurations remain unchanged.

Lemma 4. *If* $\mathbf{a}$ *is not relevant to* θ *then* $\mathsf{next}(\theta, \mathbf{a}, C) = C$.

Proof. We show that if $\mathbf{a}$ is not relevant to θ then

$$\mathsf{move}(\theta, \mathbf{a}, C) = \{\} \qquad \mathsf{stay}(\theta, \mathbf{a}, C) = C$$

and therefore $\mathsf{next}(\theta, \mathbf{a}, C) = C$. We begin by noting that by the definition of event relevance (Def. 33) if $\mathbf{a}$ is not relevant to θ then

$$\forall \mathbf{b} \in \mathcal{A}(\theta) : \neg \mathsf{matches}(\mathbf{a}, \mathbf{b}) \vee \mathsf{quantified}(\mathsf{match}(\mathbf{a}, \mathbf{b})) \neq [\ ]$$

and we can expand out the application of θ to $\mathcal{A}$ to write the equivalent

$$\forall \mathbf{b} \in \mathcal{A} : \neg \mathsf{matches}(\mathbf{a}, \mathbf{b}(\theta)) \vee \mathsf{quantified}(\mathsf{match}(\mathbf{a}, \mathbf{b}(\theta))) \neq [\ ]$$

This implies that

$$\forall \mathbf{b} \in \mathcal{A} : \neg \mathsf{okay}(\mathbf{a}, \mathbf{b}, \theta, \_).$$

This gives the result that $\mathsf{move}(\theta, \mathbf{a}, C) = \{\}$ as the condition for a configuration based on event $\mathbf{b}$ being in $\mathsf{move}(\theta, \mathbf{a}, C)$ is dependent on $\mathsf{okay}(\mathbf{a}, \mathbf{b}, \theta, \_)$. Similarly, a configuration in C is included in $\mathsf{stay}(\theta, \mathbf{a}, C)$ if and only if there is no $\mathbf{b}$ such that $\mathsf{okay}(\mathbf{a}, \mathbf{b}, \theta, \_)$ holds and there is a transition from that configuration involving $\mathbf{b}$. As $\mathsf{okay}(\mathbf{a}, \mathbf{b}, \theta, \_)$ does not hold for any $\mathbf{b}$ every configuration in C is included in $\mathsf{stay}(\theta, \mathbf{a}, C)$. □

An updated extend function can now be given that computes associated configurations for all extending bindings. Note that using Proposition 4 we can combine the two cases for when $\mathbf{a}$ is relevant or not.

Definition 39 (Extending Configurations). *Let the configuration extensions be defined as:*

$$\mathsf{extend_c}(\theta, \mathbf{a}, C) = [\theta' \mapsto \mathsf{next}(\theta', \mathbf{a}, C) \mid \theta' \in \mathsf{extensions}(\theta, \mathbf{a})]$$

We now update the single step monitoring construction given in Def. 36. This previous definition updated a monitoring state (map from bindings to projections) by stepping over a linearisation of its domain. We now replace this monitoring state with a monitoring lookup (map from bindings to sets of configurations). This requires us to make two other replacements. Firstly, the $\mathsf{extend_p}$ function is replaced by the $\mathsf{extend_c}$ function defined above. Secondly, the `next` function is used to compute the result of extending a projection. Note that the `next` function preserves configurations if an event is not relevant, as captured by Lemma 4. The single step monitoring constructions for configurations can be given as the construction in Def. 36 with these changes; we replicate it here with the described modifications for completeness.

Definition 40 (Single Step Monitoring Construction). *Given ground event* $\mathbf{a}$ *and monitoring lookup* L. *Let* $\theta_1, \ldots, \theta_m$ *be a linearisation of the domain of* L *i.e. if* $\theta_j \sqsubset \theta_k$ *then* $j > k$ *and every element in the domain of* L *is present once in the sequence, hence* $m = |L|$. *We define* $(\mathbf{a} * L) = N_m \in$ *MonitoringLookup where* N_m *is iteratively defined as follows for* $i \in [1, m]$.

$$\begin{aligned} N_0 &= [\ ] \\ N_i &= N_{i-1} \dagger \mathsf{Add}_i \dagger [\theta_i \mapsto \mathsf{next}(\theta_i, \mathbf{a}, L(\theta_i))] \end{aligned}$$

where $\mathsf{Add}_i = [(\theta' \mapsto C') \in \mathsf{extend_c}(\mathbf{a}, \theta_i, L(\theta_i)) \mid \theta' \notin \mathsf{dom}(N_{i-1})]$

This is lifted to traces in the same way as before. In this case a set of configurations capturing an empty projection is used.

Definition 41 (Stepwise Monitoring). *For a trace* $\tau = \mathbf{a}_0.\mathbf{a}_1 \ldots \mathbf{a}_n$ *we define the final monitoring lookup* L_τ *as* $\mathbf{a}_n * (\ldots * (\mathbf{a}_0 * [\ [\ ] \mapsto \{\langle q_0, [\ ]\rangle\}\ ])\ldots)$.

5.3.3 Type variable adjustment

Rather than deal with type variables in our construction, and add additional data structures, we will augment the alphabet to ensure that the domains are combined, as suggested in Sec. 3.5.2. We present a function on QEAs that achieves this. We present an implicit function we will often assume has been applied to a QEA.

Definition 42 (Type Variable Adjustment). *Given the QEA* $\mathcal{Q} = \langle \Lambda, \langle Q, \mathcal{A}, \delta, q_0, F\rangle, \mathcal{D}\rangle$ *let the type variable adjusted version be* $\mathsf{adjusted}(\mathcal{Q}) = \langle \Lambda, \langle Q, \mathcal{A}', \delta, q_0, F\rangle, \mathcal{D}\rangle$ *where if* $\Lambda = Q_1 x_1 : X_1.g_1, \ldots, Q_n x_n : X_n.g_n$ *for* $Q \in \{\forall, \exists\}$ *we have*

$$\mathcal{A}' = \mathcal{A} \cup \{e(y_1, \ldots, y_n) \mid \exists e(x_1, \ldots, x_n) \in \mathcal{A}, \forall i \in [1, n] : y_i \in \mathsf{varsOf}(\mathsf{typeOf}(x_i))\}$$

using the definitions of varsOf *and* typeOf *given in Def. 26.*

This may lead to a much larger alphabet but no bindings will be created that would not have been in the big-step semantics. Adjusting the alphabet in this way ensures that all relevant bindings are contained in the domain of the monitor lookup.

5.3.4 Acceptance

As a monitor lookup contains all our information about the trace we can give acceptance as a predicate on monitor lookups.

Definition 43 (Monitor Lookup Acceptance)**.** *For QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ and trace τ let L_τ be the final monitor lookup for trace τ and QEA* $\mathsf{adjusted}\mathcal{Q}$*. Let*

$$D_L(x) = \begin{array}{ll} \mathcal{D}(\mathsf{typeOf}(x)) & \textit{if } \mathsf{typeOf}(x) \in \mathsf{dom}(\mathcal{D}) \\ \{\theta(x) \mid \theta \in \mathsf{dom}(L) \wedge x \in \mathsf{dom}(\theta)\} & \textit{otherwise} \end{array}$$

give the domain of a variable in a monitor lookup L. $\mathcal{Q}$ accepts τ if and only if $L_\tau \vDash_\theta \Lambda$, defined as

$$\begin{array}{llll} L \vDash_\theta & (\forall x : g)\Lambda' & \textit{iff} & \textit{for all } d \textit{ in } D_L(x) \textit{ if } g(\theta \dagger [x \mapsto d]) \textit{ then } L \vDash_{\theta \dagger [x \mapsto d]} \Lambda' \\ L \vDash_\theta & (\exists x : g)\Lambda' & \textit{iff} & \textit{for some } d \textit{ in } D_L(x), g(\theta \dagger [x \mapsto d]) \textit{ and } L \vDash_{\theta \dagger [x \mapsto d]} \Lambda' \\ L \vDash_\theta & \epsilon & \textit{iff} & \begin{cases} \exists \langle q, \varphi \rangle \in L(\theta) : q \in \mathcal{E}.F & \textit{if } \theta \in \mathsf{dom}(L) \\ \mathcal{E}.q_0 \in \mathcal{E}.F & \textit{if } \theta \notin \mathsf{dom}(L) \end{cases} \end{array}$$

The check for $\theta \in \mathsf{dom}(L)$ deals with the case where θ binds a quantified variable in $\mathcal{D}$ but not in $\mathsf{Dom}(\tau)$. In this case the projected trace is necessarily empty and therefore in the initial state. Later we show that all relevant bindings are generated in the monitor lookup.

5.4 Complexity of small step semantics

We consider the time complexity of the trace checking process as defined by the small-step semantics. We discuss this in terms of the size of monitor lookup with respect to the trace seen so far and the complexity of each step with respect to the size of monitor lookup. We assume a trace τ and a QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ where $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F \rangle$. We recall the structural properties introduced in Sec. 4.3 i.e. k, the maximum arity of an event, and r, the minimum number of times a value is reused.

The size of a monitor lookup

When considering the size of the monitor lookup we should consider the number of entries and the size of each entry.

Number of entries. Let us consider the size of $\mathsf{dom}(L_{\tau.\mathbf{a}})$ by examining the process by which new bindings are added to L_τ. The set of new bindings given $\mathbf{a}$ are:

$$\bigcup_{\theta \in \mathsf{dom}(L_\tau)} \{\theta' \in \mathsf{extensions}(\mathbf{a}, \theta) \mid \theta' \notin \mathsf{dom}(L_\tau)\}$$

If we have seen $\mathbf{a}$ before then this set is empty as we will already have added all extensions. Otherwise, let b be the number of new values in $\mathbf{a}$. A given binding of size l can be extended in a maximum $\sum_{i=(k-l)}^{k} \binom{b}{i}$ ways. Therefore, the number of bindings introduced depends on the structure of $\mathsf{dom}(L_\tau)$ i.e. the average size of bindings. Note that b will typically be small and less than k.

The ratio between these two cases is controlled by k and r i.e. the number of values in an event combined with the number of times a value is reused. That is the second case (where new bindings are introduced) will happen $\frac{|\tau|k}{r}$ times. Note that later (Lemma. 16) we show that $\mathsf{dom}(L_\tau) = \mathsf{construct}(\tau)$.

Size of entries. The number of configurations for each binding. If $\mathcal{Q}$ is deterministic then this is 1, otherwise it is bounded by the maximum branching possible i.e. the maximum number of paths that could be followed by τ in $\mathcal{Q}$. As an upper bound this is $|\tau|$. To see this imagine an EA with no quantified variables and a single looping transition labelled with $\mathtt{f}(x)$ and a trace of events with different values for x. Without free variables, the number of configurations is bounded by the number of states.

The size of D_L. Assuming $\mathcal{D} = [\ ]$, D_L gives the values for each quantified variable. In the worst case, every parameter of every event could introduce a new value. If we assume that each event is repeated on average r times and on average contributes only one value to the domain of each variable then the size of each domain would be $\frac{|\tau|}{r}$.

Complexity of a step

For each step of the single step monitoring construction (Def. 40) we compute a linearisation of the monitor lookup and for each entry we construct next and extend, after combining these into a new monitor lookup we then decide acceptance. Let us consider the complexity of each of these steps.

Let us assume our monitor lookup is of size m (i.e. contains m bindings).

- **Computing linearisation.** This requires sorting of the monitor lookup, which has a bound of $O(m \log m)$.
- **Computing** next**.** This requires iterating over the configurations C and δ. If we take matching, quantifying a binding and deciding guards to be constant time then this has a bound of $O(|C||\delta|)$. Note the above bound on C.
- **Computing** extend**.** This first computes the extending bindings and then calls next for each binding. Binding extensions are computed by first iterating over $\mathcal{A}$ and then taking all submaps of produced bindings, which will be bounded by $\mathcal{A}2^k$. We then take the lub-closure of this set and then iterate over it once more. This gives a total upper bound of $O(2^{k^2})$, with k being small.
- **Deciding acceptance.** This requires us to first construct D_L by iterating over the monitor lookup and then stepping over the tree-like structure given by $\models$, therefore deciding acceptance is bounded by

$$O(|L_\tau| + \prod_{i=0}^{i=|\Lambda|} |D_{L_\tau}(\Lambda(i))|)$$

Putting this together, the complexity of each step is

$$O(m \log m + m(|C||\delta| + 2^{k^2}) + |L_\tau| + \prod_{i=0}^{i=|\Lambda|} |D_{L_\tau}(\Lambda(i))|)$$

Note that a large portion of this involves computing a verdict on each step. An optimisation we introduce in Sec. 6.2 addresses this.

Summary

It is difficult to compare this with the complexity of the big-step semantics in Sec. 4.3 as so much is dependent on the structure of the trace and QEA. However, in the worst case, the small step approach is more complex as we need to keep track of intermediate information. In the next chapter we see optimisations that alleviate much of this complexity, i.e. introducing incremental checking and reducing the size of L_τ.

5.5 Proving equivalence of semantics

We want to prove that the big step semantics introduced in Chapter 3 and small step semantics introduced earlier in this chapter coincide. To do this we show that the acceptance relations in both semantics are equivalent.

Theorem 1 (Equivalence of Acceptance Relations)**.** *For all QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ and traces τ*

$$\tau \models \Lambda \Leftrightarrow L_\tau \models \mathcal{Q}.\Lambda \tag{5.1}$$

Where the left-hand-side acceptance relation is taken from Def. 27 on page 68 and the right-hand-side acceptance relation is taken from Def. 43 on page 102.

The proof of this theorem will take the following steps. To begin with (Sec. 5.5.1) we introduce an intermediate big-step semantics for partial bindings and show that this is equivalent to big-step semantics for total bindings. We then (Sec. 5.5.2) show that the small-step and big-step semantics construct the same set of bindings from a trace. We then (Sec. 5.5.3) establish that the trace projections in monitoring states coincide with those given by the partial big-step semantics, and then (Sec. 5.5.4) that monitor lookups contain configurations that agree with these trace projections. Therefore, by going via the partial big-step semantics we finally (Sec. 5.5.5) show that the verdict produced by the big-step and small-step semantics are the same and finally prove Theorem 1.

5.5.1 A big-step semantics for partial bindings

The big-step constructions introduced in Chapter 3 assumed that we were dealing exclusively with total bindings. To deal with partial bindings we introduce partial versions of an EA's general language and the transition relation.

We begin by using the notion of event relevance to define a notion of *partial ground alphabet* to replace (total) ground alphabet in Def. 24.

Definition 44 (Partial Ground Alphabet)**.** *Let the partial ground alphabet of EA $\mathcal{E}$ with alphabet $\mathcal{A}$ be*

$$\mathsf{partial\_ground}(\mathcal{E},\theta) = \{\mathbf{a} \in \mathit{GEvent} \mid \mathsf{relevant}(\mathbf{a},\theta)\}$$

We relate this to the ground alphabet for total bindings.

Lemma 5. *For total bindings projection with respect to a partial ground alphabet is equivalent to projection with respect to a ground alphabet.*

Proof. Lemma 3 shows that for total bindings θ, $\mathsf{ground}(\mathcal{E}(\theta)) = \mathsf{partial\_ground}(\mathcal{E},\theta)$, as the set of relevant events is equivalent to the ground alphabet. □

An EA's partial general language is then defined as before (in Def. 24) using the partial ground alphabet i.e.

$$\mathcal{L}_{partial\_G}(\mathcal{E},\mathcal{B}) = \{\tau \in \mathcal{B}^* \mid \tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E})} \in \mathcal{L}(\mathcal{E})\}$$

Again, by Lemma 3 this is equivalent to total general language when θ is total.

Next we introduce the *partial transition relation* that adds the notion of event relevance to the previous definition of transition relation for EA (Def. 22) i.e. we ensure that no quantified variables are bound. As before we assume that an EA is (partially) instantiated.

Definition 45 (Partial Transition Relation)**.** *Consider an EA $\mathcal{E} = \langle Q,\mathcal{A},\delta,q_0,F\rangle$. Let $\langle q,\varphi\rangle \overset{\mathbf{a}}{\rightsquigarrow} \langle q',\varphi'\rangle$ hold if*

$$\begin{aligned}\exists \mathbf{b} \in \mathcal{A}, \exists g \in \mathit{Guard}, \exists \gamma \in \mathit{Assign} : (q,\mathbf{b},g,\gamma,q') \in \delta \;\wedge\\ \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge g(\varphi \dagger \mathsf{match}(\mathbf{a},\mathbf{b})) \wedge \varphi' = \gamma(\varphi \dagger \mathsf{match}(\mathbf{a},\mathbf{b})) \;\wedge\\ \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\;]\end{aligned}$$

We lift this relation to $\rightsquigarrow_{\mathcal{E}}$ as we did before.

Again it is necessary to show that the partial transition relation coincides with the total transition relation for total bindings.

Lemma 6. *For all traces τ and QEA $\langle\Lambda,\mathcal{E},\mathcal{D}\rangle$*

$$\forall \theta \in \mathit{Bind} : \mathsf{total}(\theta,\Lambda) \Rightarrow \{c \mid \langle q_0,[\;]\rangle \overset{\tau}{\rightsquigarrow}_{\mathcal{E}(\theta)} c\} = \{c \mid \langle q_0,[\;]\rangle \overset{\tau}{\rightarrow}_{\mathsf{E}(\theta)} c\}$$

Proof. The difference between the total and partial transition relations is that the partial relation contains the condition $\mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\;]$. If this condition does not hold then there must be some quantified variables (in Λ) which θ is not defined for. But as $\mathsf{total}(\theta,\Lambda)$ this cannot be the case and therefore this condition must always be true, and the two transition relations are equivalent. □

Note. *The condition in Lemma 6 is sufficient because of the* skip *semantics of EA. If we were to change this to a* next *semantics then it would be necessary to add the additional constraint that all events in τ are relevant to the binding used to instantiate the EA. Note that where we use this Lemma later (Lemma 19) this condition holds, so it would be possible to weaken this statement without altering the proof.*

5.5.2 Computing the same bindings

Next we show that the two semantics construct the same set of total bindings. We begin by discussing some properties of the constructed and relevant bindings for the big-step semantics as captured by $\mathsf{construct}(\tau)$ in Def. 29. As a reminder these are defined as follows:

$$\begin{aligned}\mathsf{construct}(\tau) &= \{\theta \in \mathit{Bind} \mid \forall (x \mapsto v) \in \theta : v \in \mathsf{Dom}(\tau)(\mathsf{typeOf}(x))\}\\ \mathsf{relevant}(\tau, \Lambda) &= \{\theta \in \mathsf{construct}(\tau) \mid \mathsf{total}(\theta, \mathsf{vars}(\Lambda))\}\end{aligned}$$

Lemma 7. $\mathsf{construct}(\tau)$ *is lub-closed*

Proof. For any $\Theta \subseteq \mathsf{construct}(\tau)$ assume that $\bigsqcup \Theta$ exists but $\bigsqcup \Theta \notin \mathsf{construct}(\tau)$. As $\bigsqcup \Theta \notin \mathsf{construct}(\tau)$ there must be an x such that $(\bigsqcup \Theta)(x) = v$ and $v \notin \mathsf{Dom}(\tau)(\mathsf{typeOf}(x))$ and therefore by definition of $\bigsqcup \Theta$ there must be a $\theta \in \Theta$ such that $\theta(x) = v$ but as $\theta \in \mathsf{construct}(\tau)$ we know that $v \in \mathsf{Dom}(\tau)(\mathsf{typeOf}(x))$ and we get a contradiction. $\square$

Furthermore, $\mathsf{construct}(\tau)$ is the lub-closure of all single bindings in $\mathsf{Dom}(\tau)$ i.e.

$$\mathsf{construct}(\tau) = \mathsf{close}_{\sqcup}(\{[x \mapsto v] \mid x \in \mathsf{vars}(\Lambda) \land v \in \mathsf{Dom}(\tau)(x)\})$$

Next we consider the bindings that are built using the small-step semantics. On each step we add the $\mathsf{extensions}$ of all bindings present on the previous step. This can be captured using the following inductive definition.

Definition 46 (Build)**.** *We organise the construction of* `build` *in an iterative fashion.*

$$\begin{aligned}\mathsf{build}(\epsilon) &= \{[\,]\}\\ \mathsf{build}(\tau\mathbf{a}) &= \mathsf{build}(\tau) \cup \textstyle\bigcup_{\theta \in \mathsf{build}(\tau)} \mathsf{extensions}(\theta, \mathbf{a})\end{aligned}$$

It is important to establish the equivalence between $\mathsf{build}(\tau)$ and $\mathsf{construct}(\tau)$ as this will demonstrate that the small-step semantics builds all the same bindings (later we show that this is the domain of a monitoring state). Importantly, $\mathsf{build}(\tau)$ is computed against a QEA modified to compensate for type variables as defined in Def. 42 on page 101, so we first show that this modification preserves $\mathsf{construct}(\tau)$.

Lemma 8. *Given QEA* $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ *and trace* τ *let* $\mathsf{adjusted}(\mathcal{Q}) = \langle \Lambda, \mathcal{E}', \mathcal{D} \rangle$ *then*

$$\forall x \in \mathsf{vars}(\Lambda) : x \notin \mathsf{dom}(\mathcal{D}) \Rightarrow \mathsf{Dom}^{\mathcal{E}}(\tau)(\mathsf{typeOf}(x)) = D_{L_\tau}(x)$$

where L_τ *is based on* $\mathsf{adjusted}(\mathcal{Q})$.

The proof is straightforward and is given in Appendix B.1. We can now show that $\mathsf{build}(\tau)$ and $\mathsf{construct}(\tau)$ are equivalent.

Lemma 9. *For all traces* τ *and all QEA* $\langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$

$$\mathsf{build}(\tau) = \mathsf{construct}(\tau)$$

The proof is straightforward and is given in Appendix B.1. We have shown that the bindings needed for acceptance i.e. $\mathsf{relevant}(\tau, \Lambda)$ are produced as they are a subset of $\mathsf{construct}(\tau)$.

Note. *At this point it should be noted that we do not need to construct all of* $\mathsf{construct}(\tau)$*, only those bindings necessary for the creation of* $\mathsf{relevant}(\tau, \Lambda)$*. In the next chapter we will see some optimisations that take advantage of this.*

5.5.3 Computing the same projections

In this section we show that the projection associated with a binding in a monitoring state for a given binding is the same as that given in the big-step semantics for the same binding. We do this in 3 steps by showing that

1. Every binding with a non-empty projection is considered
2. When extending a projection the maximal existing bindings is used
3. This means that the recorded projections are correct

We first establish that the domain of M_τ builds all relevant bindings by showing that the domain of M_τ is equivalent to $\mathsf{build}(\tau)$.

Lemma 10. *For all traces τ and QEA $\langle \Lambda, \mathcal{E} \rangle$*

$$\mathsf{build}(\tau) = \mathsf{dom}(M_\tau)$$

Proof. By structural induction on τ.

Base Step. For $\tau = \epsilon$, $\mathsf{build}(\epsilon) = \{[\ ]\}$ and $\mathsf{dom}(M_\epsilon) = \{[\ ]\}$.

Inductive Step. Consider the trace $\tau.\mathbf{a}$ We show that

$$\mathsf{build}(\tau.\mathbf{a}) = \mathsf{dom}(\mathbf{a} * M_\tau) \tag{5.2}$$

by showing that the additions to $\mathsf{build}(\tau)$ and $\mathsf{dom}(M_\tau)$ are equal.

In the construction of $(\mathbf{a} * M_\tau)$ (Def. 36) new bindings are introduced through the extend function applied to each binding in M_τ. The bindings introduced by the extend function are captured by the $\mathsf{extensions}$ function (Def. 34). Therefore,

$$\mathsf{dom}(\mathbf{a} * M_\tau) = \mathsf{dom}(M_\tau) \cup \bigcup_{\theta \in \mathsf{dom}(M_\tau)} \mathsf{extensions}(\theta, \mathbf{a}) \tag{5.3}$$

By using (5.3) and expanding out $\mathsf{build}(\tau.\mathbf{a})$ (Def. 46) equation (5.2) becomes

$$\mathsf{build}(\tau) \cup \bigcup_{\theta \in \mathsf{build}(\tau)} \mathsf{extensions}(\theta, \mathbf{a}) = \mathsf{dom}(M_\tau) \cup \bigcup_{\theta \in \mathsf{dom}(M_\tau)} \mathsf{extensions}(\theta, \mathbf{a}) \tag{5.4}$$

Using the induction hypothesis $\mathsf{build}(\tau) = \mathsf{dom}(M_\tau)$ equation (5.4) holds. □

We have shown that the bindings generated in big and small step semantics are equivalent. Next we show that the domain of M_τ grows monotonically.

Lemma 11. *For all traces $\tau.\mathbf{a}$ and QEA $\langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$*

$$\mathsf{dom}(M_\tau) \subseteq \mathsf{dom}(M_{\tau\mathbf{a}})$$

Proof. By definition $\mathsf{build}(\tau) \subseteq \mathsf{build}(\tau\mathbf{a})$. By Lemma, 10 $\mathsf{build}(\tau) = \mathsf{dom}(M_\tau)$ and $\mathsf{build}(\tau\mathbf{a}) = \mathsf{dom}(M_{\tau\mathbf{a}})$ so we have $\mathsf{dom}(M_\tau) \subseteq \mathsf{dom}(M_{\tau\mathbf{a}})$ □

So that we can use $\mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta)$ later we must first show that the domain of M_τ is lub-closed.

Lemma 12. *For all traces τ and QEA $\langle \Lambda, \mathcal{E} \rangle$ the set of bindings $\mathsf{dom}(M_\tau)$ is least-upper-bound closed*

Proof. By Lemmas 7, 9 and 10, $\mathsf{construct}(\tau)$ is lub-closed and $\mathsf{construct}(\tau) = \mathsf{build}(\tau) = \mathsf{dom}(M_\tau)$ □

We now show that when a binding θ is added to the monitoring state the entry for the maximum existing consistent binding ($\mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta)$) is used. This ensures that no information is lost as otherwise events from that binding's trace slice would be lost. Therefore this result is important in establishing our later result that partial projections are preserved in the monitoring state.

Lemma 13. *For all traces $\tau.\mathbf{a}$ and QEA $\langle \Lambda, \mathcal{E} \rangle$*

$$\forall \theta : \theta \notin \mathsf{dom}(M_\tau) \wedge \theta \in \mathsf{dom}(M_{\tau\mathbf{a}}) \Rightarrow M_{\tau\mathbf{a}}(\theta) = \mathsf{extend}(\mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta), \mathbf{a})(\theta)$$

Proof. By contradiction. Let us assume that for some θ that $\theta \notin \mathsf{dom}(M_\tau)$ and $\theta \in \mathsf{dom}(M_{\tau\mathbf{a}})$ but that $M_{\tau\mathbf{a}} \neq \mathsf{extend}(\mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta), \mathbf{a})(\theta)$.

By Def 36 (single step monitoring on page 98) the binding θ was introduced into $M_{\tau\mathbf{a}}$ by $\mathsf{extend}(\theta', \mathbf{a})$ for some binding θ'. By our assumption $\theta' \neq \mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta)$. By the definition of extend it is necessarily the case that $\theta' \sqsubseteq \theta$. As M_τ is linearised with respect to $\sqsubseteq$ and due to the order in which maps are combined in Def. 36 it must be the case that $\mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta) \sqsubset \theta'$. As $\theta' \in \mathsf{dom}(M_\tau)$ and M_τ is lub-closed, Lemma 2 tells us that $\mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta') = \theta'$ and therefore $\theta \sqsubset \theta'$. We have a contradiction and no such θ' can exist. □

Therefore, this property is ensured by iterating over the linearisation of the monitoring state with respect to $\sqsubseteq$.

Next we show that if a binding θ is not in the monitoring state then there exists a binding θ' in the monitoring state such that θ and θ' have the same partial projections, and that $\theta' = \mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta)$. This means that the monitoring state implicitly contains the trace projection for *every* possible binding. This fact is used later to show that when we add a binding we use this implicit information.

Lemma 14. *For all traces $\tau.\mathbf{a}$ and QEA $\langle \Lambda, \mathcal{E} \rangle$*

$$\forall \theta : \theta \notin \mathsf{dom}(M_\tau) \Rightarrow \tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))} = \tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}((\mathsf{max\ below}(\mathsf{dom}(M_\tau),\theta))))}$$

Proof. Using the definition of partial ground alphabet (Def. 44), we can restate this as:

$$\forall \theta \notin \mathsf{dom}(M_\tau), \forall \mathbf{a} \in \tau : \mathsf{relevant}(\mathbf{a},\theta) \Rightarrow \mathsf{relevant}(\mathbf{a}, \mathsf{max\ below}(\mathsf{dom}(M_\tau),\theta))$$

We continue by contradiction. Let $\theta_m = \mathsf{max\ below}(\mathsf{dom}(M_\tau),\theta)$. Consider an $\mathbf{a} \in \tau$ such that $\mathsf{relevant}(\mathbf{a},\theta)$ but $\neg\mathsf{relevant}(\mathbf{a},\theta_m)$. By considering the definition of $\mathsf{relevant}$ (Def. 33) we have that

$$\begin{array}{rcl} \neg\mathsf{relevant}(\mathbf{a},\theta_m) & = & \neg(\exists \mathbf{b} \in \mathcal{A}(\theta_m) : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\ ]) \\ & = & \neg(\exists \mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) \sqsubseteq \theta_m) \end{array}$$

Combining this with the fact that $\theta_m \sqsubseteq \theta$ (from the definition of $\mathsf{max\ below}$) gives us

$$\neg(\exists \mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) \sqsubseteq \theta)$$

But as $\mathsf{relevant}(\mathbf{a},\theta)$ we know that

$$\exists \mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) \sqsubseteq \theta$$

Leading to a contradiction, thus such an event $\mathbf{a} \in \tau$ cannot exist.

□

Finally, we show that the projections given by a monitoring state are those given by projecting the trace with respect to the partial ground alphabet. Combined with Lemma 5 this shows that for all total bindings a monitoring state contains the same projections as produced in the big-step semantics.

Lemma 15. *For all traces τ and QEA $\langle \Lambda, \mathcal{E} \rangle$*

$$\forall (\theta \mapsto \sigma) \in M_\tau : \sigma = \tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))}$$

Proof. By structural induction on τ.

Base Step. Let $\tau = \epsilon$. By definition, $M_\epsilon = [[\ ] \mapsto \epsilon]$ and $\epsilon \downarrow_{\mathsf{partial\_ground}(\mathcal{E}([\ ]))} = \epsilon$

Inductive Step. Consider the trace $\tau.\mathbf{a}$. By using our inductive hypothesis

$$\forall (\theta \mapsto \sigma) \in M_\tau : \sigma = \tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))}$$

we show that

$$\forall (\theta \mapsto \sigma) \in (\mathbf{a} * M_\tau) : \sigma = \tau\mathbf{a} \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))}$$

by considering two cases: when θ is in $\mathsf{dom}(M_\tau)$ and when it is not.

- If $\theta \in \mathsf{dom}(M_\tau)$ then by Def. 36 (single step monitoring) $\mathbf{a}$ is added to $M_\tau(\theta)$ to give σ iff $\mathsf{relevant}(\mathbf{a}, \theta)$, the same condition used in the definition of partial ground alphabet 44.
- If $\theta \notin \mathsf{dom}(M_\tau)$ then by Lemma 13 the projection for θ is given by $\mathsf{extend}(\theta_m, \mathbf{a})$ where $\theta_m = \mathsf{max\ below}(\mathsf{dom}(M_\tau), \theta)$. By Lemma 14

  $$\sigma \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))} = \sigma \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta_m))}$$

  and therefore to show that $(\mathbf{a} * M_\tau)(\theta) = \tau\mathbf{a} \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))}$ it is sufficient to show that $\mathbf{a}$ is added to $\tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta_m))}$ under the same conditions as when projecting with the partial ground alphabet. The extension of projections in the definition of `extend` (Def. 34) is dependent on event relevance; the same condition used when projecting with the partial ground alphabet.

In both cases, $\mathbf{a}$ is only added to the projection for θ under the same condition used when projecting with the partial ground alphabet, event relevance.

□

5.5.4 Computed configurations match computed projections

Now let us consider how configurations are constructed in monitor lookup L_τ. We first show that the bindings created in the construction of L_τ are exactly those created in the construction of M_τ.

Lemma 16. *For all traces τ and QEA $\langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$*

$$\mathsf{dom}(M_\tau) = \mathsf{dom}(L_\tau)$$

Proof. By structural induction on τ. For $\tau = \epsilon$ the domains of M_ϵ and L_ϵ are equal by definition. For $\tau = \sigma.\mathbf{a}$ the bindings added to M_τ and L_τ are

$$\bigcup_{\theta \in \mathsf{dom}(M_\tau)} \mathsf{extensions}(\theta, \mathbf{a}) \quad \text{and} \quad \bigcup_{\theta \in \mathsf{dom}(L_\tau)} \mathsf{extensions}(\theta, \mathbf{a})$$

respectively. So, by our inductive hypothesis, bindings added on each step are equal. □

We now consider the next function (Def. 38) used in the construction of L_τ and relate this to the partial transition function introduced in Sec. 5.5.1 in Def. 45.

Lemma 17. *For binding θ, event $\mathbf{a}$ relevant to θ, and set of configurations C*

$$\mathsf{next}(\mathbf{a}, \theta, C) = \{c' \mid \exists c \in C : c \overset{\mathbf{a}}{\rightsquigarrow}_{\theta, \mathcal{E}(\theta)} c'\}$$

The proof is straightforward and is given in Appendix B.2. We now relate the construction of configurations in the small-step semantics to that in the big-step semantics by showing that the configurations in L_τ are exactly those that satisfy the partial transition relation.

Lemma 18. *For all traces τ and QEA $\langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$*

$$\forall (\theta \mapsto C) \in L_\tau : C = \{c \mid \langle q, [\,]\rangle \overset{M_\tau(\theta)}{\rightsquigarrow}_{\theta,\mathcal{E}}\, c\}$$

Proof. By structural induction on τ.

Base Case. Let $\tau = \epsilon$. By definition, $L_\epsilon = [[\,] \mapsto \{\langle q_0, [\,]\rangle\}]$ and $\langle q_0, [\,]\rangle \overset{\epsilon}{\rightsquigarrow}_{\theta,\mathcal{E}} \langle q_0, [\,]\rangle$.

Inductive Step. Consider trace $\tau.\mathbf{a}$. We show that for a given $\theta \in \mathsf{dom}(L_{\tau\mathbf{a}})$

$$L_{\tau\mathbf{a}}(\theta) = \{c \mid \langle q, [\,]\rangle \overset{M_{\tau\mathbf{a}}(\theta)}{\rightsquigarrow}_{\theta,\mathcal{E}}\, c\} \tag{5.5}$$

by considering two cases: when $\mathbf{a}$ is relevant to θ and when it is not. Firstly, if $\mathbf{a}$ is not relevant to θ then by Def. 40 the binding θ must have existed in L_τ and $L_{\tau\mathbf{a}}(\theta) = L_\tau(\theta)$. By considering Def. 36 we see that in this case $M_{\tau\mathbf{a}}(\theta) = M_\tau(\theta)$ and therefore, by our inductive hypothesis, (5.5) holds.

In the case where $\mathbf{a}$ is relevant to θ there are two further cases to consider: where θ is in $\mathsf{dom}(L_\tau)$ and when it is not. We show that in each case there is a set of configurations C such that

$$C = \{c \mid \langle q_0, [\,]\rangle \overset{M_\tau}{\rightsquigarrow}_{\mathcal{E}(\theta)}\, c\} \quad \text{and} \quad L_{\tau\mathbf{a}}(\theta) = \mathsf{next}(\mathbf{a}, \theta, C) \tag{5.6}$$

and therefore by Lemma 17, the definition of $\rightsquigarrow$ (Def. 45) and Lemma 15 combined with the definition projection with respect to a partial ground alphabet we have that

$$\begin{array}{rcl} L_{\tau\mathbf{a}}(\theta) & = & \{c \mid \exists c' \in \mathit{Config} : \langle q, [\,]\rangle \overset{M_\tau(\theta)}{\rightsquigarrow}_{\theta,\mathcal{E}}\, c' \overset{\mathbf{a}}{\rightsquigarrow}_{\mathcal{E}(\theta)}\, c\} \\ & = & \{c \mid \langle q, [\,]\rangle \overset{M_\tau(\theta)\mathbf{a}}{\rightsquigarrow}_{\mathcal{E}(\theta)}\, c\} \\ & = & \{c \mid \langle q, [\,]\rangle \overset{M_{\tau\mathbf{a}}(\theta)}{\rightsquigarrow}_{\mathcal{E}(\theta)}\, c\} \end{array}$$

We now show that in each case there exists a C in L_τ such that (5.6) holds.

- If $\theta \in \mathsf{dom}(L_\tau)$ then by our inductive hypothesis

  $$\forall c \in L_\tau(\theta) : \langle q, [\,]\rangle \overset{M_\tau(\theta)}{\rightsquigarrow}_{\theta,\mathcal{E}}\, c$$

  we have a C satisfying the left-hand side of (5.6). By Def. 40 the configurations $L_{\tau\mathbf{a}}(\theta)$ are given by $\mathsf{next}(\mathbf{a}, \theta, L_\tau(\theta))$ (recall that $\mathbf{a}$ is relevant to θ), thus giving us the right-hand side of (5.6).

- If $\theta \notin \mathsf{dom}(L_\tau)$ then by Lemma 14, Lemma 16 and our inductive hypothesis

  $$L_\tau(\mathsf{max\ below}(\mathsf{dom}(L_\tau), \theta)) = \{c \mid \langle q_0, [\,]\rangle \overset{M_\tau}{\rightsquigarrow}_{\theta,\mathcal{E}}\, c\}$$

  Due to the linearisation of $\mathsf{dom}(L_\tau)$ in Def 40 $\mathsf{max\ below}(\mathsf{dom}(L_\tau), \theta)$ will be encountered first (i.e. will appear before any other bindings consistent with θ in the linearisation). The binding θ is then introduced and initialised using $\mathsf{next}(\mathbf{a}, \theta, L_\tau(\mathsf{max\ below}(\mathsf{dom}(L_\tau), \theta))$. Therefore (5.6) is satisfied in this case.

Therefore, (5.5) holds when **a** is relevant to θ. □

This allows us to state an important lemma that brings together the statements proved in the last few sections.

Lemma 19. *For all traces τ and QEA $\langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$*

$$\forall \theta \in \mathsf{construct}(\tau) : \mathsf{total}(\theta, \Lambda) \Rightarrow (\exists \langle q, \varphi \rangle \in L_\tau(\theta) : q \in \mathrm{F} \Leftrightarrow \tau \downarrow_{\mathsf{ground}(\mathcal{E}(\theta))} \in \mathcal{L}(\mathcal{E}(\theta)))$$

i.e. Every total constructed binding appears in L_τ with an accepting configuration if and only if the trace is in the general language of the instantiated EA.

Proof. Firstly we show that L_τ contains the correct configurations for every binding:

1 $\quad \mathsf{construct}(\tau) = \mathsf{build}(\tau) = \mathsf{dom}(M_\tau) = \mathsf{dom}(L_\tau)$ $\quad$ Lemma 9,10,16

2 $\quad \forall (\theta \mapsto \sigma) \in M_\tau : \sigma = \tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))}$ $\quad$ Lemma 15

3 $\quad \forall (\theta \mapsto C) \in L_\tau : C = \{c \mid \langle q, [\ ] \rangle \overset{M_\tau(\theta)}{\rightsquigarrow}_{\mathcal{E}(\theta)} c\}$ $\quad$ Lemma 18

4 $\quad \forall \theta \in \mathsf{dom}(L_\tau) : L_\tau(\theta) = \{c \mid \langle q, [\ ] \rangle \xrightarrow{\tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))}} c\}$ $\quad$ 2, 3

5 $\quad \forall \theta \in \mathit{Bind} : \mathsf{total}(\theta, \Lambda) \Rightarrow \{c \mid \langle q_0, [\ ] \overset{\tau}{\rightsquigarrow}_{\theta, \mathcal{E}} c\} = \{c \mid \langle q_0, [\ ] \overset{\tau}{\rightarrow}_{\mathcal{E}(\theta)} c\}$ $\quad$ Lemma 6

6 $\quad \mathsf{total}(\theta, \Lambda) \Rightarrow \tau \downarrow_{\mathsf{ground}(\mathcal{E}(\theta))} = \tau \downarrow_{\mathsf{partial\_ground}(\mathcal{E}(\theta))}$ $\quad$ Lemma 5

7 $\quad \forall \theta \in \mathsf{construct}(\tau) : \mathsf{total}(\theta, \Lambda) \Rightarrow: L_\tau(\theta) = \{c \mid \langle q, [\ ] \rangle \xrightarrow{\tau \downarrow_{\mathsf{ground}(\mathcal{E}(\theta))}}_{\mathcal{E}(\theta)} c\}$ $\quad$ 1, 4, 5, 6

By using the definitions of projection (Def. 6) and an EA language (Def. 23) we show that if we reach a final state with a trace projection then we are in an EA's language, which combines with what we showed previously to prove our lemma.

8 $\quad \forall \theta : \tau \downarrow_{\mathsf{ground}(\mathcal{E}(\theta))} \in \mathsf{ground}(\mathcal{E}(\theta))^*$ $\quad$ Def. 6

9 $\quad \forall \theta : (\exists \langle q, \varphi \rangle : \langle q_0, [\ ] \rangle \xrightarrow{\tau \downarrow_{\mathsf{ground}(\mathcal{E}(\theta))}}_{\mathcal{E}(\theta)} \langle q, \varphi \rangle \wedge q \in \mathrm{F}) \Leftrightarrow$
$\quad (\tau \downarrow_{\mathsf{ground}(\mathcal{E}(\theta))} \in \mathcal{L}(\mathcal{E}(\theta)))$ $\quad$ 8, Def. 23

10 $\quad \forall \theta \in \mathsf{construct}(\tau) : \mathsf{total}(\theta, \Lambda) \Rightarrow (\exists \langle q, \varphi \rangle \in L_\tau(\theta) : q \in \mathrm{F} \Leftrightarrow$
$\quad \tau \downarrow_{\mathsf{ground}(\mathcal{E}(\theta))} \in \mathcal{L}(\mathcal{E}(\theta)))$ $\quad$ 7,9

□

We now show that the definition of D_{L_τ} in Def. 43 coincides with the use of $\mathsf{Dom}(\tau)\dagger\mathcal{D}$ in Def. 27.

Lemma 20. *For all QEA* $\langle\Lambda,\mathcal{E},\mathcal{D}\rangle$ *and trace* τ

$$\forall x \in \mathsf{vars}(\Lambda) : D_{L_\tau}(x) = (\mathsf{Dom}(\tau)\ \dagger\mathcal{D})(x)$$

Proof. From Def 16 and Lemmas 9, 10 and 18 we have that

$$\{\theta \in \mathit{Bind} \mid \forall(x \mapsto v) \in \theta : v \in \mathsf{Dom}(\tau)(x)\} = \mathsf{construct}(\tau) = \mathsf{build}(\tau) = \mathsf{dom}(M_\tau) = \mathsf{dom}(L_\tau)$$

and therefore if $\mathsf{typeof}(x) \in \mathsf{dom}(\mathcal{D})$ then $D_{L_\tau} = \mathcal{D}(x)$ and otherwise

$$D_{L_\tau}(x) = \{\theta(x) \mid \forall(y \mapsto v) : v \in \mathsf{Dom}(\tau)(y) \wedge x \in \mathsf{dom}(\theta)\} = \{v \mid v \in \mathsf{Dom}(\tau)(x)\} = \mathsf{Dom}(\tau)(x)$$

$\square$

5.5.5 Completing the proof

We can now prove Theorem 1.

Theorem 1 (Equivalence of Acceptance Relations)**.** *For all QEA* $\mathcal{Q} = \langle\Lambda,\mathcal{E},\mathcal{D}\rangle$ *and traces* τ

$$\tau \vDash \Lambda \Leftrightarrow L_\tau \vDash \mathcal{Q}.\Lambda \tag{5.7}$$

Proof. By structural induction on Λ for $\tau \vDash_\theta \Lambda.\mathcal{E}$ in Def. 27 and $L_\tau \vDash_\theta \Lambda$ in Def. 43.

Base Case. $\Lambda = \epsilon$. The condition for $\tau \vDash_\theta \epsilon$ is

$$\tau \downarrow_{\mathsf{E}(\theta)} \in \mathcal{L}(\mathcal{E}(\theta)) \tag{5.8}$$

and the condition for $L_\tau \vDash_\theta \Lambda$ is

$$\begin{array}{ll} \exists\langle q,\varphi\rangle \in L_\tau(\theta) : q \in \mathrm{F} & \text{if } \theta \in \mathsf{dom}(L_\tau) \\ q_0 \in F & \text{otherwise} \end{array} \tag{5.9}$$

By construction, θ is in $\mathsf{construct}(\theta)$ and total, and therefore either by Lemma 19 equations (5.8) and (5.9) are equivalent or $\tau \downarrow_{\mathsf{ground}(\theta)} = \epsilon$.

Inductive Step. $\Lambda = (Q,x,g,X).\Lambda'$ for $Q \in \{\exists,\forall\}$. The cases for each Q are symmetric so we take $Q = \forall$. This case in Def. 27 on page 68 is

$$\text{for all } d \text{ in } \mathcal{D}(X) \text{ if } g(\theta\dagger[x \mapsto d]) \text{ then } \tau \vDash_{\theta\dagger[x\mapsto d]} \Lambda'$$

and in Def. 43 on page 102 it is

$$\text{for all } d \text{ in } D_L(x) \text{ if } g(\theta\dagger\ [x \mapsto d]) \text{ then } L \vDash_{\theta\dagger[x\mapsto d]} \Lambda'$$

By Lemma 20 (stating that the domains of x used in each definition are equivalent) and our inductive hypothesis that the two definitions are equivalent for Λ' we see that these two conditions are equal. $\square$

5.6 Basic Algorithm

In this section we introduce a basic (i.e. unoptimized) incremental monitoring algorithm based on the small-step semantics presented in Section 5.3. A worked example of this algorithm is given in Appendix A.2.3.

5.6.1 The algorithm

The algorithm has four parts, thus allowing future optimisations to replace individual parts. These are

1. **Initialisation.** Called once at the start of the trace
2. **Step.** Callled for each event (step of the process) and returns a verdict
3. **Update.** Uses the event to update the data structures
4. **Check.** Uses the data structures to reach a verdict

and are captured in Algorithms 2 to 5.

A note on types. The types referred to in this chapter have been introduced in Chapter 3. Recall that here the Verdict type is the boolean domain (later we will replace this). We use the following (`Scala`-like) syntax for collection types:

- Set[T] or List[T] for a set or list of things of type T
- Map[S,T] for a map from things of type S to things of type T

An overview

The functions given in Algorithms 2 to 5 operate as follows. The initialisation in Algorithm 2 creates an initial monitor lookup (see Section 5.3.2). The step function in Algorithm 3 calls the update and check functions to update the monitor lookup and compute a verdict from it. The update and check functions are separated so that they can be optimized separately. Note that they share the state defined in Algorithm 2, here the QEA $\mathcal{Q}$ and the monitor lookup L.

The update function in Algorithm 4 captures the single step monitor construction given in Definition 40 on page 101, making use of the NEXT function given in Algorithm 1 introduced in Section 5.3.2 on page 100. Note that the order (biggest to smallest) is used to enforce maximality, see Section 5.3 for a full discussion of this and the example in Appendix A.2.3 for a demonstration.

Finally the check function in Algorithm 5 computes the verdict. To do this it first computes the domain by examining L and then steps through the quantifier list to compute the verdict as

Algorithm 2 The data structures and initialisation function for the basic algorithm.

```
$\mathcal{Q}$ : QEA
L : Map[Binding, Set[Config]] $\leftarrow$ [ ]

function INIT($\mathcal{Q}$: QEA)
    L $\leftarrow$ ([ ]$\mapsto \{\langle \mathcal{Q}.\mathcal{E}.q_0, [\ ]\rangle\}$)
    $\mathcal{Q} \leftarrow \mathcal{Q}$
```

Algorithm 3 The step function for the basic algorithm.

```
function STEP($\mathbf{a}$ : GEvent) : Verdict
    UPDATE($\mathbf{a}$)
    return CHECK
```

set out in Definition 27. Note that this algorithm returns a result as soon as possible, preventing unnecessary recursive calls.

Understanding the update function

Let us consider the steps of the update function in Algorithm 4.

We begin by making a call to the MATCHING function, which constructs the closed set of bindings that matches the incoming event with events in the alphabet of the EA. It considers each event in the alphabet and extracts the quantified matching binding (if it exists) and maintains a lub-closed set.

We then iterate through M from biggest to smallest updating bindings as we did in Definition 40. To enable us to sort through the domain of L from biggest to smallest efficiently we can maintain a sorted set that tracks the domain of L. The set B′ is the set of extensions used by the extend function. Note that we must create B′ separately before iterating over it. If we iterated over B only then we might count an event twice as a binding may be added to B′ more than once. We avoid this by letting B′ be a set, thus removing duplicates.

Understanding the check function

The check function in Algorithm 5 is a relatively straightforward implementation of Def. 43 that first computes the domain of quantification and then evaluates the monitor lookup with respect to this domain.

The recursively defined CHECKING function steps through the quantification list building up bindings and checks the configurations associated with them. If the quantifier list is empty then all quantifications have been expanded and the monitor lookup should be inspected. If the quantifier list is not empty then we pick the next quantified variable and step through the possible values for that variable updating our results as we go. This enables us to return early if an early value causes a $\forall$ to become false or an $\exists$ to become true.

Algorithm 4 The update function for the basic algorithm.

```
function UPDATE($\mathbf{a}$ : GEvent)
    B : Set[Binding] ← MATCHING($\mathbf{a}$)
    for $\theta \in \mathsf{dom}(L)$ sorted from biggest to smallest do
        B′ : Set[Binding] ← {}
        for $\theta' \in B$ do
            if $\theta$ is compatible with $\theta'$ then $B' += \theta \dagger \theta'$
        for $\theta_N \in B'$ do
            if $\theta = \theta_N$ or $\theta_N \notin \mathsf{dom}(L)$ then
                $L \leftarrow L \dagger (\theta_N \mapsto \text{NEXT}(\theta_N, \mathbf{a}, L(\theta)))$

function MATCHING($\mathbf{a}$ : GEvent) : Set[Binding]
    B : Set[Binding] ← [ ]
    for $\mathbf{b} \in \mathcal{Q}.\mathcal{E}.\mathcal{A}$ do
        if $\mathsf{matches}(\mathbf{a}, \mathbf{b})$ then
            $\theta \leftarrow \mathsf{quantified}(\mathsf{match}(\mathbf{a}, \mathbf{b}))$
            for $\theta_1 \sqsubseteq \theta$ do
                for $\theta_2 \in B$ do
                    if $\theta_1$ compatible with $\theta_2$ then $B += \theta_1 \dagger \theta_2$
                $B += \theta_1$
    return B
```

5.6.2 Correctness

For this algorithm to be correct it should implement the small-step semantics introduced in Section 5.3. We show this by relating functions of the basic algorithm to definitions of the small-step semantics. Firstly, we show that the expected bindings are generated using the Matching function.

Proposition 1. *When processing event* $\mathbf{a}$ *for each binding* $\theta \in \mathsf{dom}(M)$ *the computed set* B' *is equivalent to the set* $\mathsf{extensions}(\theta, \mathbf{a}) \cup \{\theta \mid \mathsf{relevant}(\theta, \mathbf{a})\}$ *i.e. it adds* θ *to the set of extensions if and only if* θ *is relevant to* $\mathbf{a}$.

The proof is given in Appendix B.3. The next step is to show that the update function of the basic algorithm coincides with that of the small-step semantics..

Proposition 2. *The* Update *function updates the monitor lookup in the same way as the* step *function of Def. 40 i.e. if we let* L_1 *be the monitor lookup prior to a call of* Update($\mathbf{a}$) *and* L_2 *be the monitor lookup after the call then* $L_2 = \mathbf{a} * L_1$.

Proof. Firstly, ordering bindings from largest to smallest is a linearisation of the kind described in Def. 40. Therefore, the Update function iterates over the domain of L_1 in the same way. The actions taken on each step are also the same. In Def. 40 for each entry θ we use next to update the entry for θ if $\mathbf{a}$ is relevant and add new bindings in $\mathsf{extend}(\theta, \mathbf{a}, C)$ if they do not already exist. Recall that extend is defined as

$$\mathsf{extend}(\theta, \mathbf{a}, c) = [\theta' \mapsto \mathsf{next}(\theta', \mathbf{a}, C) \mid \theta' \in \mathsf{extensions}(\theta, \mathbf{a})]$$

Algorithm 5 The check function for the basic algorithm.

```
D : Map[Var,Set[Val]] ← [ ]
function CHECK : Verdict
    D ← [var ↦ {} | var ∈ vars($\mathcal{Q}.\Lambda$) ]
    for $\theta$ ∈ dom(L) do
        for (var ↦ val)∈ $\theta$ do
            D ← D † (var ↦ D(var) ∪ val)
    D ← D † $\mathcal{Q}.\mathcal{D}$                                    ▷ Overwrite with given domain
    return CHECKING($\mathcal{Q}.\Lambda$, [ ])

function CHECKING(qlist : List[({∃, ∀},Guard,Var)], $\theta$ : Binding) : Boolean
    if qlist is empty then
        if $\theta \notin$ dom($L$) then                                  ▷ $\theta$ includes values from $\mathcal{D}$
            return $\mathcal{Q}.\mathcal{E}.q_0 \in \mathcal{Q}.\mathcal{E}.F$
        for $(q, \Gamma)$ ∈ L($\theta$) do
            if $q \in \mathcal{Q}.\mathcal{E}.F$ then return True
        return False
    else
        (quantifier,var,guard)::rest ← qlist
        result ← True if quantifier is ∀ and False otherwise
        for val ∈ D(var) do
            g ← guard($\theta$ † [var ↦ val])
            r ← CHECKING(rest,$\theta$ † [var ↦ val])
            if quantifier is ∀ then
                result ← result ∧ (r ∨¬ g)
                if result is false then return false
            else                                                  ▷ quantifier is ∃
                result ← result ∨ (r ∧ g)
                if result is true then return true
        return result
```

As we previously showed that

$$\mathrm{B}' = \mathsf{extensions}(\theta, \mathbf{a}) \cup \{\theta \mid \mathsf{relevant}(\theta, \mathbf{a})\}$$

we can see that in UPDATE we use NEXT to update the entry for θ if it is relevant to **a** and add any bindings in $\mathsf{extend}(\theta, \mathbf{a}, C)$ that do not already exist.

Finally, in Section 5.3.2 we argued for the equivalence of next and NEXT. □

Next, we show that the CHECK function implements the acceptance judgement for QEA. Note the assumption that type variable adjustment has occurred and therefore, type variables can be ignored.

Proposition 3. *The* CHECK *function produces the same output as the judgement given in Def. 43.*

The proof is given in Appendix B.3. Therefore, we can conclude that a trace is accepted by the small-step semantics if and only if it is also accepted by the basic monitoring algorithm. This can be seen by replacing the appropriate functions of the basic algorithm with the corresponding equivalent definitions.

5.7 Summary

In this chapter we have introduced an algorithm for monitoring QEA and have shown that this preserves the semantics presented in the previous chapter by introducing an equivalent small-step semantics.

Chapter 6

Optimising Monitoring

The last chapter introduced a stepwise monitoring construction for QEA and an associated monitoring algorithm. However, we did not consider efficiency in this presentation. In this chapter we consider improvements to the monitoring algorithm; partly to maximise the utility of the produced result, but mainly to increase the efficiency of monitoring. This chapter explores the following:

Verdicts. We motivate and introduce a fine-grained acceptance criteria utilising a five-valued verdict domain (Sec. 6.1) and present a method for carrying out this checking incrementally (Sec. 6.2).

Global Guards. Global guards can exclude many bindings from consideration, which can significantly reduce the amount of work that needs to be done. We first consider how to filter the bindings created (Sec. 6.3). We then focus on how to optimise checking the previously defined *connectedness* global guard (Sec. 6.3.3).

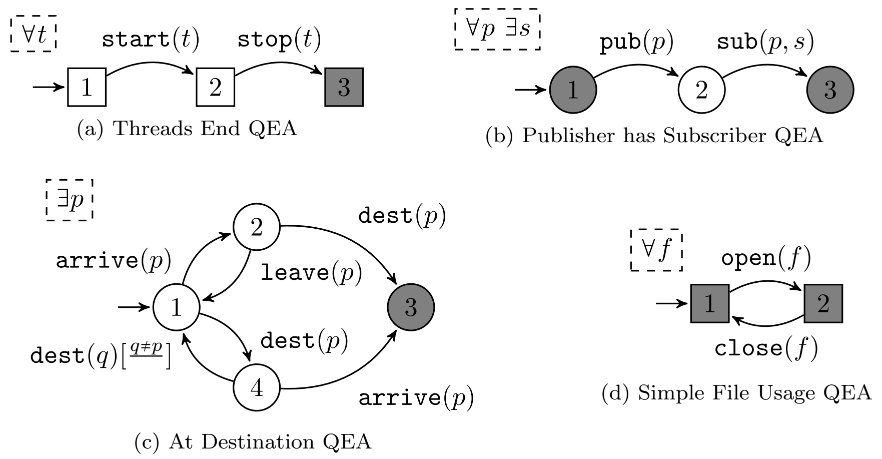
Eliminating Redundancies. We identify a number of redundancies in the basic algorithm and describe methods for eliminating such redundancies (Sec. 6.4).

Indexing. Next we identify an important property of the monitoring process, that only certain bindings are inspected when processing an event. This leads us to develop indexing strategies that look up these bindings directly (Sec. 6.5).

Reference values. Finally we discuss mechanisms introduced to deal with a common issues that arise when monitoring programs with reference objects at runtime, the notions of equality and garbage (Sec. 6.6).

6.1 More informative verdicts

In this section we explore how a five-valued verdict domain can give more informative results than a two-valued verdict domain. Recall our initial discussion about many-valued verdict



(a) Threads End QEA

(b) Publisher has Subscriber QEA

(c) At Destination QEA

(d) Simple File Usage QEA

Figure 6.1: QEAs demonstrating the utility of a four-valued verdict domain.

domains in runtime monitoring in Section 2.3.2. The verdict domain we introduce is not new but we motivate it using concepts from QEA.

6.1.1 Defining the verdicts

Using the four QEAs in Figure 6.1 we address the question *what can be concluded about* past *and* future *behaviour of the system from a finite trace?*

Firstly let us take the QEA in Figure 6.1a which captures how threads should start and stop. The following table gives traces with their verdicts.

Trace	Verdict
`start`(A)	false
`stop`(A).`start`(A)	false
`start`(A).`stop`(A)	true

The first verdict is different from the second verdict as the first indicates that the property *has not yet* been satisfied, whilst the second indicates that the property *can never* be satisfied. We would like to be able to differentiate between these two. Importantly, the second verdict violates the property with a finite trace.

Next consider the QEA $\mathcal{PS}$ in Figure 6.1b capturing the property that every publisher has at least one subscriber. Given a trace τ that satisfies $\mathcal{PS}$ we can always construct a trace $\tau.\mathbf{a}$ that does not by letting **a** be an event introducing a new publisher. Similarly given a trace τ that does not satisfy $\mathcal{PS}$ we can always construct a trace $\tau.\sigma$ that does by letting σ be a list of subscribe events for each observed publisher. We can see that no finite trace can validate or violate this property but we can always tell if the current trace (the events received so far) satisfy the property or not.

The QEA in Figure 6.1c captures the property there is a location p that we have arrived at and is the desired destination. Consider the trace $\texttt{dest}(A).\texttt{arrive}(B).\texttt{dest}(B)$. After the first two events the verdict is false but after the third event the verdict is true and can never become false again. In fact, if the verdict *ever* becomes true it will remain true for any subsequent events. Finally, if the QEA in Figure 6.1d, capturing proper file usage, ever reaches the implicit failure state for any file it will remain there for all subsequent events and will return the false verdict from then on.

The following table summarises the possible verdicts returned by each of these QEAs divided into two different interpretations.

	Possible verdicts			
	true		false	
QEA	not yet false	cannot be false	not yet true	cannot be true
Fig. 6.1a	✓		✓	✓
Fig. 6.1b	✓		✓	
Fig. 6.1c		✓	✓	
Fig. 6.1d	✓			✓

To capture these four different possible verdicts we introduce a five-valued verdict domain as used before in the RULER tool [BRH08]. We include an unknown verdict for use where the status of the current trace is unknown. This cannot happen in our current approach, but we keep the option for possible extensions. For example, if we were to use asynchronous monitoring (where a result is not returned immediately) then we might return unknown as the unprocessed event could potentially alter the verdict.

Definition 47 (Five Value Verdict Domain)**.** *Let* $\mathbb{B}_5 = \{\top_W, \bot_W, \top_S, \bot_S, ?\}$.

The verdicts in $\mathbb{B}_5$ have the following intuitions and names. We use the term *weak* to mean that the verdict could change in the future and call a verdict *ultimate* or *strong* if it cannot change.

	Current Trace	Future Extensions	Name
$\top_W$	holds	unknown	weak success
$\bot_W$	does not hold	unknown	weak failure
$\top_S$	holds	holds	(strong) success
$\bot_S$	does not hold	does not hold	(strong) failure
?	unknown	unknown	unknown

Strong verdicts imply their weak versions, giving a partial order on verdicts. We can also relate these verdicts to the notions of safety and co-safety (Sec. 4.1.3), a safety property can lead to $\bot_S$ or $\top_W$, whereas a co-safety property can lead to $\top_S$ or $\bot_W$.

6.1.2 Producing verdicts

We now consider how these verdicts can be produced by introducing a Check function for monitor lookups. The first step is introducing the concept of *strong states*. These are states from which you cannot reach a state with a different acceptance condition.

Definition 48 (Strong Success and Failure States)**.** *Given EA* $\mathcal{E} = \langle Q, \mathcal{A}, \delta, q_0, F\rangle$*, let* $\texttt{reach}(q)$ *be the set of reachable states of* $q \in Q$ *using* δ*. Define*

$$\begin{aligned} \mathsf{strongS} &= \{q \in F \mid \texttt{reach}(q) \subseteq F\} \\ \mathsf{strongF} &= \{q \in Q \backslash F \mid \texttt{reach}(q) \cap F = \emptyset\} \end{aligned}$$

Note that it is not necessarily the case that ($\texttt{strongS} \cup \texttt{strongF}$) = Q.

State p is *reachable* from state q if there is a finite sequence of transitions starting at q and ending at p. For example, let us identify the strong states in the QEAs in Fig. 6.1 where we use $q_\perp$ as the implicit failure state introduced for next states (note that this is always a strong failure state).

	Fig. 6.1a	Fig. 6.1b	Fig. 6.1c	Fig. 6.1d
strongS	$\{\}$	$\{3\}$	$\{3\}$	$\{\}$
strongF	$\{q_\perp\}$	$\{\}$	$\{\}$	$\{q_\perp\}$

We now update our notion of monitor lookup classification (previously Def. 43) to make use of these strong states to give one of the four new verdicts. We write $\mathsf{pure}(\Lambda, Q)$ if quantifier list Λ only uses the quantifier Q and $\mathsf{pure}(\Lambda)$ if Q does not matter.

Definition 49 (Monitor Lookup Classification)**.** *Let G be the combination of all guards in $\mathcal{Q}.\Lambda$, i.e, $G(\theta)$ iff $\forall(\_,\_,g) \in \Lambda : g(\theta)$. Then $\texttt{Check}(L, \mathcal{Q})$ is defined as*

$$\begin{array}{lll}
\top_S & \textit{iff} & \mathsf{pure}(\Lambda, \exists) \wedge \exists\theta \in \mathsf{dom}(L) : G(\theta) \wedge \exists\langle q, \varphi\rangle \in L(\theta) : q \in \texttt{StrongS} \\
\bot_S & \textit{iff} & \mathsf{pure}(\Lambda, \forall) \wedge \exists\theta \in \mathsf{dom}(L) : G(\theta) \wedge \forall\langle q, \varphi\rangle \in L(\theta) : q \in \texttt{StrongF} \\
\top_W & \textit{iff} & \textit{not a strong result and } L \models \mathcal{Q}.\Lambda \\
\bot_W & \textit{iff} & \textit{not a strong result and } \textit{ and } L \not\models \mathcal{Q}.\Lambda
\end{array}$$

where $L \models \mathcal{Q}.\Lambda$ is as given in Def. 43.

6.1.3 A brief example

Let us demonstrate the verdicts returned for the QEA $\mathcal{Q}$ in Fig. 6.1a when processing τ = $\texttt{start}(1).\texttt{stop}(1).\texttt{start}(2).\texttt{start}(2)$. The monitor lookups and verdicts are given below. The first event gives weak failure as we have not yet satisfied the property. After the second event we have weak success as the property is currently true but it may not be in the future. The third event introduces a new thread that is in a non-final state and the property is not satisfied again. The final event ultimately violates the property and monitoring can stop as no further events can change the verdict.

$$\begin{array}{lll}
L_{\tau_1} = \left[\begin{array}{lll} [t \mapsto 1] & \mapsto & \langle 2, [\ ]\rangle \end{array}\right] & & \mathsf{Check}(L_{\tau_1}, \mathcal{Q}) = \bot_W \\
L_{\tau_2} = \left[\begin{array}{lll} [t \mapsto 1] & \mapsto & \langle 3, [\ ]\rangle \end{array}\right] & & \mathsf{Check}(L_{\tau_2}, \mathcal{Q}) = \top_W \\
L_{\tau_3} = \left[\begin{array}{lll} [t \mapsto 1] & \mapsto & \langle 2, [\ ]\rangle \\ [t \mapsto 2] & \mapsto & \langle 2, [\ ]\rangle \end{array}\right] & & \mathsf{Check}(L_{\tau_3}, \mathcal{Q}) = \bot_W \\
L_{\tau_4} = \left[\begin{array}{lll} [t \mapsto 1] & \mapsto & \langle 2, [\ ]\rangle \\ [t \mapsto 2] & \mapsto & \langle q_\perp, [\ ]\rangle \end{array}\right] & & \mathsf{Check}(L_{\tau_4}, \mathcal{Q}) = \bot_S
\end{array}$$

6.1.4 Properties of strong states

Let us consider the properties of strong states. We first show that all states reachable from a strong state are themselves strong.

Lemma 21. *For EA $\mathcal{E}$*

$$\forall q \in \texttt{StrongS} : \texttt{reach}(q) \subseteq \texttt{StrongS} \tag{6.1}$$

$$\forall q \in \texttt{StrongF} : \texttt{reach}(q) \subseteq \texttt{StrongF} \tag{6.2}$$

Proof. Let us consider (6.1) first. Assume there is a $q \in \texttt{StrongS}$ such that $\texttt{reach}(q) \nsubseteq \texttt{StrongS}$. There must therefore exist a q' and q'' such that $q' \in \texttt{reach}(q)$, $q'' \in \texttt{reach}(q')$ and $q'' \notin \mathrm{F}$ but due to the transitivity of $\texttt{reach}$ we know that $q'' \in \texttt{reach}(q)$ and as $q'' \notin \mathrm{F}$ the state q cannot be a strong success state. Therefore, such a state q cannot exist. Our argument for (6.2) is symmetrical. □

Now let us show that the act of collating guards together in Def. 49 is sound

Lemma 22. *It is sound to write a pure quantifier list $\Lambda = Qx_1 : g_1 \dots Qx : n : g_n$ ($Q = \forall$ or $Q = \exists$) as $\Lambda' = Qx_1 : True \dots Qx_n : g_1 \wedge \dots \wedge g_n$ i.e. replace all global guards except the last with the true guard and the last guard with all guards conjoined.*

Proof. We show that for all traces and QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$.

$$\tau \models^{\mathcal{E},\mathcal{D}} \Lambda \Leftrightarrow \tau \models^{\mathcal{E},\mathcal{D}} \Lambda' \tag{6.3}$$

As Λ is pure the evaluation of $\tau \models^{\mathcal{E},\mathcal{D}} \Lambda$ is the equivalent of collecting the relevant bindings (Def. 30) and either checking if all bindings hold (if $Q = \forall$) or if at least one holds (if $Q = \exists$). Global guards restrict the set of relevant bindings as described in Def. 30. Every binding in $\mathsf{relevant}(\tau, \Lambda)$ must satisfy all guards $g_1, \dots g_n$ to be in $\mathsf{relevant}(\tau, \Lambda)$ and therefore $\mathsf{relevant}(\tau, \Lambda) = \mathsf{relevant}(\tau, \Lambda')$ and equation (6.3) holds. □

Now we show that only pure quantification lists can lead to an ultimate result.

Lemma 23. *If a quantifier list is not pure then we cannot have ultimate success or failure i.e. for any QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ and any trace τ:*

$$\neg\mathsf{pure}(\Lambda) \Rightarrow \exists \tau' : (L_\tau \models \Lambda) \neq (L_{\tau\tau'} \models \Lambda)$$

i.e. if the quantifier list is not pure there always exists a trace extension that gives a different verdict. (We assume a non-trivial $\mathcal{E}$ i.e. $\exists \tau_1, \tau_2 . \tau_1 \in \mathcal{L}(\mathcal{E}) \wedge \tau_2 \notin \mathcal{L}(\mathcal{E})$).

Proof. Intuitively this holds as universal quantification can be made false by adding to the domain and existential quantification can be made true by adding to the domain.

We show that given τ we can always construct a trace τ'. There are two cases: if τ is accepted or not. Firstly assume that $L_\tau \models \Lambda$ let τ' be a minimal trace containing only new values such that $\exists \theta \in \mathsf{relevant}(\tau', \Lambda) : \tau' \notin \mathcal{L}(\mathcal{E}(\theta))$ i.e. it reaches a non-accepting trace in a new instantiation of $\mathcal{E}$. Such a trace will exist unless $\mathcal{E}$ is trivial. Then as Λ contains at least one universal quantification and τ' necessarily extends the domain of the universally quantified variable we have that $L_{\tau.\tau'} \not\models \Lambda$. A symmetric case holds for $L_\tau \not\models \Lambda$ i.e. we construct τ' as a minimal trace containing new values such that enough new total bindings are introduced in final states. □

Next we show that entries in a monitor lookup L_τ in a strong state will remain in that strong state for all extensions of τ.

Lemma 24. *For QEA $\langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ and traces τ and τ'*

$$\forall \theta \in \mathsf{dom}(L_\tau) : \exists \langle q, \varphi \rangle \in L_\tau(\theta) : q \in \mathtt{StrongS} \Rightarrow \exists \langle q, \varphi \rangle \in L_{\tau\tau'}(\theta) : q \in \mathtt{StrongS} \tag{6.4}$$

$$\forall \theta \in \mathsf{dom}(L_\tau) : \exists \langle q, \varphi \rangle \in L_\tau(\theta) : q \in \mathtt{StrongF} \Rightarrow \exists \langle q, \varphi \rangle \in L_{\tau\tau'}(\theta) : q \in \mathtt{StrongF} \tag{6.5}$$

Proof. By Lemma 21 all states reachable from strong states are themselves strong success states. By Lemmas 10 and 18 for a given $\theta \in \mathsf{dom}(L_\tau)$

$$\forall c \in L_\tau(\theta), \forall c' \in L_{\tau.\tau'}(\theta) : c \overset{\tau\downarrow_{\mathsf{partial\_ground}(\mathcal{E},\theta)}}{\rightsquigarrow}_{\mathcal{E}(\theta)} c'$$

and as $\rightsquigarrow$ is based on reachability in δ, if c is in a strong state then c' must be in the same kind of strong state.

□

We now consider the relation between this new and old acceptance conditions for the small-step semantics. Let us call the previous acceptance condition the 2-verdict condition, and this new acceptance condition the 4-verdict condition.

A trivial result is that a weak result returned by the 4-verdict condition implies the corresponding result from the 2-verdict condition.

Corollary 1 (Weak Results). *For all QEA $\mathsf{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ and all traces τ*

$$\mathtt{Check}(L_\tau, \mathsf{Q}) = \top_W \Rightarrow L_\tau \vDash \Lambda \tag{6.6}$$

$$\mathtt{Check}(L_\tau, \mathsf{Q}) = \bot_W \Rightarrow L_\tau \nvDash \Lambda \tag{6.7}$$

Furthermore, for all non-pure Λ this is an equivalence.

Proof. By the definition of Check (Def. 49) and the fact that $\top_S \Rightarrow \top_W$ and $\bot_S \Rightarrow \bot_W$. □

The claims we can make about strong results are more complex; it is not the case that strong success in the 4-verdict condition necessarily implies success in the 2-verdict condition directly. Instead, strong success in the 4-verdict condition implies that all possible (potentially empty) extensions of the trace which lead to the creation of a total binding satisfy the 2-verdict condition. A similar claim can be made for strong failure. This means that the 4-verdict condition will detect strong success (or failure) in certain conditions earlier than the 2-verdict condition, we give an example of this shortly.

Theorem 2 (Strong Results). *For all QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ and all traces τ:*

$$\mathtt{Check}(L_\tau, \mathcal{Q}) = \top_S \Leftrightarrow \forall \tau' \in \mathit{Trace} : \mathsf{relevant}(\tau\tau', \Lambda) \neq \varnothing \Rightarrow L_{\tau\tau'} \vDash \Lambda \tag{6.8}$$

$$\mathtt{Check}(L_\tau, \mathcal{Q}) = \bot_S \Leftrightarrow \forall \tau' \in \mathit{Trace} : \mathsf{relevant}(\tau\tau', \Lambda) \neq \varnothing \Rightarrow L_{\tau\tau'} \nvDash \Lambda \tag{6.9}$$

Proof. As the cases are symmetric let us consider case (6.8) only.

Let us consider the $\Rightarrow$ direction first. From the definition of Check we know that Λ is existentially pure and there exists a binding θ in $\mathsf{dom}(L_\tau)$ such that $G(\tau)$ and $\exists\langle q,\varphi\rangle \in L_\tau(\theta)$: $q \in \mathsf{StrongS}$. Let us consider an extension τ'. If $\mathsf{relevant}(\tau.\tau',\Lambda) = \varnothing$ then the right-hand side holds trivially. If $\mathsf{relevant}(\tau\tau',\Lambda) \neq \varnothing$ then either θ is relevant or there is some relevant binding θ' such that $\theta \sqsubseteq \theta'$ as $\mathsf{dom}(L_{\tau.\tau'})$ is lub-closed. In either case we know that, by Lemma 24 there exists a configuration in $L_{\tau\tau'}(\theta)$ or $L_{\tau\tau}(\theta')$ with a strong success state. Therefore, $L_{\tau\tau'} \vDash \Lambda$.

Let us now consider the $\Leftarrow$ direction. Firstly, by Lemma 23 we know that Λ is pure, otherwise we would be able to find a non-accepting extension. Furthermore, from the proof of Lemma 23, we can see that Λ must be existentially pure as we are considering acceptance. Additionally, there must exist a binding in L_τ with a strong state, otherwise there would be an extension of τ that would be non-accepting, this follows from Lemma 24. Therefore, $\mathsf{Check}(L_\tau,\mathcal{Q})$.

□

Therefore, the 4-verdict acceptance condition gives a more refined verdict than the 2-verdict acceptance condition.

An example of where the 2 and 4 verdict conditions diverge

Let us consider an example where the 4-verdict condition returns an ultimate verdict contrary to the 2-verdict condition. As we can see from above, this only occurs when there are no relevant bindings (i.e. no total bindings have been constructed). Note that this is a peculiar edge case.

Let us consider the QEA $\mathcal{EP}$ introduced in Fig. 3.7(left) on page 71 and the monitoring of the following trace:

$$\tau = \texttt{create}(G).\texttt{create}(G).\texttt{make}(G,E).\texttt{add}(E,U).\texttt{close}(E).\texttt{destroy}(G)$$

This trace is not accepted as the group G is created twice and, as expected $L_\tau \nvDash \forall g\forall e\forall u$. However, if we consider the subtrace $\tau' = \texttt{create}(G).\texttt{create}(G)$ we have a monitor lookup of

$$L_{\tau'} = \begin{bmatrix} [\,] & \mapsto & \{\langle 1,[\,]\rangle\} \\ [g \mapsto G] & \mapsto & \{\langle q_\perp,[\,]\rangle\} \end{bmatrix}$$

where $q_\perp$ is the implicit failure state (in StrongF). Therefore $\mathsf{Check}(\tau',\mathcal{EP}) = \bot_S$ but as there are no relevant bindings and empty universal quantification is true we have $\tau' \vDash \Lambda$. The 2-verdict condition would not detect a problem until the next two events occur and a relevant binding is constructed. This does not mean that the 2-verdict condition is incorrect, but demonstrates why we need a non-empty set of relevant bindings for the two conditions to agree.

6.2 Incremental Checking

In this section we present a method for incremental checking that introduces and maintains a data structure to track the projection acceptance of total bindings. We note that this checking

Algorithm 6 Additions to initialisation.

```
$strong\_reached \leftarrow false$
$\Gamma \leftarrow Node$ of $q_0 \in F$ and [ ] and 0
```

procedure is required because of quantifier alternation and we discuss the special case of pure quantification at the end of this section.

6.2.1 Checking data structure

We introduce a tree data structure to store the total bindings of a monitor lookup in quantifier-list order. Each node stores the current verdict of the subtree below it so that when a single binding is updated only a small portion of the tree is updated.

A checking structure is a tree with nodes labelled with boolean values and integers and edges labelled with values taken from *Val*.

Definition 50 (Checking Structure).

$$\begin{aligned} \mathit{CheckingStructure} &= \mathit{Node}\ \ \mathit{of}\, \mathbb{B} \times \mathit{Map}[\mathit{Val}, \mathit{Node}] \times \mathit{Integer} \\ &= \mathit{Leaf}\ \ \mathit{of}\, \mathbb{B} \end{aligned}$$

A checking structure for a quantifier list Λ has at most depth $|\Lambda|$ and level i is implicitly labelled with the ith quantified variable. The value of a checking structure is its boolean value, written $\mathsf{value}(\Gamma)$ for checking structure Γ. The integer at each node counts the number of non-accepting values in its map and is accessed using $\mathsf{count}(\Gamma)$.

As outlined in Algorithm 6, during initialization we construct an empty checking structure and a boolean variable indicating whether a strong state has been reached.

On each step, we update the checking structure so that the current status of the trace is given by the boolean value of the root node. Algorithm 7 gives the procedure for updating the checking structure, which should be called every time we update the states associated with a total binding. To achieve this we include the statement $\Gamma \leftarrow \text{UPDATE\_CHECK}(\Gamma, 0, \theta, S)$ at the appropriate point in the monitoring algorithm.

The UPDATE_CHECK function constructs a new checking structure by recursively updating the branch consistent with the given binding. For the base case, we create a leaf with the acceptance of the set of configurations. At each level we update the *count* variable to ensure it reflects the number of non-accepting branches at that node.

At the base level we also check strong states. For the purposes of this algorithm we collect all strong success and failure states together in a set Strong and if a strong state is encountered we update the *strong_reached* variable appropriately. Note how global guards are dealt with; if a global guard is false then we trim the tree at that point, effectively removing it from the set of bindings considered.

This leads to an updated CHECK function as given in Algorithm 8 which uses the root of the checking structure and the *strong_reached* variable to return one of four verdicts. If the checking structure has not been updated then no total bindings have been added and the verdict for empty domains is given.

Algorithm 7 The checking structure update function.

```
function UPDATE_CHECK($\Gamma$,$i$,$\theta$, $S$) : $\mathit{CheckingStructure}$
    $(Q, x, g, \_) \leftarrow \Lambda(i)$
    if $i = |\Lambda|$ then
        if $(S \cap \mathsf{Strong}) \neq \varnothing$ then
            $\mathit{strong\_reached} \leftarrow \mathit{true}$
        return $\mathit{Leaf}$ of $(F \cap S) \neq \varnothing$
    if $\neg g(\theta)$ then                                        ▷ Trim the tree here
        return $\Gamma$
    $\mathit{Node}$ of $b$ and $M$ and $\mathit{count} \leftarrow \Gamma$                        ▷ Cannot be a leaf
    $\Gamma' \leftarrow \begin{cases} M(\theta(x)) & \text{if } \theta(x) \in \mathsf{dom}(M) \\ \mathit{Node} \text{ of } \mathit{false} \text{ and } [\ ] \text{ and } 0 & \text{if } \theta(x) \notin \mathsf{dom}(M) \wedge i < |\Lambda| \\ \mathit{Leaf} \text{ of } (q_0 \in F) & \text{otherwise} \end{cases}$
    $\mathit{prev} \leftarrow \mathsf{value}(\Gamma')$
    if $\theta(x) \notin M$ and $q_0 \notin F$ then count+=1
    $\Gamma' \leftarrow$ UPDATE$(\Gamma', i+1, \theta, S)$
    $M \leftarrow M \dagger [\theta(x) \mapsto \Gamma']$
    $\mathit{count} \leftarrow \begin{cases} \mathit{count}+1 & \text{if } \mathit{prev} \wedge \neg\mathsf{value}(\Gamma') \\ \mathit{count}-1 & \text{if } \neg\mathit{prev} \wedge \mathsf{value}(\Gamma') \\ \mathit{count} & \text{otherwise} \end{cases}$
    if $(Q = \forall \wedge \mathit{count} > 0) \vee (Q = \exists \wedge \mathit{count} = |\Gamma|)$ then
        return $\mathit{Node}$ of $\mathit{false}$ and $M$ and $\mathit{count}$
    else
        return $\mathit{Node}$ of $\mathit{true}$ and $M$ and $\mathit{count}$
```

Algorithm 8 The incremental check function.

```
function CHECK : Verdict
    if $\Gamma$ has not been updated then return $\begin{cases} q_0 \in F & \text{if } \Lambda = \epsilon \\ \bot_W & \text{if } \Lambda \text{ begins with } \exists \\ \top_W & \text{otherwise} \end{cases}$
    if $\mathit{strong\_reached}$ then
        if all quantifiers are $\forall$ and $\neg\mathsf{value}(\Gamma)$ then return $\bot_S$
        if all quantifiers are $\exists$ and $\mathsf{value}(\Gamma)$ then return $\top_S$
    if $\mathsf{value}(\Gamma)$ then return $\top_W$
    else return $\bot_W$
```

An example of this algorithm is given in Appendix A.2.4.

Dealing with given domain

If we have a given domain we need to include this information in the checking structure to ensure that the relevant total bindings are constructed. To achieve this, we initialise the monitor lookup with the given domain and add any resulting total bindings to the checking structure. This ensures that the necessary total bindings will be created when updating the monitor lookup. To initialise the monitor lookup we add all bindings of the form $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ where $x_i \in \mathsf{vars\_of}(X)$ and $v_i \in \mathcal{D}(X)$.

6.2.2 Special case

There are three special cases. The first two are when we have zero or one quantifier and are discussed later in Sec. 6.5. The third is where the quantifier list is *pure*. Instead of the complex checking structure we only need to maintain counts of accepting and non-accepting bindings, or a flag indicating the previous status that could be updated when this changes, as is done in JavaMOP. Whilst the majority of properties in this thesis use pure quantification, and can use this special case, it is important that we introduce a reasonably efficient method for checking in the general case.

6.2.3 Analysis

We consider the *correctness* and *time complexity* of this incremental algorithm.

Correctness

We can think of this incremental approach as keeping in memory the evaluation structure of the acceptance relation; when we evaluate $L_\tau \vDash \Lambda$ we implicitly explore this tree structure. Therefore, we can map each step of this incremental approach back to the evaluation of $\vDash$ and we produce the same results. Note that the treatment of global guards is equivalent as in $\vDash$ we exclude from consideration all bindings where the associated global guard fails to hold.

Complexity

Previously to check a verdict it was necessary to compute the domains of quantified variables, build possible bindings and then check them until we had a result. Here we store much of this computation from step to step. The time complexity of the previous approach was

$$O(|L_\tau| + \prod_{i=0}^{i=|\Lambda|} |D_{L_\tau}(\Lambda(i))|)$$

as we needed to process the monitor lookup and then step through our implicit checking tree. In this approach we have $|\Lambda|$ calls to the UPDATE function, each of which carries out a constant number of operations. Therefore, complexity is $O(|\Lambda|)$. and we have vastly increased the efficiency of our approach. However, we should note that the $|L_\tau|$ element of the previous approach can be easily removed by storing the domain separately.

Let us consider the additional space requirements of this approach. We are now storing a checking structure so we will place an upper bound on the size of this structure. Note that a checking structure has a fixed height of $|\Lambda|$ and its branching at level i is given by the size of the domain of x_i i.e. the ith quantified variable. Let d_i by the size of the domain of quantified variable x_i. Without global guard trimming the checking structure will be

$$1 + d_1 + d_1(d_2 + d_2(d_3 + d_3(\dots d_n)))$$

which is potentially exponential in the length of Λ.

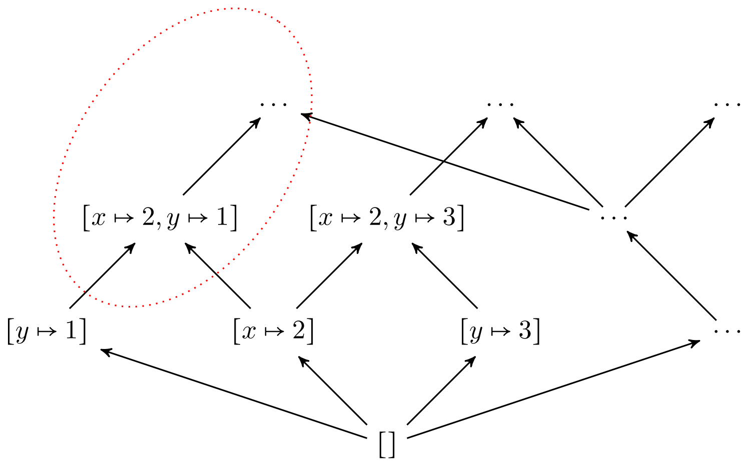


Figure 6.2: Global guards can be used to trim the search space. This demonstrates the effects of filtering based on the global guard $y > x$. Not only do we trim bindings where $y \leq x$ but also any bindings that might be constructed from them.

6.3 Filtering with global guards

We introduce the mechanisms used to trim the search space when using global guards. We begin by introducing the basic additions used for global guards in general before focussing on connectedness.

6.3.1 Filtering for global guards

Global guards exclude some total bindings from being considered in acceptance. When updating the checking structure (Algorithm 7) we use global guards to trim branches of the structure, but do not consider the bindings generated.

Here we apply the same technique to binding generation. If a newly generated binding does not satisfy the global guards it is not added to the monitor lookup. This can drastically reduce the number of bindings that are generated, as illustrated in Figure 6.2. To deal with partial bindings we only evaluate global guards for the variables that are defined in a binding, therefore given a quantification list $\Lambda = Q_1 x_1.g_1 : X_1 \dots Q_n x_n.g_n : X_n$ we define the *composite* global guard:

$$G^{\Lambda}(\theta) = \forall i.1 \leq i \leq n \Rightarrow g_i(\theta) \vee (\exists j.j < i \wedge x_j \notin \mathsf{dom}(\theta))$$

The global guard filtering extension then adds the check $G^{\Lambda}(\theta)$ to the addition of bindings to the monitor lookup. We show that this approach preserves the verdicts returned .

Proposition 4. *The global guard filtering extension preserves verdicts, i.e. for a given trace and QEA the same verdict is given with and without the extension.*

Proof. As we demonstrated in Section 6.2.3, bindings that do not pass global guards are ignored when deciding acceptance, and therefore removing them from the monitor lookup has no effect. Note that we cannot build a binding that passes global guards from one that does not. □

Let us consider the effect this filtering can have on the number of bindings generated given

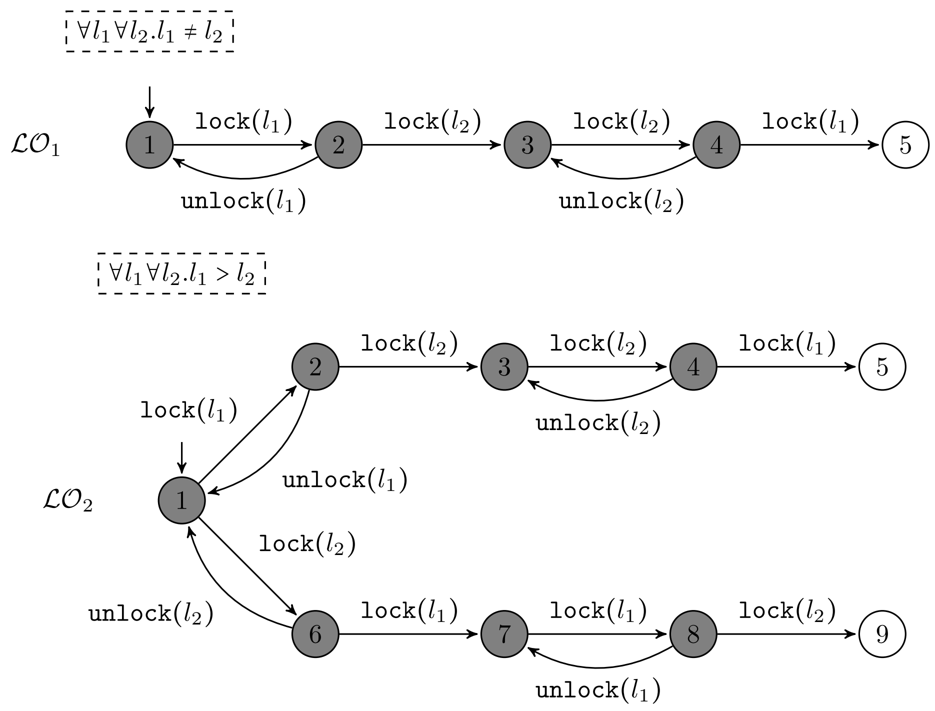


Figure 6.3: Two QEAs for the lock ordering problem.

a trace τ and a QEA $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$. Let $x^d = |(\mathsf{Dom}(\tau)\dagger\mathcal{D})(x)|$ be the size of the domain of x. Let $\Lambda = Q_1 x_1.g_1 : X_1 \ldots Q_n x_n.g_n : X_n$. Let e_i be the proportion of values in the domain of x_i excluded by global guards. The number of bindings generated in total (i.e. the size of $\mathsf{dom}(L_\tau)$) and the number of bindings generated with the global guard filtering extension are then give as

$$1 + \sum_{X \subseteq \mathsf{vars}(\Lambda)} \left(\prod_{x \in X} x_d \right) \qquad\qquad 1 + \sum_{X \subset \mathsf{vars}(\Lambda)} \left(\prod_{x \in X} x_d - e_i \right)$$

respectively. Therefore, reducing the size of the domain of a variable has a polynomial effect on the number of bindings produced.

For example, consider the quantification list $\forall x.x > 0 \forall y.y > x$. Given domains $[x \mapsto \{-99, \ldots, 100\}, y \mapsto \{1, \ldots, 100\}]$ the number of generated bindings would be $(200 * 100) + 200 + 100 + 1 = 20,301$. The number of bindings generated with the global guard filtering extension would be $\frac{100*101}{2} + 100 + 100 + 1 = 5251$.

As a second example consider the quantification list $\forall l_1 \forall l_2 : l_1 > l_2$ where the domains of l_1 and l_2 are the natural numbers up to n. The number of bindings excluded by $l_1 > l_2$ is $n^2 - \frac{n(n-1)}{2}$ and therefore the total number of bindings is $1 + 2n + \frac{n(n-1)}{2}$, compared to $1 + 2n + n^2$ without the global guard filtering.

6.3.2 Evaluating the global guard filter

Here we briefly demonstrate how trimming the search space by filtering using a global guard can improve performance. We consider the lock ordering property given first on page 346 and

Table 6.1: The results of evaluating the global filtering extension using the LockOrdering example.

Locks	Time (seconds)		Configurations		$\|\tau\|$	Time speedup	Configuration decrease
	$\mathcal{LO}_1$	$\mathcal{LO}_2$	$\mathcal{LO}_1$	$\mathcal{LO}_2$			
15	0.35	0.22	341	136	310	1.60	2.50
30	10.6	6.2	931	496	2220	1.70	1.87
45	88.2	52.5	2071	1081	7230	1.67	1.91
60	420.3	231.8	3661	1891	16840	1.81	1.94
75	1255.9	692.6	5701	2926	32550	1.81	1.95
90	2872.2	1561.2	8191	4186	55860	1.84	1.96

reproduced in Fig. 6.3. This figure also gives an equivalent alternative formulation of the lock ordering problem, assuming some ordering on locks. The second QEA works by selecting only one instance of any pair of locks and considers the case where either lock is taken first.

To evaluate the global filtering extension we include this filtering technique in the basic monitoring algorithm. We generate a set of traces of the following form for a given i:

$$\begin{aligned}
\tau &= \texttt{lock}(1).\tau_1.\texttt{unlock}(1).\ldots.\texttt{lock}(i).\tau_1.\texttt{unlock}(i)\\
\tau_1 &= \texttt{lock}(i).\tau_2.\texttt{unlock}(i).\ldots.\texttt{lock}(2i).\tau_2.\texttt{unlock}(2i)\\
\tau_2 &= \texttt{lock}(2i+1).\texttt{unlock}(2i+1).\ldots.\texttt{lock}(3i).\texttt{unlock}(3i)
\end{aligned}$$

This generates a correct trace with $3i$ locks with a nesting of 3 deep. This reflects what might occur when iterating over a data structure where each node has an individual lock.

Table 6.1 presents the results. In this case, global filtering allows us to almost half the size of the search space, and therefore almost half the running times. This demonstrates an obvious correlation between the number of bindings produced and the running time.

6.3.3 Special case: Connectedness

We now focus on connectedness as an example of a complex global guard that requires special machinery. Connectedness was explained in Section 3.5.8. To recap, this guard only accepts 'connected' bindings, and a binding is connected if there is a set of events in the trace whose extracted bindings have non-empty intersections and connect together to form the binding.

The connectedness guard represents a special set of global guards that we will call *trace-predicate* global guards as they are predicated on the *whole* trace. These global guards need to be treated differently from standard global guards as they cannot necessarily be used to trim the search space as trace extensions might cause the global guard to become true. We might be able to analyse a trace-predicate global guard to find instances where no trace extensions can make it true, and therefore trim the search space, but this is not the case with connectedness, so we do not consider it here.

To apply the connectedness filter we maintain a special data structure that records the connectedness status of generated bindings. This is then used as a global guard when deciding acceptance. As the status of this guard can change from step to step we must keep track of which parts of the checking structure are effected.

Data structure

To track connectedness information at runtime we use a *union-find data structure* [CLRS01]. Let us call a set of bindings a *partition*. A union-find datastructure is a set of partitions implemented such that two operations are efficient: the union of two partitions and finding whether two bindings are in the same partition. Here, two bindings are in the same partition if they are connected. We include a third operation add, which adds a new partition containing a binding.

When we observe a new event $\mathbf{a}$ we generate the set of bindings $B = \{\theta \mid \exists \mathbf{b} \in \mathcal{A} : \theta \sqsubseteq \mathsf{match}(\mathbf{a}, \mathbf{b}) \wedge \theta \neq []\}$, add all bindings in B to the data structure, and then for all pairs of bindings b_1 and b_2 in B we call $\mathsf{union}(b_1, b_2)$; recording that these bindings are connected and causing the new bindings to become connected to any existing partitions they intersect with. Note that b_1 and b_2 may already occur in the data structure and the point of this step is to connect any bindings that are already connected to them.

As an example consider the alphabet $\{\mathtt{f}(w,x), \mathtt{g}(y,z), \mathtt{h}(w,z)\}$ and trace $\tau = \mathtt{f}(1,2).\mathtt{g}(3,4).\mathtt{h}(1,4)$. After receiving $\mathtt{f}(1,2)$ we add $[w \mapsto 1]$, $[x \mapsto 2]$ and $[w \mapsto 1, x \mapsto 2]$ and union their partition together to get:

$$\{\{[w \mapsto 1], [x \mapsto 2], [w \mapsto 1, x \mapsto 2]\}\}$$

We repeat the same process with $\mathtt{g}(3,4)$, leading to two partitions:

$$\{\{[w \mapsto 1], [x \mapsto 2], [w \mapsto 1, x \mapsto 2]\}, \{[y \mapsto 3], [z \mapsto 4], [y \mapsto 3, z \mapsto 4]\}\}$$

When we receiving $\mathtt{h}(1,4)$ we carry out the same process but this time the two partitions collapse as we call $\mathsf{union}([w \mapsto 1], [z \mapsto 3])$:

$$\{\{[w \mapsto 1], [x \mapsto 2], [w \mapsto 1, x \mapsto 2], [y \mapsto 3], [z \mapsto 4], [y \mapsto 3, z \mapsto 4]\}\}$$

Acceptance

Bindings begin unconnected and can become connected, after which they will always be connected. Therefore, we can begin by assuming that all total bindings in the checking structure are unconnected and therefore ignored. A binding is connected if it can be built entirely from bindings in a partition of the data structure. Therefore, whenever we update a partition in the data structure we compute all total bindings that can be built from bindings in that partition, and update the status of all such total bindings in the checking structure. Note that we still need to track non-connected bindings in the monitor lookup as they may later become connected. Therefore, the connectedness global guard does not restrict the search space.

6.3.4 Evaluating the connectedness global guard

Let us consider the UnsafeMapIter example introduced in Appendix A.3.2 on page 344 and replicated in Fig. 6.4, along with a version using the connected guard. Consider a trace that has two maps (c_1) and from each produces two collections (c_2), such as sets of values or keys,

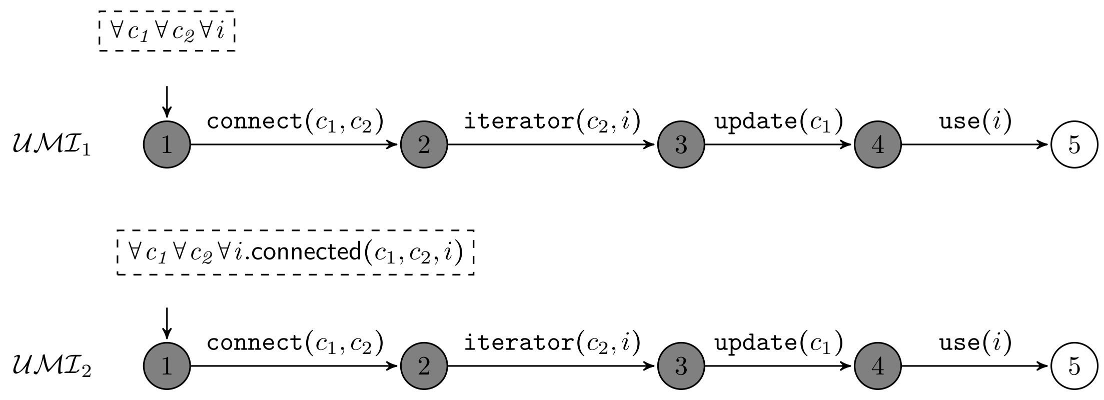


Figure 6.4: Two QEAs for UnsafeMapIter: one using the connected guard and one not.

Table 6.2: Results of connectedness experiment.

c_1	c_2	i	Bindings	Events	Times (millisecs) Standard	Connected	Speedup
1	1	1	8	5	8.3	10.6	0.78
2	2	2	135	24	258.6	244	1.06
3	3	3	1120	69	4850	3664	1.32
4	4	4	5525	152	48874	34456.6	1.42
5	5	5	19656	285	318679.3	217705.7	1.46

and creates 2 iterators (i) for each of these. In total we have two maps, six collections and twelve iterators giving 144 total bindings, but only 12 connected ones.

We briefly evaluate the effects of using the connectedness guard. We do not expect the connectedness guard to significantly effect performance as we have the added cost of computing connectedness, but remove a large number of updates to the checking structure. We create artificial traces which create sequences of collections, sub-collections and iterators and then exercise these. We monitor different lengths of traces with different values using the basic algorithm for both the standard and connected QEAs.

Table 6.2 gives the results of our experiment, with times taken as the average over three runs. It is clear that using the connectedness global guard improves efficiency. This is due to the decrease in bindings that are considered for deciding the verdict. The more bindings we have the greater the speedup, as the percentage of bindings that are non-connected increases.

We carried out this experiment for relatively few collections and iterators as the number of bindings produced increases dramatically, although the majority of these are redundant. We consider redundancy elimination in the next section.

6.4 Redundancy Elimination

In this section we consider the elimination of redundancies in the basic algorithm. We are mainly concerned with bindings that are created or become redundant in the sense that they can be removed without altering the outcome. As the complexity of our approach is polynomial in the number of bindings created, we want to remove any redundant bindings. We begin by introducing a condition on QEAs that allows us to apply later optimisations before considering different binding redundancies.

6.4.1 Normal QEAs

The optimisations given in this section and later (Sec. 6.5) rely on a certain quality of a QEA, which we will call normality. A QEA is normal if we could add or remove bindings with empty projections without changing the outcome.

For example, consider the case where we have a QEA with an alphabet consisting of a single event $\mathtt{f}(x, y)$ and then observe the trace $\mathtt{f}(1,2).\mathtt{f}(1,4).\mathtt{f}(3,4)$. The binding $[x \mapsto 3, y \mapsto 2]$ will have an empty projection. Let us consider the following checking structure where we have two quantifiers for x and y, Q_x and Q_y, and four acceptance values, A, B, C, D, where D is the acceptance value associated with the empty projection.

$$\begin{tikzcd}
& & & Q_x \arrow[dll, "1"'] \arrow[drr, "3"] & & & \\
& Q_y \arrow[dl, "2"'] \arrow[dr, "4"] & & & & Q_y \arrow[dl, "2"'] \arrow[dr, "4"] & \\
A & & B & & C & & D
\end{tikzcd}$$

The truth of this checking structure relies on Q_x and Q_y. Let us consider the effects of the empty projection acceptance value for different values of Q_x and Q_y.

Q_x	Q_y	truth	$q_0 \in F$	$q_0 \notin F$
$\forall$	$\forall$	$(A \wedge B) \wedge (C \wedge D)$	$(A \wedge B) \wedge C$	$(A \wedge B) \wedge \bot$
$\forall$	$\exists$	$(A \vee B) \wedge (C \vee D)$	$(A \vee B) \wedge \top$	$(A \vee B) \wedge C$
$\exists$	$\forall$	$(A \wedge B) \vee (C \wedge D)$	$(A \wedge B) \vee C$	$(A \wedge B) \vee \bot$
$\exists$	$\exists$	$(A \vee B) \vee (C \vee D)$	$(A \vee B) \vee \top$	$(A \vee B) \vee C$

We can see that when $Q_y = \forall$ and $q_0 \in F$ the empty projection verdict can be safely removed and that the same goes for when $Q_y = \exists$ and $q_0 \notin F$. Therefore, under these conditions we can omit total bindings mapped to the empty projection. We use this observation to give our definition of a normal QEA.

Definition 51 (Normal QEA)**.** *A QEA is* normal *if the initial state is final when the innermost quantification is universal and non-final when the innermost quantification is existential. A QEA with no quantifications is always normal.*

Note that all QEA defined so far are normal, except for the ExistsLeader property from the rovers case study first given in Fig. A.36 on page 354.

6.4.2 Empty projection redundancy

This first form of redundancy we consider is where a total binding would be associated with an empty projection. Note that we only consider total bindings as partial bindings with empty projections might be needed to create new bindings as they store information about the domains of quantified variables.

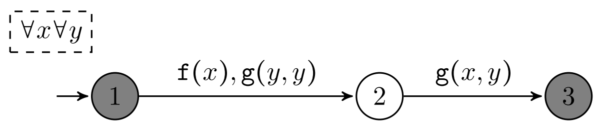


Figure 6.5: A QEA demonstrating why empty redundancy cannot be applied to partial bindings.

We show that for normal QEA's we do not need to store total bindings associated with the initial configuration (i.e. empty projection) in a monitor lookup.

Lemma 25. *Given a monitor lookup L_τ for a normal QEA we have*

$$\mathsf{Check}(L_\tau) = \mathsf{Check}([(\theta \mapsto C) \in L_\tau \mid \neg(\mathsf{total}(\theta) \wedge C = \langle q, [\ ]\rangle)])$$

Proof. Let us consider the checking structure in each case. As the QEA is normal we know that at the bottom level of the checking structure, removing a total binding with an empty projection will not effect the truth of that node (as demonstrated earlier). Therefore, the overall truth of the checking structure will be unaffected. Additionally, due to the QEA being normal it is not possible for the initial state to be a strong state that would lead to a strong verdict. Finally, if an event occurs later to create the total binding with a non-empty projection then not including earlier does not prevent this. □

Therefore, if a QEA is normal then when we do not create total bindings in the monitor lookup or checking structure if they have an empty projection.

Finally, to note whey this optimisation can only apply to total bindings consider the QEA in Fig. 6.5 and the trace $\tau = \mathtt{g}(1,2).\mathtt{f}(1)$. We have two total bindings: $[x \mapsto 1, y \mapsto 1]$ and $[x \mapsto 1, y \mapsto 2]$. However, if we did not record the partial binding $[y \mapsto 1]$ after the first event, which would have an empty projection, we would not create the total binding $[x \mapsto 1, y \mapsto 1]$ and would return the incorrect verdict (weak success rather than weak failure).

6.4.3 Trivial binding redundancy

Let us now extend our previous discussion. We have identified the conditions for excluding total bindings with empty projections but we want to identify a larger set of bindings, i.e. those whose contribution is *trivial*.

Definition 52 (Trivial binding). *Given a QEA $\mathcal{Q}$ with alphabet $\mathcal{A}$, a trace τ and a binding θ, θ is trivial for $\mathcal{Q}$ on τ if $\tau\downarrow_{\mathcal{A}(\theta)}$ does not take any transitions in $\mathcal{Q}$. Let $\mathsf{trivial}^{\mathcal{Q}}_{\tau}(\theta)$ indicate that θ is trivial for $\mathcal{Q}$ on τ where $\mathcal{Q}$ and τ are omitted if obvious from context.*

If a total binding is trivial then it can be omitted, as we saw above. For the case of partial bindings we need to ensure that removing the binding would not mean we fail to create a necessary binding, as we saw in the previous example.

We cannot remove a binding θ if there is an event $\mathbf{a}$ and an event $\mathbf{b} \in \mathcal{A}$ such that θ extended with $\mathsf{match}(\mathbf{a}, \mathbf{b})$ is non-trivial and $\mathsf{match}(\mathbf{a}, \mathbf{b})$ does not contain θ. The second condition states that θ is *necessary* to construct the new non-trivial binding. The first condition requires $\mathbf{b}$ to label a transition out of the initial state. Let us define trivial redundancy.

Definition 53 (Trivial-redundant binding). *Given a QEA $\mathcal{Q}$ with alphabet $\mathcal{A}$, a trace τ and a binding θ, θ is trivial-redundant for $\mathcal{Q}$ on τ if it is trivial and*

$$\forall \mathbf{a} \in \mathit{Event}, \forall \mathbf{b} \in \mathcal{A} : \mathsf{trivial}(\theta \sqcup \mathsf{match}(\mathbf{a}, \mathbf{b})) \Rightarrow \theta \sqsubseteq \mathsf{match}(\mathbf{a}, \mathbf{b})$$

in which case we say that $\mathsf{trivial\_red}(\theta)$ *holds.*

We can therefore remove all trivial-redundant bindings without effecting the verdict computed (we show this later).

Elimination method

Let us now define a method for trivial-redundant binding elimination. We will do this at the source i.e. where new bindings are introduced (as the redundancy of a binding does not change). New bindings are introduced by the extensions function, so we consider this.

A binding is non-trivial if there is an event in $\mathcal{A}$ that causes a transition from the initial state or can contribute to the creation of a necessary binding when taking a transition from the initial state. Therefore, we define this set of non-trivial events for an EA.

An event $\mathbf{a} \in \mathcal{A}$ is *nontrivial* if it either labels a transition (in δ) from q_0 or there is an event $\mathbf{b}$ labelling a transition from q_0 such that the variables of $\mathbf{a}$ are not contained in the variables of $\mathbf{b}$.

Definition 54 (Nontrivial events). *Given an EA $\langle Q, \mathcal{A}, q_0, \delta, F\rangle$ we define the set*

$$\mathsf{nontrivial}(\mathcal{E}) = \{\mathbf{a} \in \mathcal{A} \mid \exists (q_0, \mathbf{b}, \_, \_, \_) \in \delta : \mathsf{vars}(\mathbf{a}) \subset \mathsf{vars}(\mathbf{b}) \Rightarrow \mathbf{b} = \mathbf{a}\}$$

Therefore, when adding a new binding θ that was created from a trivial binding we apply the following check for trivial-redundancy.

$$\mathsf{check}(\theta, \mathbf{a}) = \forall \mathbf{b} \in \mathsf{nontrivial}(\mathcal{E}) : \mathsf{match}(\mathbf{b}, \mathbf{a}) \notsqsubseteq \theta \wedge \mathsf{dom}(\theta) \subset \mathsf{vars}(\mathbf{b})$$

If a $\mathsf{check}(\theta, \mathbf{a})$ is true then we do not add θ to the monitor lookup or checking structure.

Correctness

Let us demonstrate that this approach preserves verdicts.

Lemma 26. *Given a normal QEA $\mathcal{Q}$ and a trace τ let L_τ be a monitor lookup computed using the standard semantics and K_τ be a monitor lookup computed when applying the above check. We have that*

$$\forall \theta \in \mathsf{dom}(L_\tau) : (\mathsf{total}(\theta) \wedge \theta \notin \mathsf{dom}(K_\tau)) \Rightarrow \langle q_0, [\ ]\rangle \xrightarrow{\tau \downarrow_{\mathcal{A}(\theta)}} \langle q_0, [\ ]\rangle$$

i.e. every total binding in L_τ not computed in K_τ is trivial.

Proof. Let θ be a total binding in L_τ but not in K_τ. Let $\sigma = \tau \downarrow_{\mathcal{A}(\theta)}$ be the sequence of events relevant to θ and therefore those events that caused θ to be created in L_τ. As θ is not in K_τ it must be the case that

$$\forall \mathbf{a} \in \sigma, \forall \mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{b}, \mathbf{a}) \Rightarrow \neg \mathsf{check}(\mathsf{match}(\mathbf{b}, \mathbf{a}), \mathbf{a})$$

and no events in σ label a transition out of an initial state and we have $\langle q_0, [\ ]\rangle \xrightarrow{\sigma} \langle q_0, [\ ]\rangle$. □

Therefore, by the argument in the previous section (for total bindings with empty projections) we have that the verdicts are preserved by this optimisation.

An example

Consider the UnsafeIter example discussed previously (page 92) and the trace

$$\texttt{iterator}(A, 1).\texttt{iterator}(B, 2).\texttt{use}(1).\texttt{update}(B)$$

Firstly, when we process `iterator`$(A, 1)$ the bindings $[c \mapsto A]$ and $[i \mapsto 1]$ are redundant as both have empty projections, and the only event that would lead to a non-trivial binding is another `iterator` event which would reintroduce the bindings. So we only produce the binding $[c \mapsto A, i \mapsto 1]$. Therefore, on the second event we do not produce the total bindings $[c \mapsto A, i \mapsto 2]$ or $[c \mapsto B, i \mapsto 1]$, as they are both trivial. Finally, for the last two events, we do not introduce the bindings $[i \mapsto 1]$ or $[c \mapsto B]$ as they are trivial. Therefore, when we would have previously constructed eight bindings, we only construct two. For this property in general if we have c collections and i iterators the standard approach would create $ci + c + i$ bindings, whereas the new approach would construct i bindings, as each iterator can only be connected to one collection.

Relation to creation events

This check for triviality achieves the same effect as the creation events used in JavaMOP. The main difference is that creation events are manually identified, whereas we have an automated procedure for identifying non-trivial events. However, JavaMOP allows the user to label non-trivial events as non-creation events, and therefore break the semantics of which bindings should be created. Although usually this is accompanied by some user knowledge that those bindings will be in some way redundant. To achieve the same flexibility we could also allow the user to label events as trivial.

6.4.4 Given domain redundancy

Currently if a domain is given it is only considered during checking and bindings that will not be considered (i.e. have values not in the given domain) are still created. We can use the given domain to restrict the created bindings to only those considered for acceptance i.e. introduce a further form of redundancy.

The global guard we introduce is

$$g(\theta) = \forall X \in \mathsf{dom}(\mathcal{D}), \forall x \in \mathsf{vars\_of}(X) : x \in \mathsf{dom}(\theta) \Rightarrow \theta(x) \in \mathcal{D}(X)$$

this is trivially true if $\mathcal{D} = [\ ]$ and is omitted in this case.

6.4.5 Redundant processing

This is a simple optimisation that attempts to use a quick check to remove a lot of event processing. The idea is that until we see a nontrivial event then all bindings created would be trivial and could be omitted. Therefore, we do not start monitoring until we see such an event.

6.4.6 Summary

The main redundancy elimination method introduced in this section identified bindings that did not need to be created as their contribution would be trivial i.e. they would not effect the verdict. Later (Sec. 7.1.5) we show that where this elimination method is applicable it can improve performance, but that applicability can be limited.

6.5 Indexing Strategies

In this section we consider methods for using an event to lookup the bindings necessary for monitoring, we call this *indexing*. We begin by discussing the indexing problem and identifying the issues we will need to address. Particular structural properties of the specification QEA can lead to special cases of indexing and we present algorithms for these cases. We then introduce two alternative indexing techniques: one based on a previous approach using data values in an event and a new approach using the event itself. Finally, we discuss a mechanism for automatically selecting an indexing strategy.

6.5.1 The problem

To help us introduce the indexing problem we first introduce the concept of event consistency.

Definition 55 (Event consistency). *A binding θ is* consistent *with an event* $\mathbf{a}$ *iff it is consistent with a binding that can be constructed from* $\mathbf{a}$ *i.e.*

$$\mathsf{consistent}(\mathbf{a},\theta) = \exists \mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{b},\mathbf{a}) \wedge \mathsf{consistent}(\theta, \mathsf{match}(\mathbf{b},\mathbf{a}))$$

Note that relevance implies consistency. It can easily be observed that when iterating over a monitoring lookup only the entries related to consistent bindings are used.

Lemma 27. *For QEA $\langle \Lambda, \mathsf{E}, \mathcal{D}\rangle$ and trace $\tau.\mathbf{a}$ we have*

1. $\forall \theta \in \mathsf{dom}(L_\tau) : \neg\mathsf{consistent}(\mathbf{a},\theta) \Rightarrow L_\tau(\theta) = L_{\tau.\mathbf{a}}(\theta)$
2. $\forall \theta \in \mathsf{dom}(L_{\tau.\mathbf{a}}) : \theta \notin \mathsf{dom}(L_\tau) \Rightarrow \mathsf{consistent}(\mathbf{a}, \mathsf{max\ below}(\mathsf{dom}(L_\tau),\theta))$

i.e. if a binding is not consistent with an event the associated entry remains unchanged, and all new bindings are added using a consistent binding.

Algorithm 9 All parts for the zero quantifiers algorithm.

```
$\mathcal{Q}$ : QEA
C : Set[Config] ← {}

function INIT($\mathcal{Q}$: QEA)
    C ← {⟨$\mathcal{Q}.q_0$, [ ]⟩}
    $\mathcal{Q}$ ← $\mathcal{Q}$
function UPDATE(**a** : GEvent)
    C ← NEXT([ ], **a**, C)

function CHECK : Verdict
    S ← states in C
    if $\mathsf{StrongS} \cap S \neq \emptyset$ then return $\top_S$
    if $S \subseteq \mathsf{StrongF}$ then return $\bot_S$
    if $F \cap S \neq \emptyset$ then return $\top_W$
    else return $\bot_W$
```

Proof. The first point above is straightforward as, by the definition of NEXT, if there is no event **b** in $\mathcal{A}$ such that $\mathsf{consistent}(\theta, \mathsf{match}(\mathbf{b}, \mathbf{a}))$ then there will be no transitions that can be taken, and therefore the configurations will remain unchanged.

To establish the second point we note that

$$\neg\mathsf{consistent}(\mathbf{a}, \theta) \Rightarrow \mathsf{extensions}(\mathbf{a}, \theta) = \emptyset \tag{6.10}$$

and therefore, as bindings are only added via extensions it is necessarily the case that a binding is added using a consistent binding. Again, it is straightforward to establish equation (6.10) by noting that if θ is not consistent with **a** then the from set (in the construction of extensions) will be empty. □

The indexing problem can therefore be phrased as follows. Given a monitor lookup L_τ and ground event **a** identify the set

$$\{\theta \in \mathsf{dom}(L_\tau) \mid \mathsf{consistent}(\mathbf{a}, \theta)\}$$

i.e. the bindings in the domain of L_τ consistent with **a**.

6.5.2 Specific cases

We consider specific cases of the indexing problem given assumptions about the QEA specification $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$.

Zero quantifiers

In this case where $\Lambda = \epsilon$, i.e. there are no quantifications, we do not need to use a monitor lookup as we have a single set of configurations. The monitoring algorithm can then be trivially given using the NEXT function as seen in Algorithm 9, which gives all three stages of the algorithm i.e. INIT, UPDATE and CHECK. We could optimise this even further if we knew that the QEA were deterministic.

Algorithm 10 Init, update and check functions for the single quantifier algorithm.

```
$\mathcal{Q}$ : QEA
$(q, x, g)$ : $(\{\forall, \exists\}$, Var, Guard)
C : Set[Config] $\leftarrow \{\}$
M : Map[Value,Set[Config]] $\leftarrow$ [ ]
collapse,count $\leftarrow$ false,0

function INIT($\mathcal{Q}$: QEA)
    C $\leftarrow \{\langle \mathcal{Q}.q_0, [\ ]\rangle\}$
    $\mathcal{Q} \leftarrow \mathcal{Q}$
    $q, x, g \leftarrow \mathcal{Q}.\Lambda$

function UPDATE($\mathbf{a}$ : GEvent)
    B $\leftarrow \{\mathsf{match}(\mathbf{a}, \mathbf{b}) \mid \mathbf{b} \in \mathcal{Q}.\mathcal{A}\}$
    if $\exists\theta \in$B : $x \notin \mathsf{dom}(\theta)$ then
        PROCESS([ ])
        for $v \in \mathsf{dom}$(M)} do
            PROCESS($[x \mapsto v]$)
    else
        for $\theta \in$B do
            if $g(\theta)$ then PROCESS($\theta$)

function CHECK : Verdict
    if collapse then return $\top_S$ if $q = \forall$ and $\bot_S$ otherwise
    else  return $\top_W$ if $q = \forall \wedge$ count = 0 and $\bot_W$ otherwise

function PROCESS($\theta$:Binding)
    Old $\leftarrow \begin{cases} \text{C} & \text{if } \theta = [\ ] \text{ or } \theta(x) \notin \mathsf{dom}(\text{M}) \\ \text{M}(\theta(x)) & \text{otherwise} \end{cases}$
    New $\leftarrow$ NEXT($\theta, \mathbf{a}$, Old)
    if $\theta = [\ ]$ then C $\leftarrow$ New
    else
        $(prev, new) \leftarrow (\exists\langle q, \varphi\rangle \in$ Old $.q \in F, \exists\langle q, \varphi\rangle \in$ New $.q \in F)$
        if $\theta(x) \notin M$ and $\neg$ prev then count+=1
        $count \leftarrow \begin{cases} count + 1 & \text{if } prev \wedge \neg this \\ count - 1 & \text{if } \neg prev \wedge this \\ count & \text{otherwise} \end{cases}$
        M $\leftarrow$ M $\dagger$ [ $\theta(x) \mapsto$ New]
        if ($q = \forall$ and $\exists\langle q, \varphi\rangle \in$New : $q \in \mathsf{StrongS}$) then collapse $\leftarrow$ true
        if ($q = \exists$ and $\forall\langle q, \varphi\rangle \in$New : $q \in \mathsf{StrongF}$) then collapse $\leftarrow$ true
```

Single quantifier

In the case where there is a single quantified variable we can use a map from values to sets of configurations as an index. Furthermore, in this case the process of checking can be made simpler and we can replace the CHECK function appropriately.

An algorithm for this special case is given in Algorithm 10. This algorithm incorporates the ideas from incremental checking and global filtering but not redundancy elimination. The algorithm works by first constructing the set of bindings that can be built from the event, and then iterating over these to update their associated configurations. This works as the maximal binding for every bindings is either itself or the empty binding.

The PROCESS function updates the configurations associated with the given binding and then uses the previous and new acceptance values for the binding to update count. Finally, we check the strong states to decide whether an ultimate verdict has been reached. The updated CHECK function uses the *collapse* and *count* variables to return a verdict.

Algorithm 11 The update part for the disjoint alphabet algorithm.

```
$\mathcal{Q}$ : QEA
C : Set[Config] ← {}
M : Map[Bind,Set[Config]] ← [ ]
$strong\_reached$ ← $false$
Γ ← $Node$ of $q_0 \in F$ and [ ] and 0

function INIT($\mathcal{Q}$: QEA)
    C ← {⟨$\mathcal{Q}.q_0$, [ ]⟩}
    $\mathcal{Q}$ ← $\mathcal{Q}$

function UPDATE(**a** : GEvent)
    B ← {match(**a**, **b**) | **b** ∈ $\mathcal{Q}.\mathcal{A}$}
    for $\theta \in B$ do
        Old ← $\begin{cases} C & \text{if } \theta \notin \mathsf{dom}(M) \\ M(\theta) & \text{otherwise} \end{cases}$
        New ← NEXT($\theta$, **a**, Old)
        if $\theta$ = [ ] then
            C ← New
        else
            M ← M † [ $\theta \mapsto$ New]
            S ← {$q$ | $\exists\langle q, \varphi\rangle \in$ New}
            Γ ← UPDATE_CHECK(Γ, 0, $\theta$, $S$)
```

Disjoint alphabet indexing

The reason that the previous indexing approach is possible is that all created bindings are disjoint. We can extend this notion to the case where all bindings with a non-empty projection are disjoint.

If we can define a property of QEA that captures this then we can apply the single indexing strategy to QEAs with more than one quantified variable.

Definition 56 (Disjoint alphabet)**.** *Given a set of quantified variables X, an alphabet $\mathcal{A}$ is disjoint if and only if*

$$\forall\langle e, \overline{x}\rangle \in \mathcal{A} : X \subseteq \overline{x}$$

We assume a normal QEA as we omit bindings with empty projections.

The update function in Algorithm 11 is similar to that for single quantifier indexing, except here we have multiple co-occurring values to index on. We can adapt the previous algorithm to use a binding wherever we saw a value. As we might have multiple quantifiers we use the standard checking function, hence the addition of Γ to initialisation and the call of UPDATE_CHECK. Note that θ is necessarily total.

Another way of viewing this approach is to view the set of quantified variables X as a single variable x and using the single quantifier approach and if we have a pure quantifier list we can do exactly this.

6.5.3 Value-based indexing

This technique is based on a method previously developed for JavaMOP (see [MJG+11]) but needs to extended to our setting, we also formalise the approach for the first time. This strategy is called *value-based* as it looks up consistent bindings using the values in an event.

The general idea

The idea behind this approach is that we map bindings that can be extracted from events to consistent bindings in the domain of the monitor lookup. This can be used to directly lookup the consistent bindings given an event.

The previous work assumed that given a ground event $\mathbf{a}$ we could construct a unique binding θ that captures the values in $\mathbf{a}$, let us write this unique binding as $\mathsf{match}(\mathbf{a})$. We can then note the following relationship:

$$\{\theta \in \mathsf{dom}(L) \mid \mathsf{consistent}(\mathbf{a}, \theta)\} \subseteq \{\theta \in \mathsf{dom}(L) \mid \exists \theta' : \theta' \sqsubseteq \theta \wedge \theta' \sqsubseteq \mathsf{match}(\mathbf{a})\}$$

Note that the set on the right can only contain a few inconsistent bindings and will be significantly smaller than $\mathsf{dom}(L)$. The map described above captures the $\theta' \sqsubseteq \theta$ relationship, allowing us to iterate over $\theta' \sqsubseteq \mathsf{match}(\mathbf{a})$ to construct our set.

Our approach does not restrict the alphabet so that only a single binding can be created from an event. This does not restrict the idea, but does require us to adapt the techniques previously used for creating and updating the map.

The theory

Let us describe the properties that we want our lookup map to have, we shall call this map $\mathbb{U} : \mathit{Bind} \to 2^{\mathit{Bind}}$ as it points up to the desired bindings. $\mathbb{U}$ should have two properties given a monitor lookup L:

1. $\mathbb{U}$ should be 'submap-closed' i.e. contain every submap of a binding in L:
$$\forall \theta \in \mathsf{dom}(L), \forall \theta' \in \mathit{Bind} : \theta' \sqsubset \theta \Rightarrow \theta' \in \mathsf{dom}(\mathbb{U})$$
2. $\mathbb{U}$ should be 'supermap-closed' i.e. every entry in $\mathbb{U}$ should point to the larger, consistent bindings in L:
$$\forall \theta \in \mathsf{dom}(\mathbb{U}), \forall \theta' \in \mathsf{dom}(L) : \theta \sqsubseteq \theta' \Rightarrow \theta' \in \mathbb{U}(\theta)$$

Given these two properties it is guaranteed that $\mathbb{U}$ can be used to lookup all relevant bindings as previously discussed. However, we note that these properties are stronger than necessary as we do not need every submap of a binding in L to appear in $\mathbb{U}$, only those that could be created by matching with an event in $\mathcal{A}$.

The algorithm

We present the replacement parts for the value-indexing algorithm. This extends the method of JavaMOP [MJG⁺11] by removing the assumption that every ground event has a unique binding.

Initialisation. The initialisation given in Algorithm 12 declares the monitor lookup L and two 'up' maps, one for storing the main $\mathbb{U}$ map and the other to be used as a temporary structure to store the bindings created on each step, we will see why this is necessary later. We also declare the components necessary for incremental checking.

Update. The update function given in Algorithm 13 performs two steps to update L. Firstly we extract the bindings B from the incoming event as before. Then we

Algorithm 12 The data structures and initialisation function for the value-based algorithm.

```
Q : QEA
L : Map[Binding, Set[Config]] ← [ ]
$\mathbb{U}$ : Map[Binding, Set[Binding]] ← [ ]
$\mathbb{U}'$ : Map[Binding, Set[Binding]] ← [ ]
$strong\_reached \leftarrow false$

$\Gamma \leftarrow Node$ of $q_0 \in F$ and [ ] and 0

function INIT($\mathcal{Q}$ : QEA)
    $M \leftarrow M \dagger ([\,] \mapsto \{(\mathcal{Q}.\mathcal{E}.q_0, [\,])\})$
    Q ← Q
```

1. Extend L using the bindings in B, and ensure that $\mathbb{U}$ is submap and supermap closed.
2. Update the entries in L given by $\mathbb{U}$

To extend L appropriately we first find the *maximal* binding in L for θ and use that binding to create a new entry in $\mathbb{U}'$ for θ. We then find all consistent bindings in $\mathsf{dom}(L)$ that could be extended using θ to add a new binding. This is achieved by iterating over submaps of θ and using $\mathbb{U}$ to lookup the bindings in $\mathsf{dom}(L)$ consistent with that submap.

Our two closure properties above are preserved by the ADD function, used to add a new binding. This first adds the binding to L with the appropriate configurations (i.e. those of the maximal binding) and then updates $\mathbb{U}'$ with each submap of θ. This second activity preserves submap-closure and super-map closure as it adds all submaps to $\mathbb{U}$ and ensures that they point to the consistent, larger bindings in L. Note that we need to be careful to ensure that elements added to $\mathbb{U}'$ are correctly inserted into $\mathbb{U}$. We use $\mathbb{U}'$ as otherwise a binding added on the current step might be identified as the maximal binding for a binding in B processed later.

Redundancy elimination (see the previous section) would occur in the ADD function i.e. check for redundancy of θ and add it only if it is not redundant.

Because we have updated L with new bindings we now only need to update the configurations associated with *relevant* bindings i.e. those whose projection the event should be added to. These are given by the following set:

$$\{\theta \in \mathsf{dom}(L) \mid \mathsf{relevant}(\mathbf{a}, \theta)\}$$

Relevance can be defined as those bindings consistent with and larger than any binding that can be extracted from an event as follows:

$$\{\theta \in \mathsf{dom}(L) \mid \exists \mathbf{b} \in \mathcal{A} : \mathsf{match}(\mathbf{b}, \mathbf{a}) \sqsubseteq \theta\}$$

As $\mathbb{U}$ has been updated we can use it to straightforwardly lookup the *relevant* bindings as follows:

$$\{\theta \in \mathsf{dom}(L) \mid \exists \mathbf{b} \in \mathcal{A} : \theta \in \{\mathsf{match}(\mathbf{b}, \mathbf{a}\} \cup \mathbb{U}(\mathsf{match}(\mathbf{b}, \mathbf{a}))\}$$

If we update a total binding we update the checking structure as appropriate.

Algorithm 13 The update function for the value-based algorithm.

```
function UPDATE($\mathbf{a}$ : GEvent)
    B : Set[Binding] ← MATCHING($\mathbf{a}$)
    U′ ← [ ]
    for $\theta \in$ B do
        if $\theta \notin$ dom(L) then
            $\theta_M$ ← [ ]
            for $\theta'$ less than $\theta$ from biggest to smallest do
                if $\theta' \in$ dom(L) then $\theta_M \leftarrow \theta'$
            ADD($\theta_M$, $\theta$)
            for $\theta_1$ less than $\theta$ from biggest to smallest do
                for $\theta_2 \in \mathbb{U}(\theta_1)$ do
                    if consistent($\theta_1, \theta_2$) and $(\theta_1 \dagger \theta_2) \notin$ dom(L) then
                        ADD($\theta_2$, $\theta_1 \dagger \theta_2$)
    for $(\theta \mapsto C) \in \mathbb{U}'$ do
        $\mathbb{U} \leftarrow \mathbb{U} \dagger [\theta \mapsto (\text{GET}(\theta, \mathbb{U}) \cup C)]$
    $\mathbb{U}' \leftarrow [\ ]$
    from ← $\{\mathsf{match}(\mathbf{b}, \mathbf{a} \mid \mathbf{b} \in \mathcal{A}(\theta) : \mathsf{matches}(\mathbf{b}, \mathbf{a})\}$
    for $\theta \in$ from $\cup \bigcup_{\theta' \in \text{from}}$ GET($\theta'$, $\mathbb{U}$) do
        C ← NEXT($\theta$,$\mathbf{a}$,L($\theta$))
        L ← L † ($\theta \mapsto$ C)
        if total($\theta$) then $\Gamma \leftarrow$ UPDATE_CHECK($\Gamma, 0, \theta, \{q \mid \exists \langle q, \varphi \rangle \in \text{C}\}$)

function ADD($\theta_M$ : Binding, $\theta$ : Binding)
    L ← L † ($\theta \mapsto$ L($\theta_M$))
    for $\theta'$ less than $\theta$ do
        $\mathbb{U}' \leftarrow \mathbb{U}' \dagger [\theta' \mapsto (\text{GET}(\theta', \mathbb{U}') \cup \{\theta\})]$

function GET($\theta$ : Binding, A : Map[Binding,Set[Binding]])
    if $\theta \in$ dom(A) then return A($\theta$)
    else return {}
```

An example

Let us demonstrate the usage of this indexing approach using the LockOdering property, first introduced in Fig. 4.2 on page 77, and the short trace

$$\tau = \texttt{lock}(1).\texttt{lock}(2).\texttt{unlock}(2)$$

We only consider in detail the events that add new bindings, as we are interested in how $\mathbb{U}$ is updated. We do not consider redundancy elimination here. As there are no free variables we replace configurations by the states that they contain for conciseness.

We begin with empty structures as given by the INIT function. On receiving `lock`(1) we compute

$$\text{B} = \{[l_1 \mapsto 1], [l_2 \mapsto 1], [l_1 \mapsto 1, l_2 \mapsto 1]\}$$

and as none of these bindings exist in L we go through the process of adding them. In this case, for each $\theta \in B$ we find θ_M to be [ ] and add θ using this. As we start with $\mathbb{U}$ empty, the set returned by $\text{GET}(\theta_1, \mathbb{U})$ will be empty and no bindings will be considered here. After updating $\mathbb{U}$ we find the relevant bindings, in this case this is all of B, and process them appropriately. At the end of this step we have the following for M and $\mathbb{U}$:

$$\begin{array}{cc}
\text{M} & \mathbb{U} \\
\left[\begin{array}{lcl}
[\,] & \mapsto & \{1\} \\
[l_1 \mapsto 1] & \mapsto & \{2\} \\
[l_2 \mapsto 1] & \mapsto & \{1\} \\
[l_1 \mapsto 1, l_2 \mapsto 1] & \mapsto & \{2\}
\end{array}\right]
&
\left[\begin{array}{lcl}
[\,] & \mapsto & \{[l_1 \mapsto 1], [l_2 \mapsto 1], [l_1 \mapsto 1, l_2 \mapsto 1]\} \\
[l_1 \mapsto 1] & \mapsto & \{[l_1 \mapsto 1, l_2 \mapsto 1]\} \\
[l_2 \mapsto 1] & \mapsto & \{[l_1 \mapsto 1, l_2 \mapsto 1]\}
\end{array}\right]
\end{array}$$

Let us now consider what is done when receiving `lock`(2). We begin by computing

$$\text{B} = \{[l_1 \mapsto 2], [l_2 \mapsto 2], [l_1 \mapsto 2, l_2 \mapsto 2]\}$$

and again add each binding using [ ]. Now we also consider the bindings in $\mathbb{U}$ consistent with each θ in B, leading us to add $[l_1 \mapsto 2, l_2 \mapsto 1]$ as an extension of $[l_2 \mapsto 1]$ and $[l_1 \mapsto 1, l_2 \mapsto 2]$ as an extension of $[l_1 \mapsto 1]$.

$$\begin{array}{cc}
\text{M} & \mathbb{U} \\
\left[\begin{array}{lcl}
[\,] & \mapsto & \{1\} \\
[l_1 \mapsto 1] & \mapsto & \{2\} \\
[l_2 \mapsto 1] & \mapsto & \{1\} \\
[l_1 \mapsto 2] & \mapsto & \{2\} \\
[l_2 \mapsto 2] & \mapsto & \{1\} \\
[l_1 \mapsto 1, l_2 \mapsto 1] & \mapsto & \{2\} \\
[l_1 \mapsto 2, l_2 \mapsto 2] & \mapsto & \{2\} \\
[l_1 \mapsto 1, l_2 \mapsto 2] & \mapsto & \{3\} \\
[l_1 \mapsto 2, l_2 \mapsto 1] & \mapsto & \{2\}
\end{array}\right]
&
\left[\begin{array}{lcl}
[\,] & \mapsto & \left\{\begin{array}{c}
[l_1 \mapsto 1][l_1 \mapsto 1, l_2 \mapsto 1], \\
[l_1 \mapsto 2], [l_1 \mapsto 2, l_2 \mapsto 2], \\
[l_2 \mapsto 1], [l_1 \mapsto 1, l_2 \mapsto 2], \\
[l_2 \mapsto 2], [l_1 \mapsto 2, l_2 \mapsto 1]
\end{array}\right\} \\
[l_1 \mapsto 1] & \mapsto & \{\ [l_1 \mapsto 1, l_2 \mapsto 1], [l_1 \mapsto 1, l_2 \mapsto 2]\ \} \\
[l_2 \mapsto 1] & \mapsto & \{\ [l_1 \mapsto 1, l_2 \mapsto 1, [l_1 \mapsto 2, l_2 \mapsto 1]\ \} \\
[l_1 \mapsto 2] & \mapsto & \{\ [l_1 \mapsto 2, l_2 \mapsto 2], [l_1 \mapsto 2, l_2 \mapsto 1]\ \} \\
[l_2 \mapsto 2] & \mapsto & \{\ [l_1 \mapsto 2, l_2 \mapsto 2], [l_1 \mapsto 1, l_2 \mapsto 2]\ \}
\end{array}\right]
\end{array}$$

Now on receiving `unlock`(2) all bindings in B are present in M and therefore we do not add any new bindings and proceed straight to retrieving all relevant events and updating them using NEXT.

Analysis

Let us discuss the correctness and time complexity of this indexing algorithm.

Correctness. To determine the correctness of this approach note that the same bindings are added to L as in the basic approach, and the relevant bindings in L are used to update configurations. To see that the same bindings are added to L assume that there is a binding θ that is not. As θ should be in L then there is an event $\mathbf{a}$ in τ that causes its addition and another binding θ' in the monitor lookup at that stage that it extends (possibly [ ]). In our approach we attempt to add all possible combinations of bindings extracted from $\mathbf{a}$ and consistent bindings in the monitor lookup, therefore either θ is added or $\mathbf{a}$ and θ' do not exist.

Complexity. Let us consider two cases when processing an event $\mathbf{a}$:

- *The event* $\mathbf{a}$ *is new.* In this case we need to go through the process of adding all bindings that can be generated from $\mathbf{a}$ and the domain of L, and ensure that $\mathbb{U}$ is correctly updated. As we iterate over all bindings in L when we consider compatibility with the bindings in $\mathbb{U}([\ ])$ it is easy to see that this step is linear in the number of bindings.
- *We have seen* $\mathbf{a}$ *before.* In this case we just look up the relevant bindings directly and iterate through them calling Next. Therefore, the complexity is bounded by the cost of computing matching bindings, the lookup calls and the Next calls. We can take all three processes as constant time, assuming a reasonably sized QEA.

Therefore this approach performs best when the common case is where events are seen multiple times. As we see in the next chapter, this is often the case in real world programs.

6.5.4 Symbol-based indexing

We now introduce an alternative indexing technique that uses the symbols of an instantiated EA as an index, rather than the values associated with it. The overall idea is similar to tree-based term-indexing methods used in automated theorem proving [SRV01, Gra95], an idea we explore further in further work (Sec. 12.2.3). As before, we give an overview of the idea before presenting and discussing an updated algorithm.

The general idea

The general idea behind this approach is to use the alphabet of an instantiated event automaton to lookup the configurations associated with that instantiation. We note the following straightforward relationship for total bindings:

$$\{\theta \in \mathsf{dom}(L) \mid \mathsf{consistent}(\mathbf{a}, \theta)\} = \{\theta \in \mathsf{dom}(L) \mid \exists \mathbf{b} \in \mathcal{A}(\theta) : \mathsf{matches}(\mathbf{a}, \mathbf{b})\}$$

This tells us that if have a map from the alphabets of instantiated EAs to the instantiating binding then we can lookup consistent bindings. However, as we have seen previously, we need to deal with partial bindings and therefore some events in the alphabet of an instantiated EA will contain variables. For example, we need to identify $\mathtt{f}(1, y)$ using the event $\mathtt{f}(1, 2)$. To do this we replace y with a special symbol $\_$ and consider all versions of $\mathtt{f}(1{,}2)$ with values replaced by $\_$, therefore simplifying the approach of matching the events by ignoring the variables used.

We introduce $\prec$ as a relationship on events where $\mathbf{b} \prec \mathbf{a}$ if $\mathbf{a}$ and $\mathbf{b}$ have the same event name and $\mathbf{b}$ has the same parameters as $\mathbf{a}$, except that some parameters may be replaced with the special symbol $\_$. For example, $\mathtt{f}(\_, \_) \prec \mathtt{f}(1, \_) \prec \mathtt{f}(1, 2)$. If $\mathbf{a} \prec \mathbf{b}$ we say that $\mathbf{b}$ is *more specific* than $\mathbf{a}$. Let $\mathsf{ver}(\mathbf{a})$ be the smallest event such that $\mathsf{ver}(\mathbf{a}) \prec \mathbf{a}$ i.e. replacing all parameters by $\_$.

Our relationship now needs to be refined to become:

$$\{\theta \in \mathsf{dom}(L) \mid \mathsf{consistent}(\mathbf{a}, \theta)\} \subseteq \{\theta \in \mathsf{dom}(L) \mid \exists \mathbf{b} \in \mathcal{A}(\theta) : \mathsf{ver}(\mathbf{b}) \prec \mathbf{a}\}$$

Algorithm 14 The data structures and initialisation function for the symbol-based algorithm.

```
Q : QEA
L : Map[Binding,Set[Config]] ← [ ]
Index : Map[Event,List[Binding]] ← [ ]
$strong\_reached \leftarrow false$
$\Gamma \leftarrow Node$ of $q_0 \in F$ and [ ] and 0

function INIT($\mathcal{Q}$ : QEA)
    M ← M † ([ ] ↦ {⟨$q_0$, [ ]⟩})
    for $\mathbf{b} \in \mathcal{A}$ do
        Index ← Index † ($\mathsf{ver}(\mathbf{b}) \mapsto \epsilon$)
    Q ← Q
```

To understand why this is no longer equality note that $\mathsf{ver}(\mathtt{f}(x,x)) = \mathtt{f}(\_,\_)$, which matches with $\mathtt{f}(1,2)$. This over-approximation is due to our choice to ignore variable names and is small (usually empty) and easily filtered.

We note that an event is only relevant to a binding if it appears in the instantiated alphabet (or suitably updated to take into account free variables):

$$\mathsf{relevant}(\mathbf{a},\theta) \Leftrightarrow \mathbf{a} \in \mathcal{A}(\theta)$$

This comes from the definition of relevance; if θ is not relevant then some quantified variables are captured in matching, and therefore there would be some _ symbols in the instantiated alphabet event. Furthermore we note that if **a** appears in our index from events to bindings then no bindings associated with less specific versions of **a** can lead to new bindings. This is simply because any extensions to bindings associated with a less specific versions of **a** would also be extensions to bindings associated with **a**, with the exception of extensions which were already bindings associated with **a**.

The algorithm

We present the two replacement parts for the symbol-indexing algorithm.

Initialisation. As well as introducing the normal monitor lookup and checking structure, the instantiation given in Algorithm 14 declares an index that maps events to *lists* of bindings. The fact that we have a list of bindings is important as it preserves our notion of maximality. Bindings will be added to the front of the list and therefore bindings created most recently are at the beginning. It is the case that if one binding is created before another then the first binding is necessarily smaller than or inconsistent with the second binding. Therefore, the lists of bindings in the index are linearised with respect to $\sqsubseteq$.

Update. The replacement UPDATE function in Algorithm 15 uses the index to find consistent bindings. This consists of first computing the compatible bindings using different versions of the event, and then iterating through these bindings adding new bindings and updating configurations.

The COMPATIBLE function uses the process described above to find all consistent bindings i.e. we generate all versions of **a** and use this to lookup the bindings whose instantiated alphabets those version-events belong to. The different versions are created by considering subsets of the

Algorithm 15 The update function for the symbol-based algorithm.

```
function UPDATE(a : GEvent)
    B ← MATCHING(a)
    for $\theta$ in COMPATIBLE(a) (in order) do
        C ← L($\theta$)
        if $\theta$ ∈GET(a) then ADD($\theta$,NEXT($\theta$,C,a),false)
        for $\theta'$ in B from largest to smallest, where $\theta'$ is consistent with $\theta$ do
            if $\theta'$ and $\theta\dagger\theta'$ not in L then ADD($\theta\dagger\theta'$,NEXT($\theta\dagger\theta'$,C,a),true)

function COMPATIBLE(a : GEvent) : List[Binding]
    if Index contains a then return a
    B ← $\epsilon$
    for $X \subseteq \mathsf{vals}(\mathbf{a})$ from smallest to largest do
        version ← a with values in $X$ replaced by _
        for $\theta$ in GET(a) do
            if $\theta$ not in B then B = B.$\theta$
    return B

function ADD($\theta$ : Binding,C : Set[Config], new : boolean)
    L ← L † ($\theta \mapsto$ C)
    if total($\theta$) then $\Gamma \leftarrow$ UPDATE_CHECK($\Gamma, 0, \theta, \{q \mid \exists\langle q,\varphi\rangle \in \mathrm{C}\}$)
    if new then
        for a in $\mathcal{A}(\theta)$ do
            version ← ver(a)
            Index ← Index † [version ↦ ($\theta$.GET(a)) ]

function GET(a:Event)
    if Index contains a then return Index(a)
    else return $\epsilon$
```

values in **a** and replacing these with our special _ symbol. It is important that we generate these versions from most to least specific i.e. we first consider $X = \{\}$ and generate the version of **a** with no replacements. This is because of the following simple observation for any two bindings θ and θ' and any two events **a** and **b**:

$$(\mathbf{a} \prec \mathbf{b} \wedge (\exists \mathbf{a}' \in \mathcal{A}(\theta) : \mathsf{ver}(\mathbf{a}') \prec \mathbf{a}(\wedge(\exists \mathbf{b}' \in \mathcal{A}(\theta') : \mathsf{ver}(\mathbf{b}') \prec \mathbf{b})) \Rightarrow \theta \sqsubseteq \theta'$$

This says that if **a** is more specific than **b** then any binding that matches with **a** in its alphabet is no larger than any binding that matches with **b** in its alphabet. Therefore, as each list of bindings in the index is a linearisation, the list of compatible bindings is a *linearisation* with respect to $\sqsubseteq$.

We optimise the COMPATIBLE function so that if **a** is in Index then we do not return any smaller versions of **a**. This breaks our contract that we will consider all consistent bindings, however, as noted above the bindings mapped to by any smaller versions would be redundant i.e. not be relevant nor lead to new bindings.

For each compatible binding we first update the configurations if the binding is relevant i.e. $\mathbf{a} \in \mathcal{A}(\theta)$. We then add the binding and then attempt to extend it using bindings in B (the bindings matching **a**). As the compatible bindings are linearised this is the same process we go through in the basic algorithm, only we do not consider inconsistent bindings. The check for consistency and the fact that NEXT only updates configurations for relevant bindings ensures that we do not create any incorrect bindings or configurations.

The ADD function simply updates L and the checking structure if appropriate and then if the binding is new it updates the index appropriately i.e. adds the variable-free versions of the events in the instantiated alphabet. Note that bindings are added to the front of the list of bindings, as we discussed earlier.

An example

Let us demonstrate this approach using the UnsafeMapIter example previously given in Fig. 6.4 on page. 133 and the following short trace:

$$\tau = \texttt{connect}(A, B).\texttt{iterator}(B, 1).\texttt{use}(1)$$

We are interested in how the index is used and updated. Again we replace configurations by the states that they contain for conciseness. We begin with the index populated with the empty binding:

$$\text{Index} = \begin{bmatrix} \texttt{connect}(\_,\_) & \mapsto & [\;] \\ \texttt{iterator}(\_,\_) & \mapsto & [\;] \\ \texttt{update}(\_) & \mapsto & [\;] \\ \texttt{use}(\_) & \mapsto & [\;] \end{bmatrix}$$

On receiving connect(A,B) we firstly compute

$$\text{B} = \{[c_1 \mapsto A], [c_2 \mapsto B], [c_1 \mapsto A, c_2 \mapsto B]\}$$

and then compute

$$\textsc{Compatible}(\texttt{connect}(\text{A,B})) = [\;]$$

by attempting to access the index with connect(A,B), connect(A,_), connect(_,B) and connect(_,_). The last event returns [ ]. We do not update the configurations associated with [ ] as it is not relevant. We then add the three new bindings from B and update Index as follows:

$$\text{Index} = \begin{bmatrix} \texttt{connect}(\_,\_) & \mapsto & [\;] \\ \texttt{iterator}(\_,\_) & \mapsto & [c_1 \mapsto A].[\;] \\ \texttt{update}(\_) & \mapsto & [c_2 \mapsto B].[\;] \\ \texttt{use}(\_) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B].[c_1 \mapsto A].[c_2 \mapsto B].[\;] \\ \texttt{connect}(A,\_) & \mapsto & [c_1 \mapsto A] \\ \texttt{connect}(\_,B) & \mapsto & [c_2 \mapsto B] \\ \texttt{connect}(A,B) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B] \\ \texttt{iterator}(B,\_) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B].[c_2 \mapsto B] \\ \texttt{update}(A) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B].[c_1 \mapsto A] \end{bmatrix}$$

On receiving `iterator`(B,1) we compute

$$\textsc{Compatible}(\texttt{iterator}(\text{B},1)) = [c_1 \mapsto A, c_2 \mapsto B].[c_2 \mapsto B].[c_1 \mapsto A].[\ ]$$

by attempting to access the index with the different versions of $\texttt{iterator}(B, 1)$ i.e. `iterator`(B,1), `iterator`(B,_), `iterator`(_,1) and `iterator`(_,_). It is with the second and last events that we build our list of bindings. We update the configurations and extend each binding with bindings in B where appropriate.

It is here we see that the ordering of consistent bindings is important. If we had processed [ ] before $[c_2 \mapsto B]$ then we would have added $[c_2 \mapsto B, i \mapsto 1]$ using the incorrect configurations. The resulting index is as follows:

$$\text{Index} = \left[\begin{array}{lcl}
\texttt{connect}(\_,\_) & \mapsto & [i \mapsto 1].[\ ] \\
\texttt{iterator}(\_,\_) & \mapsto & [c_1 \mapsto A]..[\ ] \\
\texttt{update}(\_) & \mapsto & [c_2 \mapsto B, i \mapsto 1].[c_2 \mapsto B].[\ ] \\
\texttt{use}(\_) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B].[c_1 \mapsto A].[c_2 \mapsto B].[\ ] \\
\texttt{connect}(A,\_) & \mapsto & [c_1 \mapsto A, i \mapsto 1].[c_1 \mapsto A] \\
\texttt{connect}(\_,B) & \mapsto & [c_2 \mapsto B, i \mapsto 1].[c_2 \mapsto B] \\
\texttt{connect}(A,B) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B, i \mapsto 1].[c_1 \mapsto A, c_2 \mapsto B] \\
\texttt{iterator}(B,\_) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B].[c_2 \mapsto B] \\
\texttt{update}(A) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B, i \mapsto 1].[c_1 \mapsto A, c_2 \mapsto B].[c_1 \mapsto A] \\
\texttt{iterator}(\_,1) & \mapsto & [c_1 \mapsto A, i \mapsto 1].[i \mapsto 1] \\
\texttt{iterator}(B,1) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B, i \mapsto 1].[c_2 \mapsto B, i \mapsto 1] \\
\texttt{use}(1) & \mapsto & [c_1 \mapsto A, c_2 \mapsto B, i \mapsto 1].[c_2 \mapsto B, i \mapsto 1]. \\
 & & \quad [c_1 \mapsto A, i \mapsto 1].[i \mapsto 1]
\end{array}\right]$$

When processing `use`(1) we will consider only the bindings in Index(`use`(1)) as the event appears in Index, therefore we have:

$$\textsc{Compatible}(\texttt{use}(1)) = \quad [c_1 \mapsto A, c_2 \mapsto B, i \mapsto 1].[c_2 \mapsto B, i \mapsto 1].[c_1 \mapsto A, i \mapsto 1].[i \mapsto 1]$$

The event is relevant to all of these bindings and does not lead to any extensions, and therefore no new additions to Index.

Analysis

Let us consider the correctness and time complexity of this indexing algorithm.

Correctness. To determine the correctness of this approach note that our iteration over $\textsc{Compatible}(\mathbf{a})$ performs the same actions as the monitor lookup step construction (Def. 40 on page 101). Importantly, every binding in the domain of L not in $\textsc{Compatible}(\mathbf{a})$ is not relevant and does not lead to new bindings being added. This is the key idea behind this approach.

Complexity. Let us consider two cases when processing an event $\mathbf{a}$:

- *The event* **a** *is new.* In this case, we will consider all versions of **a**, which is bounded by 2^k if **a** has k parameters, and lookup all bindings associated with each version. Then for each binding we will call NEXT and consider all extensions with B. The number of consistent bindings is determined by the alphabet of the QEA and, of course, the number of values for each quantified variable. The number of consistent bindings should be significantly smaller than the set of all bindings.

- *We have seen* **a** *before.* In this case we look up the relevant bindings directly and iterate through them. The complexity here is minimal, and for a reasonably sized QEA we can take this as constant time.

Again, this approach performs best when the common case is where events are seen multiple times. However, here we see that the work to be completed when adding a new binding is effected by the alphabet of the QEA, not the number of bindings.

6.5.5 Algorithm selection

We have introduced multiple indexing strategies with different complex conditions for applicability. It is not reasonable to expect the user to be able to decide which strategy to use. Therefore, we introduce a function that selects the most appropriate strategy based on the QEA being monitored.

Firstly, if a connectedness global guard is used then we select variations of the algorithms that incorporate the appropriate structures and alternative checking algorithms to handle connectedness.

Secondly, the mechanisms required to handle features such as non-determinism and free variables are unnecessarily complicated when the monitored properties do not use them. Therefore, we construct reduced algorithms to be used in these cases that are automatically selected if the associated features are not used. Additionally, we include options for turning different optimisations, such as redundancy elimination, off when creating a monitor.

Finally, we select an indexing strategy based on structural properties of the QEA. We select Zero indexing if $\Lambda = \epsilon$ and Single indexing if $|\Lambda| = 1$. If the alphabet is disjoint we select Disjoint indexing. Otherwise we select Symbol indexing by default (as later experiments show this performs better most often) with the option of selecting Value indexing instead. We also select the optimised checking algorithm if quantification is pure.

6.6 Dealing with Reference Values: Garbage and Equality

So far we have assumed that values are symbols with a notion of equality. In the case of integer or boolean values the notion of equality is straightforward, but if these values are objects taken from an object-orientated programming language, as is often the case, then there are two issues we must consider. Firstly, we need to fix our notion of equality for such object values, and secondly we should handle the case where these object values become ‘garbage’ i.e. unreachable in their original program. In this section we discuss these two issues and propose solutions as extensions of the techniques we have already developed.

The implementation of our tool is given in the `Scala` programming language and we generally assume we are working within the family of JVM (Java Virtual Machine) based languages. However, the ideas in this section can be mapped across to concepts in other high-level languages where we have a notion of garbage collection (automatic or manual) and differing actual and semantic equality.

6.6.1 Exploring the issues

Let us consider the two issues introduced above.

Equality

With reference objects there are two notions of equality we might want to use:

- Identity - the two reference objects refer to the same point in memory. In `Java` this can be achieved by using `==` or `System.IdentityHashCode`.
- Semantic equality - the two reference objects are semantically equivalent via some user defined notion. In `Java` this can be achieved by using `equals` or `hashCode`, and often coincides with usage of the `Comparable` interface.

The first case will be the most common, but there are cases where we may wish to use the second. For example, in `Java`, objects such as `Integer` and `String` are not always interned, and therefore `(new Integer(200)) == (new Integer(200))` will be false. Additionally, there may be some domain-specific reason semantic equality is required.

We therefore want the user to be able to specify which form of reference equality they want to be used when comparing two reference values. Note that we will also have a class of *primitive* values in any language, for example `int` and `byte` in `Java`, which may have their own notion of equality. In `Java` we use `==`.

Garbage

Reference objects are generally stored in a separate area of memory from the main program stack and are accessed via pointers. If an object cannot be reached via any sequence of pointers from the program stack it is said to be *unreachable* and can no longer take part in the program execution. This objects are referred to as *garbage* and it is common to have techniques to remove such objects as they are unnecessarily consuming memory. In JVM-based languages this is via automated garbage collection.

We note that garbage references cannot take part in any future events. Firstly, this means that there are sets of bindings in our monitor lookup that are now redundant and using up space. But more importantly, the states associated with these bindings can now be considered final i.e. as the strong versions of themselves. By noting this we can make two optimisations:

1. If quantifiers are pure it might be the case that viewing states as strong might lead directly to a strong result i.e. if our quantification is $\forall x, \forall y$ and a total binding $[x \mapsto A, y \mapsto B]$ is in a non-final state when the object A becomes garbage then we have strong failure.

2. If quantifiers are pure but the binding does not cause an ultimate result, or if the quantifiers are alternating, we can apply the optimisations discussed in Sec. 6.4 for removing redundant bindings.

In the following we discuss our extensions that deal with the issues of equality and garbage.

6.6.2 Garbage events

To abstract away from the problem of detecting garbage objects at runtime (we might also be monitoring offline), we introduce the concept of *garbage events*. These are a special form of event that indicate to the monitoring algorithm that a certain object is now redundant. A garbage event consists of the name `garbage`[1] and a finite sequence of values that have become garbage.

Next we consider how garbage events should be handled. We can use a set of garbage values to identify a set of bindings which are *garbage redundant*.

A binding θ in configurations C is *garbage-redundant* if there are no transitions in $\delta(\theta)$ such that a configuration in C can reach a state with a different acceptance.

Definition 57 (Garbage Redundant Bindings)**.** *Given a set of garbage values G and a QEA* $\mathcal{Q} = \langle \lambda, \langle Q, \mathcal{A}, q_0, \delta, F\rangle, \mathcal{D}\rangle$, $\mathsf{garbage\_redundant}(\theta, C)$ *holds if*

$$\forall \langle q, \varphi\rangle \in C, \forall (q_1, e(\overline{x}), g, \gamma, q_2) \in \delta . q = q_1 \Rightarrow$$
$$(\forall x \in \overline{x} . x \in \mathsf{vars}(\Lambda) \vee \theta(x) \notin G) \Rightarrow \begin{array}{ll} q_1 = q_2 \vee q_2 \in \mathsf{StrongS} & \textit{if } \exists \langle q', \varphi'\rangle \in C . q' \in F \\ q_1 = q_2 \vee q_2 \in \mathsf{StrongF} & \textit{otherwise} \end{array}$$

Garbage redundant bindings should be removed from any indexing structures and the monitor lookup. Here care must be taken not to prevent the creation of later bindings that would effect the verdict returned. They should also be removed from the checking structure. This may lead to a strong state being reached and an ultimate verdict being returned. We only check for garbage events at intervals (every 100 events) to prevent the processing of garbage events from interfering with the normal monitoring process and to amortize the cost of removing garbage as it requires us to iterate over a number of collections.

6.6.3 Online monitoring

Here we consider the processes we go through when monitoring a program to produce garbage events and deal with the notion of equality.

Extending QEA

We add two additional (optional) components to QEA: the reference variables and the kept variables. The reference variables record the notion of equality to use for a variable and the kept variables are those whose value we need to observe in guards or assignments.

The reference variables consist of a pair $\langle IRef, SRef\rangle$ such that $IRef \cap SRef = \emptyset$, $IRef \cup SRef \subseteq \mathsf{vars}(\mathcal{E})$ and for every two quantified variables x and y, if they have the same type

[1]This is therefore a reserved event name.

variable then they are in the same reference set. The kept variables are a set $K \subseteq \mathsf{vars}(\mathcal{E})$ such that $K \subseteq (IRef \cup SRef)$.

These sets will then be used to appropriately process events when they are received. These components are optional and by default all variables are considered to be in $IRef$ and not in K. One possible future extension would be to examine guards and assignments and attempt to automatically detect variables that should be in K.

Preprocessing

When monitoring online (when the monitored program is running) we do the following to each incoming event:

1. Replace values in events with either their identity hash code if they are in $IRef$ or their semantic hash code if they are in $SRef$.
2. Store weak references to reference values along with a pointer to a `ReferenceQueue`. When the reference values are garbage collected the weak references will be placed on this queue. A weak reference is one that does not prevent garbage collection i.e. does not effect the reachability of an object.
3. At intervals during the execution of the monitor, check the reference queue and create garbage events for any garbage objects, using the appropriate hash code. We must also ensure that garbage events are not accidentally created before all events for that value have been processed.
4. Map hash codes to these weak references if they are related to a kept variable in K. This map is then accessed in guards and assignments to access the true values.

When collecting traces for offline monitoring we follow the same procedures, only storing events rather than passing them to a monitor.

Important choice

Using semantic equality can be problematic when the identify of a monitored object changes during the monitored process. A notable example of this behaviour is with collections where semantic equality is based on their contents. It is therefore important that the correct notion of equality is used, although the default behaviour (of identity equality) is most likely to be the desired one.

6.7 Summary

In this chapter we have considered five different ways in which the runtime monitoring of QEAs can be optimised. Firstly, we introduced a five-valued verdict domain and methods for incrementally tracking checking information. Secondly, we considered how global guards could be used to trim the search space and discussed how we could deal with special trace-predicated guards such as connectedness. Thirdly, we looked at redundancy elimination, focusing on removing redundant bindings. The fourth optimisation introduced indexing strategies for quickly identifying the bindings required for updating monitor lookups. Finally, we discussed mechanisms for dealing with the pragmatic issues of equality and garbage that arise when dealing with reference values.

Chapter 7

Evaluating Monitoring Techniques

In this chapter we evaluate the incremental runtime monitoring technique and optimisations discussed in Chapters 5 and 6. As outlined in Section 1.1, our aim is to develop a highly expressive runtime monitoring technique that can compete with the most-efficient techniques and improve on the efficiency of more expressive techniques. To this end, we compare our technique to one of the most efficient tools (JavaMOP) and one of the most expressive tools (RuleR). We acknowledge that our system is relatively immature compared with systems such as JavaMOP, which have received years of optimisation.

Structure. We use the sets of specifications discussed in Sec. 4.1.1 to evaluate our techniques. We begin (Sec. 7.1) by using the planetary rover scenario to address research questions, such as the effect of the number of quantifiers on monitoring overhead, and the relative performance of different indexing techniques. We then (Sec. 7.2) consider the application of QEA to real-world programs by using the DaCapo benchmark suite to monitor a number of `Java` library properties. In both cases we compare QEA the with JavaMOP and RuleR.

Implementation. We have implemented algorithms in the `Scala` programming language [OSV08] as its combination of functional and object-oriented features make it particularly suited to implementing QEA as an internal domain specific language (DSL). We abbreviate implementations with no, zero quantifier, single quantifier, disjoint alphabet, symbol and value indexing strategies Basic, Zero, Single, Disjoint, Symbol and Value respectively.

Experimental machine. All experiments are carried out on an eight core Mac Pro containing two 2.26GHz Quad-Core Intel Xeon 5500 series processors and 16GB of RAM. We use version 1.6 of `Java` running on Oracle's HotSpot 64-Bit Server VM.

The other tools

In this chapter we will compare QEA with both JavaMOP and RuleR systems, previously described in Section 4.5. As it is will be helpful when discussing our results we briefly review the important points of each implementation.

We use version 3.0 of JavaMOP. The tool generates an `AspectJ` file that contains all instrumentation and monitoring code. There are a range of optimisation techniques incorporated here, including extensive caching, but the most important is their approach to indexing and garbage. They have implemented a range of custom mapping structures that combine hashing and `WeakRerefences`[1] to ensure that they can quickly locate the correct monitors and clear monitors relating to garbage. The initial garbage implementation was *unsound* with respect to an implicitly universally quantified specification as bindings that could be extended to failure were removed if they contain garbage (see Sec. 6.6). This issue has been fixed in later work [JMGR11], and would not effect the properties monitored here.

RuleR maintains a set of rule activations and updates this by iterating over the set. Some kinds of rule activations require checking all other activations to see if they can fire. This linear search can become very expensive when many data values are being tracked. However, rule systems can explicitly remove rule activations, meaning that some well-written, simple specifications can ensure that this set is kept small. Garbage is dealt with by storing data in `WeakReference`s and occasionally scanning for garbage rule activations. By default RuleR stops monitoring when it reports an error. To allow us to detect many errors we restart the monitor on an error, although this is unsound and has the potential to report false positives as the data structures will be reset and some previous information is lost.

7.1 Planentary rover case study

We use the planetary rover case study to address a number of research questions.

7.1.1 Research questions

We aim to answer the following questions:

RQ1 How do the indexing techniques based on structural properties (zero and one variable and disjoint alphabet) compare to basic monitoring, and do they represent an improvement on the general indexing techniques?

RQ2 For each indexing approach, which structural properties, if any, effect running time, and to what extent?

RQ3 To what extent do the redundancy elimination techniques (including garbage removal) effect running times?

RQ4 How does our technique compare with the efficient JavaMOP and expressive RuleR runtime verification systems?

[1]In `Java` a weak reference does not prevent an object being marked for garbage collection. This prevent data leaks and keeps data structures small. We use a different approach in our work.

Table 7.1: A summary of the properties used in this case study. CC = cyclomatic complexity.

Name	$\|\Lambda\|$	$\|Q\|$	$\|\mathcal{A}\|$	CC	disj.	norm.	has triv.	Fig/page
GrantCancelS	1	4	3	4	yes			A.25/350
GrantCancelD	2	4	2	4				A.25/350
ResourceLifecycle	1	4	5	6	yes		no	A.26/350
ReleaseResource	3	4	4	4				A.27/350
RespectConflictsS	1	4	5	6				A.29/351
RespectConflictsD	2	4	5	7				A.28/351
RespectPriorities	2	11	10	28				A.30/352
ExactlyOneSuccess	1	5	3	5	yes			A.31/352
IncreasingIdentifiers	0	3	2	4				A.32/353
CommandAcks	1	4	4	6				A.33/353
NestedCommands	2	4	4	3			no	3.3/61
ExistsSatellite	2	3	2	1	yes			A.34/353
ExistsSatelliteS	1	2	3	2	yes			A.35/354
ExistsLeader	2	3	2	1	yes	no	no	A.36/354
HashCorrect	2	2	2	4	yes			A.37/354

The hypthetical planetary rover system used in this section mainly reflects a log-file analysis scenario, however we make some modifications to allow us to evaluate the effects of garbage collection. Section 7.2 is concerned with monitoring real world programs online. These research questions are addressed in Sections 7.1.3 to 7.1.6.

7.1.2 Properties and workloads

We give an overview of the properties and workloads used in this study.

Properties overview

Table. 7.1 gives an overview of the properties (see Appendix A.4) that we will be using in this evaluation. There is a slight discrepancy between the number of states given in Table. 7.1 and the graphical representations as we must add an implicit failure state if next-style states are used. We also give the *cyclomatic complexity* [McC76] of each QEA, which measures the complexity of the transition structure in the QEA and is given as

$$|\{(q_1, \mathbf{b}, g, \gamma, q_2 \in \delta \mid q_2) \notin \mathsf{StrongF}\}| - |Q \backslash \mathsf{StrongF}| + 2|F|$$

where $\mathsf{StrongF}$ is the set of strong failure states as defined in Sec. 6.1.2. Therefore, cyclomatic complexity gives the number of linearly independent paths to an accepting state.

Property workloads

For each QEA property we design a suitable workload parameterised by the number of values for each quantified variables and the number of events. These workloads are designed to spread events related to each binding across the trace and randomise the paths taken through the QEA where appropriate. All but the NestedCommands workloads use a set of persistent objects to represent the objects being monitored. NestedCommands creates commands in a recursive

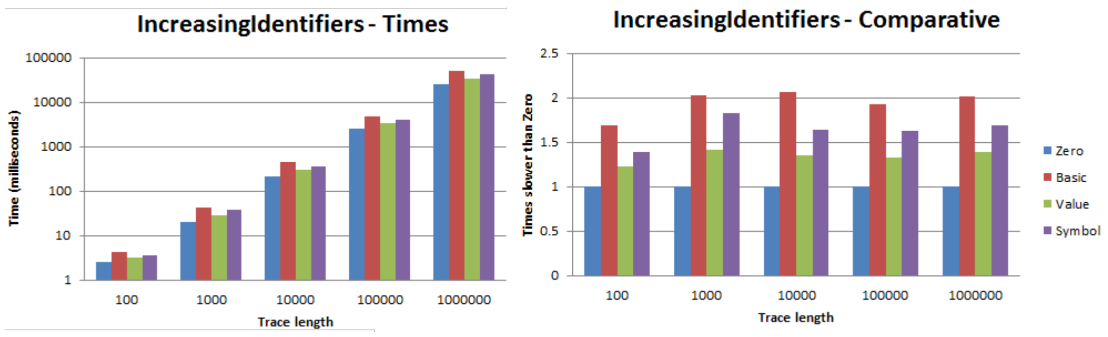


Figure 7.1: The results of RQ1 with zero quantifiers for IncreasingIdentifiers.

structure and the object representing a command goes out of scope shortly after that command succeeds. To evaluate the effects of different redundancy elimination techniques we create custom workloads, described later. Appendix C.2 describes the workloads further.

7.1.3 RQ1: Comparing indexing approaches

We compare Zero, Single and Disjoint with Basic, Value and Symbol.

Zero quantifiers

The IncreasingIdentifiers QEA has zero quantified variables and can be monitored using Zero. We compare our strategies across a range of workloads (from 100 to a million events). The results are given in Fig. 7.1; we also show how much slower other techniques are compared to the Zero strategy. We can see that the Zero strategy performs better than the other approaches, running between 1.5 and 2 times faster than the basic monitoring approach. Symbol performs slightly better than Value as in this case its book-keeping overhead is smaller. Zero processes events at roughly one every 0.024 milliseconds, or 42k events a second.

Single quantifier

We have five QEA's with a single quantifier in our set of properties: ResourceLifecycle, ExactlyOneSuccess, ExistsSatelliteS, GrantCancelS and RespectConflictsS. We inspect ResourceLifecycle in detail and then review the results for the other properties.

The workload for the ResourceLifecycle consists of randomly producing e events for r resources. The graphs in Fig. 7.2 give the results for the different cases of r/e. We give both the running time (in milliseconds) and compare the Single approach to the other strategies. The basic monitoring approach is substantially slower than the indexing techniques, running 170 times slower than the single quantifier indexing approach for 5k resources over 10k events. Single performed the best, with the symbol-based strategy almost performing as well. Value performed very badly when the number of resources was very large; in the 5k/10k case, where on average every second event introduces a new value, Value was 4 times slower than Single. This is due to Value needing to iterate over the set of existing bindings when adding a new

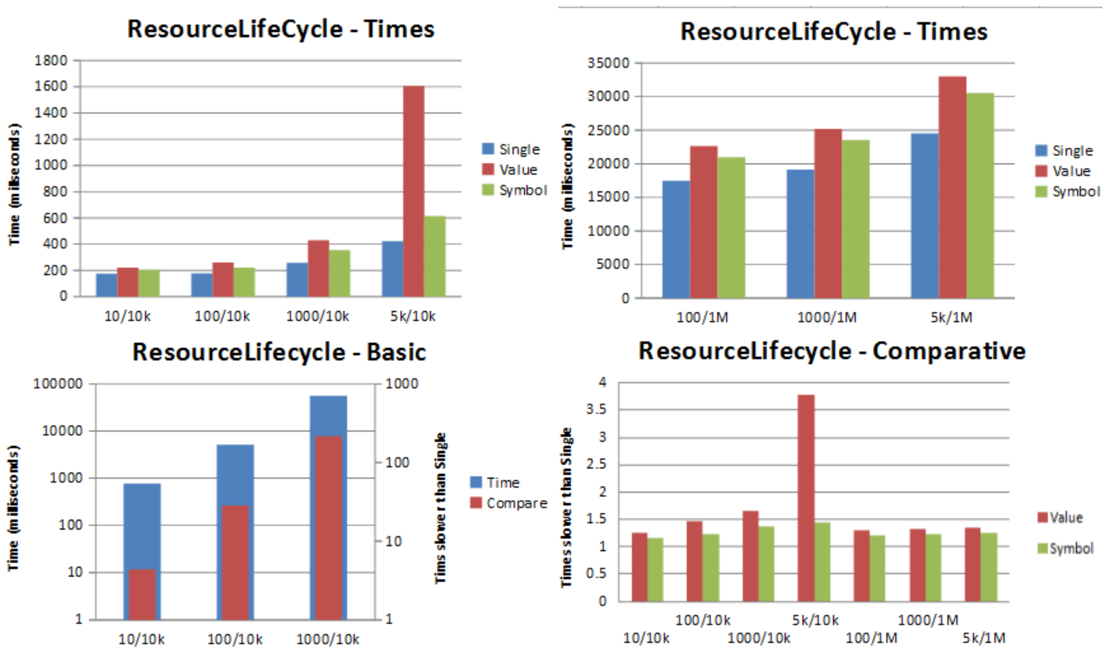


Figure 7.2: The results of RQ1 with single quantifiers for ResourceLifeCycle.

Table 7.2: Comparing the Single approach with other strategies.

Property	Single is X times faster than									Throughput	
	Basic			Value			Symbol			ms/	events/
	min	avg.	max	min	avg.	max	min	avg.	max	event	sec
ResourceLifeCycle	4.3	148	475	1.3	1.7	3.8	1.2	1.3	1.4	0.024	46k
ExactlyOneSuccess	3	37	96	1.5	3.4	7.6	1	1.5	1.7	0.036	28k
ExistsSatelliteS	2.4	42	102	1	1.4	1.7	1.6	1.8	2.1	0.038	26k
GrantCancelS	8.4	46	121	2	2.7	5.1	2.3	3	3.6	0.012	103k
RespectConflictS	7.6	15.7	27	5.7	9.7	13.3	1.3	1.6	2	0.007	144k

binding. On average compared to the basic monitoring strategy, Single performed 148 times faster, Value ran 59 times faster and Symbol ran 105 times faster.

Tables 7.2 records the average speedup achieved using the Single strategy for each QEA, we give detailed analysis of the other properties in Appendix C.2.2. The Symbol approach is the closest in performance to the Single approach and we achieve large speedups compared to the Basic approach. Table 7.2 also gives the average milliseconds per event and events processed per second for each QEA using the Single strategy. The GrantCancelS and RespectConflictS have far better throughput as the value reuse rate is far higher than with the other properties.

In summary, the Single strategy consistently performed much better than alternative strategies whenever it was applicable.

Disjoint alphabet

There are two QEAs with disjoint alphabets: ExistsSatellite and HashCorrect. Note that ExistsLeader is not suitable as it is not a normal QEA (see page 134 for a discussion).

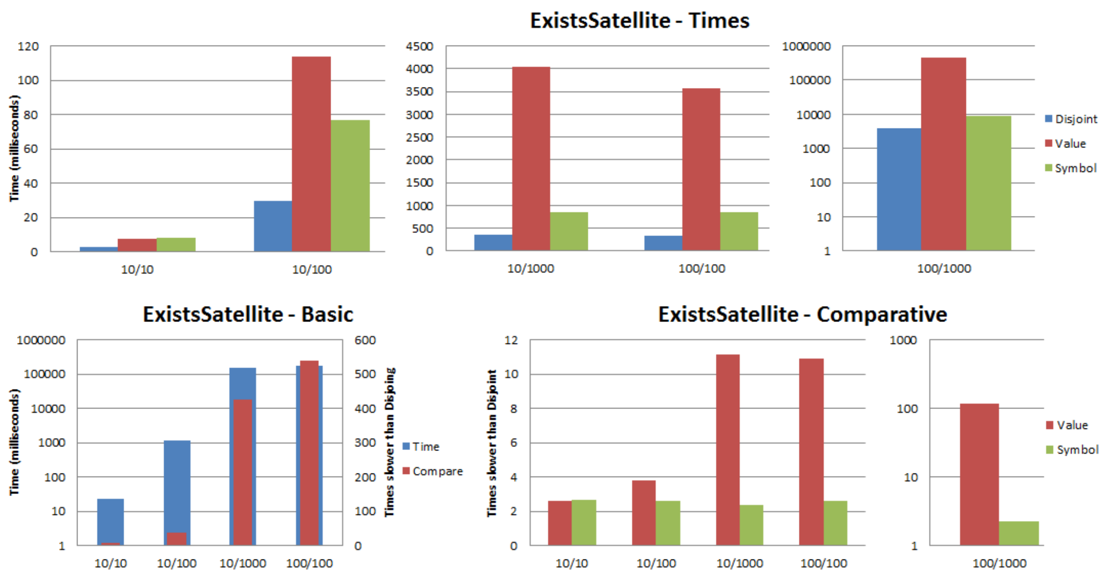


Figure 7.3: The results of RQ1 for ExistsSatellite.

Table 7.3: Comparing the Disjoint approach with other strategies.

Property	Disjoint is X times faster than									Throughput	
	Basic			Value			Symbol			ms/	events/
	min	avg.	max	min	avg.	max	min	avg.	max	event	sec
ExistsSatellite	8	253	540	2.6	29	118	2.3	2.5	2.7	0.034	30k
HashCorrect	6.8	248	650	2.3	21	97	2.8	3.2	4.2	0.035	30k

Let us consider the ExistsSatellite QEA. The workload for this QEA consists of randomly selecting a non-empty set of satellites to acknowledge each rover and producing the appropriate `ping` and `ack` events. The results are given in Fig. 7.3. Here we can see that the disjoint alphabet indexing approach far outperforms all other approaches. We note that roughly half of the bindings produced are disjoint. Value performs badly as it is having to consider many irrelevant bindings frequently, i.e. when introducing a new binding. On average, Symbol takes 2.5 the time of the disjoint alphabet approach, whereas Value takes almost 30 times as long. On average compared to the basic monitoring strategy, Single performed 253 times faster, Value ran 25 times faster and Symbol ran 101 times faster.

Table 7.3 records the average speedup achieved using the Disjoint. Again, we analyse the HashCorrect QEA in Appendix C.2.2. This time the Value strategy performed very poorly due to many irrelevant bindings being created and added. The Symbol strategy avoids this as it indexes using events instead of bindings. Table 7.3 also gives the average milliseconds per event and events processed per second for each QEA using the Disjoint strategy. This is roughly similar to the Single strategy as the mechanisms involved are similar.

In summary, the Disjoint strategy consistently performed much better than alternative strategies whenever it was applicable.

Table 7.4: Average speedup for Value and Symbol.

Property	Value	Symbol	Difference ($\frac{\text{Symbol}}{\text{Value}}$)
IncreasingIdentifiers	**1.44**	1.19	0.82
ResourceLifeCycle	59.54	**105.84**	1.78
ExactlyOneSuccess	15.06	**22.64**	1.5
ExistsSatelliteS	**29.87**	23.74	0.79
GrantCancelS	14.86	**15.10**	1.02
RespectConflictS	1.56	**9.40**	6.02
ExistsSatellite	25.30	**101.67**	4.02
HashCorrect	29.73	**70.67**	2.38
GrantCancelD	34.70	**76.42**	2.2
ReleaseResource	9.39	**31.78**	3.38
RespectConflictsD	12.98	**14.81**	1.14
RespectPriorities	1.59	**2.73**	1.72
NestedCommands	6.36	**8.28**	1.3
CommandsAcks	**0.99**	0.61	0.62
ExistsLeader	**16.67**	14.96	0.9

Summary

We have shown that Zero, Single and Disjoint are always more efficient than general indexing or basic monitoring where they are applicable.

7.1.4 RQ2: The effects of structural properties

Here we consider how structural properties outlined in Table. 7.1 effect monitoring times.

A summary of all experiments

We review how the Value and Symbol strategies perform on our properties, including those discussed previously. Table 7.4 reports the speedup (with respect to the Basic strategy) for the Value and Symbol strategies. On average the Symbol strategy achieved almost 2 times better speedup than the Value strategy. Overall Symbol performs best in 11 cases and Value performs best in 4 cases. Appendix C.2.3 gives further details.

The CommandAcks QEA is interesting as, on average, we see no speedup using indexing approaches. The workload consists of c commands being sent and either acknowledged or resent. The results for different c are given in Fig. 7.4. The Value strategy only just outperforms the basic strategy and for 10 commands the basic strategy runs the fastest as we only see 20 events and the overhead for Value and Symbol dominates. Symbol is consistently outperformed by the basic monitoring approach as most events introduce a new binding. Therefore, the common case that the two indexing strategies optimise for (the reuse of data values) is not seen.

Quantified variables

We consider the three properties in our set that have versions using both one and two quantified variables where the first is constructed from the second using the quantifier stripping 'trick'. We compare the monitoring times for different indexing strategies for the two different versions. We

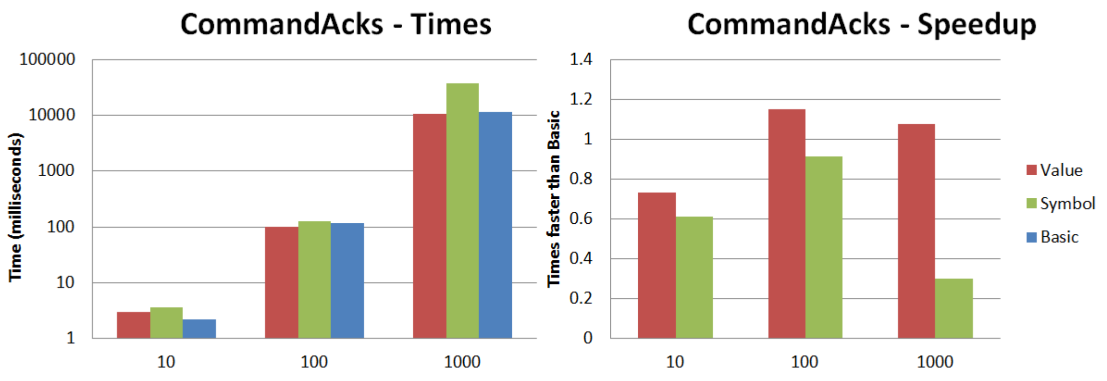


Figure 7.4: The results of RQ1 for CommandAcks.

expect the single quantifier versions to be monitored more efficiently, which is the case, and we present our results in terms of how much faster monitoring these versions is. Fig. 7.5 compares the running times for each version. In the case of ExistsSatellite we also include results from the Disjoint strategy i.e. we compare the Single strategy on the single quantifier version with the Disjoint strategy on the double quantifier version.

For ExistsSatellite the Disjoint strategy with two quantifiers is more efficient than the Single strategy with one as the additional computation required when removing one quantifier makes no impact when we use the Disjoint strategy, which is an adaptation of the Single strategy. In this case the quantifier stripping trick was not effective.

In the two other cases the single quantifier versions are always more efficient i.e. for all strategies. The Basic and Value strategies performed worst; this is expected as the additional bindings mean that the strategies have large structures to search through. However, in the case of RespectConflicts with relatively short traces (10k events) we see the Value strategy vastly outperforming the Symbol strategy, but when we move to longer traces (1M events) we see the performance of the Value strategy decrease again. This is due to the higher value reuse rate in both cases i.e. the same number of bindings are used, only reused more often. This shows that the Value strategy is very sensitive to reuse rate, where the Symbol strategy is not.

The RespectConflicts graph demonstrates a link between the size of domain and the slow-down caused by an additional quantifier for Basic. Larger domains leads to more bindings, leading to larger data structures to search through on each event. This effect is multiplied with additional quantified variables, as suggested in Sec. 4.3.1 and demonstrated here.

In summary, the number of quantified variables has a drastic impact on efficiency and any possible action to reduce the number of quantified variables should be taken. This observation applies to any trace-slicing based approaches, including JavaMOP.

Size of alphabet and transition complexity

The graphs in Fig. 7.6 plot speedup against alphabet size and transition complexity,the RespectPriorities QEA is not included.

Firstly, let us consider alphabet size. If we look at the graph plotting alphabet size against speedup we can see a weak negative correlation, which is stronger for the Symbol strategy.

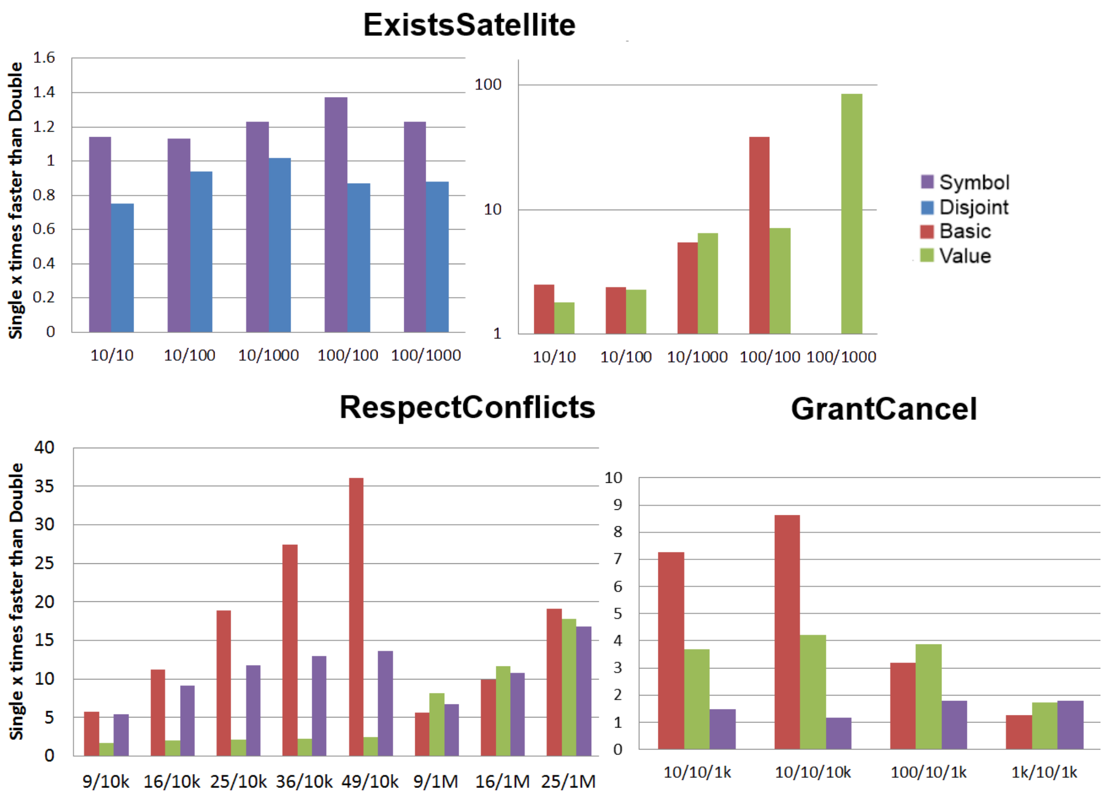


Figure 7.5: Comparing versions of properties with one and two quantifers.

Recall that this measures speedup with respect to the Basic strategy, i.e. not using indexing. The Symbol strategy is more sensitive to alphabet size as it uses the alphabet to construct the index and the time taken to search and update this index is related to the size of the alphabet. The two data points in the top right represent an outlier and are for the ResourceLifecycle QEA. Here the single quantifier compensates for the larger alphabet. There does not appear to be as strong a connection between transition complexity and speedup.

Table 7.4 shows that the Value strategy performs best with small, simple properties. However, there are exceptions: ExistsLeader performs best with Value whilst ExistsSatellite performs far better with Symbol even though though they have the same structure. The difference between these two properties are the quantification alternation, which would have limited effect, and the workloads. We have a far greater reuse rate with ExistsLeader, which is a trace property we have already highlighted as being important for the Value strategy. This suggests that trace properties are often more important than structural properties.

The RespectPriorites QEA has the largest alphabet and transition complexity and makes extensive use of guards and assignments. Therefore, the time to process each event is greatly increased, as the large transition set must be searched and guards and assignments evaluated.

In summary, whilst the size of alphabet and transition complexity of a QEA do effect monitoring time, they do not appear as important as other factors, such as the number of quantified variables or the value reuse rate in the trace.

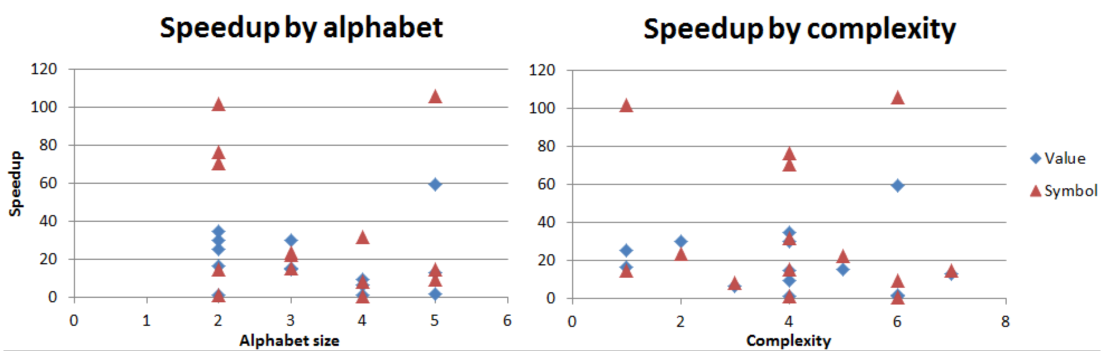


Figure 7.6: Comparing speedup with alphabet size and transition complexity.

Trace properties

Let us return our attention to the ResourceLifecycle QEA discussed on page 159 with graphs given in Fig. 7.2. If we consider the *reuse rate* for resources we can confirm that the performance of the Value strategy is strongly related to the ratio of times a resource is reused. This is consistent with our statement that Value optimises the common case where the values have been seen previously (page 146).

r	e	reuse rate	events/sec
10	10k	1000	44,843
100	10k	100	38,314
1000	10k	10	23,266
5k	10k	2	6,223

r	e	reuse rate	events/sec
100	1M	100k	44,193
1000	1M	10k	39,717
5k	1M	2k	30,285

The events per second is given for Value. We see that it is at its worst when the reuse rate is 2. We also see that the number of bindings can also effect event throughput, supporting our claims in Section 4.3.1.

Summary

The number of quantified variables in the QEA and the value reuse rate in the trace are the two most important structural factors for determining efficiency.

7.1.5 RQ3: Redundancy elimination

We now consider the two redundancy elimination optimisations: removing bindings that have trivial projections and removing bindings whose values become ‘garbage’.

Trivial projection redundancy

Only two of our properties have usable trivial events: ExactlyOneSuccess and ReleaseResource. The other properties’ trivial events are not usable as they match with non-trivial events, and some have no semantically correct trace that would introduce a trivial event. The trivial events for each QEA are given in Appendix C.2.

Table 7.5: Speedup for trivial projection redundancy elimination on custom workloads.

ExactlyOneSuccess			ReleaseResource		
Workload	Symbol	Value	Workload	Symbol	Value
10	0.8	1.33	5/5/100	1.23	13.17
100	0.88	2.78	5/5/1000	1.04	42.14
1000	1.1	5.66	10/10/100	1.44	27.18
10000	1.2	3.12	10/10/1000	1.15	66.28

To evaluate the effects of removing redundant events we construct workloads for Exactly-OneSuccess and ReleaseResource that for half of the events introduce a redundant command. Table 7.5 summarises the speedup achieved using the trivial projection redundancy filtering technique. This shows that redundancy elimination is far more effective for Value as the number of created bindings effects this strategy the most. In cases where we have relatively few bindings the cost of redundancy checking is not effective for Symbol, but for reasonably sized workloads we see a positive effect. The effects are greater for ReleaseResource as we have three quantifiers, which amplifies the effects of introducing a new command value.

In summary, this redundancy elimination can be very effective but there may be limited scenarios in which it is useful. One case where it will be useful is where we have looping transitions on the initial state (or it is a skip state) for events whose orderings are only prescribed after some other event occurs. For example, in UnsafeIter an update to a collection should be ignored unless it occurs after the creation of an iterator.

Garbage redundancy

To explore this redundancy elimination method we use the NestedCommands and the ResourceLifecyle properties with updated workloads. We should note here that the rest of the other evaluations in this section have (with the exception of NestedCommands) used workloads without garbage, meaning that bindings are never removed. We test garbage frequency of every 1, 10, 100 and 1000 events.

In Figure 7.7 we see the results for NestedCommands, this shows that different workloads achieve greatest speedup with different collection frequencies and indexing strategies. For the Value strategy we see a frequency of every one event achieving the greatest speedup with complex workloads. This is because the majority of events require us to iterate over the main data structure. For the Symbol indexing strategy a frequency of 100 achieves the greatest speedup. The speedup with Symbol is much lower than with Value as the effects of garbage are less. In the case of Symbol the overhead of checking for and clearing garbage only becomes worthwhile when there is enough garbage to clear.

The ResourceLifecyle workload produces a less garbage than NestedCommands as it uses one resource object at a time and then lets it go out of scope; the full results are reported in Appendix C.2.4. In this case we see a frequency of 100 performing well across both strategies for the less complex workloads. However, with the more complex workloads we see a frequency of 1 working best for Value again and a frequency of 10 performing well for Symbol. The most likely explanation here is that with longer running experiments the number of resources that

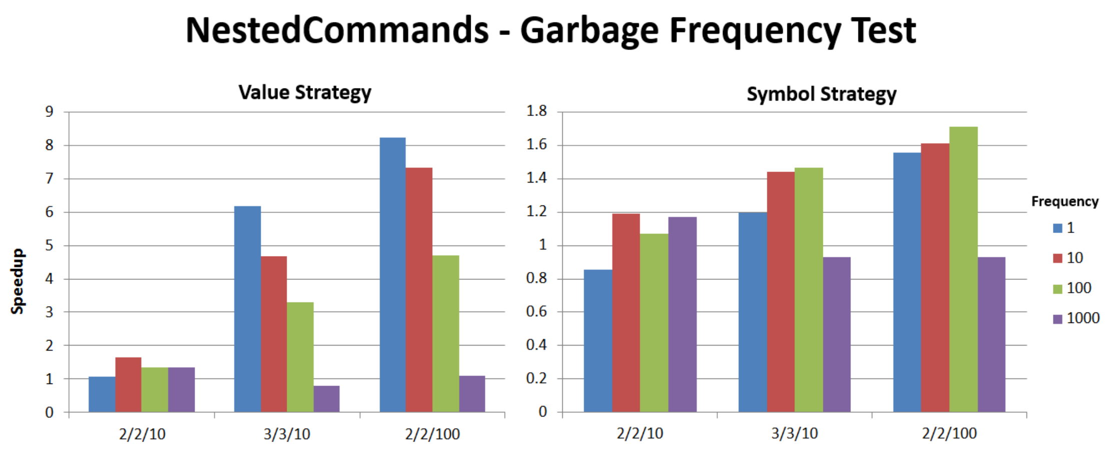


Figure 7.7: Garbage frequency test with NestedCommands.

become garbage over the lifetime of the experiment increases, therefore the likelihood that there will be garbage to remove increases.

In summary, garbage removal is very important and should be frequent. Further work might explore methods by which the frequency can be altered automatically. But it is evident that checking for garbage is not detrimental in long running programs.

7.1.6 RQ4: Comparison with JavaMOP and RuleR

We compare QEA with JavaMOP and RuleR for Expressiveness, Elegance and Efficiency. The implemented properties for these tools can be found in Appendix C.1.

Expressiveness

As RuleR is very expressive it can capture all of our properties. However, as QEA are implemented as an internal DSL in `Scala` we are able to use an arbitrary hash function in the HashCorrect property and as RuleR uses an external DSL we do not have that option. To overcome this in these experiments we use a simple 'hash' function that just takes the negative of the number. RuleR would not be able to compute arbitrary hash functions, although we note that whether a different implementation of QEA would depend on the expressiveness of the guard and assignment languages.

The JavaMOP tool is less expressive, yet can be made to capture half of the properties in this case study, in some cases by using extra code. We cannot capture the IncreasingIdentifiers, CommandAcks, HashCorrect nor RespectPriorities as they use free variables and in some cases at least one event that contains no quantified variables. We cannot capture ExistsSatellite nor ExistsLeader as they make use of existential quantification. All of these properties could be captured using JavaMOP along with extensive additional coding but we feel that they would as easily be captured using pure `AspectJ` without JavaMOP.

For some of the properties we had to apply the following trick. As JavaMOP does not allow us to associate an event name with more than one list of parameters we rewrite, for example, $\texttt{grant}(t_1, r)$ and $\texttt{grant}(t_2, r)$ into $\texttt{grant}_1(t_1, r)$ and $\texttt{grant}_2(t_2, r)$. We have to apply this trick

Table 7.6: Comparing conciseness.

Property	QEA	JavaMOP	RuleR
GrantCancel	**13**	28/20	18
ResourceLifecycle	**15**	25/16	28
ReleaseResource	**14**	35/24	24
RespectConflicts	26	31/**22**	24
RespectPriorities	66	-	**31**
ExactlyOneSuccess	**12**	28/18	22
IncreasingIdentifiers	**12**	-	15
CommandAcks	**16**	-	24
NestedCommands	**15**	35/27	24
ExistsSatellite	**20**	-	21
ExistsLeader	**15**	-	31
HashCorrect	34	-	**17**

to GrantCancel, RespectConflicts and NestedCommands. The issue with this renaming of events is that they are both created by the same program events, which only becomes an issue when, for example, $t_1 = t_2$, as the two events belong to the same trace slice and the JavaMOP semantics does not allow for this. Therefore, in the JavaMOP versions of these properties we must add checks to ensure this is not the case, this is okay for these properties but requires us to add some code to the monitors.

Therefore, we were able to translate ResourceLifecycle, ReleaseResource and ExactlyOneSuccess into JavaMOP without additional code and were able to translate GrantCancel, RespectConflicts and NestedCommands by adding some extra code.

Unlike QEA and RuleR, the JavaMOP tool does not have the notion of weak verdicts; one can either return the standard boolean verdicts, or attach verdicts to states when using the finite state machine plugin. This means that we do not get the fine-grained verdicts for some properties (such as ReleaseResource) that we would get with the other two tools. However, as we can attach advice to different states we can take different actions (print different messages) on different kinds of error, which is not possible with QEA or RuleR.

Elegance

To give a measure of elegance we consider the issue of conciseness. Table 7.6 measures conciseness as number of lines of code needed to write the property (in a readable format) in the respective DSL. Where there is a choice we show the more efficient QEA. We give JavaMOP specifications with and without the event definitions as the other tools using separate `AspectJ` files to generate events.

This is a crude measure of conciseness, let alone elegance, but overall QEA specifications are shorter. Two obvious exceptions are RespectPriorities and HashCorrect. It turns out that the RespectPriorities requires us to communicate between trace slices. To do this we use free variables, however RuleR's parameterised rules are a for more elegant mechanism for capturing such behaviour. The QEA description of the HashCorrect property is long as it is parameterised by a hash function and then instantiated with a given hash function, which is functionality not provided by the RuleR tool. Generally, JavaMOP specifications are long as they combine

Table 7.7: Speedup (times faster) for QEA over JavaMOP and RuleR.

Property	QEA Basic			QEA Best		
	simple	complex	average	simple	complex	average
JavaMOP						
GrantCancel	0.01	0.002	0.005	0.08	0.09	0.16
ResourceLifecycle	0.001	0.0001	0.0003	0.005	0.005	0.005
ReleaseResource	0.008	0.002	0.02	0.08	0.06	0.32
RespectConflicts	0.007	0.005	0.005	0.06	0.06	0.06
ExactlyOneSuccess	0.005	0.0001	0.003	0.02	0.01	0.02
NestedCommands	0.17	0.07	0.12	0.9	0.64	0.77
RuleR						
GrantCancel	0.149	**1.03**	0.338	**1.25**	**246.7**	**65.0**
ResourceLifecycle	0.114	**5.22**	**1.36**	0.5	**2433.5**	**537.0**
ReleaseResource	0.011	0.001	0.005	0.112	0.07	0.073
RespectConflicts	**2.97**	**291.3**	**83.0**	**23.17**	**7998.3**	**2009.5**
ExactlyOneSuccess	0.33	**1.31**	0.63	1	**126.1**	**43.3**
IncreasingIdentifiers	0.076	0.04	0.064	0.128	0.082	0.124
CommandAcks	**1.06**	0.2	0.5	0.77	0.22	0.43
NestedCommands	0.005	0.0002	0.002	0.028	0.002	0.012
ExistsSatellite	0.313	0.079	0.083	1	**9.1**	**3.38**
ExistsLeader	0.181	0.111	0.12	0.555	**3.72**	**1.74**
HashCorrect	0.051	0.005	0.022	0.35	**2.36**	**1.1**

instrumentation with the specification but even with these removed the additional code required to implement the specifications make them longer.

Efficiency

We compare monitoring time of JavaMOP and RuleR with that of the QEA strategy selected by the algorithm selection routine for each property. We choose the more efficient QEA representations where there is a choice i.e. single quantifier.

Table. 7.7 reports the speedup (number of times faster) given by QEA, where this is less than one the other tool performed better i.e. if the speedup is 0.5 this is a 2 times slowdown, if it is 0.01 this is a 100 times slowdown.

The JavaMOP tool is much more efficient for the properties that it can express. The best QEA algorithm typically runs a few hundred times slower. We only beat JavaMOP once, when monitoring the ReleaseResource property when we have a large number of objects with respect to the number of events. One reason JavaMOP runs so much faster is that it weaves the monitoring code directly into the monitored code, whereas QEA is implemented as an external monitor. This means that calls to the monitor are often inlined by the `Java` just-in-time compiler and as the majority of running time is monitoring time these small differences have a large impact. The QEA monitor also creates a new object per event, which JavaMOP avoids. We can learn a lot from the engineering decisions of JavaMOP.

For RuleR we note that there appears to be a divide between the specifications: those where QEA greatly outperforms RuleR and those where RuleR greatly outperforms QEA. Let us consider the case where QEA performs best. Firstly, we have the singly quantified properties that make use of the efficient Single strategy: GrantCancel, ResourceLifecycle, RespectConflicts, ExactlyOneSucces and ExistsSatellite. In this case we target the source of RuleR's efficiency issue. Then we have ExistsLeader and HashCorrect. In both cases QEA's Symbol indexing strategy of QEA proves more efficient than the linear search method of RuleR. RuleR outperforms QEA for one reason; it can actively remove bindings from consideration, thus maintaining a small set of activations and allowing its linear search to be effective. In the case of IncreasingIdentifiers both systems track a small amount of information, but the additional matching and checking mechanisms of QEA add unnecessary overhead. For the other cases we have complex quantification where QEA must keep track of many bindings. In the case of NestedCommands we saw RuleR running hundreds of times faster than QEA as each command only lasts for a few events, yet QEA keeps track of them and combines them with other commands. RuleR also beats JavaMOP for ReleaseResource and NestedCommands.

7.1.7 Discussion

In this section we have used an artificial case study to analyse the performance of monitoring algorithms for QEA and answer a number of research questions. Overall we showed that our monitoring techniques are efficient and that the optimisations we made were worthwhile. The Symbol indexing strategy performed the best in cases where specific structural-based indexing strategies did not apply. We briefly review the conclusions of our four research questions.

RQ1 We compared the Zero, Single and Disjoint strategies against the Basic strategy and the general strategies of Value and Symbol. In all cases where this specialised strategies applied they performed the best. In some cases we saw these strategies performing hundreds of times faster than Basic, tens of times faster than Value and 2-3 times faster than Symbol.

RQ2 We examined the relationships between structural properties of the QEA and monitoring efficiency. We found, as expected, that more quantified variables leads to greater monitoring overhead. We concluded that alphabet size and transition complexity played a role, but are not as important as trace properties such as object reuse rate.

RQ3 We demonstrated that both of our redundancy elimination techniques were effective. Trivial projection redundancy elimination may be of limited applicability but, as is confirmed in the next section, garbage redundancy elimination is very important.

RQ4 We compared QEA with JavaMOP and RuleR along the three dimensions of Expressiveness, Elegance and Efficiency. We showed that QEA is as expressive as RuleR and strictly more expressive than JavaMOP, and that QEA is the most concise (using one crude measure of conciseness). We found that JavaMOP was far more efficient than QEA for the properties it could express and that RuleR was very efficient for some small properties, but far less efficient than QEA for complex properties.

7.2 Evaluating with the DaCapo benchmark suite

This section uses a collection of real world applications (the DaCapo benchmark suite) to evaluate the applicability of the QEA monitoring algorithms. The aim is to firstly establish that our technique can be applied to real world applications, and secondly to compare the performance to existing tools JavaMOP and RuleR.

We begin by introducing the experimental setup. We then look at the HasNext property in detail, inspecting the errors found and discussing the performance of our algorithms with respect to the other tools. Finally we present and discuss our results for the remaining properties.

7.2.1 Experimental Setup

We introduce the the monitored applications and the monitored properties. We use the experimental machine introduced earlier and restrict all experiments to two hours.

The DaCapo suite

The DaCapo benchmark suite [BGH+06] was originally designed to evaluate the performance of Java Virutal Machines. It has been adopted as a useful benchmark suite for evaluating program analysis techniques and is often used to evaluate a runtime verification tools i.e. [MR10, MJG+11, BHL+07, BLH10]. Table 7.8 describes the programs that make up the suite, giving the number of threads used and the time it takes to run on our experimental machine (described earlier on page 156). In some cases programs use all available hardware threads, which is 16 in our case. The programs included in the suite were chosen to represent a range of workloads, however are all relatively short-running. We use version 9.12 of the benchmark suite with default workloads.

The DaCapo benchmark suite has a number of parameters to control how the individual programs are ran and how results are collected. We use the `-converge` option with `window -5` which means that the program should be repeated until the running time is within 3% of the mean running time over the last 5 runs. All running times reported in this section are the average of 5 runs. Note that this means we are monitoring multiple runs of a program using the same monitor. Systems that use garbage collection should be able to effectively reset data structures as all objects will become unreachable. We also use the `--no-validation`, `-preserve` and `-scratch-directory` options to allow us to print information during monitoring.

Some programs are multi-threaded, although not all threads will exhibit monitored behaviour. We give a rough indication of the level of concurrency as threads do not necessarily last for the full run of the system. All monitoring systems used have a single monitor lock to keep monitors thread-safe.This single-lock can have the effect of adding significant overhead to multi threaded programs in the form of waiting time. A number of multithreaded benchmarks use different classloaders to create threads, which means that a separate monitor will be created per thread as new `AspectJ` instances will be created in each classloader. We treat these threads separately where they occur.

The DaCapo benchmark suite has been chosen as it has been used previously to evaluate runtime verification tools and represents a range of applications. However, we note that the

Table 7.8: The DaCapo benchmarks.

Name	Description	Threads	Time (ms)
avrora	A set of simulation and analysis tools in a framework for AVR micro-controllers.	~23	5823
batik	An SVG toolkit produced by the Apache foundation. The benchmark renders a number of svg files.	2	1613
eclipse	Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.	~1	27317
fop	Parses it and formats an XSL-FO file, generating a PDF.	1	517
h2	Is an in-memory database benchmark.	> 32	7209
jython	Interprets the pybench Python benchmark.	~1	5757
luindex	Uses lucene to index a set of documents; the works of Shakespeare and the King James Bible.	~6	853
lusearch	Uses lucene to do a text search of keywords over the works of Shakespeare and the King James Bible.	~16	1425
pmd	Analyzes a set of `Java` classes for source code problems.	~16	2722
sunflow	A raytracing rendering system for photo-realistic images.	~16	2241
tomcat	Uses the Apache Tomcat servelet container to run some sample web applications.	~16	1878
tradebeans	Runs the Apache daytrader workload ”directly” (via EJB) within a Geronimo application server.	> 30	10211
tradesoap	Identical to the tradebeans workload, except that client/server communications is via soap protocols.	> 30	17313
xalan	Transforms XML documents in HTML.	~1	990

range of monitored behaviours only represents low-level properties found in `Java` programs. Ideally, more extensive benchmarks would be developed, but this is beyond the scope of this work. The interested reader should refer to the international runtime verification competition starting in 2014, which aims to collect a larger range of benchmarks and properties.

The Properties

Table 7.9 describes the specifications used in this evaluation, taken from our specification of the `Java` Standard Library in Appendix A.3. Some of these specifications have been used in previous examples. All properties, apart from LockOrdering, property can be described using the less expressive JavaMOP system. These are relative simple specifications only using universal quantifications and only using guards and assignments to reduce complexity (i.e. they are not necessary). Appendix C.1 gives the JavaMOP and RuleR versions of these properties.

7.2.2 The HasNext Property

Let us first consider the HasNext property, a canonical example for runtime monitoring.

Understanding the traces

We look at two properties of the traces produced by the monitored programs. Firstly, the number of occurrences of each event, and secondly the amount of time an iterator object is ‘alive’. These properties should indicate the level of overhead expected during monitoring.

Table 7.10 gives the number of `hasNext` events returning true or false and the number of

Specification	Description	Fig/page
HasNext	Every call of `Java`next must directly be preceded by a call to `Java`hasNext returning true.	3.2 / 61
UnsafeIter	Do not update a `Java`Colletion when using the `Java`Iterator interface to iterate its elements.	A.16 / 344
UnsafeMapIter	Do not update a `Java`Map when using the `Java`Iterator interface to iterate its values or keys.	A.17 / 344
UnsafeSyncCollection	If a `Java`Collection is synchronized, then its iterator also should be accessed synchronously.	A.20 / 346
UnsafeSyncMap	If a `Java`Map is synchronized, then its iterators on values and keys also should be accessed synchronously.	A.21 / 346
UnsafeFileWriter	Do not write to a `Java`FileWriter after closing.	A.12/ 341
LockOrdering	If two locks are taken in a certain order they should not be taken in the opposite order elsewhere.	A.22/ 347
ConsistentHashes	If an object is placed in a `Java`HashSet or as a key in a `Java`HashMap then its `Java`hashcode should not be updated until it is removed.	A.18/ 345
CloseFiles	If a file is opened in a method then it should be closed in the same method, this is a software development good practice.	A.14/ 342

Table 7.9: Specifications of behaviour found in the `Java` library used for evaluation.

`next` events occurring in a *single run* of each benchmark. We can assume that an iterator only returns false for `hasNext` once and calculate the average number of iterations per iterator. Note that for a few benchmarks we have many events, but for others we have very few. Three benchmarks (h2, luindex and sunflow) have HasNext activity on separate threads created by different class loaders (see above). Table 7.10 gives the total number of events across all threads, but the events per second and lifetimes are averaged across threads.

Table 7.10 also gives the average lifetime of an iterator object. This demonstrates the length of time an iterator will be present in the monitor if garbage collection is used. It should be noted that these times can be skewed by the act of recording them, however they should give a rough guide to the average size of data structures in the monitor. Note that the maximum lifetimes often far exceed the running times of the benchmark, as it is running in a JVM with the DaCapo harness and objects created whilst the benchmark ran may not be collected until after it has completed. Overall, we see many iterators not becoming garbage during the running time of the benchmark, the main exceptions being eclipse and jython.

Errors found

There were a number of violations of this property. We manually inspect all reported errors and report those results here. Table 7.11 gives the errors reported. We class an error as a real error if the method in which the reported error occurred does not prevent the error i.e. assuming that two external data structures are the same size is a real error, but calling next after checking that the data structure is non-empty is not a real error. The reasoning is that whilst there might be some class invariant that guarantees correct behaviour, small changes could break this contract and there are no local checks in place to ensure that they still hold. We give different results for different monitors when different errors are captured by each system. JavaMOP fails to

Table 7.10: The trace statistics for the HasNext property.

	Events					
	`hasNext` (true)	`hasNext` (false)	`next`	**total**	**mean iterations**	**events per ms**
avrora	247,026	908,978	350,547	1,506,551	0.39	259
batik	11,913	24,277	12,559	48,749	0.52	30
eclipse	80,989	4,346	60,472	145,807	13.91	5.3
fop	477,917	72,692	461,585	1,012,194	6.35	1,958
h2	61,710,546	61,710,216	36,521,060	159,941,822	1.69	4,437
jython	395,759	85,977	256,432	738,168	2.98	128
luindex	6,795	6,795	2,835	16,425	2.4	1.28
lusearch	256	128	256	640	2.00	0.4
pmd	3,947,524	804,529	3,640,109	8,392,162	4.52	3,083
sunflow	35,754,264	35,764,120	2,827,552	74,345,936	12.64	4,739
tomcat	0	0	0	0	-	-
tradebeans	0	0	0	0	-	-
tradesoap	0	0	0	0	-	-
xalan	0	0	0	0	-	-

	Lifetime (milliseconds)			Overall
	min	mean	max	runtime
avrora	2,335	25k ± 70k	63k	5,823
batik	222	6.5k ± 2.5k	11k	1,613
eclipse	239	6k ± 8k	29k	27,317
fop	1870	29k ± 29k	50k	517
h2	24	53k ± 17k	656k	7,209
jython	71	3k ± 1.5k	68k	5,757
luindex	730	1.1k ± 1.4k	1841	853
lusearch	11	523 ± 2k	3k	1,425
pmd	376	22k ± 157k	580k	2,722
sunflow	510	6k ± 28k	999k	2,241

capture any potential errors that stem from calling `next` after `hasNext` returns false. QEA fails to capture multiple errors for the same iterator due to the way the non-failing mode is implemented, it could be extended to support the same behaviour as JavaMOP i.e. removing the monitor and allowing it to be recreated. RuleR has the same restart issue as QEA but also gives false positives as mentioned previously.

In fop we see a few cases of custom iterator operations such as `nextIndex` and `previous` causing false positives. In one case the false positive was difficult to determine but was caused by caching in the custom iterator which meant that `hasNext` returned false before a valid call to `next`. The JavaMOP monitor did not catch this false positive as it is an incorrect behaviour not captured by their version of the property as they cannot capture free variables.

One source of false positives comes from non-empty checks on the collection being iterated as in this example, taken from `org.apache.batik.ext.awt.image.AbstractRable`.

```
if (this.srcs.size() != 0) {
    Iterator i = srcs.iterator();
    Filter src = (Filter)i.next();
```

Table 7.11: Errors reported when monitoring HasNext.

Property	Reported errors	Average repetitions	Real errors
avrora (JavaMOP,QEA)	1	103k	1
avrora (RuleR)	10	23k	1
batik	3	65	2
fop (JavaMOP,RuleR)	14	214k	8
fop (QEA)	20	97k	7
h2 (QEA)	4	8	0
h2 (JavaMOP,RuleR)	1	30	0
pmd (JavaMOP,QEA)	3	2k	1
pmd (RuleR)	7	3	0
sunflow (RuleR)	1	5	0

Sometimes the programmer has defended against the error. For example, in this code taken from `avrora.sim.radio.Medium$BasicArbitrator.mergeTransmissions` we see an assert statement being used.

```
public char mergeTransmissions(Receiver receiver,
                List it, long bit, int Milliseconds) {
    assert(it.size() > 0);
    Iterator i = it.iterator();
    Transmission first = (Transmission) i.next();
```

We find some real bugs, verified by manual inspection of the code. For example, this code taken from `org.apache.batik.gvt.renderer.StrokingTextPainter` assumes that `aciList` is not empty but inspection of the rest of the method could not verify that this will always hold, as it is dependent on external data structures.

```
// copy the text chunks into an array
AttributedCharacterIterator[] aciArray =
    new AttributedCharacterIterator[aciList.size()];
Iterator iter = aciList.iterator();
for (int i=0; iter.hasNext(); ++i) {
    aciArray[i] = (AttributedCharacterIterator)iter.next();
}
```

Another source of errors is where we iterate over two collections at the same time, as seen in `org.apahce.batik.bridge.SVGTextElementBridge`. Here two iterators are created from `strings` and `attributes` and the programmer has relied on the (unchecked) property that these collections are the same size. The call to `next` on line 7 is incorrect, however the call on line 12 is a false positive as it is necessarily the case that the key set and value set of a map are the same size.

```
1 Iterator sit = strings.iterator();
2 Iterator ait = attributes.iterator();
3 int idx = 0;
4 while (sit.hasNext()) {
5     String s = (String)sit.next();
6     int nidx = idx + s.length();
```

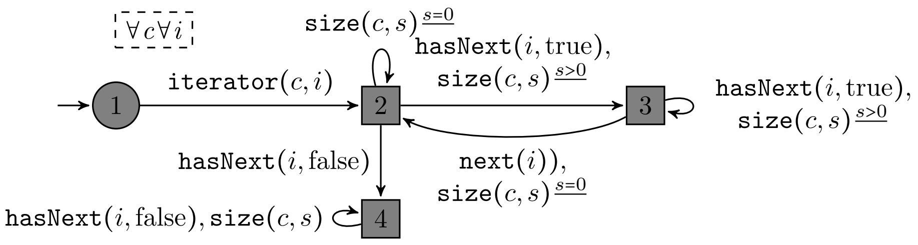

∀c∀i
size(c, s) s=0
hasNext(i, true), size(c, s) s>0
iterator(c, i)
1
2
3
4
hasNext(i, true), size(c, s) s>0
hasNext(i, false)
next(i)), size(c, s) s=0
hasNext(i, false), size(c, s)

Figure 7.8: A refined HasNext QEA

```
    Map m = (Map) ait.next();
    Iterator kit = m.keySet().iterator();
    Iterator vit = m.values().iterator();
    while (kit.hasNext()) {
        Attribute attr = (Attribute)kit.next();
        Object val = vit.next();
        result.addAttribute(attr, val, idx, nidx);
    }
    idx = nidx;
}
```

The fact that only 46% of identified errors are real errors demonstrates that programming rules such as HasNext might be too specific, and we might consider incorporating additional information. For example, if the size of the base collection is confirmed as nonempty and not updated before an iterator is created and used with `next`. Figure 7.8 captures a refined QEA that incorporates checking the size of a collection. However, there are some semantic behaviours, such as two collections being assumed to be the same size, which may be beyond the scope of runtime monitoring.

Timing results

Table 7.12 gives the average slowdown for each tool, the actual times are reported in Appendix C.3.2. The fastest tool for each benchmark is highlighted. In some cases the percentage overhead is less than one, this phenomenon has been previously reported and is due to instrumentation causing the code to undergo different optimisations by the JIT compiler or different thread orderings caused by synchronisation on the monitor.

On average QEA performed 3.94 times faster than JavaMOP and 22.63 times faster than RuleR but if we exclude fop QEA performs 1.06 times faster than JavaMOP and 1.43 times faster than RuleR. This is mainly due to the Single indexing strategy that is specialised for the case where we have only one quantified variable. As HasNext is a very simple property it is unsurprisingly that RuleR performs well generally. We consider the results more closely.

Firstly, let us consider those benchmarks that have relatively few or no events i.e. luindex, lusearch, tomcat, tradebeans, tradesoap and xalan. Slowdown is between 0.82 and 1.17. The last four properties have zero events and no instrumentation occurs and therefore slowdown can only be due to variants in the timing of the original program.

Finding errors can have a large impact on running time. RuleR performs well on avrora

Table 7.12: Slowdown results for HasNext

	Slowdown			Single compared to	
	QEA-Single	JavaMOP	RuleR	JavaMOP	RuleR
avrora	**1.99**	6.30	4.11	**3.16**	**2.07**
batik	**1.06**	1.32	2.15	**1.24**	**2.02**
eclipse	1.00	**0.99**	1.01	1.00	**1.02**
fop	**3.54**	146.51	981.75	**41.35**	**277.06**
h2	1.51	**1.06**	-	0.70	-
jython	0.56	**0.51**	0.78	0.92	**1.40**
luindex	1.11	**1.09**	1.12	0.98	**1.01**
lusearch	0.99	**0.96**	1.04	0.97	**1.05**
pmd	5.17	**2.27**	-	0.44	-
sunflow	2.33	**0.99**	10.23	0.42	**4.38**
tomcat	1.03	1.03	1.05	0.99	**1.02**
tradebeans	1.03	1.03	0.96	1.00	0.94
tradesoap	0.83	**0.83**	0.86	0.99	**1.03**
xalan	0.94	**0.93**	0.99	0.99	**1.05**

due to the false positives returned as we restart the monitor on every error, clearing the data structures. For fop QEA outperforms JavaMOP and RuleR, partly due to the errors identified as QEA will only report one error per iterator and then stop considering that iterator, but both JavaMOP and RuleR will continue to process these events. In batik and pmd the errors are infrequent with respect to the rest of the behaviour.

QEA sees a vast improvement for fop compared to the other monitors, which cannot only be explained by error handling. We should first note that fop is a short running program (507 milliseconds) and any overhead will lead to large slowdowns. It is also the case that the majority of iterators are live for the full length of the program. The most likely explanation is that the indexing structures making use of weak references used by JavaMOP added unneeded overhead as JavaMOP relies on quickly clearing its data structures to remain efficient.

The h2 and sunflow benchmarks were both monitored with multiple monitors as different parts were loaded by different class loaders. This spread the workload between monitors but we still saw a very high throughput. For sunflow each monitor handled 10M events on average, and for h2 it was almost 35M events per monitor. Sunflow has relatively long iterations so it is likely that JavaMOP's caching mechanism allowed it to achieve its very small slowdown. It is likely that the different threads are monitored by different monitors preventing synchronisation on the single monitor lock.

Both eclipse and jython have relatively small slowdown given the number of events. If we inspect Table 7.10 we see that the majority of iterators are short-lived, meaning that for these benchmarks data structures remain reasonably small. Figure 7.9 plots the slowdown overhead against the average Iterator lifetime as a percentage of benchmark running time, meant to indicate the level of garbage collected during the run. We see that as the average lifetime increases above 100% slowdown increases dramatically, and that slowdown is at its worst for JavaMOP when we have the most garbage. RuleR is a slight exception as it can directly remove rule activations without waiting for them to become garbage, so it is the number of simultaneously live iterators that effects its performance. This shows that the removal of

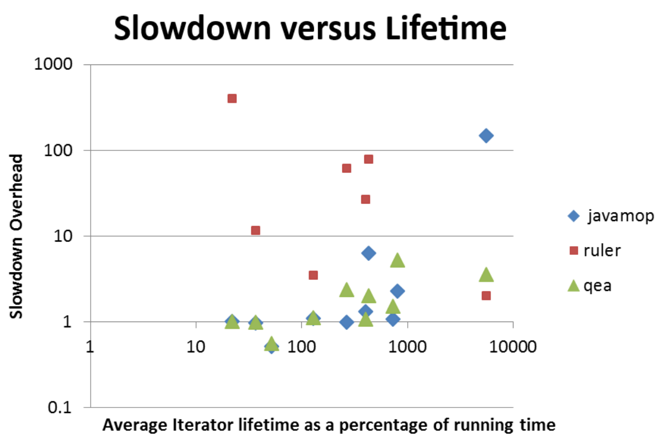


Figure 7.9: Comparing the average lifetime of iterators with overhead.

Table 7.13: Inside overhead and throughput whilst monitoring HasNext. We report the time spent inside the monitor in terms of percentage of total monitoring time and percentage of overall overhead, we also report the average events processed per millisecond.

	JavaMOP			RuleR			QEA-Single		
	% total	% over	e/ms	% total	% over	e/ms	% total	% over	e/ms
avrora	87	103	45	-	-	-	43	82	282
batik	23	56	78	46	77	26	7	20	265
eclipse	0	0	3639	0	3	235	0	1	323
fop	98	99	13	-	-	-	44	56	903
h2	24	195	2549	89	1.51	353	39	1.09	1116
jython	1	-4	8877	27	5.59	289	6	-0.28	1729
luindex	0	0	481	1	6	22	1	4	20
lusearch	0	2	179	5	23	7	5	20	7
pmd	38	56	2696	-	-	-	44	54	1284
sunflow	13	68	4216	39	44	165	21	35	1243

garbage is of key importance.

Additionally, we note that eclipse and jython are single-threaded benchmarks, suggesting that much of the overhead for other benchmarks is due to synchronization on the single monitor. We add additional instrumentation to track the time spent inside each monitor (i.e. holding the monitor lock). Table 7.13 reports the time spent inside the monitor as a percentage of the total monitoring time and of the overhead; where the overhead is negative this will also be negative. We also report throughput as time time spent inside the monitor divided by the number of events processed. From this we can see that it is often the case that a minority of the overhead is spent in the actual monitor; the rest will be a form of interference caused by synchronising on the monitor lock. For example, pmd has 16 threads and time spent inside the monitor makes up just over half of the total overhead time.

Summary

The HasNext property is often used as a representative example for runtime monitoring, however our analysis of errors found in the DaCapo benchmark suite show that it may be too simplistic

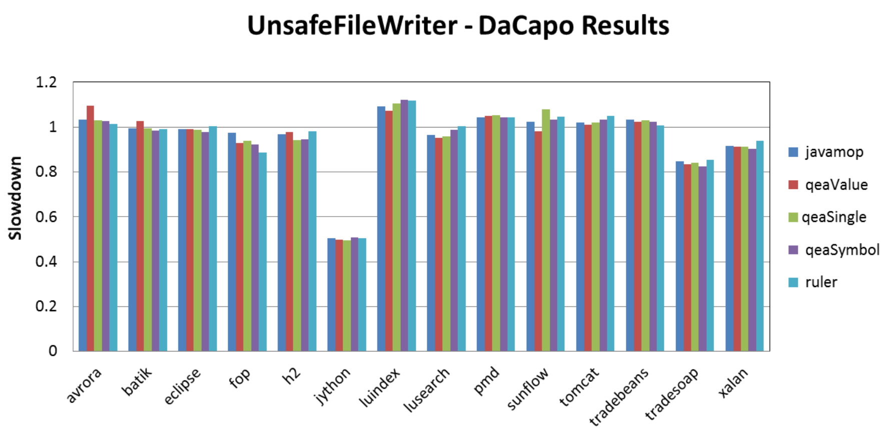


Figure 7.10: Overhead for monitoring the UnsafeFileWriter property.

as a reflection of common coding practice. Our Single indexing strategy allowed us to monitor this property very efficiently. Furthermore, we found that garbage collection is a *fundamental requirement* and that the monitor lock can cause negative interference.

7.2.3 Other properties

In this section we consider the remaining properties in slightly less detail. In some cases we group similar properties together. Full details are given in Appendix C.3.

UnsafeFileWriter

This is a very simple property that has been used previously in runtime monitoring experiments using DaCapo [MJG$^+$11]. However, we found that none of benchmarks make use of FileWriters. This allows us to examine the effects of monitoring an unused property. When we monitor an unused property we construct a monitor at the beginning and load the necessary classes into memory. We also pass the code through the `AspectJ` compiler to carry out instrumentation, which should not effect uninstrumented code.

Figure 7.10 gives the slowdown for all monitoring approaches used in our experiments. This shows that some properties are sped up by the small changes mentioned previously. This could be accounted for in our other discussions of slowdown. For example, if we take this as jython's baseline then we get positive (but small) slowdown for HasNext across all monitors. We see little variation in the overhead introduced by different approaches.

The connected properties

We consider UnsafeIter, UnsafeMapIter, UnsafeSyncCollection and UnsafeSyncmap together as they all consider objects created from other objects. These properties also demonstrate an issue with the current implementation of the QEA monitoring algorithms i.e. how they deal with garbage.

Table 7.14 summarises the average slowdown across all benchmarks for these four properties, for QEA this is somewhat skewed by a few properties that have hundreds of times slowdown. In

Table 7.14: Average slowdown for each property for each monitoring approach.

	QEA	JAVAMOP	RULER
UnsafeIter	110.65	1.64	4.02
UnsafeMapIter	135.29	1.29	1.72
UnsafeSyncCollection	103.99	1.12	-
UnsafeSyncmap	163.50	1.06	-

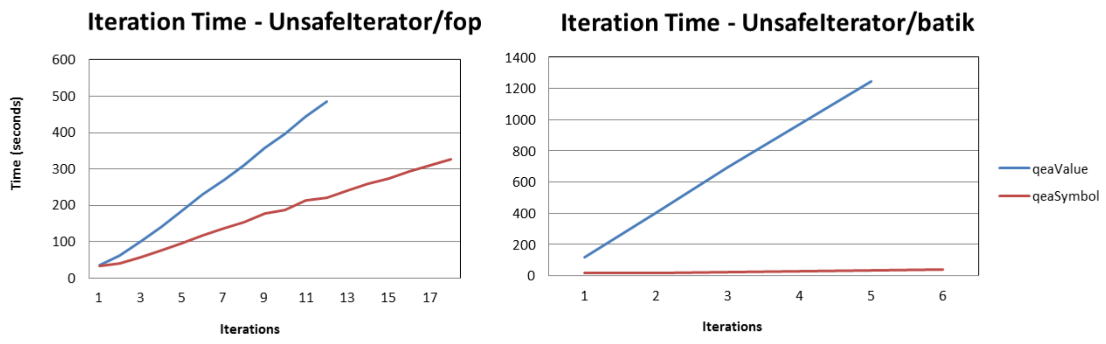


Figure 7.11: The garbage problem in UnsafeIter.

these cases QEA runs on average 100 times slower than JAVAMOP, and RULER. The issue here is that the garbage removal technique for QEA cannot deal with these connected properties properly. as we take all possible combinations of collections and iterators, keeping bindings alive necessarily as there is still an extension of the trace slice for that binding that could lead to failure. This means that data structures get very large. Figure 7.11 demonstrates this garbage issue. We show the increasing time it takes to complete an iteration in UnafeIter for both the Symbol and Value strategies for fop and batik respectively. On every iteration the time increases (we would expect it to decrease as JIT optimisations apply) as the monitors are storing more and more values. The problem is far worse with the Value indexing technique, compared to Symbol, as this approach has a larger index. RULER performs reasonably here as it has fine-grained control over when it creates rule activations, and many of the events can be ignored.

This is an issue that needs resolving to make our tool applicable to this class of properties. It should be possible as the underlying structure is similar to that of JAVAMOP.

ConsistentHashes

This property checks that objects stored in collections using hashing maintain their hashCode. We consider both `HashSet` and `HashMap`. We need to write a specification for each in JAVAMOP but can monitor both together in QEA. To implement the properties in JAVAMOP a small amount of programming is required, whereas QEA uses free variables. For JAVAMOP we need to add a variable to each monitor to store the previous hashcode. It should be noted that the use of `IdentityHashCode` to identify different objects is of key importance here.

Table 7.15 records the total number of events and slowdown per benchmark. We only include benchmarks with more than zero events, additionally an exception persistently occurred when

Table 7.15: Average slowdown for the ConsistentHashes property.

	Events		QEA-Disjoint	QEA-Symbol	JavaMOP	RuleR
	add	observe				
avrora	1,984	713,809	1.88	7.72	**1.03**	-
batik	10,872	23,130	1.24	2.19	**1.0**	866
eclipse	7,865	25,367	1.02	1.04	**1.01**	-
fop	29,735	359,969	97.9	41.0	**1.36**	2310
jython	10,626	3,146,865	2.1	10.88	**0.52**	-
luindex	218	124,565	1.91	5.52	**1.02**	452
lusearch	9,202	1,049,474	64.4	121.2	**2.0**	-
pmd	149,039	764,939			**0.59**	-
xalan	29,534	462,500	10.84	24.27	**1.25**	-

Table 7.16: Warnings when monitoring for the LockOrdering property.

	QEA		RuleR	
	Warnings	Frequency	Warnings	Frequency
avrora	6	8	13	34k
batik	0	0	1	35
fop	0	0	1	2
h2	≥16	≥94	≥31	≥ 307k
luindex	5	5	11	1.2k
lusearch	33	11k	21	30k
pmd	4	194	5	115
sunflow	8	11	8	35
xalan	2	2	18	240

monitoring h2 so it has been excluded. JavaMOP outperforms QEA, although we should note that it spreads events across two monitors, reducing the size of data structures. For QEA we see that performance degrades with the number of objects added, for example with fop and xalan. RuleR performs very badly for this property as it involves storing a large number of rule activations, and searching through them on each step. For lusearch the slowdown is amplified as it is multithreaded, with 16 threads all accessing hashing structures at the same time causing synchronisation on the single monitor lock.

Warning properties

The last two properties are different from the rest as it captures a desired rather than required property. Therefore we refer to reports of breaking this property as warnings rather than errors.

Let us first consider the LockOrdering property, which cannot be captured using JavaMOP. Some of the benchmarks produce a lot of events as they make heavy use of locking.

Table 7.16 captures all of the warnings given during monitoring. As before, RuleR returning false positives drastically reduces the number of locks being tracked, improving efficiency. This suggest a possible future work that eagerly discards bindings, trading soundness for efficiency. Benchmarks that use locking heavily, such as lusearch, receive many warnings. We have not explored these warnings; a possible exercise, which we have not explored, would be to check the warnings using standard techniques for detecting potential deadlocks.

Table 7.17: Average slowdown for the LockOrdering property.

	Total events	Live (mean/max)	QEA-Value	QEA-Symbol	RuleR
avrora	3,355,930	6.19/12	**91.9**	95.3	-
batik	104,882	0.52/4	680.1	526.8	**1.32**
eclipse	237,732	4.25/10	-	-	-
fop	7,106	0.5/2	7.15	7.8	**0.97**
h2	≥43492240	32.45/35	-	-	-
jython	33,101,230	-	-	-	-
luindex	428,608	2.85/6	671.7	642.7	**7.13**
lusearch	3,289,700	8.42/61	-	-	-
pmd	17,714	8.76/17	136.5	**130.6**	-
sunflow	1648	8.63/17	1.26	1.45	**1.1**
xalan	8,935,120	37.84/65	-	-	**1641**

Table 7.18: Average slowdown for CloseFiles.

	JavaMOP	QEA-Symbol	QEA-Value	RuleR
batik	-	108.2	**92.7**	185.9
eclipse	-	**1.02**	1.04	1.23
fop	627.1	430.4	**404.3**	457.5
h2	**54.8**	-	-	-
pmd	-	462.2	**444.2**	-
sunflow	**547.7**	-	-	-
tomcat	-	1.01	1.04	**1.0**
tradebeans	-	1.02	**0.98**	1.01
tradesoap	-	0.85	**0.82**	0.83

It is important to note that producing warnings can effect the efficiency of the approach. Table 7.17 gives the total number of events and slowdown for each benchmark and monitoring approach. We also record the mean and maximum number of locks held at any time, giving a very coarse measure of how likely a lock ordering problem is to occur. In many cases there are too many locks for the monitors to efficiently deal with them all. Both approaches need to store a large amount of data, and as before, QEA suffers from not being able to remove garbage bindings as they could be extended to failure. QEA is able to deal with over 3M locks in the case of avrora and a relatively large number of live held at once, in pmd. Only RuleR was able to monitor xalan successfully, which had the most locks held at once. In general, to efficiently monitor this property we need to implement better methods for trimming the search space.

The CloseFiles property is interesting as it is context-free. This means that we use the context-free-grammar plugin for JavaMOP and our QEA makes use of counters to keep track of the depth of method nestings. We also have a lot of activity as this property requires us to capture every method entrance and exit. Table 7.18 reports the slowdown for each monitoring approach, we only include benchmarks where at least one monitor completes.

JavaMOP and QEA are reasonably even in their performance. For fop, QEA outperforms JavaMOP considerably, but in two other cases JavaMOP completes monitoring when QEA does not. In the case of batik and pmd, QEA is at an advantage as it only produces one warning per thread, allowing it to monitor more efficiently. RuleR's performance was reasonable, outperforming JavaMOP for fop. For tomcat we only saw 124 events, in contrast to the 200M events for batik and 3.2M events for eclipse. Of those benchmarks that did not complete, on average QEA had processed 400 million events after two hours. They also typically had many warnings; JavaMOP reports 2 different warnings in luindex 7.6 million times.

Given its nature we suggest that this property is not well-suited to runtime monitoring and might more effectively be checked using simple static analysis as it does not require intra-procedural checks.

7.2.4 Discussion

From the experiments in this section we can conclude two things:

1. Our technique scales to real world applications. It can outperform other techniques in cases where we have developed efficient algorithms and compete with more efficient techniques in other settings.
2. Further work is required to improve how our monitoring algorithms deal with garbage. In particular, the case where we fail to remove objects as their trace slices could be extended to failure, but our external knowledge that there is a connected semantics means that it is safe to remove these objects.

7.3 Summary

In this chapter we have used a hypothetical case study to explore a number of research questions relating to the efficiency of our monitoring algorithms and then applied these algorithms to some real world programs from the DaCapo benchmark suite to establish the applicability of our techniques.

We have shown that the optimisations described in the previous two chapters have been worthwhile but that there is still some work to be done to tackle the issue of garbage collection. There are further optimisations that can be applied. importantly, as we make use of the trace slicing concept we can adapt any of the optimisations used by JavaMOP. Our algorithm selection technique can be extended to consider more specific optimisations based on structural and trace properties. One observation is that the Symbol indexing technique, which is novel to this work, outperforms the Value indexing technique, which was introduced by JavaMOP.

Part III

Specification Mining

Chapter 8

Mining Quantified Event Automata

We now consider the problem of inferring QEA. The problem we address is that of producing a (form of) QEA from a set of labelled traces produced by a system presumed to be correct. We refer the reader to Section 2.4 for a discussion of this learning process.

Here we take an approach that is well suited to our development; a generate-and-check style approach that harnesses the efficient runtime verification algorithms developed in Part II. Our technique *generates* a set of possible 'patterns' and then *checks* them against the traces using our runtime verification process. From an abstract point of view these patterns can be seen as hypothesis QEAs, however in practice we carry out mining at the propositional level and use the quantifications to move between the parametric and propositional settings. It is useful to produce large specifications and therefore we introduce a *combination* stage to our approach that combines together extracted patterns.

This process relies on a number of different concepts coming together. Firstly, we require a notion of pattern that can be easily and effectively combined. Secondly, we need an efficient method for generating and checking such patterns. Part of the efficient checking will come from the QEA monitoring algorithms presented in Part II, and part will depend on our method of combining patterns together to be checked simultaneously.

We note here that this chapter focuses on mining a restricted form of QEA that does not allow free variables, we discuss methods for extending this approach in Sec. 12.2.8.

Structure. We begin by introducing the general area of *specification mining* (Sec. 8.1). This is followed by an overview of the mining process used here (Sec. 8.2). Although the mining process is *generate*, *check*, *combine* the first stage we will consider is that of combination, as this informs our choice of pattern formalism, therefore Section 8.3 introduces *open automata* as our notion of pattern. Next we define appropriate mechanisms for generating and checking sets of patterns against traces (Sec. 8.4) before bringing these definitions together in the mining framework (Sec. 8.5). We then demonstrate how this mining process applies to a trace (Sec. 8.6). Next we present an extension of the mining framework to take into account the *connectedness* global guard (Sec. 8.7). The chapter concludes by comparing the approach taken here with existing work from the fields of specification mining and specification inference (Sec. 8.8).

8.1 Pattern-based specification mining

Here we consider our chosen specification inference approach; a pattern-based specification mining technique that uses a set of template patterns to generate potential specifications, and then checks these against the traces.

8.1.1 Daikon:an important beginning

An early generate and check approach for state specifications, or state invariants, is the Daikon tool [ECGN01, NE02]. The tool can label different program points, such as method entry/exit, with invariants over the accessible variables. The approach uses a set of template invariants, which it instantiates and checks over execution traces recording the value of variables at different program points. Inferred invariants are filtered for redundancies.

Daikon infers many kinds of invariants, for example: a variable being constant, coming from a small set, being in a range, being no-zero, or being equal to some constant modulo some other constant; two numerical variables having some linear relationship, ordering relationship, or functional equivalence; sequences having minimum and maximum conditions, (lexicographical) orderings, invariants holding over the whole sequence, or relationships with other linear sequences such as elementwise linear relationship, comparison, or subsequence relationships.

Daikon is often cited as an inspiration for pattern-based trace specification mining.

8.1.2 The problem

Gabel and Su [GS08b] describe the pattern-based specification mining problem as finding an instantiation of a template finite state machine with events in a trace. They attempt to formalise this as follows:

Problem 1 (Pattern-Based Specification Mining)**.** *Given a finite state machine $\mathcal{A}$ over an alphabet Σ_1 and execution trace $\tau \in \Sigma_2^*$ such that $\Sigma_1 \cap \Sigma_2 = \varnothing$. Does there exist a total injective function $\rho : \Sigma_1 \to \Sigma_2$ such that $\tau \downarrow_{\Sigma_1} \in \mathcal{L}(\rho(\mathcal{A}))$ where $\tau \downarrow_{\Sigma_1}$ is τ with all elements* not *in Σ_1 removed and $\rho(\mathcal{A})$ is equal to $\mathcal{A}$ with $\delta_{\rho(\mathcal{A})}(s, \rho(a)) = \delta_{\mathcal{A}}(s, a)$.*

Importantly, as they take the trace projection, an inferred model only captures interactions between the symbols in Σ_2. Note that there may be many solutions ρ and each represents an inferred specification. Their problem description does not discuss how to differentiate between these different solutions. Furthermore, the template finite state machine $\mathcal{A}$ determines the specifications extracted and the identification of good templates is a separate problem.

Although some processes may be missing from this problem description, it captures the core activity of the pattern-based approach. Gabel and Su [GS08b] also give a polynomial reduction from their definition of the problem to the well known NP-complete HamPath problem [Kar72]. This shows that the pattern-based approach, in general, is NP-complete.

8.1.3 The generate-check-refine process

The pattern-based specification mining approach can be realised by a generate-check-refine process. Concrete patterns are *generated* from templates and an alphabet, these are *checked* over the set of traces and those that pass are *refined* to form a description of the system. We now discuss these three stages in more detail.

Generate

The generation stage will always take one or more pattern templates and an alphabet and combine them together to make many instances of the pattern. If the pattern template has n symbols and the alphabet has m symbols then the number of patterns generated is bounded by n^m. We note that the generation stage is often conceptual i.e. we do not create the pattern instances directly but may represent them when checking. However, the set of pattern templates and alphabet determine the pattern instances that must be checked.

Let us now consider the choice of pattern templates. Later (Chap. 9) we will explore the choice of patterns in more detail and give a more thorough review (Sec. 9.2) of previously used patterns. For now we review the different tools and the kinds of patterns they use.

Engler et al. [ECH+01] use this approach to uncover bugs as inconsistent behaviour and 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. Perracotta [YE04a, YE04b, YEB+06], uses small (2 element) regular expression patterns based on the *Response* pattern [DAC99] that says whenever P happens, S must also eventually happen. The Javert tool [GS08b, GS08a] introduced the concept of combing successful patterns to produce larger patterns and the (2 or 3 symbol) patterns reflect this. Li et al[LFS10] 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 [WN05]) only use the alternating pattern $(ab)^*$.

All of the approaches given here take the symbols in the trace to be the alphabet, causing many patterns to be generated. The idea is that it is difficult to know a priori what an interesting set of symbols will be (each inferred specification describes the interaction between a subset of the alphabet) and therefore all should be used. However, a decision must be made of what to record in the execution trace and the decision of what symbols may be interesting is deferred.

Check

Generated patterns are checked against the traces (previous approaches only use positive traces).

Peracotta [YEB+06] introduce a simple matrix approach for checking binary patterns[1]. For each pattern and alphabet of n symbols an $n \times n$ matrix is constructed with each cell representing a state in the finite state automaton 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 states being updated accordingly. Li et al. [LFS10] extends the matrix approach in their SAM tool. As their framework allows multiple events to occur at the

[1]Patterns over two symbols

same time (this makes sense in their setting of hardware systems) they extend the approach to ensure that no cell is updated more than once on a single cycle.

Javert [GS08b, GS08a] 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 $(|\Sigma| * \lceil \mathtt{log}_2(|\Sigma'|) \rceil) + \lceil \mathtt{log}_2(|Q|) \rceil$ boolean variables. The 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 [GS10] 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.

All approaches are 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.

Refine

The set of all successful patterns can be used to describe the system but as there might be many such patterns (with varying levels of accuracy and relevance) we can refine this set in three ways: ranking, pruning, and combination.

The patterns can be *ranked* based on measurements of *support* and *confidence* as well as other factors that might indicate relevance, such as number of symbols. 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. These notions of support and confidence only make sense in scenarios where lots of small rules are being extracted from many traces. The set can be *pruned* by removing patterns based on heuristics; for example, Peracottta [YEB+06] uses a reachability heuristic and a name similarity heuristic. OCD [GS10] aggressively prunes specifications during the mining process based on frequency of occurrence.

Perhaps the most effective way to refine the successful patterns is to *combine* some together to form larger, more descriptive specifications. Yang et al. [YEB+06] proposed a method for *chaining* inferred alternating patterns using the rule:

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

Where $A \rightarrow B$ is a pattern stating that if A occurs then B should occur later.

Gabel et al. have created two tools Javert [GS08a] and `OCD` [GS10] 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 as it does not need a relationship between A and C. The two inference rules used are as follows:

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

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

The first rule allows us to take the disjunction of behaviour that is observed to happen within the bounds of other events. For example, if both `read` and `write` were observed to happen between `open` and `close`. The second rule is an extension of Yang et al.'s sequencing rule.

Li et al.'s work [LFS10] with more complex temporal patterns also introduced similar inference rules over their more complex patterns.

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

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

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

Generally these combination rules are used to *close* the set of patterns i.e. carry on applying them until they can no longer be applied.

8.1.4 Dealing with data

Few previous approaches consider an events contextual information, for example parameter or return values. Yang et al. [YEB+06] can either perform a context-sensitive rewriting of events ($\texttt{f}(1)$ becomes $\texttt{f}_1$) or context-slicing where traces are sliced (in the same way we use the term previously) based on the callee object. This will therefore extract typestate properties, restricted to methods called on a particular object. Gabel and Su [GS10] also take a context-slicing approach by relating each symbol in a binary rule to an object - note that this generalises Yang et al.'s approach. Neither of these approaches can deal with multiple quantifications.

8.1.5 Applications

In Sec. 2.4.3 we discussed general applications for specification inference, but specification mining has particular strengths. Specification mining extracts small patterns of behaviour concerning a few events of interest. These may be combined together to form larger patterns but in general the mined specifications describe the behaviour of parts of a system only. Therefore, the extracted patterns can be useful in understanding these particular snapshots. For example, the usage of a particular interface in a code-base. The advantage of extracting small specifications is that they exclude the clutter of unimportant interactions. A key area where specifications extracted in this way can be used is in automated processes. For example, the OCD tool [GS10] mines and checks patterns simultaneously; effectively checking for self-consistency of coding rules. Another example would be where one tested system was replaced by an untested, optimised one; a set of patterns could be extracted describing the first and checked against the second. Automated uses such as these play to the strengths of specification mining as specifications do not need to be presented in a comprehensive format.

8.1.6 Summary

There are two computational problems to be solved in pattern-based specification mining: checking patterns against traces and combing patterns together. The first of these is exactly the role of runtime verification. This is our reason for selecting this approach to specification inference; we can directly use any runtime verification techniques developed for our new specification formalism.

8.2 Overview of approach

The previous section outlined the general specification mining approach. Here we discuss the approach taken in this work and relate this back to the runtime verification work carried out in Part II.

8.2.1 Target QEA

In this chapter we focus on mining QEA without free variables (we discuss this limitation at the end of the chapter). This form of QEA does not fit exactly into the hierarchy developed in Chapter 3 so we introduce a new kind of QEA for use here.

Definition 58 (Target QEA)**.** *A* target QEA *is a pair* $\langle \Lambda, \mathcal{E} \rangle$ *where ,* $\Lambda \in (\{\forall, \exists\} \times \mathsf{vars}(\mathcal{E}))^*$ *is a list of quantified variables and* $\mathcal{E}$ *is a simple event automaton (as introduced in Definition 8 in Section 3.2) i.e. a tuple* $\mathcal{E} = \langle Q, \mathcal{A}, q_0, \delta, F \rangle$ *where* Q *is a finite set of states,* $\mathcal{A}$ *is a finite alphabet of events,* $q_0 \in Q$ *is an initial state,* $\delta \subseteq (Q \times \mathcal{A} \times Q)$ *is a finite set of transitions, and* $F \subseteq Q$ *is a finite set of final states. It is required that* $\mathsf{vars}(\mathcal{E}) = \mathsf{vars}(\Lambda)$.

This adds existential quantification to SQSEA and differs from full QEA in that it does not include free variables, type variables, global guards or given domains.

8.2.2 The stages

The approach is generate-check-combine where the three stages (demonstrated in Fig. 8.1) are:

1. *Generating.* A pattern library and candidate alphabet are used to produce a set of possible patterns. To be more precise, an alphabet and pattern library are used to produce a *pattern checker* (introduced in Sec. 8.4.2) that extracts patterns from traces.

2. *Checking.* These patterns are checked against a set of given traces using concepts introduced in Chapter 3; traces are projected for bindings of quantified variables to produce a set of subtraces per input trace. The pattern checker is then used to extract a set of patterns that hold in each subtrace and a supplied quantifier list is used to select successful patterns with respect to the quantification.

3. *Combining.* The passing patterns are combined to form a specification.

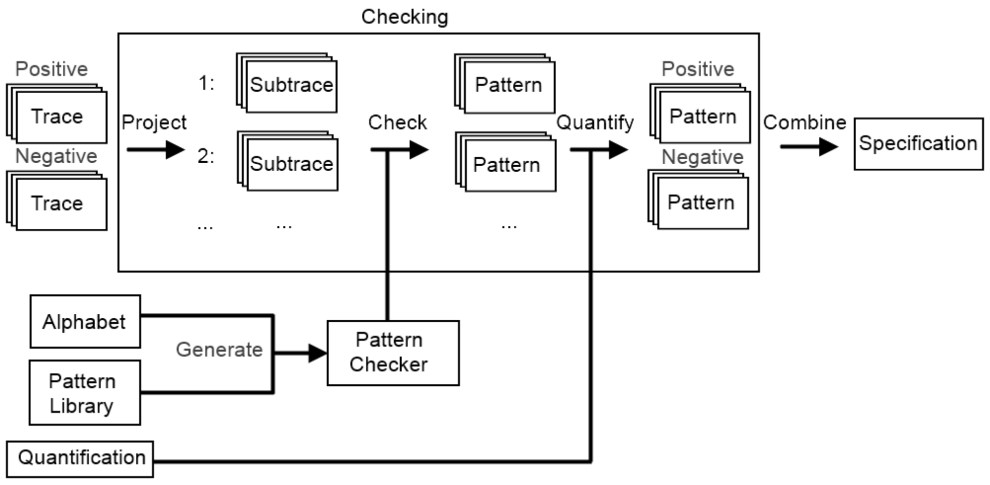


Figure 8.1: Pattern checking in the parametric setting.

8.2.3 The inputs

There are four inputs to the monitoring process which determine the specification produced. As illustrated in Figure 8.1 these are:

A) A pattern library.

B) An alphabet.

C) A list of quantifications over the variables in the alphabet.

D) A set of positive traces (and possibly a set of negative traces)

In the following, we discuss how these inputs may be produced, the effect they may have on the extracted specification and any limitations they introduce.

A) Pattern library. These should represent common patterns found in real-world specifications. The hypothesis is that by using many small, general patterns we can build powerful yet concise specifications, we explore this notion further in Chapter 9. The choice of pattern library dictates the specifications that can be extracted and later (Sec. 9.1.1) we discuss the theoretical bounds introduced by pattern libraries. In Section 8.3 we discuss our requirements for a pattern language and introduce *open automata* to be used as patterns in this work.

B) Alphabet. This is the set of events that will be used to identify the traces to record and will form the alphabet of the extracted specification. The need to identify a set of events to extract is a limitation, which is discussed further in Sec. 8.9.1, and we do not address the problem of detecting likely alphabets in this work. Adding or removing events from an alphabet can have a significant impact on the results obtained and furthermore that the entire mining process must be repeated for each alphabet considered as partial results are not possible due to our use of open automata. We discuss this further later.

C) Quantifications. The checking stage only uses the set of quantified variables, and whether the quantifications are $\forall$ or $\exists$ is immaterial at this stage. Therefore, all possible combinations can be explored efficiently after the traces have been traversed. However, this may result in a separate specification per quantification list as the successful patterns may be different in each case. In general, the assumption might be that if a specification can be found using only universal quantification then this is the most general specification, however this may be an over-generalisation. We demonstrate this later.

D) Traces. As discussed in Section 2.4, the mining process is a generalisation from the set of input traces. We will produce a specification that accepts at least the positive traces and rejects at least the negative traces. If such a specification does not exist in the set of specifications that can be built from the pattern-library then no specification will be returned. The assumption is that we have no control over the input traces, but if we do we should aim for good coverage, as discussed in Sec. 8.8.3.

It is generally difficult to generate negative traces and the mining process does not require them. We include negative traces as they serve as an additional source of external knowledge that can contribute towards the final specification. For example, if we know that a trace cannot begin with a certain event we can supply a negative trace to that effect.

8.3 Patterns that combine

In this section, we present *open automata* for capturing patterns along with a sound and complete method for combination i.e. the combination process does not lose or add any information. We begin by discussing what we require of our pattern formalism, and then discuss why standard deterministic finite automata are inadequate before introducing open automata and their properties.

8.3.1 Requirements for patterns

Our approach involves checking many small patterns and then combining the successful patterns together to form a large specification. Each of these small patterns will describe some behaviour of a subset of the events involved in the final specification. Here we consider the ideal qualities a pattern formalism would have with respect to combination; the matter of checking traces against patterns will be considered in the next section.

Metasymbols in patterns. Our approach requires us to define patterns up-front yet the patterns we extract should refer to events in our given alphabet. We will therefore define patterns over *metasymbols* to be instantiated before checking against given traces. For example, we may define a pattern over two metasymbols a and b and then given an alphabet of n events we will create 2^n instantiated patterns to check at runtime.

When checking patterns containing metasymbols we could either think of instantiating the pattern with events or rewriting the trace with the inverse of the instantiation. We will use the first approach here, but the fact that this is equivalent to the second approach is why throughout this section it will make sense to talk about a pattern accepting a trace of metasymbols.

A pattern formalism. Let us assume we have a pattern formalism such that a pattern p consists of a set of metasymbols Σ and a language $\mathcal{L}(p)$ such that $\mathcal{L}(p)$ is the set of traces (of metasymbols) accepted by p.

Small alphabets. Generally patterns will be defined for a small set of symbols i.e. if we have k metasymbols in our pattern and n distinct events in an input trace it is likely that $k < n$. To allow us to talk about a pattern accepting a trace over a larger set of symbols we will use the concept of *projection* as we did with the definition of acceptance for QEA.

Definition 59 (Projection)**.** *Given a trace of symbols τ and set of symbols Σ let $\tau\downarrow_\Sigma$ be the trace τ with all symbols not in Σ removed. Given a language $\mathcal{L}(p)$ let $\mathcal{L}^\Sigma(p) = \{\tau\downarrow_\Sigma \mid \tau \in \mathcal{L}(p)\}$.*

Combining patterns. The first concept we introduce is that of combination i.e. adding together the information given by a set of patterns. So what does this mean? A pattern represents a very general approximation of the behaviour of the system that produced the trace. We choose to take a safe notion of acceptance, rather than rejection, when combining patterns i.e. ensure that we maintain all information about traces that are *rejected*.

Therefore, combination can be given as an intersection of languages; if p is the combination of p_1 and p_2 then $\mathcal{L}(p) = \mathcal{L}(p_1) \cap \mathcal{L}(p_2)$. However, the language of a pattern is given as a set of traces over the patterns alphabet and the two patterns may not have the same alphabet. To address this, we expand these patterns so that they reflect the traces they accept based on the notion of projection. Expanding a pattern includes the symbols in the other patterns alphabet.

Definition 60 (Expansion)**.** *Given a pattern p defined over alphabet Σ, the expansion of p for alphabet Σ' is a new pattern $p^{\Sigma'}$ defined over the alphabet $\Sigma \cup \Sigma'$ such that $\mathcal{L}(p^{\Sigma'}) = \mathcal{L}^\Sigma(p)$.*

Let $\cap_p$ be an intersection operation on patterns with the same alphabet. Combination then becomes expansion followed by intersection.

Definition 61 (Combination)**.** *Given two patterns p_1 and p_2 over alphabets Σ_1 and Σ_2 respectively, let their combination be $p_1 \cap p_2 = p_1^{\Sigma_2} \cap_p p_2^{\Sigma_1}$.*

The 'combination space'. Let us consider the conceptual space of patterns defined by this combination operator. We use the combination operator to define a partial order on patterns such that $p_2 \sqsubseteq p_1$ iff there exists a p_3 such that $p_1 \cap p_3 = p_2$. We can read this as 'p_2 is more precise than p_1' as it contains more information about what traces should be rejected.

Given a set of patterns P, the patterns that we can extract from this set is the closure of P under combination. Next, let us also categorise patterns based on the size of their alphabet i.e. the k-patterns are those patterns with an alphabet of size k. Figure 8.2 illustrates this partial order and categorisation; arrows connect patterns to other patterns more precise than them. If there existed a k such that every k-pattern is a combination of patterns in a lower category then we could place an upper limit on the number of symbols required in patterns.

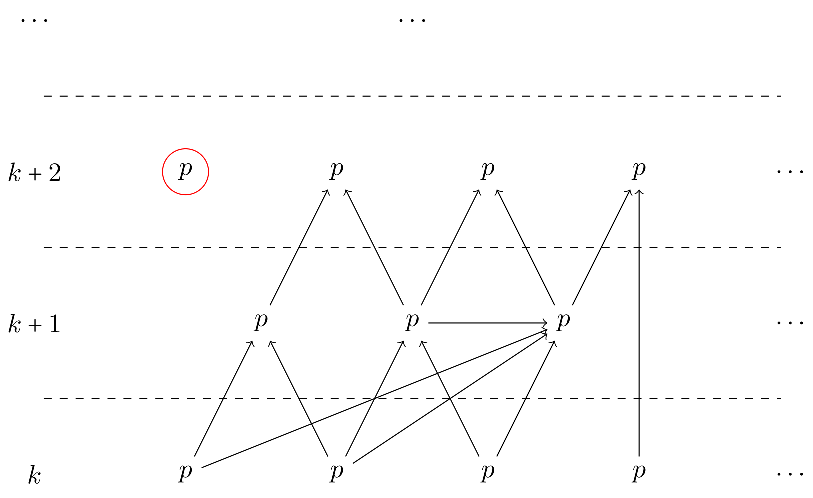


Figure 8.2: A partial order on patterns

However, if we can show that for every category there exists at least one pattern that is not the combination of patterns in a lower category then we learn something about the limitations of this approach i.e. that we require an approach that allows patterns of an arbitrary size.

The inverse of combination. The notions of expansion and combination are what we need for our mining process. However, to discuss patterns we also introduce their (theoretical) inverse operations: *contraction* and *decombination.*

Definition 62 (Contraction)**.** *Given a pattern p defined over alphabet $\Sigma \cup \Sigma'$, the contraction of p for Σ' is a new pattern $p^{\overline{\Sigma'}}$ defined over the alphabet Σ such that $\mathcal{L}(p^{\overline{\Sigma'}})$ such that $\mathcal{L}^{\Sigma}(p) = \mathcal{L}(p^{\overline{\Sigma'}})$.*

Definition 63 (Decombination)**.** *A set of patterns $\{p_1, \ldots p_n\}$ is a decombination of pattern p if $\bigcap\{p_1, \ldots, p_n\} = p$ and $p \notin \{p_1, \ldots p_n\}$. This is called a $k-n$ decombination if every p_i is in category $k-n$ when p is in category k. Furthermore, it is called a* covering *$k-n$ decombination if for every subset Σ of metasymbols of size k there is a pattern p_i with Σ as an alphabet.*

Contraction is the inverse of expansion as for pattern p we have $\mathcal{L}((p^{\Sigma'})^{\overline{\Sigma'}}) = \mathcal{L}(p)$ as $\mathcal{L}^{\Sigma}(p^{\Sigma'}) = \mathcal{L}((p^{\Sigma'})^{\overline{\Sigma'}})$ and $\mathcal{L}^{\Sigma}(p^{\Sigma'}) = \mathcal{L}(p)$. Decombination is the inverse of combination by definition.

It is clear that if a decombination operator exists and is totally defined (i.e. for all patterns) then we solve our above problem of finding a set of patterns that can be used to build all other patterns. If $k-1$ decombination is defined for all k bigger than some n then we find our maximum number of symbols required. However, we will see later that selecting a pattern formalism is a trade off between *generality* of decombination and *coverage* of combination such that decombination can only be perfect if it does not generalise. The goal is therefore to find a pattern formalism that offers an appropriate level of generality in decombination and coverage in combination.

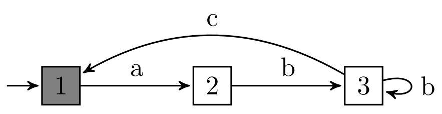


Figure 8.3: A DFA capturing the resource usage pattern $(ab^+c)^*$, note the use of next states.

8.3.2 Considering deterministic finite automata

Let us consider deterministic finite automata as a possible pattern formalism. As discussed in Section 8.1 they have been used previously in propositional generate-and-check specification mining approaches. We begin by formalising what we mean by standard automata, i.e. deterministic finite automata (DFA), and their combination.

Definition 64 (DFA)**.** *The tuple $\mathcal{D} = \langle \Sigma, Q, \delta, q_0, F\rangle$ is a deterministic finite automaton (DFA) where Σ is a finite set of symbols, Q is a finite set of states, $\delta : \Sigma \times Q \to Q$ is a transition function, $q_0 \in Q$ is an initial state and $F \subset Q$ is a set of final states. $\mathcal{L}(\mathcal{D})$ is defined in terms of δ and F in the standard way.*

As we have a standard notion of intersection for DFA we only need to introduce the concept of expansion to give us notions of combination and decombination.

Definition 65 (DFA Expansion)**.** *The Σ' expansion of DFA $\mathcal{D}$ with alphabet Σ is the DFA $\mathcal{D}^{\Sigma'}$ which is obtained by adding a looping transition to each state for each new symbol i.e. each symbol that does not already occur in Σ.*

It was shown by Gabel and Su [GS08a] that there exist three symbol DFA that cannot be decombined into two symbol DFA. We reproduce this proof here by giving a 3-symbol DFA that cannot be detecting using any set of 2-symbol DFA patterns.

Proposition 5. *The resource usage automaton in Fig. 8.3, let us call it $\mathcal{R}$, has no two-symbol decombination.*

Proof. We can assume, without loss of generality, that the decombination of $\mathcal{R}$ consists of three DFA; one for each pair of symbols in its alphabet (two DFA with the same alphabet can be combined). Let us call these DFA $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$, with alphabets $\Sigma_1 = \{a, b\}$, $\Sigma_2 = \{b, c\}$ and $\Sigma_3 = \{a, c\}$ respectively, and let Σ be the alphabet of $\mathcal{R}$. From the definition of decombination we have that

$$\mathcal{L}(\mathcal{R}_1^{\Sigma}) \cap \mathcal{L}(\mathcal{R}_2^{\Sigma}) \cap \mathcal{L}(\mathcal{R}_3^{\Sigma}) = \mathcal{L}(\mathcal{R})$$

and therefore for $i \in \{1, 2, 3\}$

$$\mathcal{L}(\mathcal{R}) \subseteq \mathcal{L}(\mathcal{R}_i^{\Sigma}).$$

As contraction is the inverse of expansion we also get

$$\mathcal{L}(\mathcal{R}^{\overline{\Sigma_i}}) \subseteq \mathcal{L}(\mathcal{R}_i)$$

for $i \in \{1, 2, 3\}$. We can then apply contraction to get

$$\mathcal{R}^{\overline{\Sigma_1}} = (ab^+)^* \qquad \mathcal{R}^{\overline{\Sigma_2}} = (b^+c)^* \qquad \mathcal{R}^{\overline{\Sigma_3}} = (ac)^*$$

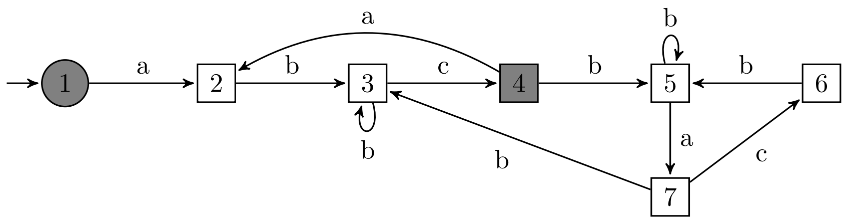


Figure 8.4: A DFA for the DFA combination of $(ab^+)^*$, $(b^+c)^*$ and $(ac)^*$.

The trace

$$a.b.b.c.b.b.a.b.b.c$$

is in the language of each of the contractions, but not in the language of $\mathcal{R}$. Therefore we have a contradiction and $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$ cannot exist. To be more precise, $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$ are not a *precise* decombination of $\mathcal{R}$. □

The key problem here is that the contractions $(ab^+)^*$, $(b^+c)^*$ and $(ac)^*$ do not contain any information about what is happening in the rest of the trace and we can insert symbols wherever we want, in this case a c in the middle of the bs of $(ab^+)^*$. The decombination is *too general* as it does not capture any context. To illustrate this Fig. 8.4 gives the combination of the decombination of the resource usage pattern, which accepts more traces than the original.

Our solution of open automata captures the lost context by replacing removed symbols by holes when performing contraction. However, this will still introduce some ambiguity in combination. We could introduce a two-hole open automata to deal with this second level of context but this process could continue until decombination did not generalise i.e. we use as many kinds of hole as we have removed symbols.

Later we will show that contraction for open automata is necessarily more precise (i.e. retains more information) than contraction for DFA, meaning that in our diagram above (Fig. 8.2) using open automata we will be able to cover more patterns on a given level using patterns at a lower level.

8.3.3 Syntax and Semantics of Open Automata

We introduce *open automata* as a formalism for patterns.

Definition 66 (Open Automata). *An* open automaton *is a tuple* $\langle \Sigma, Q, q_0, \delta, F, q_\perp \rangle$ *where* Σ *is a finite alphabet,* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $\delta \subseteq (Q \times \Sigma \cup \{\bullet\} \times Q)$ *is a set of transitions where* $\bullet$ *is a special* hole *symbol,* $F \subseteq Q$ *is the set of final states and* $q_\perp \in Q \backslash F$ *is the error state.*

Let *Pattern* be the set of all open automata. The special hole symbol $\bullet$ can match any symbol not in Σ, we assume a universal set of symbols U. We define a function Δ which returns a set of states given a trace over U for a given open automaton $p = \langle \Sigma, Q, q_0, \delta, F, q_\perp \rangle$ as follows:

$$\begin{array}{lll} \Delta_p(S,\epsilon) & = & S \\ \Delta_p(\{\},\sigma) & = & \{q_\perp\} \\ \Delta_p(S,a\sigma) & = & \Delta_p(S',\sigma) \text{ for } S' = \left\{ q' \mid \exists q \in S : \begin{array}{ll} (q,a,q') \in \delta & \text{if } a \in \Sigma \\ (q,\bullet,q') \in \delta) & \text{if } a \notin \Sigma \end{array} \right\} \end{array}$$

This adds transitions to an error state if no other transition can be made. The language of p is then

$$\mathcal{L}(p) = \{\sigma \mid \Delta_p(\{q_0\},\sigma) \cap F \neq \varnothing\}.$$

Every DFA is a form of open automata; the translation is the straightforward addition of self-looping hole transitions on each state. This is consistent with the notion of acceptance with respect to projection introduced in the previous section.

8.3.4 Combining Open Automata

Open automata expansion can be defined as adding transitions for the new symbols where a hole transition exists. It should be obvious that DFA expansion described previously is a special case of this.

Definition 67 (Open Automata Expansion). *Given open automaton $p = \langle \Sigma, Q, q^0, \delta, F, q^\perp \rangle$, the expansion for Σ' is $p^{\Sigma'} = \langle \Sigma \cup \Sigma', Q, q^0, \delta', F, q^\perp \rangle$ where*

$$\delta' = \delta \cup \{(q_1, \bullet, q_2) \mid \exists a \in \Sigma' \backslash \Sigma : (q_1, a, q_2) \in \delta\}$$

The definition of contraction is similarly straight-forward: replace symbols to be removed with hole symbols, it helps that open automata are non-deterministic.

We can define combination as expansion followed by intersection. Note that this can be seen as synchronizing on hole symbols.

Definition 68 (Open Automata Combination). *The combination of the two open automata $p_1 = \langle \Sigma_1, Q_1, q_1^0, \delta_1, F_1, q_1^\perp \rangle$ and $p_2 = \langle \Sigma_2, Q_2, q_2^0, \delta_2, F_2, q_2^\perp \rangle$ is*

$$p_1 \cap p_2 = \langle \Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, (q_1^0, q_2^0), \delta, F, (q_1^\perp, q_2^\perp) \rangle$$

where $(q_1, q_2) \in F$ iff $q_1 \in F_1$ and $q_2 \in F_2$, and $((q_1,q_2), a, (q_1', q_2')) \in \delta$ iff both

$$\begin{array}{ll} 1. & (q_1, a, q_1') \in \delta_1, \textit{ or } a \in \Sigma_2 \backslash \Sigma_1 \textit{ and } (q_1, \bullet, q_1') \in \delta_1, \textit{ and} \\ 2 & (q_2, a, q_2') \in \delta_2, \textit{ or } a \in \Sigma_1 \backslash \Sigma_2 \textit{ and } (q_2, \bullet, q_2') \in \delta_2 \end{array}$$

Decombination can be defined using contraction for a given set of symbols.

Definition 69 (Open Automata Decombination). *Given open automaton $p = \langle \Sigma, Q, q^0, \delta, F, q^\perp \rangle$ such that $|\Sigma| = k$ and a $n < k$ the $k - n$ decombination is the set of open automata*

$$\mathsf{D}(p, n) = \{p^{\overline{\{a_1, \dots, a_n\}}} \mid a_1, \dots, a_n \in \Sigma\}$$

This is equivalent to the set of contractions for each subset of Σ with n symbols removed. The size of $\mathsf{D}(p,n)$ is $\binom{k}{n}$ as it is equivalent to the number of n-sized subsets of Σ.

8.3.5 Properties

We need to convince ourselves that by introducing open automata we have gained something useful over using DFA. Firstly, we show that combination for open automata is sound and complete, i.e. no new traces are accepted or rejected, this is important as it demonstrates that open automata are not worse than DFA in that respect.

Proposition 6. *Given two open automata p_1 and p_2 we have that $\mathcal{L}(p_1) \cap \mathcal{L}(p_2) = \mathcal{L}(p_1 \cap p_2)$.*

We prove this by showing that, for a given trace, the states reached by both p_1 and p_2 are exactly those reached by $p_1 \cap p_2$ and therefore if an accepting state is reached by both patterns it is reached by their combination.

Proof. Let $p_1 = \langle \Sigma_1, Q_1, q_1^0, \delta_1, F_1 \rangle$ and $p_2 = \langle \Sigma_2, Q_2, q_2^0, \delta_2, F_2 \rangle$. Let

$$S_1 = \Delta_{p_1}(\{q_1^0\}, \tau) \qquad S_2 = \Delta_{p_1}(\{q_1^0\}, \tau) \qquad S = \Delta_{p_1 \cap p_2}(\{(q_1^0, q_2^0), \tau)$$

It is sufficient to show that $(q_1, q_2) \in S$ if and only if $q_1 \in S_1$ and $q_2 \in S_2$. We therefore prove that

$$S_1 \times S_2 = S$$

by structural induction on the trace τ. For $\tau = \epsilon$ our conditions hold by definition:

$$S_1 = \{q_1^0\} \qquad S_2 = \{q_2^0\} \qquad S = \{(q_1^0, q_2^0)\}$$

For $\tau = \sigma.a$ let S_1^σ, S_2^σ and S^σ represent S_1, S_2 and S for σ. Note that our induction hypothesis gives us that $S_1^\sigma \times S_2^\sigma = S^\sigma$. Let us consider the four different cases for an input symbol a.

The cases where $a \in \Sigma_1 \cap \Sigma_2$ and $a \notin \Sigma_1 \cap \Sigma_2$ are symmetrical. In the following let $\alpha = a$ in the former case and $\alpha = \bullet$ in the latter. Firstly,

$$S_1 = \{q' \mid q \in S_1^\sigma \wedge (q, \alpha, q') \in \delta_1\} \qquad S_2 = \{q' \mid q \in S_2^\sigma \wedge (q, \alpha, q') \in \delta_2\}.$$

We then show that $S_1 \times S_2 = S$ by rewriting S by expanding the definition of δ for $p_1 \cap p_2$ and using our induction hypothesis:

$$\begin{array}{rl} S = & \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge ((q_1, q_2), \alpha, (q_1', q_2')) \in \delta\} \\ = & \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1 \wedge (q_2, \alpha, q_2') \in \delta_2\} \\ = & \{(q_1', q_2') \mid q_1 \in S_1^\sigma \wedge q_2 \in S_2^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1 \wedge (q_2, \alpha, q_2') \in \delta_2\} \\ = & \{q_1' \mid q_1 \in S_1^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1\} \times \{q_2' \mid q_2 \in S_2^\sigma \wedge (q_2, \alpha, q_2') \in \delta_2\} \end{array}$$

The second two cases are also symmetric.Without loss of generality assume $a \in \Sigma_1$ but $a \notin \Sigma_2$, we first have that

$$S_1 = \{q' \mid q \in S_1^\sigma \wedge (q, a, q') \in \delta_1\} \qquad S_2 = \{q' \mid q \in S_2^\sigma \wedge (q, \bullet, q') \in \delta_2\}$$

and again by rewriting S by expanding δ for $p_1 \cap p_2$ and using our induction hypothesis we get

$$\begin{aligned} S = &\quad \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge ((q_1, q_2), a, (q_1', q_2')) \in \delta\} \\ = &\quad \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge (q_1, a, q_i') \in \delta_i \wedge a \in \Sigma_i \backslash \Sigma_j \wedge (q_j, \bullet, q_j') \in \delta_j\} \\ = &\quad \{(q_1', q_2') \mid q_1 \in S_1^\sigma \wedge q_2 \in S_2^\sigma \wedge (q_i, a, q_i') \in \delta_1 \wedge (q_j, \bullet, q_j') \in \delta_j\} \\ = &\quad \{q_1' \mid q_1 \in S_1^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1\} \times \{q_2' \mid q_2 \in S_2^\sigma \wedge (q_2, \alpha, q_2') \in \delta_2\} \end{aligned}$$

□

Next let us consider decombination for open automata. Firstly, we show that given an open automaton without holes we have perfect $k-1$ decombination. This means that given all patterns with k symbols we could detect all hole-less patterns with $k+1$ symbols. However, as we see shortly, as we only cover hole-less patterns, this will not extend to patterns with $k+n$ symbols.

Proposition 7. *Given a hole-less open automaton p its $k-1$ decombination can be combined to give p exactly.*

Proof. Straightforwardly, the contractions of the decombination would only have holes in them that were given by removing a single symbol therefore during combination these holes would be filled with exactly those symbols that were removed so each contraction would become p. □

This means that the resource usage pattern has a perfect 2 symbol decombination, and therefore we can extract it using open automata, demonstrating already that open automata have greater coverage than DFA.

As mentioned above, this is not the case for an open automaton in general as if there are pre-existing holes the contractions would have a larger language when expanded. This means that we cannot extend decombination to the $k-2$ case and beyond in a precise way. But we can show that open automata gives us more coverage in the solution space then DFA by including contextual information. Recall that a pattern p is more precise than a pattern q if $\mathcal{L}(p) \subseteq \mathcal{L}(q)$.

Firstly, we show that as we remove more information from each contraction the resulting decombination is less precise.

Proposition 8. *Open automata $k-n$ decombination is at least as precise as $k-n-1$ decombination for $n \geq 0$ i.e.*

$$\mathcal{L}(\bigcap \mathsf{D}(p, n-1)) \subseteq \mathcal{L}(\bigcap \mathsf{D}(p, n))$$

Proof. The first thing to note is that all contractions of p will have the same structure modulo holes replacing symbols. This makes our proof more straightforward. We show that every trace in $\mathcal{L}(\bigcap \mathsf{D}(p, n-1))$ is necessarily in $\mathcal{L}(\bigcap \mathsf{D}(p, n))$. We proceed by contradiction. Assume that there exists a trace τ such that $\tau \in \mathcal{L}(\bigcap \mathsf{D}(p, n-1))$ but $\tau \notin \mathcal{L}(\bigcap \mathsf{D}(p, n))$. There most be at least one contraction $p^{\overline{X}} \in \mathsf{D}(p, n)$ such that $\tau \notin \mathcal{L}(p^{\overline{X}})$ and a contraction $p^{\overline{Y}} \in \mathsf{D}(p, n-1)$ such that $Y \subset X$, $|X| = n$ and $|Y| = n-1$. Let $q_0 \xrightarrow{\tau} q$ be an accepting path of τ on $p^{\overline{Y}}$; it will also be a path on $p^{\overline{X}}$ with the symbol in Y but not X replaced by a hole and therefore matching the missing symbol. Therefore, every accepting path of $p^{\overline{Y}}$ is an accepting path of $p^{\overline{X}}$ and $\mathcal{L}(p^{\overline{Y}}) \subseteq \mathcal{L}(p^{\overline{X}})$ leading to a contradiction, and therefore our assumption that $\tau \notin \mathcal{L}(\bigcap \mathsf{D}(p, n))$ cannot hold. □

This property implies that $\mathcal{L}(p) \subseteq \mathcal{L}(\bigcap \mathsf{D}(p,1))$ as $\mathcal{L}(p)$ is the $k-0$ decombination. We now want to show that open automata decombination is more precise than DFA decombination.

Proposition 9. *Open Automata $k-n$ decombination is more precise than DFA $k-n$ decombination.*

Proof. Given a DFA p with k symbols we consider the $k-n$ decombination of p in terms of DFA and open automata. In both cases we produce a set of $\binom{k}{n}$ patterns $\{p_1,\ldots,p_m\}$. Let us label these p_i^{DFA} in the case of DFA contraction and p_i^{OA} in the case of open automata contraction. Firstly, assuming that the *ith* decombination removes the symbols Σ_i, we show that

$$\mathcal{L}(p_i^{OA}) \subseteq \mathcal{L}(p_i^{DFA})$$

by showing that every trace in $\mathcal{L}(p_i^{DFA})$ must be in $\mathcal{L}(p_i^{OA})$. Given a trace $\tau \in \mathcal{L}(p)$ let $\tau' = \tau \downarrow_{\Sigma_i}$ i.e. the projection for Σ_i. The set $\mathcal{L}(p_i^{OA})$ then contains any variant of τ' that inserts a symbol not in Σ_i where it occurred in τ. The set $\mathcal{L}(p_i^{DFA})$ contains any variant in τ' that inserts any symbol in Σ_i at any point. Obviously every variant that satisfies the first case satisfies the second case.

It is then straightforward to see that

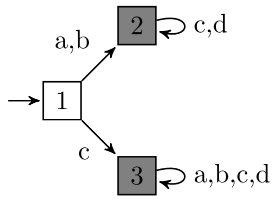
$$(\mathcal{L}(p_1^{OA}) \cap \ldots \cap \mathcal{L}(p_m^{OA})) \subseteq (\mathcal{L}(p_1^{DFA}) \cap \ldots \cap \mathcal{L}(p_m^{DFA}))$$

as if there were a trace τ in $(\mathcal{L}(p_1^{OA}) \cap \ldots \cap \mathcal{L}(p_m^{OA}))$ but not in $(\mathcal{L}(p_1^{DFA}) \cap \ldots \cap \mathcal{L}(p_m^{DFA}))$ then there must exist an i such that $\tau \in \mathcal{L}(p_i^{OA})$ and $\tau \notin \mathcal{L}(p_i^{DFA})$, but as we saw previously this cannot be the case. $\square$

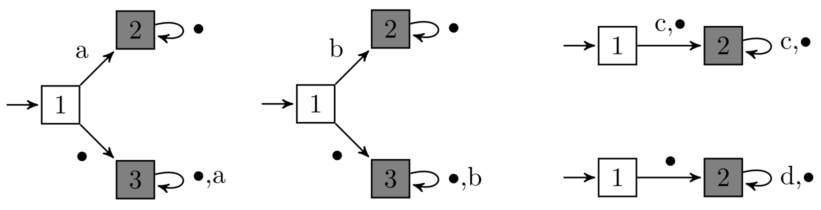
This means that, for some k, every pattern that can be extracted with k-symbol DFA can also be extracted with k-symbol open automata, and there are some patterns that can only be extracted using k open automata.

8.3.6 Understanding precision

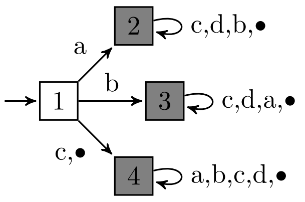
Let us consider a few examples that demonstrate the previous discussions about precision. Firstly let us consider the following DFA, let us call this p.



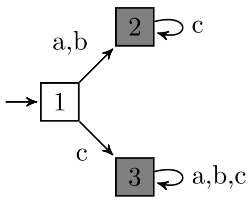
Using open automata decombination we get the following for $\mathsf{D}(p,3)$.



Now we can see that the trace $a.c.b$ is accepted by each of these automata, but not the original automaton. This demonstrates the imprecision of open automata decombination. Indeed, the combination of these automata is given as follows.



Now let us consider a slightly different DFA, which we shall call q.



This is the same as p except that we do not include d, therefore $\mathsf{D}(q,2)$ consists of the patterns in $\mathsf{D}(p,3)$ except for the one with alphabet $\{d\}$. Now, the combination of the patterns in $\mathsf{D}(q,2)$ gives us q exactly i.e. there is no imprecision.

So what is it about the d symbol in p that causes the imprecision of the decombination? The decombination for a conflates the behaviour of b, c and d and the other patterns can differentiate b and c but not b and d.

However, this is not a realistic example and the reason for that is that majority of the specifications used in the rest of this document have a precise decombination. Therefore, we claim that open automata represent a reasonable compromise between generality (precision) of decombination and coverage of combination.

8.3.7 Converting to Target QEA

Open automata can be instantiated with a mapping from symbols to ground events, ground events only as target QEA do not allow free variables. This process will allow us to generate target QEA from extracted patterns.

Definition 70 (Instantiation). *Given an open automaton $p = \langle \Sigma, Q, q_0, \delta, F, q_\perp \rangle$, an instantiation φ is a map from Σ to GEvent such that $p(\varphi) = \langle \mathcal{A}, Q, q_0, \delta(\varphi), F, q_\perp \rangle$ where $(q, \varphi(a), q') \in \delta(\varphi)$ iff $(q, a, q') \in \delta$ and $\mathcal{A}$ is the co-domain of φ.*

The conversion is then straight-forward; holes are removed and symbols replaced with their appropriate events. Note that this generates a SEA, as mining is undertaken within the context of a given Λ this can then be combined with the produced SEA to give a TQEA.

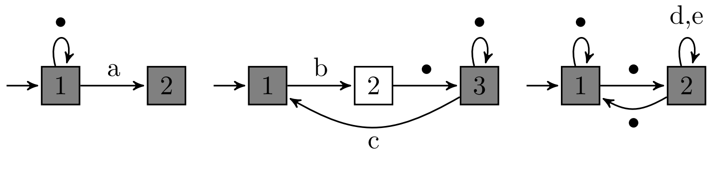
Definition 71 (Conversion). *Given a pattern $p = \langle \Sigma, Q, q^0, \delta, F, q^\perp \rangle$ and an instantiation φ we can create the SEA by applying the instantiation, removing the hole symbols and introducing transitions to the error state i.e. $\mathcal{E} = \langle Q, \mathcal{A}, \delta', q_0, F \rangle$ where*

$$\delta' = \{(q_1, \varphi(a), q_2) \mid a \neq \bullet \wedge ((q_1, a, q_2) \in \delta \vee ((q_1, a, q_2) \notin \delta \wedge q_2 = q_\perp))\}$$

During conversion it is necessary to add transitions to the error state as open automata have a next-style semantics, whereas target QEA have a skip-style semantics.

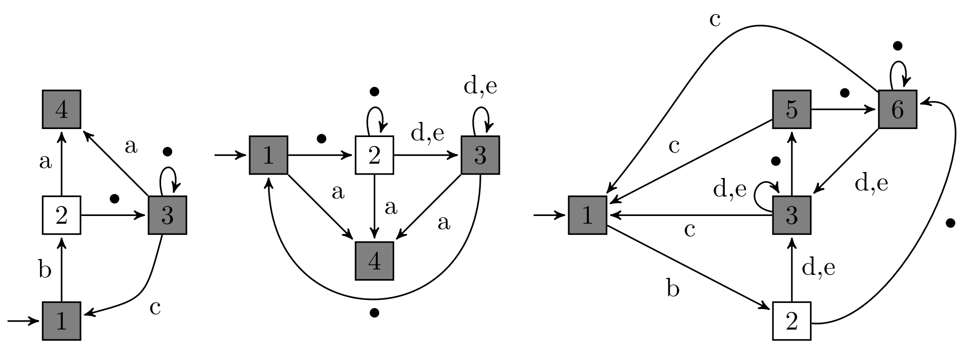
8.3.8 Example

Here we give an example of three patterns and how they are combined. Consider the following three patterns.

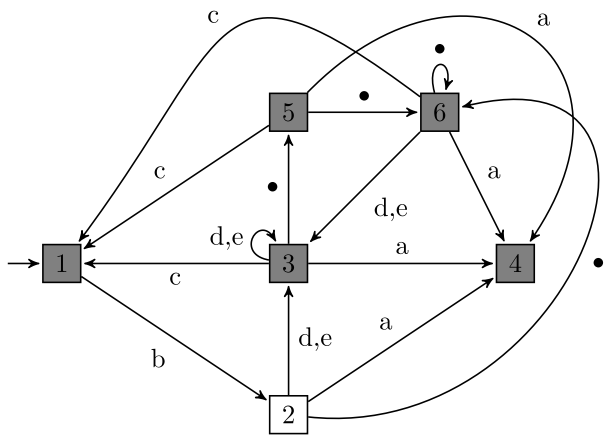


Pattern p_1 Pattern p_2 Pattern p_3

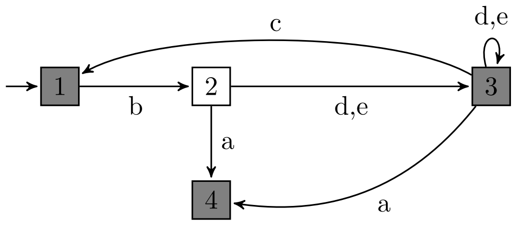
We combine each pair of patterns in turn, giving the three patterns below. The first (left) pattern is the result of combining p_1 and p_2, the second (middle) is the result of combining p_1 and p_3 and the third (right) is the result of combining p_1 and p_3.



When we combine all three together we get the following pattern:



And if we remove hole symbols (as we do when converting to SEA) we get:



8.4 Generating and checking patterns efficiently

In this section we consider how we can generate and check sets of patterns efficiently. We begin by describing the sets of patterns that we will consider, which are called *pattern libraries*, then we introduce a structure that allows us to check these sets of patterns efficiently, before describing how we construct such a structure from a pattern library.

8.4.1 Pattern libraries

Our approach takes a predefined pattern library as an input. A k-pattern library is a list of k-patterns with the same alphabet, which we will take as the first k letters of the English alphabet i.e. $\{a, b, c, \ldots\}$. A pattern library is a set of k-pattern libraries. We can have any value k, but we focus on small k for two reasons. Firstly, for efficiency reasons as the complexity of checking is dependent on the maximum pattern size, and secondly, the intuition behind our approach is that small *general* patterns can be combined to create larger specifications. The more symbols used in a pattern the more specific it is.

8.4.2 Pattern checkers

We begin by introducing a *pattern checker* that maps a trace of metasymbols to a set of patterns that will accept the trace.

Definition 72 (Pattern Checker)**.** *Let* $\mathsf{C} = \langle Q, \Sigma, q_0, \Rightarrow, \Gamma\rangle$ *be a* pattern checker *where* Q *is a set of states,* Σ *is a set of symbols,* $q_0 \in Q$ *is the initial state,* $\Rightarrow \in (Q \times \Sigma) \to Q$ *is the transition function and* $\Gamma \in Q \to 2^{Pattern}$ *is the output function. Lifting* $\Rightarrow$ *to traces in the standard way, define* C*'s successful patterns for the trace* τ *as*

$$\mathsf{C}(\tau) = \{p \in \Gamma(q) \mid q_0 \overset{\tau}{\Rightarrow} q\}$$

A pattern checker can be specialised with an *instantiation* (Def. 70) to extract patterns from traces of events as follows. Let us define an *event pattern checker* as a pattern checker, a binding and an instantiation. An event pattern checker maps a trace of events to a set of instantiated patterns that accept the trace (given the binding). Let $\varphi^{-1} \in GEvent \rightharpoonup \Sigma$ be the inverse of instantiation φ.

Definition 73 (Event Pattern Checker)**.** *An event pattern checker* $\mathsf{EC} = \langle \mathsf{C}, \varphi, \theta \rangle$ *with pattern checker* C*, instantiation* φ *and binding* θ *produces the following instantiated pattern from the trace* $\tau \in$ *GEvent***:*

$$\begin{aligned}\mathsf{EC}(\tau) = \quad & \{p(\varphi) \mid \exists \sigma \in \mathsf{C}.\Sigma^* : |\sigma| = |\tau| \wedge p \in \mathsf{C}(\sigma) \wedge \\ & \quad \forall i.0 \leq i < |\sigma|. \exists \mathbf{b} \in \varphi^{-1}(\sigma[i]).\mathsf{matches}(\tau[i], \theta(\mathbf{b}))\}\end{aligned}$$

This states that a pattern is included if there exists a trace of metasymbols that produces the pattern and has a mapping onto the trace of events using the instantiation.

We include a binding to allow us to separate the abstract alphabet of a pattern and the concrete one i.e. a pattern might match against concrete events $\mathtt{f}(2)$ and $\mathtt{e}(4)$ but when we extract the pattern we want it to be over the abstract alphabet $\mathtt{f}(x)$ and $\mathtt{e}(y)$.

We lift this concept to a set of event pattern checkers F such that F's successful patterns for trace τ are

$$\mathsf{F}(\tau) = \bigcup_{\mathsf{EC} \in \mathsf{F}} \mathsf{EC}(\tau)$$

8.4.3 Generating pattern checkers

We now build a generation function G that uses a pattern library and an alphabet of events to construct an event pattern checker.

Firstly we construct a pattern checker from a k-pattern library as follows. Given a k-pattern library L over the set of symbols Σ let $\langle Q, \Sigma, q_0, \Rightarrow, \Gamma \rangle$ be the pattern checker of L where

- $Q = (\ldots \times 2^{\mathsf{L}[i].Q} \times \ldots)$ and $q_0 = \langle \ldots, \{\mathsf{L}[i].q_0\}, \ldots \rangle$ for $0 < i < |\mathsf{L}|$
- $\langle \ldots, S_i, \ldots \rangle \overset{a}{\Rightarrow} \langle \ldots, S_i', \ldots \rangle$ iff for $0 \leq i < |\mathsf{L}|$

$$S_i' = \begin{cases} \mathsf{L}[i].\delta(q^i, a) & \text{if } a \in \Sigma \\ \mathsf{L}[i].\delta(q^i, \bullet) & \text{otherwise} \end{cases}$$

- $\mathsf{L}[i] \in \Gamma(\langle \ldots, q^i, \ldots \rangle)$ iff $q^i \in \mathsf{L}[i].F$ for $0 < i < n$

where we use the standard dot notation to access the elements of a pattern i.e. $\mathsf{L}[i].q_0$ is the initial state of pattern $\mathsf{L}[i]$. Recall that all patterns in L will have the same alphabet.

We construct a set of event pattern checkers G from a k-pattern library L and an alphabet of events $\mathcal{A}$ by using C, the pattern checker of L, as follows:

$$\mathsf{G}(\mathsf{L}, \mathcal{A}, \theta) = \{\langle \mathsf{C}, \varphi, \theta \rangle \mid \forall a \in \mathsf{C}.\Sigma : a \in \mathsf{dom}(\varphi) \wedge \varphi(a) \in \mathcal{A}\}$$

Finally, the generation function for pattern library P and alphabet of events $\mathcal{A}$ is

$$\mathsf{G}(\mathsf{P}, \mathcal{A}, \theta) = \bigcup_{\mathsf{L} \in \mathsf{P}} \mathsf{G}(\mathsf{L}, \mathcal{A}, \theta)$$

In this approach, pattern checkers check all patterns together in one operation, so the size of the pattern library effects the size of the pattern checkers produced but not the time it takes to find the successful patterns for a given trace.

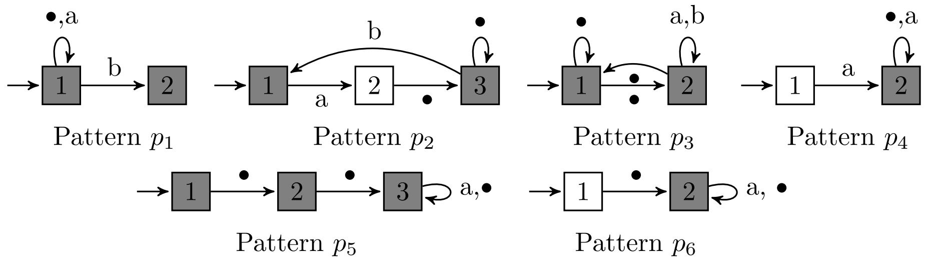


Figure 8.5: Six patterns used to create a pattern library

For efficiency, pattern checkers can be precompiled from pattern libraries and then combined with alphabets to produce event pattern checkers later. These can be compiled once and used many times. If additional patterns are added we can combine pattern checkers at runtime. This is important as it further reduces the effect of a large pattern library.

8.4.4 Example

Here we demonstrate how pattern checkers are generated and used. Consider the six patterns in Fig. 8.5. The first three are 2-patterns, as they have alphabets of size 2, and the second three are 1-patterns, as they have alphabets of size 1. Recall that patterns with the same number of symbols should have the same alphabets if they are to be used in a pattern library together.

The pattern checker for the first three patterns is given in Fig. 8.6 and the pattern checker for the second three patterns is given in Fig. 8.7. The patterns written on each state are those given by the output function for that state i.e. those that accept traces ending at that state.

8.5 Mining framework

In this section we describe how we mine Target QEA using patterns (i.e. open automata). As outlined previously, our approach is split into three main stages, illustrated in Figures 8.2 and 8.8. We give a more detailed overview of these three stages.

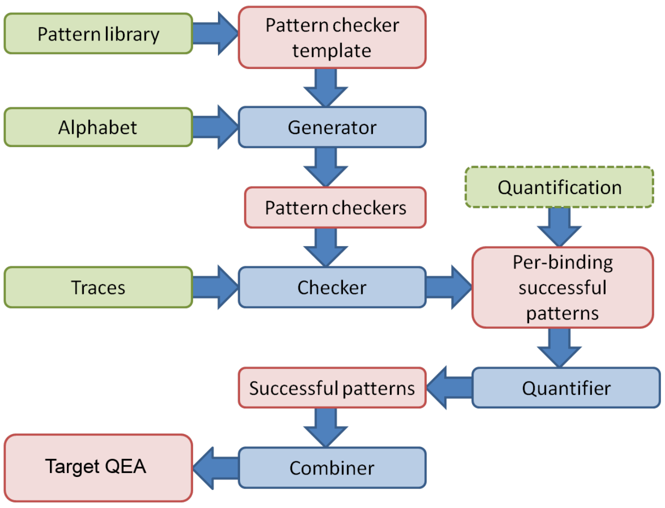


Figure 8.8: An overview of the tool.

1. **Generator.** Patterns in the pattern library are defined in terms of metasymbols and the job of the generator stage is to use the given alphabet to instantiate those metasymbols to produce patterns which can be checked against the trace.

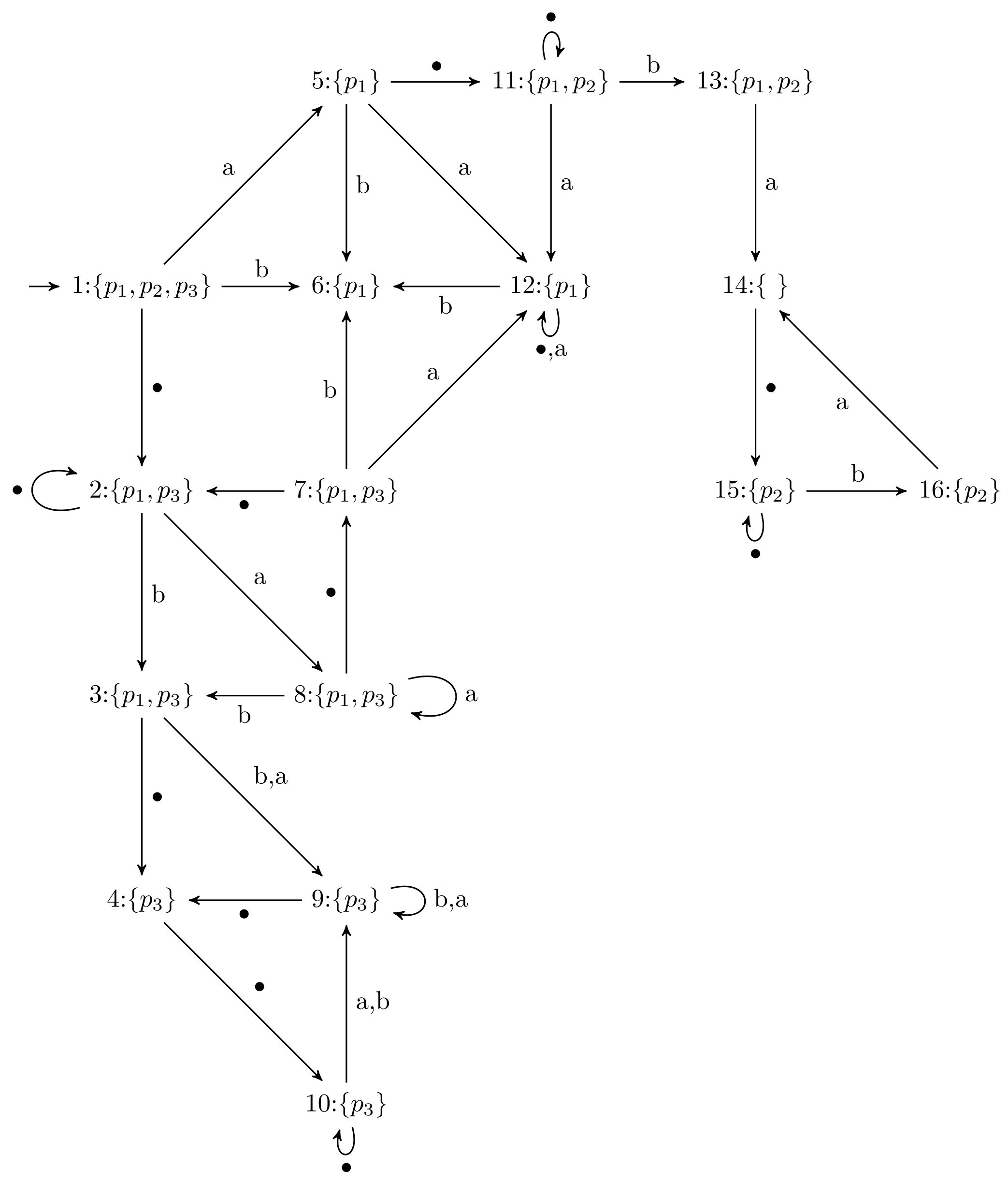


Figure 8.6: An example pattern checker for a 2-pattern library.

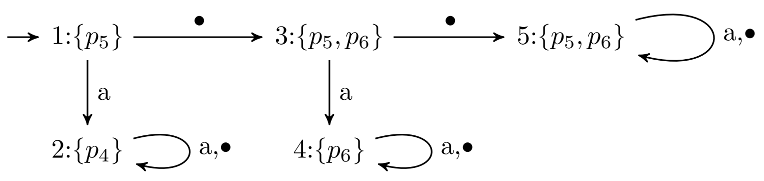


Figure 8.7: An example pattern checker for a 1-pattern library.

As it would be inefficient to check these patterns separately, we introduced in the previous section the concept of *event pattern checkers*, which check all patterns for a set of events simultaneously. We have already introduced the G function that produces a set of event pattern checkers from a pattern library and set of events.

2. **Checker.** The checking stage has two steps. Firstly, the concept of projection from the development of QEA is used to project the trace into subtraces, and secondly the event pattern checkers are used to find the patterns which hold in each subtrace. A sub-stage then uses a list of quantifications to select the positive and negative patterns.

3. **Combiner.** The notion of combination for open automata is used to produce an instantiated open automaton that, with the quantification list, forms a Target QEA.

We have implemented this framework in the `Scala` programming language.

8.5.1 Checking patterns

In this section we use the notion of projection and acceptance from QEA to produce sets of positive and negative successful patterns. First we compute *scorecards*, which capture, for each trace, which patterns hold for each binding of quantified variables.

Definition 74 (Scorecard)**.** *A scorecard is a map from bindings to sets of patterns:*

$$\mathsf{Scorecard} \quad = \quad \mathit{Binding} \rightharpoonup 2^{\mathit{Pattern}}$$

We use projection and the generator function G to generate a scorecard $\mathsf{S}(\tau)$ for a trace τ.

Definition 75 (Scoring)**.** *Given a pattern library* P*, alphabet* $\mathcal{A}$*, set of quantified variables* X *and trace* τ*, let the set of relevant bindings be*

$$\mathsf{relevant} = \{\theta \mid \mathsf{dom}(\theta) = X \land \forall x \in X : \theta(x) \in \mathsf{Dom}(\tau)(x)\}$$

where $\mathsf{Dom}(\tau)$ *is as given previously in Def. 15 on page 60. The scorecard* $\mathsf{S}(\tau)$ *defined as*

$$\mathsf{S}(\tau) = [\theta \mapsto \mathsf{G}(\mathsf{P}, \mathcal{A}, \theta)(\tau \downarrow_{\mathcal{A}(\theta)}) \mid \theta \in \mathsf{relevant}]$$

where the $\downarrow$ *projection operator is as given in Def. 6 on page 52 and the notion of alphabet instantiation* $\mathcal{A}(\theta)$ *is that given on page 56.*

This definition can be implemented directly to produces scorecards. However, it requires passing over each trace multiple times. Instead, we adapt the techniques explored in Chapters 5 and 6 to produce an efficient algorithm for checking these patterns against traces.

Once scorecards have been constructed the list of quantifications is used to extract successful patterns following the approach given for QEA acceptance in Def. 27 on page 68.

Definition 76 (Successful patterns). *Given a scorecard* $\mathsf{S}(\tau)$*, the successful patterns for the list of quantifications* Λ *are given by* $\mathsf{pat}(\mathsf{S}(\tau), [\ ], \Lambda)$*, defined as*

$$\begin{array}{lcl} \mathsf{pat}(\mathsf{S}, \theta, \forall x \Lambda') & = & \bigcap_{d \in \mathsf{Dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger [x \mapsto d], \Lambda') \\ \mathsf{pat}(\mathsf{S}, \theta, \exists x \Lambda') & = & \bigcup_{d \in \mathsf{Dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger [x \mapsto d], \Lambda') \\ \mathsf{pat}(\mathsf{S}, \theta, \epsilon) & = & \mathsf{S}(\theta) \end{array}$$

The positive and negative successful patterns for positive traces T_+ and negative traces T_+ are therefore given as

$$\mathsf{suc}_\alpha = \bigcap_{\tau \in T_\alpha} \mathsf{pat}(\mathsf{S}(\tau), \langle\ \rangle, \Lambda) \qquad \text{for } \alpha \in \{-, +\}$$

i.e. a pattern is positively (negatively) successful if it is a successful pattern for every positive (negative) trace.

8.5.2 Combining patterns

This stage straightforwardly combines the successful positive and negative patterns produced by the checking stage using the notion of combination given in Def. 68 on page 197.

Definition 77 (Combination). *The combination of a set of positive successful patterns* suc_+ *and a set of negative successful patterns* suc_- *is given as*

$$\left(\bigcap_{p \in \mathsf{suc}_+} p\right) \cap \overline{\left(\bigcap_{p \in \mathsf{suc}_-} p\right)}$$

where the complement $\overline{q}$ *of an open automaton* q *is given by inverting accepting states.*

The result is an open automaton with events as symbols; the translation to TQEA is then as described in Sec. 8.3.7.

8.5.3 Complexity

To calculate the time complexity of the mining process we examine each stage separately.

Generation happens once per pattern library as the pattern checkers are pre-compiled. As this involves simulating all patterns in parallel this process is exponential in the size of the pattern library i.e. if $|p|$ is the average size of a pattern then the complexity of the generation stage is $O(|p|^{|\mathsf{L}|})$.

In the worst case the checking stage passes over each trace τ twice; once to construct the bindings and once to compute the passed patterns. Given quantified variables X, the number of bindings is $O(|X|^{|\tau|})$, as the domain of each quantified variable is bounded by the length of the trace and we consider each combination of values. For each event we update the event pattern checkers associated with the event's relevant bindings. The number of event pattern checkers is given by $|\mathcal{A}|^k$ for each k-pattern library. Letting k be the maximum k-library, t be the average trace length and T be the number of traces, the complexity of the checking stage is $O(|X|^t + Tt|\mathcal{A}|^k)$.

The combination stage takes a divide-and-conquer approach to combining a set of patterns. The complexity of combining two open automata via the standard product construction is quadratic in the size of the automata and therefore the complexity of the combination stage is $O(n^2 \mathsf{log} m)$ where n is the average size of automata being combined and m is the number of automata. As m, the number of successful patterns, is bounded by $|\mathcal{A}|^k|\mathsf{P}|$, the number of all possible patterns, this can be rewritten $O(n^2\mathsf{log}|\mathcal{A}|^k|\mathsf{P}|)$.

Therefore, at runtime, the complexity of mining is $O(|X|^t + Tt|\mathcal{A}|^k + n^2\mathsf{log}|\mathcal{A}|^k|\mathsf{P}|)$. This grows quickly with k and the size of $\mathcal{A}$ and, to a lesser extent, the size of X. Therefore, the two limiting factors for performance are the size of alphabet and number of quantified variables.

8.6 Demonstrating the Mining Process

We demonstrate the mining process by considering a system that deals with files. Assume that we have instrumented the system to produce events from the following alphabet:

$$\mathcal{A} = \{\texttt{open}(f), \texttt{read}(f), \texttt{write}(f), \texttt{close}(f), \texttt{delete}(f)\}$$

As there is a single quantified variable, and we want the property to hold for all files, we consider the quantification $\forall f$. We then take following trace

$$\begin{aligned}\tau_1 = \; & \texttt{open}(1).\texttt{write}(1).\texttt{open}(2).\texttt{read}(2).\texttt{read}(1).\texttt{close}(1).\texttt{write}(2).\\ & \texttt{open}(1).\texttt{read}(1).\texttt{write}(1).\texttt{close}(2).\texttt{open}(2).\texttt{read}(2).\texttt{write}(1).\\ & \texttt{read}(1).\texttt{delete}(1).\texttt{write}(2).\texttt{delete}(2)\end{aligned}$$

Lastly, let us take the pattern library given in Sec. 8.4.4 and therefore the pattern checkers given in Fig. 8.6 and Fig. 8.7, which we will call C_3 and C_2 respectively.

Our first task is to construct a scorecard for the trace. The relevant bindings are $\{[f \mapsto 1], [f \mapsto 2]\}$. Firstly we compute $\tau_1 \downarrow_{\mathcal{A}(\theta)}$ for each relevant binding as follows

$$\begin{aligned}\tau_1 \downarrow_{\mathcal{A}([f\mapsto 1])} &= \sigma_1 = \texttt{open}(1).\texttt{write}(1).\texttt{read}(1).\texttt{close}(1).\texttt{open}(1).\texttt{read}(1).\\ &\qquad \texttt{write}(1).\texttt{write}(1).\texttt{read}(1).\texttt{delete}(1)\\ \tau_1 \downarrow_{\mathcal{A}([f\mapsto 2])} &= \sigma_2 = \texttt{open}(2).\texttt{read}(2).\texttt{write}(2).\texttt{close}(2).\texttt{open}(2).\texttt{read}(2).\\ &\qquad \texttt{write}(2).\texttt{delete}(2)\end{aligned}$$

We then compute $\mathsf{G}(\mathsf{P}, \mathcal{A}, \theta)$ for each binding. As we have two pattern checkers we have

$$\mathsf{G}(\mathsf{P}, \mathcal{A}, \theta) = \{\langle \mathsf{C}_3, [a \mapsto \mathbf{a}, b \mapsto \mathbf{b}], \theta\rangle \mid \mathbf{a}, \mathbf{b} \in \mathcal{A}\} \cup \{\langle \mathsf{C}_2, [a \mapsto \mathbf{a}], \theta\rangle \mid \mathbf{a} \in \mathcal{A}\}$$

for each binding. As $|\mathcal{A}| = 5$ we have $|\mathsf{G}(\mathsf{P}, \mathcal{A}, \theta)| = 25 + 5$, giving us 60 event pattern checkers in total, 30 for each binding. Finally, we compute the set of instantiated patterns given by each trace for each event pattern checker.

Before we give the results, let us briefly consider the patterns extracted from σ_2 by the event pattern checker $\langle \mathsf{C}_3, [a \mapsto \texttt{close}(f), b \mapsto \texttt{delete}(f)], [f \mapsto 2]\rangle$. The following shows the states

Table 8.1: The reached states for traces σ_1 and σ_2 for the patterns and their event checker for the 2-pattern library, - indicates no states were reached.

φ		σ_1				σ_2			
a	b	p_1	p_2	p_3	C_3	p_1	p_2	p_3	C_3
`open`(f)	`read`(f)	–	–	–	–	–	–	–	–
`open`(f)	`write`(f)	–	–	–	–	–	–	–	–
`open`(f)	`close`(f)	–	3	–	$15 : \{p_2\}$	–	3	–	$15 : \{p_2\}$
`open`(f)	`delete`(f)	2	–	–	$6 : \{p_1\}$	2	–	–	$6 : \{p_1\}$
`read`(f)	`open`(f)	–	–	–	–	–	–	–	–
`read`(f)	`write`(f)	–	–	1	$4 : \{p_3\}$	–	–	1	$4 : \{p_3\}$
`read`(f)	`close`(f)	–	–	–	–	–	–	–	–
`read`(f)	`delete`(f)	2	–	2	$3 : \{p_1,p_3\}$	2	–	–	$6 : \{p_1\}$
`write`(f)	`open`(f)	–	–	–	–	–	–	–	–
`write`(f)	`read`(f)	–	–	1	$4 : \{p_3\}$	–	–	1	$4 : \{p_3\}$
`write`(f)	`close`(f)	–	–	–	–	–	–	1	$4 : \{p_3\}$
`write`(f)	`delete`(f)	2	–	–	$6 : \{p_1\}$	2	–	2	$3 : \{p_1,p_3\}$
`close`(f)	`open`(f)	–	–	–	–	–	–	–	–
`close`(f)	`read`(f)	–	–	–	–	–	–	–	–
`close`(f)	`write`(f)	–	–	–	–	–	–	1	$4 : \{p_3\}$
`close`(f)	`delete`(f)	2	–	2	$3 : \{p_1,p_3\}$	2	–	2	$3 : \{p_1,p_3\}$
`delete`(f)	`open`(f)	–	–	–	–	–	–	–	–
`delete`(f)	`read`(f)	–	–	2	$9 : \{p_3\}$	–	–	–	–
`delete`(f)	`write`(f)	–	–	–	–	–	–	2	$9 : \{p_3\}$
`delete`(f)	`close`(f)	–	–	2	$9 : \{p_3\}$	–	–	2	$9 : \{p_3\}$

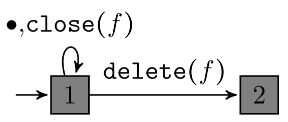
that the trace σ_2 passes through.

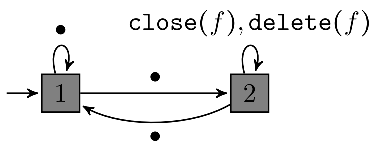
$$1 \xrightarrow{\texttt{open(2)}} 2 \xrightarrow{\texttt{read(2)}} 2 \xrightarrow{\texttt{write(2)}} 2 \xrightarrow{\texttt{close(2)}} 8 \xrightarrow{\texttt{open(2)}} 7 \xrightarrow{\texttt{read(2)}} 2 \xrightarrow{\texttt{write(2)}} 2 \xrightarrow{\texttt{delete(2)}} 3$$

Note that we can think of the processing part of the event pattern checker as rewriting the trace as follows:

$$\bullet.\bullet.\bullet.a.\bullet.\bullet.\bullet.b$$

Therefore, for this event pattern checker there are two passing patterns at state 3 in C_3, giving us the following two instantiated patterns:





If we apply this process to all event pattern checkers we reach the final states in the patterns and pattern checkers as given in Tables. 8.1 and 8.2. We include the final states in the individual patterns to show that the pattern checker achieves its aim of finding the patterns that match a trace. Here we can see that wherever one of the patterns reaches an accepting state the pattern checker reaches a state that produces that pattern.

We can now construct the sets of successful patterns. Table 8.3 describes the successful

Table 8.2: The reached states for traces σ_1 and σ_2 for the patterns and their event checker for the 1-pattern library, - indicates no states were reached

φ	σ_1				σ_2			
a	p_4	p_5	p_6	C_2	p_4	p_5	p_6	C_2
`open`(f)	2	–	–	$2 : \{p_4\}$	2	–	–	$2 : \{p_4\}$
`delete`(f)	–	3	2	$5 : \{p_5,p_6\}$	–	3	2	$5 : \{p_5,p_6\}$
`write`(f)	–	–	2	$4 : \{p_6\}$	–	3	2	$5 : \{p_5,p_6\}$
`read`(f)	–	3	2	$5 : \{p_5,p_6\}$	–	–	2	$4 : \{p_6\}$
`close`(f)	–	3	2	$5 : \{p_5,p_6\}$	–	3	2	$5 : \{p_5,p_6\}$

Table 8.3: The successful patterns.

a	b	Patterns
`open`(f)	`close`(f)	$\{p_2\}$
`open`(f)	`delete`(f)	$\{p_1\}$
`read`(f)	`write`(f)	$\{p_3\}$
`read`(f)	`delete`(f)	$\{p_1\}$
`write`(f)	`read`(f)	$\{p_3\}$
`write`(f)	`delete`(f)	$\{p_1\}$
`close`(f)	`delete`(f)	$\{p_3,p_1\}$
`delete`(f)	`close`(f)	$\{p_3\}$

a	Patterns
`open`(f)	$\{p_4\}$
`delete`(f)	$\{p_5,p_6\}$
`write`(f)	$\{p_6\}$
`read`(f)	$\{p_6\}$
`close`(f)	$\{p_5,p_6\}$

patterns, as we have universal quantification this is the set of instantiated patterns that hold in both traces.

These can then be combined to give a final pattern that can be converted into a SEA. Figure 8.9 gives the combined open automaton that is the result of combining all successful patterns in Table 8.3. Figure 8.10 gives the resulting SEA. Holes can be removed as we can take the universal set of symbols to be the alphabet.

8.7 Connectedness extension

We introduce the additional input of a global guard to the mining process. The notion is straightforward: the input global guard is used to restrict the bindings used to select patterns. However, by also allowing the *connectedness* global guard (see Sec. 3.5.8) we can improve the mining process considerably.

We begin by motivating the usefulness of a connectedness mode and then describe the additional mechanisms required to achieve it, finishing with an example. Note that this extension also allows us to include other global guards over the quantified variables, for example that they are not equal.

8.7.1 Why connectedness is useful

Let us briefly remind ourselves what connectedness is. As first described in Section 3.5.8, we can introduce a special global guard that takes the trace seen so far as input and only evaluates to true for a binding if there exist events in the trace that 'connect' the binding's contents.

In Definition 31 on page 70 we defined the notion of a connected binding as follows. Given

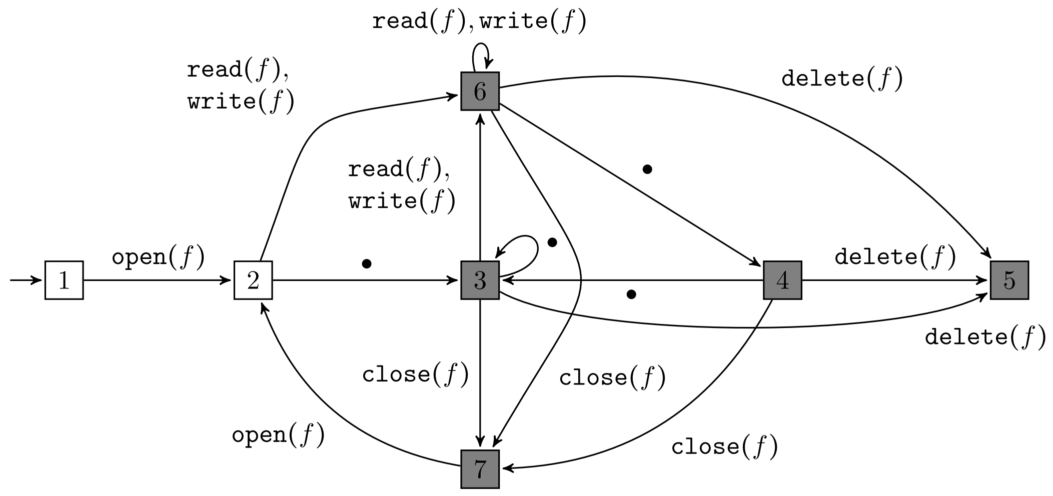


Figure 8.9: The combined open automaton with holes.

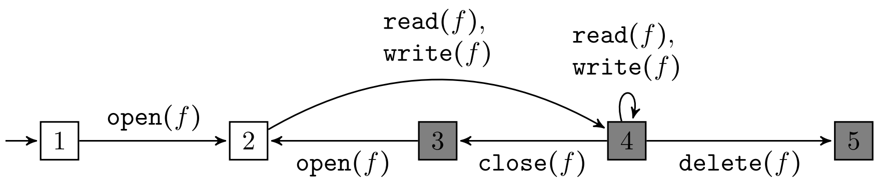


Figure 8.10: The resultant SEA.

a trace τ and an EA $\mathcal{E}$ with alphabet $\mathcal{A}$, let the set of τ-connected bindings be the smallest set $C(\tau)$ such that:

$$\begin{array}{lll} \varphi & \in C(\tau) \text{ iff} & \exists \mathbf{a} \in \tau, \exists \mathbf{b} \in \mathcal{A} : \varphi \sqsubseteq \mathsf{quantified}(\mathsf{match}(\mathbf{a}, \mathbf{b})) \\ \varphi_1 \sqcup \varphi_2 & \in C(\tau) \text{ iff} & \varphi_1, \varphi_2 \in C(\tau) \wedge \varphi_1 \cap \varphi_2 \neq \varnothing \end{array}$$

For example, the binding $[c \mapsto 1, i \mapsto 2]$ is connected if $\texttt{iterator}(c, i) \in \mathcal{A}$ and $\texttt{iterator}(1, 2)$ is in the trace. Here the intuition is that when inspecting method calls, if we see two (or more) objects used in the same method then it is likely that the behaviour of those objects is (in some way) connected. Connectedness mode is therefore useful for two reasons:

1. Extracted specifications are simpler. We saw this previously when using the connectedness guard to specify QEA. As the behaviours of unconnected objects are ignored we do not need to specify what this behaviour should be.

2. Extracted specifications are more accurate (and more likely to be found). As traces related to unconnected bindings can be ignored the mining process will not be effected by the 'noise' they introduce. Additionally, in the case where the patterns in the pattern library are unable to capture the complex behaviour needed to describe the behaviour of unconnected events it is more likely that the pattern library can explain the more simple connected behaviour.

We explore these claims later (Sec. 10.3.4).

We note that the specification mining tool JMiner [LCR11] using a similar notion of projection to our tool but runs exclusively in connected mode i.e. only considers the behaviour associated with connected bindings.

8.7.2 Including global guards

Extending the mining framework to also use a global guard G as an input is straightforward. We redefine successful patterns (previously Def. 76) as follows:

Definition 78 (Successful patterns)**.** *Given a scorecard* $\mathsf{S}(\tau)$*, the successful patterns for the list of quantifications* Λ *and global guard* G *are given by* $\mathsf{pat}(\mathsf{S}(\tau), [\ ], \Lambda, G)$*, defined as*

$$\begin{array}{lcl}
\mathsf{pat}(\mathsf{S}, \theta, \forall x \Lambda', G) & = & \bigcap_{d \in \mathsf{Dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger [x \mapsto d], \Lambda', G) \\
\mathsf{pat}(\mathsf{S}, \theta, \exists x \Lambda', G) & = & \bigcup_{d \in \mathsf{Dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger [x \mapsto d], \Lambda', G) \\
\mathsf{pat}(\mathsf{S}, \theta, \forall x \epsilon, G) & = & \bigcap \{\mathsf{S}(\theta \dagger [x \mapsto d]) \mid d \in \mathsf{Dom}(\tau)(x) \wedge G(\theta \dagger [x \mapsto d])\} \\
\mathsf{pat}(\mathsf{S}, \theta, \exists x \epsilon, G) & = & \cup \{\mathsf{S}(\theta \dagger [x \mapsto d]) \mid d \in \mathsf{Dom}(\tau)(x) \wedge G(\theta \dagger [x \mapsto d])\}
\end{array}$$

This ignores all bindings that do not pass the global guard G, i.e. we do not include their successful patterns.

To compute connectedness we do not need to follow the method described in Sec. 6.3.3 as we are provided with whole traces. Instead, we can first extract all bindings from the trace by matching each event in $\mathcal{A}$ with each event in the trace, and then close this set by repeatedly taking the union of any two bindings with a non-empty intersection.

8.7.3 An example

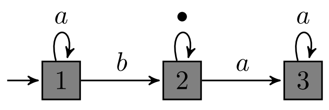
To demonstrate mining in connected mode let us consider the following trace that could be created by iterating over two separate collections:

$$\texttt{iterator}(A,1).\texttt{use}(1).\texttt{use}(1).\texttt{update}(B).\texttt{iterator}(B,2).\texttt{use}(2).\texttt{update}(A).\texttt{update}(A)$$

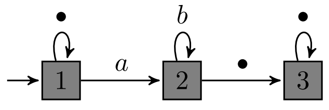
Firstly, we can create the following trace projections for the three relevant bindings:

	c	i	$\tau \downarrow_{\mathcal{A}(\theta_i)}$
θ_1	A	1	`iterator`(A,1).`use`(1).`use`(1).`update`(A).`update`(A)
θ_2	A	2	`use`(2).`update`(A).`update`(A)
θ_3	B	1	`use`(1).`use`(1).`update`(B)
θ_4	B	2	`update`(B).`iterator`(B,2).`use`(2)

Note that bindings θ_1 and θ_4 are connected, whilst θ_2 and θ_3 are not. Let us now take the following patterns as a pattern library:



p_1

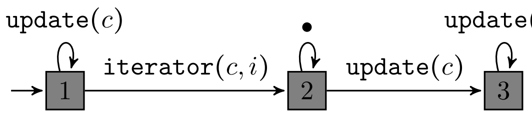


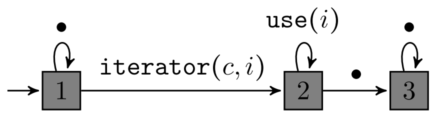
p_2

And consider the successful patterns. The following table shows for each instantiation of a and b which patterns hold for each of the trace projections. Given our updated notion of successful patterns we only need to take the intersection of those patterns associated with connected bindings, here identified as $\theta_1 \cap \theta_4$. If we do not filter the results in this way then there would be no successful patterns.

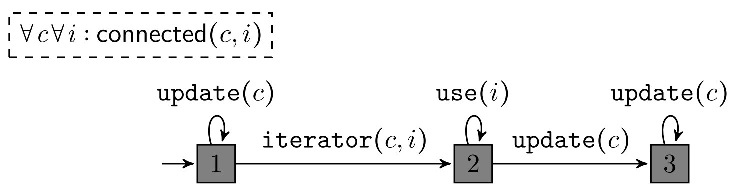
a	b	θ_1	θ_2	θ_3	θ_4	$\theta_1 \cap \theta_4$
`iterator`(c,i)	`update`(c)					
`iterator`(c,i)	`use`(i)	p_1,p_2	p_1		p_2	p_2
`update`(c)	`iterator`(c,i)	p_1		p_2	p_1, p_2	p_1
`update`(c)	`use`(i)		p_1			
`use`(i)	`iterator`(c,i)		p_2			
`use`(i)	`update`(c)		p_2	p_1	p_1	

The two successful instantiation patterns are:





Which, when combined, give us the following specification of the UnsafeIter property:



8.8 Alternative inference techniques

We compare our approach to existing work in the area of specification mining and specification inference. We provide information about alternative approaches for completeness.

8.8.1 Generate-and-check specification mining approaches

No previous generate-and-check techniques consider the parametric setting. We therefore extend the expressiveness of this approach (discussed in Sec. 8.1). We also extend the notion of pattern using open automata. Unlike previous approaches we assume a given alphabet.

One of the main idioms behind previous generate and check techniques is that frequent behaviour is correct. We lose this notion in our work as we do not distinguish between patterns that 'occur' frequently and those that do not. This may lead to infrequent and unnecessary behaviour being taken as a requirement. We could include frequency in two ways; by measuring the (respective) lengths of projected slices for pattern alphabets, or the (respective) lengths of trace slices.

We do not focus on efficiency, but it is likely that including additional information makes our approach less efficient than propositional techniques.

8.8.2 Static approaches

There are a large range of techniques that generate specifications using source code [LZ05a, WZL07, RGJ07, SYFP07, KTB+06, LPH+07, WML02, DLWZ06]. We review the common approach of frequent itemset mining [GZ03] here. This technique, from data-mining, takes a set of itemsets (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.

PR-Miner [LZ05a] finds associations among elements by identifying those which are frequently used together, with no explicit concept of order. Each function is turned into an itemset by taking the set of all symbols. Frequent itemset mining is applied to extract patterns of the form $set_1 \Rightarrow set_2$ read as 'if we see the symbols in set_1 then we should see the symbols in set_2'. Jadet [WZL07] takes a similar approach but uses a form of call-graph to construct their itemsets. An item is an ordered pair (m, n) indicating that n can be called after m and a function is turned into an itemset by extracting the valid pairs (m, n). Ramanathan et al. [RGJ07] use a combination of frequent itemset mining and sequential pattern mining [MTP05] to develop a technique called *predicate mining* that identifies the preconditions that must hold whenever a procedure is called. Flow-analysis labels program statements with sets of predicates, frequent itemset mining infers data-flow predicates, where an itemset represents the predicates at a call point, and sequential pattern mining infers control-flow predicates.

We do not consider extracting specifications from source code in our work.

8.8.3 Regular inference

Regular inference asks whether a finite state machine can be extracted from a set of traces. Here we briefly review the main concepts and algorithms. This is an alternative to the generate-and-check approach taken by previous mining techniques that make use of slicing - Secondly, we compare with other tools. We are strictly more expressive than previous work that uses trace slicing i.e. JMiner [LCR11] and Pradel and Gross [PG09]. Our approach is more expressive than these as we consider existential quantification and, more importantly, non-connected bindings.

Nerode's right congruence

Nerode's right congruence $\equiv_{\mathcal{L}}$ is a congruence on the set of words Σ^* for some regular language $\mathcal{L}$ that splits 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. It is given by the following for $u, v \in \Sigma^*$

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

This has a number of consequences, the main one being that two states accepting the same sets of words can be merged as they cannot be differentiated.

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 $\delta(\mathbf{u}_{\mathcal{L}}, a) = \mathbf{ua}_{\mathcal{L}}$.

Passive state-merging

The main approach to passive regular inference is that of state-merging. The general idea is to construct an automaton that accepts and rejects exactly those traces given (called an augmented prefix tree acceptor) and then merge states to generalise this.

From Nerode's right congruence we can derive a condition for exact learning with passive information. We say that a sample is characteristic if for every transition and state in the hidden automaton there is a positive trace that visits them, and for every two non-equivalent states there exist traces that can separate them. A characteristic set is of size $O(|Q|^2|\Sigma|)$, which is reasonable, but there is no guarantee that we can extract this. The Regular Positive Negative Inference (RPNI) [OG91, OGV93, Lan92] algorithm begins by constructing the prefix tree acceptor for the positive sample set S^+ and then attempts to merge each pair of states. If the resulting automaton is consistent with S^- it is kept and the process repeated until no more pairs of states can be merged. There have been some refinements to help select pairs of states to merge. The Blue-Fringe algorithm [LPP98] maintains a subset of mergeable states and the (Evidence Driven State Merging) EDSM [CK02] algorithm scores a merge pair by comparing the state transitions from each state. If the sample of traces is not characteristic then learning will be approximate. Note that negative traces are required to form a characteristic sample.

Another approximate approach is that of Biermann. The ktails algorithm [BF72] merges two states if they have identical suffixes of length k, so the level of approximation can be controlled using k. Leucker [Leu07] notes that this algorithm can be reduced to a constraint satisfaction problem over the natural numbers. This algorithm has been extended in a number of ways. Most notably the sk-strings [RP97] algorithm infers a probabilistic finite state automaton (PFSA) where states are merged if they agree on the top s percent of their most probable k-strings

There have been a few parametric extensions. GkTail [LMP08] combines the Daikon invariant-detection tool with the k-tails approach to mine a form of Extended Finite State Machine (EFSM). Traces are annotated with Daikon and used to construct a canonical automaton accepting all traces. States which share the same k-futures are then merged, where state equivalence can be based on different forms of predicate subsumption. JMiner [LCR11] mines parametric specifications with universally quantified variables using sk-strings to infer a PFSA from trace slices. A recent approach [WTD13] extracts EFSMs with 'local' guards (i.e. based only on the values in the transition label) by combining classifiers with the EDSM approach. They train classifiers (taken from WEKA [HFH+09]) to predict the next parametric event given a previous parametric event and then use this information to restrict merges.

Active learning

An active learning approach to grammar inference was first taken by Dana Angluin [Ang87]. She introduced the seminal L^* algorithm which has been studied, extended and applied in many places in the literature. Berg et al. [BJLS03] give a good introduction to the algorithm that is more approachable than the original paper.

The setup has a *teacher* or *oracle* who can answer *queries* put to it by the *learner*. Queries can either be *membership* queries (is this word in the language) or *equivalence* queries (is this the answer). 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

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

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

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 $T(u)(\epsilon) = \mathit{true}$. If the teacher 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 exactly learn a canonical minimal DFA in time polynomial in the number of states and the maximum counterexample length. For this reason oracle implementations aim to reduce counterexample length.

This approach has been extended to inexperienced teachers [Leu07, FGMP94], Mealy Machines [SG09], infinite languages [MP95] and timed automata [GJL10].

In recent years there has been much interest in extending L* to the parametric setting. Some make use of abstraction and refinement to deal with parametric events through guards [BJR06, AHK$^+$12] as well as learning an automaton they learn an abstraction function. More recently there has been an interest in forms of Register Automata. Both Howar et al. [HSJC12] and Bollig et al. [BHLM13] define regular canonical representations for their target automata using symbolic representations of data words. Howar et al. [HSJC12] extract a subclass of Register Automata (described in [CHJ$^+$11]) that restrict the way state variables are stored. Bollig et al. [BHLM13] extract so-called *session automata* that can either write a value into a register as a fresh value, or read an existing value and compare it against the input. The tool TzuYu [XSL$^+$13] makes use of a refiner to split alphabet symbols using guards when two states cannot be distinguished. They implement their refiner using support vector machines [Joa98] and therefore their approach only works if states are linearly separable. All of these techniques need to translate propositional queries into parametric ones, and parametric counterexamples into propositional ones.

These active approaches focus on free notion of data, rather than the quantified view we take in our work.

8.8.4 Data mining

Lo et al. [LKL07, LKL08a, LKL08b, LK08, LRRV09] have carried out extensive work using the data-mining technique of frequent itemset mining to mine specifications. Their general approach is to prune the traces with respect to support before generating patterns they are confident in. This basic approach (two event patterns) has complexity $O(n + ab)$ 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 produced sets of closely occurring events but later work introduces orderings between these events. One interesting approach [LMP09] uses the inferred temporal patterns to steer the previously described ktails algorithm for regular inference.

The same group have also produced parametric extensions. For example, they mine live sequence charts enriched with invariants by first identifying frequent charts and then using these to identify subtraces for invariant mining [LM12]. They also produce the TARK [LCH+09, LRRV12] tool that identifies predetermined quantified binary temporal rules with equality constraints.

Our approach is more expressive than Tark as we consider general orderings of events.

8.8.5 Evolutionary approaches

The earliest attempt at grammar inference using genetic algorithms was by Fogel et al. [FOW66] in 1966, who attempted to evolve DFAs for regular languages. More recently Dupont presented the GIG method [Dup94] that evolves a partitioning of states and Pawar and Nagaraja [PN02] extend this work with different forms of structural mutations. Hingston [Hin01] attempts to evolve a partitioning using the concept of Minimum Message Length [WB68] to measure fitness.

Lucas and Reynolds [LR05, LR03] evolve the transition function for a DFA and Niparnan and Chongstitvatana [NC02] evolve finite state machines in the form of Mealy Machines. Bongard and Lipson [BL05] have developed an *active* approach which they describe as *coevolutionary.*

There have also been approaches that look at context-free languages. Lankhorst [Lan95] and Naidoo and Pillay [NP07] evolve pushdown automata, whilst Wyard [Wya93] and Pandey [Pan10] evolve grammars directly.

8.9 Summary

In this chapter we have introduced a method for extracting a form of QEA from traces making use of a set of predefined patterns. We began by discussing the form these patterns should take in Sec. 8.3 before introducing *open automata* as a formalism that allow us to combine extracted patterns together. We then explored methods for efficiently generating and checking patterns in Sec. 8.4, showing that we can use a set of patterns to produce a checker object that checks lots of patterns simultaneously. We used the work of the first two sections to introduce our mining framework in Sec. 8.5, which is demonstrated in Sec. 8.6. Finally, we extended our framework with a notion of the connectedness global guard in Sec. 8.7.

8.9.1 Limitations

We discuss the limitations of our technique and what further work is required to address these.

Correct traces

One limitation of this technique is the assumption that all given traces are correct, therefore if a pattern is satisfied by 99% of a trace it will not be recorded. We address this in Chapter 11.

Pattern library

Another limitation is that we must provide a pattern library. On one hand, we may fail to extract a specification if the library is too general as only specifications that are some combination of patterns in the pattern library can be mined. On the other hand, we may fail to generalise from the given traces if the pattern library is too specific. This limitation is inherent in our approach and in the next chapter we will address it by exploring what makes a good pattern library.

Prior knowledge

An alphabet must be supplied, requiring some prior knowledge of the system under inspection. This may be addressed with additional computation. For example, we could attempt to enumerate all possible alphabets given a set of traces. However, this would become inefficient quickly and examining the source code heuristically, as is done in the JMiner work [LCR11], is likely to be necessary. This need not be a restriction, as in many applications we know the events whose behaviour we wish to mine.

Free variables

Finally, our current approach only targets QEA without free variables. One of the main motivations for introducing QEA was that they included free variables and these are necessary for writing expressive and concise specifications. We have developed an initial approach for extracting QEA with free variables but as this is not fully developed we discuss later when considering further work (Sec. 12.2.8).

Chapter 9

Exploring Pattern Libraries

The previous chapter introduced our mining process extracting a form of QEA from execution traces. This mining process depends on a pattern library. What such a pattern library should look like and how it should be developed is not obvious. Here we begin to address the question "what makes a good pattern library?"

Ideally we would produce a general 'pattern base' i.e. a pattern library from which all specifications could be generated. As shown in Sec. 8.3 this is not possible without placing restrictions on the size of alphabets. Furthermore, due to the issues of overspecification discussed in Sec. 8.2 we do not want to be able to generate overly specific specifications as (without sufficient negative traces) this will prevent generalisation from the traces, which is our aim. Therefore, a pattern library should form a good basis for a particular domain i.e. be able to generate specifications for general forms of common behaviour. Throughout this chapter we consider methods for developing such patterns; for example, in Sec. 9.4 we consider specifications of common forms of behaviour and the patterns required to generate them.

Structure. We begin by discussing what we want from a pattern library (Sec. 9.1). This is followed by a review of previous notions of pattern used in specification and mining (Sec. 9.2). Next we ask whether suitable patterns can be generated automatically (Sec. 9.3), the answer is generally 'no' but the discussion identifies interesting methods to automatically expand or augment an existing library. Previous patterns for mining are based on general intuition about useful or common behaviours and we build on this by looking at the patterns obtained when decombining common specifications (Sec. 9.4). We conclude by summarising the pattern library we will use for evaluation (Sec. 9.5).

9.1 What do we want from a pattern library?

To understand what we want form a pattern library let us look at how the choice of pattern library effects the specification extracted.

9.1.1 Solution space

Given a pattern library P and an alphabet $\mathcal{A}$ we can generate the *solution space* i.e. the set of specifications that could possibly be extracted from a set of traces. This is simple to construct as follows:

$$\begin{aligned}\mathsf{instances}(\mathsf{P},\mathcal{A}) &= \{p(\varphi) \mid p \in \mathsf{P} \wedge \forall a \in p.\Sigma, \exists \mathbf{a} \in \mathcal{A} : \varphi(a) = \mathbf{a}\}\\ \mathsf{solutions}(\mathsf{P},\mathcal{A}) &= \{\textstyle\bigcap \mathsf{I} \mid \mathsf{I} \subseteq \mathsf{instances}(\mathsf{P},\mathcal{A})\}\end{aligned}$$

i.e. all possible combinations of instances of patterns in P using events in $\mathcal{A}$. Let us consider the *size* and *shape* of the solution space.

The size of the solution space

Given a pattern library P containing p_k patterns with k symbols and alphabet $\mathcal{A}$ consisting of a events we have that

$$\begin{aligned}|\mathsf{instances}(\mathsf{P},\mathcal{A})| &= \textstyle\sum_{k=1}^{k=n} p_k a^k\\ |\mathsf{solutions}(\mathsf{P},\mathcal{A})| &= 2^{|\mathsf{instances}(\mathsf{P},\mathcal{A})|}\end{aligned}$$

If we take $p_2 = 50$ and $p_3 = 10$. with $p_k = 0$ for all other k. and $a = 5$ then we have $|\mathsf{instances}(\mathsf{P},\mathcal{A})| = 12.5k + 12.5k = 25k$ and $|\mathsf{solutions}(\mathsf{P},\mathcal{A})| = 2^{25k}$, which is obviously a vast solution space. However, the solution space will be significantly smaller than this for the following reasons:

1. Repetitions in $\mathsf{instances}$. If we instantiate a pattern with two or more of the same symbol then there is a chance that the language of this pattern collapses to that of another pattern, or to the trivially total language. For example take the pattern that captures an alternating relationship between two symbols interspersed by holes, if these are the same symbols then any trace matches this pattern.

2. Empty combinations in $\mathsf{solutions}$. If the language intersection between two patterns is empty then their combination will represent the empty language. This is highly likely to occur as patterns are designed to capture different behaviours. Additionally, it is unlikely for instances of the same pattern to be compatible, for example if a pattern states that the trace must begin with a certain symbol then two instances of this pattern will have an empty combination.

3. Repeated combinations in $\mathsf{solutions}$. If we combine patterns p_1, p_2, p_3 where $p_3 \sqsubseteq p_1$ and $p_3 \sqsubseteq p_2$ then their combination will be the same as the combination for p_1 and p_2. This is likely to occur where we have patterns in our pattern library that include others. However, this is often necessary for capturing slight variations of a pattern.

Therefore, to maximise the size of the solution space we should ensure that patterns are mostly distinct yet there are enough similarities to allow for compatible combinations.

The 'shape' of the solution space

Let us consider the *precision* hierarchy of the solution space, recall that if p_1 is more precise than p_2 then $\mathcal{L}(p_1) \subseteq \mathcal{L}(p_2)$. Firstly, take the instances. We would expect a reasonably flat structure as most instances would not contain another instance. However, two or three levels might be observed if we had different variations of the same kind of pattern.

The solutions would experience a much more complex structure. The maximum height is $log_2|\mathsf{instances}(\mathsf{P}, \mathcal{A})|$ if every pair of instances were consistent. However, this is unlikely. Again, the more distinct the patterns in the pattern library, the greater complexity and depth we will see in the precision hierarchy.

We want our solution space to exhibit a deep precision hierarchy as this shows that we can differentiate between different levels of preciseness in a specification i.e. we do not just return an over-general fit to the traces.

9.1.2 Exploring the solution space

Our approach can be phrased as an exploration of the solution space. We are looking for the smallest/most precise pattern that satisfies our traces i.e.:

$$\mathsf{min}(\{p \in \mathsf{solutions}(\mathsf{P}, \mathcal{A}) \mid \forall \tau \in T_+ : \tau \in \mathcal{L}(p) \land \forall \tau \in T_- : \tau \notin \mathcal{L}(p)\})$$

We do this by identifying the instances that satisfy our traces and combining them i.e.:

$$\bigcap\{p \in \mathsf{instances}(\mathsf{P}, \mathcal{A}) \mid \tau \in T_+ : \tau \in \mathcal{L}(p) \land \forall \tau \in T_- : \tau \notin \mathcal{L}(p)\}$$

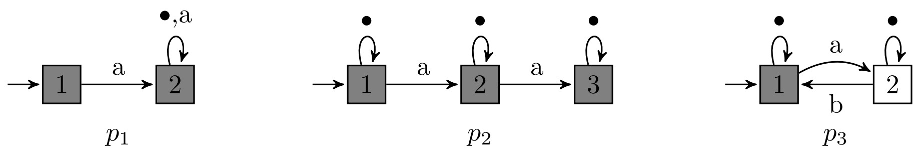
This is necessarily the minimum solution, as if there were a smaller/more precise solution it would have to be based on the combination of additional instances, but by construction there are no other compatible instances.

9.1.3 The over-specification problem

The mining process detects successful patterns and then takes their intersection. If a combination of patterns exactly describes the set of input traces then this is the specification that will be returned. The whole notion of specification mining is to generalise from a set of traces and a pattern-mining approach achieves this through using patterns that are appropriately generic.

An example of over-specification

Consider a pattern library that contains the following patterns:



Given the traces

$$\tau_1 = \texttt{f.g.h}$$
$$\tau_2 = \texttt{f.g.h.f.g.h}$$

we would extract the patterns

$$p_1([a \mapsto \texttt{f}]), p_2([a \mapsto \texttt{f}]), p_2([a \mapsto \texttt{g}]), p_2([a \mapsto \texttt{h}]),$$
$$p_3([a \mapsto \texttt{f}, b \mapsto \texttt{g}]), p_3([a \mapsto \texttt{f}, b \mapsto \texttt{h}]), p_3([a \mapsto \texttt{g}, b \mapsto \texttt{h}])$$

and combine these to form the following specification:

$$\begin{tikzcd}
{} \arrow[r] & \boxed{1} \arrow[r, "\texttt{f}"] & \boxed{2} \arrow[r, "\texttt{g}"] & \boxed{3} \arrow[r, "\texttt{h}"] & \boxed{4} \arrow[r, "\texttt{f}"] & \boxed{5} \arrow[r, "\texttt{g}"] & \boxed{5} \arrow[r, "\texttt{h}"] & \boxed{5}
\end{tikzcd}$$

The unfolding of p_3 is caused by p_2, which captures the fact that each event occurs at most twice. However, from manually inspecting the traces we might conclude that these are special instances of the more general specification:

$$\begin{tikzcd}
{} \arrow[r] & \boxed{1} \arrow[r, "\texttt{f}"] & \boxed{2} \arrow[r, "\texttt{g}"] & \boxed{3} \arrow[ll, bend left, "\texttt{h}"]
\end{tikzcd}$$

We can either think of this as an over-specification in our pattern library, or a problem with the quality of the information received. If we had also received the trace $\tau_3 = \texttt{f.g.h.f.g.h.f.g.h}$ then we would have returned the more general specification.

9.1.4 The under-specification problem

This is a more common problem than over-specification. If we do not include a necessary pattern in our pattern library then it is possible that we have no compatible instances in our solution space, and no solution is returned. One cause of this is that there are small errors in the input traces, we consider this scenario in Chapter 11.

An example of under-specification

Let us consider a similar situation to the over-specification example. Consider a pattern library that contains the following patterns:

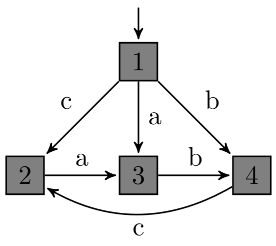
$$\begin{tikzcd}
{} \arrow[r] & \boxed{1} \arrow[r, "a"] & \boxed{2} \arrow[loop above, "\bullet{,}a"]
\end{tikzcd}
\qquad
\begin{tikzcd}
{} \arrow[r] & \boxed{1} \arrow[loop above, "\bullet"] \arrow[r, bend left, "a"] & \boxed{2} \arrow[loop above, "\bullet"] \arrow[l, "b"]
\end{tikzcd}$$

p_1 $\qquad$ p_2

Given the traces

$$\tau_1 = \texttt{f.g.h}$$
$$\tau_2 = \texttt{g.h.f.g.h.f.g.h}$$

we would not extract any patterns as the two traces begin with different events and no two events are in complete alternation as all pairs finish in state 2 of pattern p_2. Extrapolating from the two traces we might conclude that the two traces belong to the following general specification, that there is an order between events but no start or end event:



9.1.5 Summary

We have seen that the choice of patterns can greatly effect the size and shape of the solution space. Therefore, our aim is to develop a set of patterns that capture *distinct* yet *common* patterns of behaviour in traces. In the rest of this chapter we attempt to develop such patterns through exploring previous work in the area of specification patterns, automatic generation, and inspecting common specification forms.

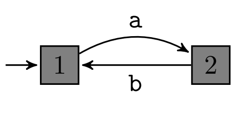
9.2 Previous work on patterns

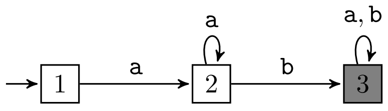
The idea of using patterns as general templates to describe common (program) behaviours has been explored previously. In some cases there has been a focus on small patterns that can be used to build larger patterns, but in others the focus has been on single independent patterns. Here we review previous work and draw some conclusions about which elements of this work we want to include in our approach.

Recall that we are interested in finite-state, finite-trace properties i.e. ones that can be described by finite-state automata. As patterns are applied in a propositional setting we will not be considering quantification in this section.

9.2.1 General concepts of safety and co-safety

We considered the high-level classification of finite-state, finite-trace properties of *safety* and *co-safety* previously in Section 4.1.3. In terms of patterns these represent a distinction between patterns that capture behaviour that should always occur and are prefix closed (all states are accepting) and behaviour that should eventually occur (some set of end states are accepting). The following gives an example of a safety (left) and co-safety (right) pattern.





Patterns do not to fall strictly within these classifications, therefore it would be more accurate to say that there are three classes with the third being a combination of safety and co-safety.

9.2.2 Patterns used for specification

Here we explore how patterns have been used as templates for specifying programs.

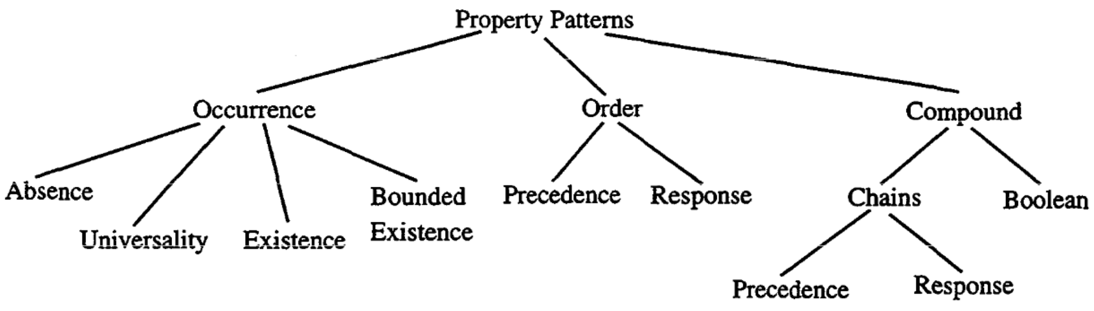


Figure 9.1: The pattern hierarchy (copied from [DAC99]).

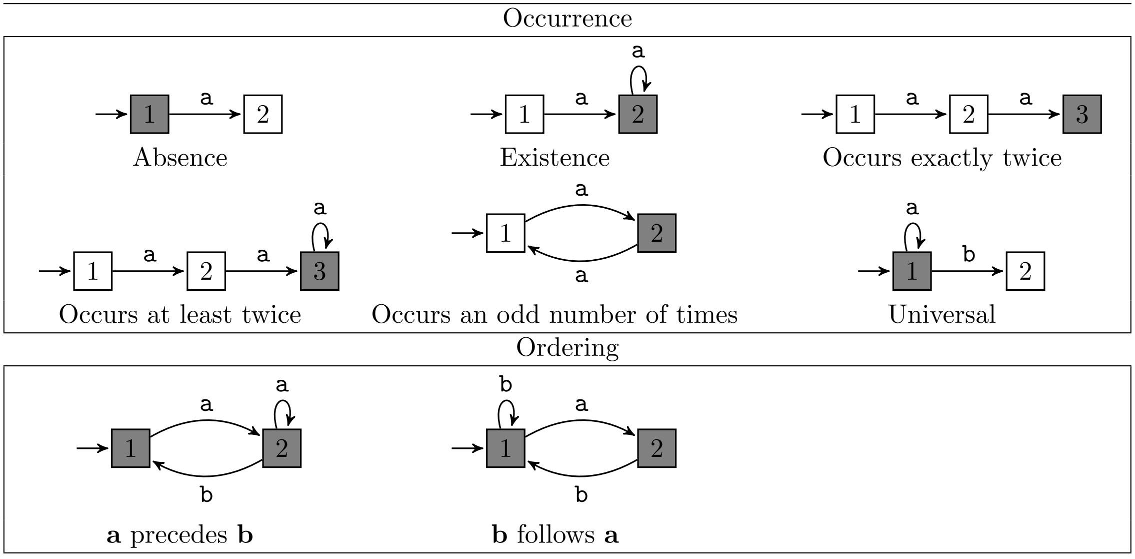


Figure 9.2: Demonstrating the pattern hierarchy of [DAC99].

Specification Pattern System

In 1999 a survey was carried out to collect examples of temporal specifications used in industry and academia. The result was the Specification Pattern System (SPS) [DAC99], a hierarchy for discussing finite-state properties. Examples were given in Linear Temporal Logic, Computation Tree Logic, Graphical Interval Logic and Quantified Regular Expressions.

These specifications are slightly different from the kind we are concerned with as they are over program states, therefore many propositions can hold on the same time step as many things are true about the current program state, as opposed to our view where a single event occurs on each time step.

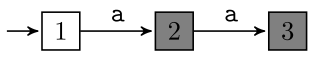
The hierarchy, illustrated in Fig. 9.1, organises different kinds of patterns in terms of how events should occur in a trace. We give an overview of the kinds of patterns in the following, and give illustrative examples in Fig. 9.2.

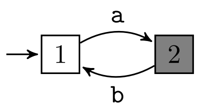
- **Occurrence.** About the occurrence of events in a trace.
  - **Absence.** An event does not occur. This is captured by a single transition to a non-final state.

  - **Existence.** An event must occur. This is captured by a single transition to a final state.
  - **Bounded Existence.** An event must occur at least/at most k times. Here we illustrate three versions of this property. The first two capture an event occurring exactly and at least twice, the third captures an event occurring an odd number of times. This last property probably falls outside the usual interpretation of this pattern.
  - **Universality.** An event occurs throughout. In our setting (where only one event can occur on each step) this is equivalent to the absence of all other events.

- **Ordering.** About the ordering of events in a trace.
  - **Precedence.** An event **a** must always be preceded by event **b**. This is captured by alternation with a loop on the second state. This means that we can have **a**.**a**.**b** but not **b**.**a**.**b** i.e. **b** cannot occur without being preceded by **a**.
  - **Response.** An event **a** must always be followed by event **b**. This is often called *request-response* where **a** is seen as a request, and **b** as the response. We capture this in a symmetric fashion to precedence; a **a** cannot occur unless it is immediately followed by a **b**.

- **Compound.** About the combination of patterns. We do not illustrate these as they are natural extensions of the previous patterns.
  - **Chain Precedence.** A sequence of precedence relationships between a sequence of events.
  - **Chain Response.** A sequence of response relationships between a sequence of events.
  - **Boolean Combinations.** We can take boolean combinations of other patterns. This might benefit from illustration. For example, we might take the disjunction of a occurs exactly once and a occurs exactly twice, or the conjunction of **a** precedes **b** and **b** follows **a**, which give the following two patterns respectively:





These patterns are combined with a notion of scope i.e. when in the trace they must apply. These are illustrated in Fig. 9.3 and are as follws:

- Global - apply to the whole trace.
- Before R - must hold on the finite trace before R first occurs.
- After Q - most hold on the finite trace after Q first occurs.
- Between Q and R - most hold on any subtrace between the first occurrence of Q and the first occurrence of R.

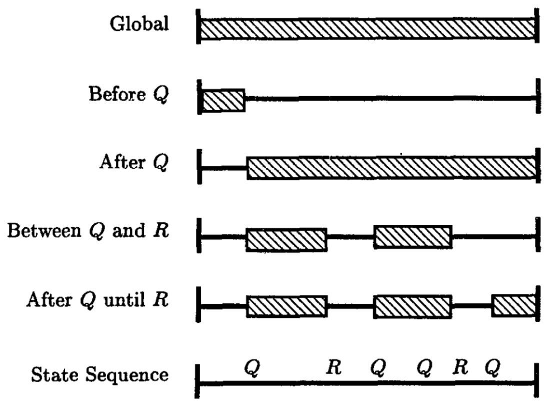


Figure 9.3: Illustrating scopes (copied from [DAC99]).

- After Q Until R - the same as between Q and R, except if no R occurs before the end of the trace the property should still hold.

Whilst many of the specification languages cannot capture these scopes directly the authors note that they are very important in practice. Note that some patterns (for example Universality) have significantly different meanings when combined with a scoping operator.

Extending SPS

This hierarchy has been extended and adapted in different ways.

Mondragon et al. [MGR] describe the Prospec tool for specifying properties and replace the notion of chain patterns with *composite patterns*. Composite patterns identify relationships between propositions and can be interpreted for either sequential or concurrent behaviour. These patterns can assert at least one proposition occurring, many happening in parallel or in some order consecutively or not. Salamah et al. [SGKR07] translate this notion of composite pattern back into LTL.

In 2005, Konrad and Cheng [KC05] extended the hierarchy to consider *real time* patterns (they label the previous patterns 'qualitative'). They are concerned with the amount of time a state formula (assertion about the state of the system) should hold. These can be captured in an event-based system by generating an event when the state formula changes truth value. The new categories include state formulas holding for given durations or occurring with a certain period, and events happening with a minimum or maximum time between them.

As mentioned above, the SPS framework focused on state propositions. At a similar time Chechick and Păun [CP99] considered how to interpret the system using an event system. A key concern of theirs is *stuttering*, where the interpretation of a property is changed by adding transitions that leave the system in the same state. The problem with formulas that can stutter is that they place restrictions on the level of abstraction being used. The main issue is, obviously, with the next operator.

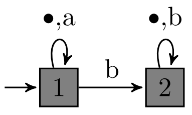
Table 9.1: Temporal property patterns used in Perracotta, adapted from [YE04b]. QRE have been replaced by normal regular expressions.

Name	Regular Expression	Valid Examples	Invalid Examples
Response	$b^*(a^+b^+)^*$	f.g.g.f.f	f.g.g.f.f.g
Alternating	$(ab)^*$	f.g.f.g	f.g.g, f.f.g, g.f.g
MultiEffect	$(ab^+)^*$	f.g.g	f.f.g, g.f.g
MultiCause	$(a^+b)^*$	f.f.g	f.g.g, g.f.g
EffectFirst	$b^*(ab)^*$	f.g.f	f.g.g, f.f.g
CauseFirst	$(a^+b^+)^*$	f.f.g.g	g.f.g.g, g.f.f.g
OneCause	$b^*(ab^+)^*$	g.f.g.g	f.f.g.g, g.f.f.g
OneEffect	$b^*(a^+b)^*$	g.f.f.g	f.f.g.g, g.f.g.g

Summary

We will keep the general hierarchy of the SPS framework, but refine it slightly given our event-based setting. We do not consider stuttering, although note that the user will need to select the correct level of abstraction to extract certain specifications. We present our pattern hierarchy and selected patterns at the end of this chapter.

We will capture *scope* using additional patterns, such as the following pattern that specifies that all occurrences of event a occur before the first occurrence of event b.



9.2.3 Pattern-based specification mining

We now consider the patterns used by previous pattern-based specification mining tools.

Initial work

Engler et al. [ECH+01] consider some language-specific patterns, for example *do not dereference null pointer* $< p >$ and whether a lock protects a certain object. They also utilise two general patterns $a(\neg b)^*$ and ab. The patterns are very specific as they are looking for specific kinds of deviant behaviour that imply bugs.

Weimer et al. [WN05, GW09]) only use the alternating pattern $(ab)^*$, concentrating on other aspects of specification mining such as reducing false positives.

Perracotta

Perracotta [YE04a, YE04b, YEB+06] used two-symbol regular expression patterns based on the *Response* pattern [DAC99] that says whenever P happens, S must also eventually happen. Table. 9.1 gives the eight patterns that they used, with examples. They use combination only for their chaining pattern i.e. all extracted specifications are either chains of alternation or instances of one of the other patterns.

Table 9.2: Temporal property patterns used in OCD [GS10].

ab	Sequencing	$ab?$	Precondition	a^+b^*	General Precon.
ab^+	LoopBegin	$a?b$	Postcondition	a^*b^+	General Postcon.
a^+b	LoopEnd	$ab \mid ba$	Association	$(a^+b^+) \mid (b^+a^+)$	General Assoc.

Javert and OCD

Later work combines patterns to create final specifications and the patterns chosen reflect this. Javert [GS08a] used $(ab)^*$ and $(ab^*c)^*$ although they also report using $(ab^+c)^*$ [GS08b]. Their later work in OCD [GS10] does not use combination, as they are generating and checking patterns online and do not need to present specifications to a user. The patterns used are summarised in Table. 9.2, note that $b?$ means that b is optional i.e. can occur 0 or 1 times. It should be noted that many of these are similar to those used in Perracotta.

Li et al

Li et al. [LFS10] focus on mining temporal properties for hardware design and build a binary pattern language over temporal and timing operators. Their miner SAM (Scalable Assertion Miner) uses $(ab)^*$, $\mathbf{G}(a \rightarrow \mathbf{X}(a\mathbf{U}b))$, $\mathbf{G}(a \rightarrow \mathbf{X}b)$ and $\mathbf{G}(a \rightarrow \mathbf{XF}b)$. They carry out combination and some additional merging to take into account timing bounds.

Summary

The two symbol pattern we see in every approach is that of alternation i.e. $(ab)^*$, often combined with some notion of chaining. Only Gabel and Su use a three symbol pattern in Javert, that is resource usage of the form $(ab^*c)^*$.

We will use all of the patterns discussed here. We note that the LTL patterns of Li et al. have straightforward automata interpretations.

9.3 Automatically Generating Patterns

When considering a basis of patterns to use as a pattern library the first process we consider is automatic generation. Firstly we show that it is not practical to enumerate possible patterns and then we consider approaches to automatically augment a small set of given patterns.

9.3.1 Can we enumerate patterns?

It would be useful if we could enumerate all patterns up to a certain size as we could then give guarantees about the specifications that could be extracted. However, this approach will explode very quickly. If we consider all possible sets of transitions given m states and n symbols we have

$$m^{mn}$$

possible transition combinations. With 4 states and 3 symbols this is 16,777,216 different versions of δ. Then we need to consider all non-trivial combinations of accepting states, which is

$$2^m - 2$$

as we do not consider the full or empty subset. Therefore the number of total patterns is

$$(m^{mn})(2^m - 2)$$

The number of patterns for relatively small values of m and n quickly grows prohibitively large, as demonstrated in the following tables. At the minimum we require 3 states and 2 symbols; recall that one of these states will be an error state. Realistically, we would like to consider patterns with at least 4 states and 3 symbols, note that • counts as a symbol here.

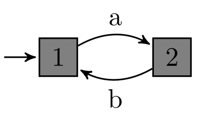
m	n	Patterns
2	2	32
2	3	128
2	4	512
3	2	4,374
3	3	118,098
3	4	3,188,646

m	n	Patterns
4	2	917,504
4	**3**	**234,881,024**
4	4	60,129,542,144
5	2	292,968,750
5	3	915,527,343,750
5	4	2,861,022,949,218,750

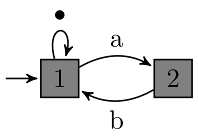
Even though many of these will not be fully connected and this set will contain a large amount of redundancy, there are too many variations for this to be a realistic approach. Note that whilst the checking time is not greatly effected by the number of patterns being checked, the combinition time is dependent on the number of patterns that pass.

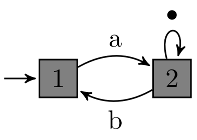
9.3.2 Opening patterns

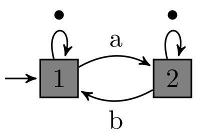
Given a pattern p that contains no holes we can generate a set of *open versions* of the pattern that places holes in different locations. For example, given the pattern



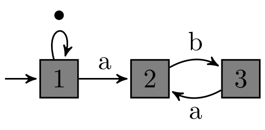
We can generate the following variants that add looping transitions with holes:

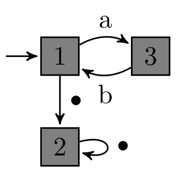


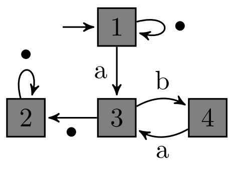




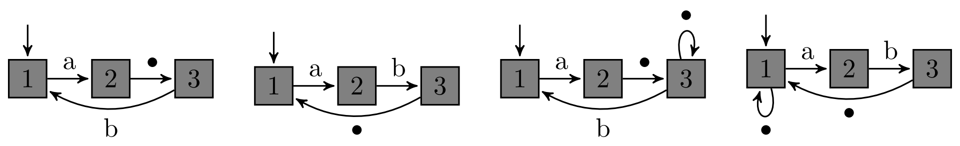
We can also generate variants that place sequences of holes before or after the given pattern, in a sense framing the pattern and capturing the fact that it should occur at the beginning, middle or end of a trace. These patterns therefore belong to the *combination* classification:







Finally, we can introduce non-empty sequences of holes into patterns:

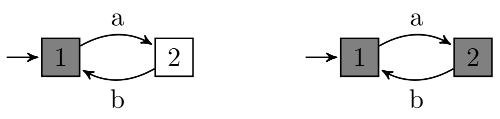


We can see that ‘opening’ a pattern does not just consist of adding the looping • transitions on each state as was done for finite automata in Javert, but also adding intuitive hole behaviours.

9.3.3 Prefix-closing patterns

A pattern p is prefix-closed if $\forall \tau_1.\tau_2 \in \mathcal{L}(p) : \tau_1 \in \mathcal{L}(p)$ i.e. the prefix of every trace accepted by p is also accepted by p. All of the patterns used in previous pattern-based specification mining tools (discussed in Sec. 9.2.3) are not prefix-closed. Prefix-closing a pattern represented in automata form is straightforward; all non failing states should be made final.

Let us briefly discuss whether prefix-closed patterns are desirable using the alternating pattern. Below we have the alternating pattern as was described in Sec. 9.2.3, and the prefix-closed version:



Which is preferable? Well, both have their uses. If we consider a = `open` and b = `close` we might prefer the non prefix-closed version so that we capture the fact that not closing is erroneous. However, if we consider a = `left_motor_on` and b = `right_motor_on` we might not care if we finish with the left motor on, rather than the right. Therefore, it is useful to have both prefix-closed and non prefix-closed versions of patterns.

9.3.4 Fuzzing patterns

Another approach to consider is taking a small set of patterns and ‘fuzzing’ them i.e. changing them in small ways to generate similar patterns. The idea here is that the small set of given patterns represent likely behaviours and certain fuzzing operations represent likely variations. This generalises the notion of opening patterns. Possible fuzzing operations include:

- Changing the accepting states
- Changing the initial state
- Adding or removing hole or symbol transitions
- Making sections loop i.e. one transition, a path or the whole pattern
- Swapping symbols

One interesting direction not explored in this work is the possibility of using these techniques as genetic operators to *evolve* a pattern library using a genetic algorithm and a set of canonical specifications to generate traces from.

9.3.5 Summary

We have concluded that automatically generating a pattern library is not feasible, but that there are some interesting techniques that can be applied to existing sets of patterns to produce new patterns. Later we apply pattern opening and prefix-closing to our constructed set of patterns.

9.4 Exploration through decombination

In this section we attempt to use common specifications to generate the patterns that are needed to mine them. We do this through the process of *decombination* for open automata, which is given in Def. 69 on page 197. This is an imprecise activity as we saw in Sec. 8.3.6, but many useful specifications do decombine precisely.

9.4.1 Common specifications.

We identify a number of common specifications and apply decombination to them, analysing the resulting patterns. When we decombine a specification we are looking for *smaller* patterns (in terms of alphabet) that can combine together to form that specification. Therefore we mainly concentrate on specification patterns with large alphabets. However, we begin by considering two small but common specification patterns:

- **Resource usage.** A resource is created or opened before being used and then destroyed or closed. This process can be repeated.
- **Mutual exclusion.** Only one resource can be used at any time.

Next let us consider the following larger common specification patterns:

- **Ordering.** Events must occur in a sequential order.
- **Reaching goodness.** A number of operations must occur in a certain order but no restrictions are placed on intermediate events. The idea is that the final state is some good state we need to reach.
- **Dependent resources.** A resource A is marked as dependent on another resource B and therefore A can only be used when B is active. This can be extended to many resources.
- **Choice.** Using events to choose between two possible (similar) behaviours.

In the next sections we consider the decombinations for these specification patterns through manual and automated techniques i.e. we first attempt to automatically decombine the specifications and then manually inspect and refine the results.

We note that it could be argued that this process biases our approach towards the specifications considered in this section. However, this is equivalent to developing a heuristic by examining common cases; as the heuristic will always be biased towards these cases but this is the point, *we want to cover common cases.*

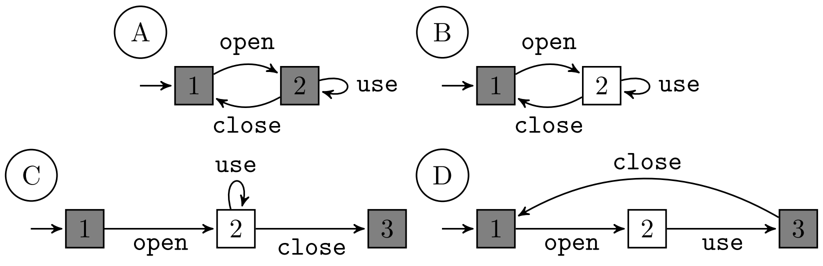
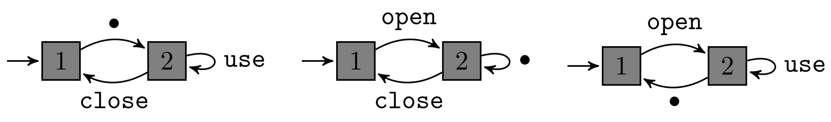


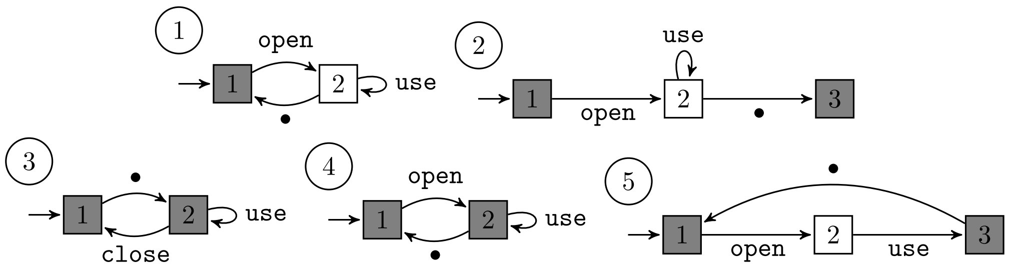
Figure 9.4: Four different versions of the resource usage pattern.

9.4.2 Resource usage

Fig. 9.4 contains four versions of the resource usage specification. Each pattern can be decombined into three different open automata with alphabets of size two. For example, the specification A decombines into the following pattern instances:



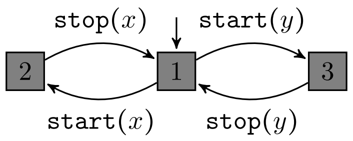
Which combine together to form the specification, actually any two of these pattern instances will combine together to form the original. Therefore, to extract these four different versions of the resource usage specification we would require eight patterns, two for each. However, if we inspect the twelve decombinations we note that there are 81 different ways of choosing five of those decombinations that can create all four resource usage specifications. One such set is as follows:



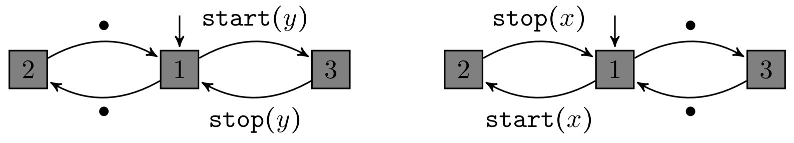
One way in which the resource usage specifications from Fig. 9.4 can be constructed is as follows: 3 and 4 combine to make A, 1 and 3 combine to make B, 1 and 2 combine to make C and 1 and 5 combine to make D.

9.4.3 Mutual exclusion

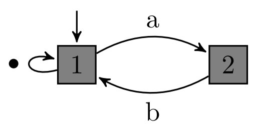
Let us now consider the mutual exclusion specification, which can be written as followed for two instances of a resource x and y:



The two-symbol decombination of this property contains nine pattern instances but we only need the following two to reconstruct the specification:



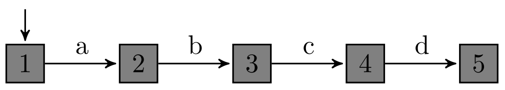
But both of these are an instance of the following pattern, therefore to extract specifications of this form we only need one pattern.



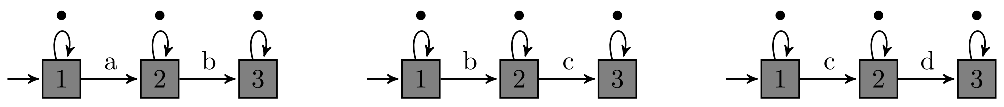
If we have any number of mutually exclusive states we can use the same pattern.

9.4.4 Ordering

Let us consider the following specification that captures the behaviour that four events appear in sequence:

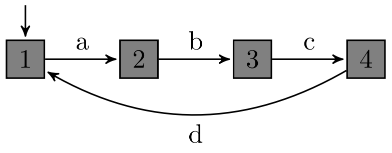


This straightforwardly decombines into the following three instances of a simple *chaining* pattern:

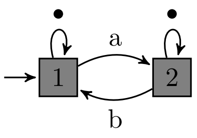


The pattern instance involving d allows further symbols to occur after d, but the other pattern instances ensure that it is not possible for any symbols to occur here.

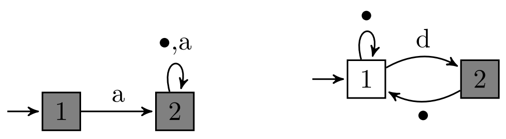
Next we consider an extension of this specification that loops the sequence of events:



then we can modify our chaining specification to capture this looping as follows:

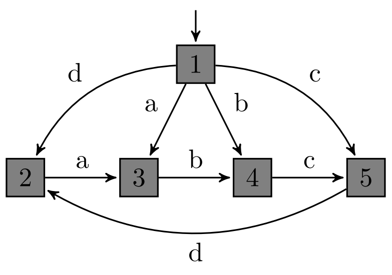


If we then combine this pattern with either of the following patterns we will also capture the original specification without looping. These patterns indicate that the trace begins with a, or ends with d respectively.

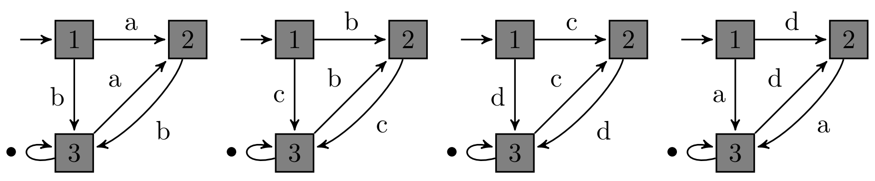


Therefore, we can use two patterns to extract any ordered or ordered-looped specification.

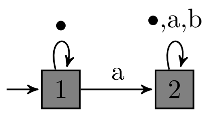
Finally, let us consider a third form of ordering where the start and end event does not matter, only the order of events, for example the following specification :



To capture this we only need one pattern with instances per pair of adjacent symbols:



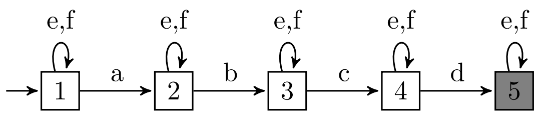
This pattern is a generalisation of the chaining pattern that allows us to start on either symbol. Combining this with the following pattern that indicates, which symbol should appear first out of two symbols, gives us the chaining pattern we saw previously.



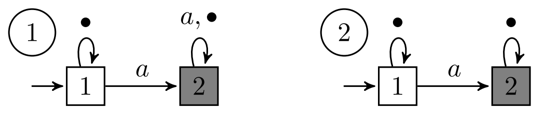
Therefore we only need three patterns to capture ordering.

9.4.5 Reaching goodness

The next specification is a form of ordering, but one where we aim to reach a certain state after a sequence of events, with some events not taking part in that sequence. An example of this specification is as follows:



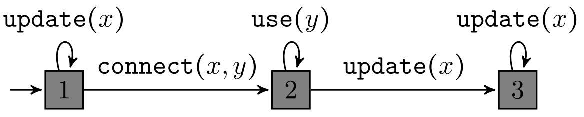
To build this specification we can use the chaining pattern introduced for ordering along with the following two patterns:



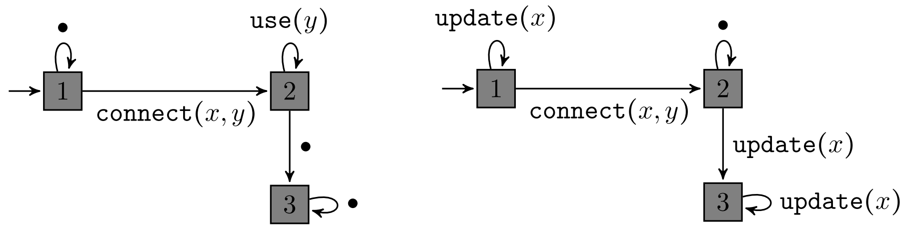
Pattern 1 indicates that an event occurs, here used with e and f and pattern 2 indicates that an event leads to success, here used with d. Pattern 2 forces the sequence to end with d and ensures that we only see an accepting state once d occurs.

9.4.6 Dependent resources

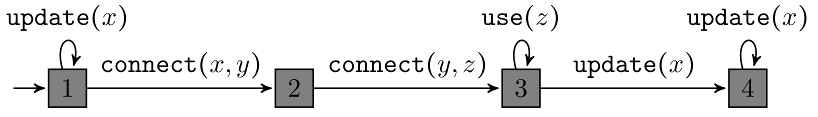
Now let us consider the dependent resources specification. An example of this specification is the (connected) version of UnsafeIter taken from the `Java` API examples (see Sec. A.3.2):



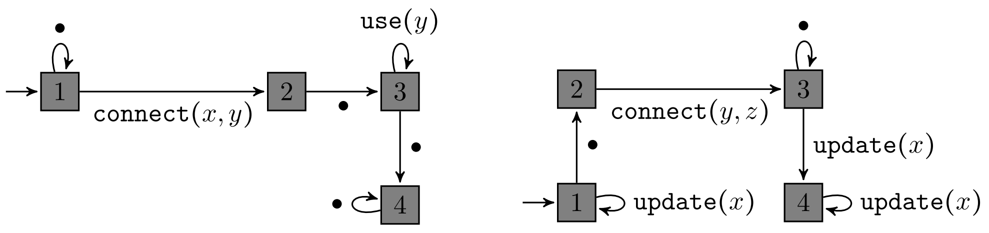
This automatically decombines into two 2 symbol pattern instances as follows:



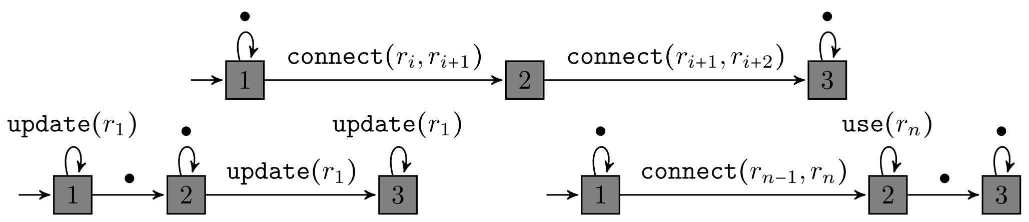
Now let us consider a more complicated version of the dependent resource pattern, this time taken as the UnsafeMapiterator property. Here there are three dependent resources.



Again this automatically decombines into two 2 symbol pattern instances as follows:

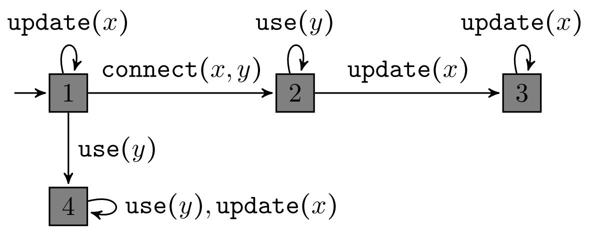


Inspecting these pattern instances we note that the general case with n connected resources can be captured with the following pattern instances:

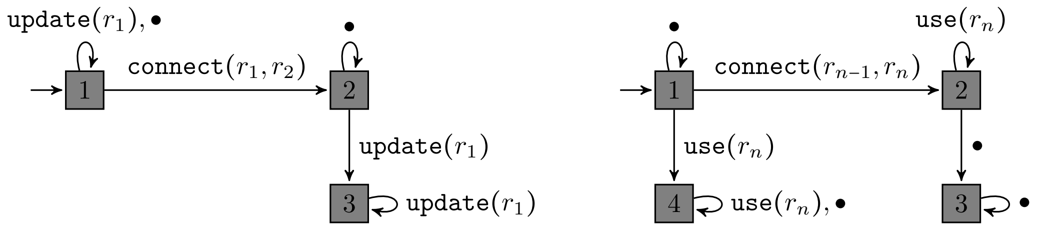


The first pattern is not required in the case where we have two connected resources.

We should also consider the unconnected version of the UnsafeIter property. This is a bit more complicated as it needs to account for the situation where x and y are not connected:

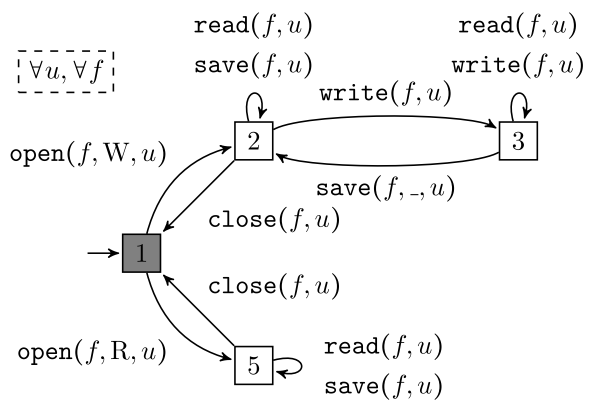


To capture this property we replace the second two general pattern instances we had before with the following instances. We note that the first of these could be used in place of what we had before.

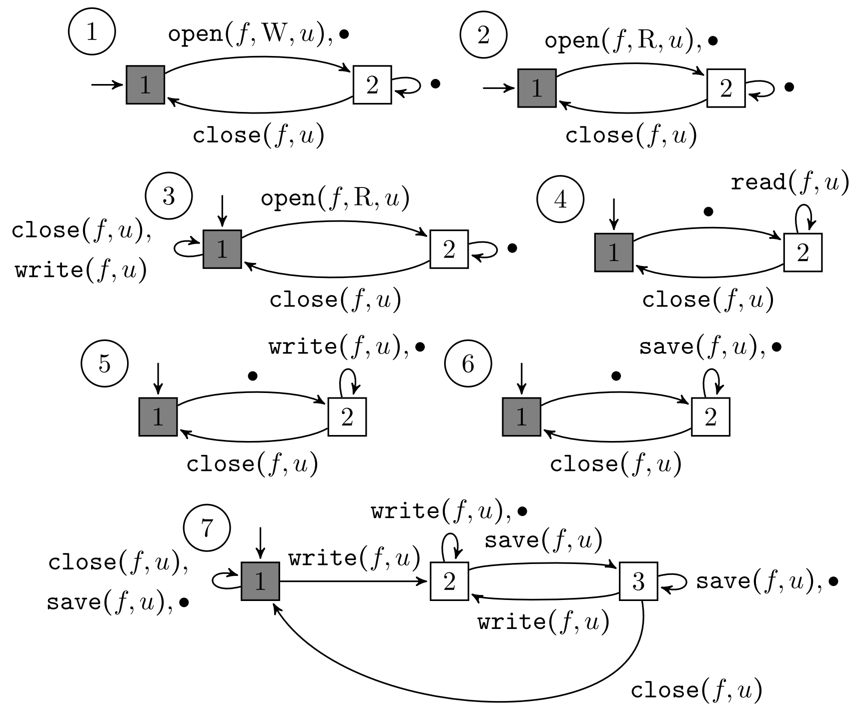


9.4.7 Choice

We choose a complicated specification as the basis of our exploration of the choice specification. Take the following specification based on the QEA given in Fig. A.38 on page 355 for file usage.



This chooses two behaviours based on the mode in which a file is opened. Firstly, we note that the 2-decombination of this pattern is imprecise. However, if we include 3 symbol patterns we can produce the following precise decombination:



Pattern instances 1 and 2 are instances of the same pattern, a simple variant of alternation for the `open` and `close` events. Pattern instance 3 uses three symbols to capture the fact that a `write` event cannot occur between an `open` event in read mode and a `close` event. Pattern instances 4, 5 and 6 are also instances of the same pattern, which restrict `read`, `write` and `save` events with respect to `close`. Pattern instance 7 also uses three symbols to capture the correct order between `write`, `save` and `close`. Therefore, we can capture this complicated choice property using four relatively simple patterns.

Note that it is pattern instance 3 that led to the choice behaviour, this pattern restricts what can happen between two other symbols. This shows that we need three symbol patterns to capture complex behaviours.

9.5 The Library

As a result of the experiments and observations made in this chapter we have designed a pattern library that extends and refines the Specification Pattern System (SPS) [DAC99]. Note that our patterns are event-based and the categories have been chosen with the mining method in mind i.e. the fact that we will use combination.

The library is explored in detail in Appendix D.1. Briefly, it consists of four categories:

1. Occurrence - as before, these events place bounds on the number of times events, or sets of events, should occur.

2. Ordering - we consider the concepts of case (precedence) and effect (response) as well as alternation.
3. Scope - this new category consists of restriction patterns such as *a* only occurs at the end of a trace, or after some other event *b*.
4. Compound - this considers the arbitrary chaining and disjunction of patterns.

We also construct an augmented version of this library by applying the pattern opening and prefix closing augmentation techniques.

9.6 Summary

This chapter began to address the question "what makes a good pattern library?" by looking at how a pattern library can effect mining, reviewing previous choices and exploring techniques for generating patterns from other patterns or common specifications. We finished with an overview of the pattern library to be used in evaluation. There are a few outstanding questions; for example, what role does redundancy play in a pattern library?

Chapter 10

Evaluating the Specification Mining Technique

In this chapter we evaluate the pattern mining framework introduced in Chapter 8 using the pattern library developed in Chapter 9. As discussed in Section 2.4.5 there are two general approaches to evaluating specification mining techniques. The first is an ad-hoc exploration of code with the aim of finding something interesting. The second measures accuracy against some predetermined ground truth (known specification). We take the second approach here using both real-world and automatically generated traces.

We have chosen this evaluation technique as it gives a strong quantitative measure of the effectiveness of our framework and allows us to explore the effects of targeting different kinds of specifications with varying levels of information in the traces used. When taking this approach it is important that the chosen specifications are representative of realistic scenarios, we aim to achieve this by taking examples from a range of domains and from both real-world projects and the literature.

Our general evaluation method works by first generating or extracting a set of training traces and using them to extract a QEA, and then generating a set of testing traces and using them to measure accuracy.

In Section 10.1 we introduce the ground truths, or models, we will target, including some specifications we monitored in Chapter 7. Section 10.2 then considers the traces we will be using. As well as generating automatically generating traces from known specifications, we also extract traces from the DaCapo benchmark suite where appropriate. Finally, Section 10.3 presents our results using different pattern libraries and traces with different properties.

10.1 Selecting models

In this section we select the models to target in our evaluation. We begin by discussing our selection process and then present the chosen models.

10.1.1 Selection process

In an attempt to make our evaluation representative we aim to select models from a range of different domains, focussing on those represented by possible application domains. Our selection process consists of the following three searches.

Specification studies. We review previous studies [BKA11, LJMR12] that have looked at common protocol specifications. These studies typically focus on the correct usage of the classes in the `Java` standard library.

Previous mining techniques. We consider properties mined by previous tools in both structured evaluations like ours and ad-hoc evaluations where the technique is applied to code to find interesting properties. We are specifically interested in the comparable JMiner [LCR11] and Tark [LRRV12] tools, and use specifications identified in this work. Pradel et al. [PBG10] developed a set of ground truths of `Java` standard library properties for their specification mining evaluation approach.

Other work using specifications. We look at previous academic work that formally specifies temporal behaviour. One obvious area of literature is that of runtime verification. We take specifications from the planetary rover examples given in previous work [BH11b] and used in our runtime verification evaluation. We also look at applications of runtime verification, such as the Orchids intrusion detection system [GLO08] .

This selection process has led to a range of models, however we accept that it is biased towards `Java` programming properties as this is an area where there has been a concentration of work formalising trace properties.

10.1.2 The models

The models we have selected are summarised in a table at the end of this section. For each model the table gives a description and reports the number of states, size of alphabet, number of quantified variables and cylcomatic complexity (explained in Sec. 7.1.2).

We discuss these models below, organised into a number of different domains. As well as covering a variety of domains, these models also capture different levels of complexity, although we note that the majority of specifications only use universal quantification.

`Java` API

The first domain we consider is the most popular for current specification mining techniques: the `Java` standard library. We divide the properties used here into two kinds: *summary* and *full*. The *summary* specifications only refer to the methods relevant to the behaviour being captured and have been taken from our work formalising `Java` API properties (Appendix A.3). The *full* specifications capture similar kinds of behaviour but include all methods of the relevant classes and have been taken from [PBG10].

The summary specifications we consider are *JFreeChart*, *SocketOutput*, *ColIter*, *HasNext*, *InputStream* and *OutputStream*. These represent both singly and doubly quantified properties. We will use the last three with extracted traces from the DaCapo benchmark suite.

The full specifications we consider are *Formatter*, *URL* and *Socket*. These contain many methods that are not constrained by the order in which they occur with respect to other methods. For example, `getUserInfo` for URL. We expect these to be more difficult to learn due to the large alphabets. Socket is also the most complex model we have with 15 states and 39 events.

Communication

We consider models related to communication. The *James* model is taken from early work in the development of JMiner [CR08] and is a specification extracted from the Apache James mail server program. To extract the specification they took a nonstandard mapping of method calls to events, requiring some understanding of the program being analysed.

The other models in this category are about communication between entities, these were previously discussed in our planetary rover case study (Appendix A.4). The *Satellites-All* and *Satellites-Each* models are the only ones using existential quantification in our set. The *Commands* and *NestedCommands* models represent singly and doubly quantified nested alternation respectively.

Rovers

We also consider examples from our planetary rover case study that relate to the internal behaviour of the rovers. We have chosen the *ResourceLifecycle*, *ReleaseResource*, and *RespectConflicts* properties as they have reasonably complex behaviour, for example, ReleaseResource uses three quantifiers.

Concurrency

In this category we consider descriptions of common desired concurrent behaviour. The *MutualExcl* and *ReadWriter* models capture the standard and many-reader-one-writer forms of mutual exclusion respectively. The *LockOrder* model captures the previously discussed lock ordering property. Both LockOrder and ReadWriter are complex properties; ReadWriter requires 5 quantifications and LockOrder needs 12 states to capture all possible interleavings.

Security

Here we use a single model that captures a slightly simplified version of the ptrace attack [GLO08], a local-to-root exploit giving root privileges to a user with local access via a sequence of ptrace system calls. This specification was introduced in the context of runtime monitoring for intrusion detection and is a validation property i.e. acceptance indicates an attack.

Drivers

The last three models we consider are based on rules described in the SDV (Static Driver Verifier) tool [SDV]. These are all models mined by the Tark tool [LRRV12] and capture the expressive capabilities of this tool.

The following table summarises the chosen models; these are illustrated in Appendix D.2.

Name	Description	$\lvert Q\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert X\rvert$	CC
	`Java` API				
HasNext	Calls of next and hasNext alternate on an iterator	4	3	1	3
SocketOutput	A stream is used after being connected to a socket and not after that socket is closed	6	3	2	13
ColIter	An iterator created from a collection is not used after the collection is updated	10	5	2	32
Socket	The correct behaviour of a Socket with respect to connections and input/output streams, also a Socket must be closed	15	39	1	188
URL	Only call getContent or openStream at most once	6	16	1	58
Formatter	A Formatter if used must be flushed and must eventually be closed	5	6	1	8
JFreeChart	If a chart and plot are connected then they notify each other if they are changed	10	5	2	15
InputStream	An InputStream should not be read from after being closed	4	3	1	4
OutputStream	An OutputStream if used must be flushed and must eventually be closed	5	4	1	5
	Communication				
James	The SMTP protocol (without authentication) as used in the Apache JAMES mail server	7	5	1	8
Satellites- All	Every field unit establishes a communication link with some satellite	4	2	2	5
Satellites-Leader	There exists a satellite that has established a communication link with all known field units	4	2	2	5
Commands	Issued commands succeed and are only reissued after failure	4	4	1	3
Nested- Commands	Command sending and acknowledgment should nest	6	4	2	13
	Rovers				
ResLifecycle	The lifecycle of a resource - from being requested to granted to canceled	4	5	1	6
ReleaseRes	Resource taken by a task completing a command must be released before that command is finished	5	4	3	7
RespectConf	Conflicts between resources should be respected when granting resources to tasks	5	5	2	11
	Concurrency				
MutualExcl	No lock should be held by two threads at the same time	4	4	3	7
LockOrder	Locks are always taken in a consistent order.	12	4	2	31
ReadWriter	Many readers may access a file but only one writer	7	8	5	18

Security					
PTrace	The sequence of system calls that identifies a local-to-root exploit	7	6	2	32
Drivers					
IOCallDriver	The I/O stack must be setup before calling the IOCallDriver	4	2	1	6
KeAcquire-SpinLock	Locks and releases of a spin lock alternative, with two alternate locking calls	3	3	1	3
ZwRegistry-Create	Registry keys are created before being used, open when used and not used after being deleted	5	5	1	10

10.2 Traces

This section describes the traces used for training and testing. We first consider the problem of generating random traces from the language of a QEA before presenting some statistics about the selected traces. Whilst the generation task is straightforward for state machines, the quantifications, and how they relate to event parameters, make this a complex task for QEA.

10.2.1 Trace generation

We consider how to generate traces from Target QEA (Def. 58 on page 190). To randomly generate traces we require a set of possible values for each quantified variable to give a set of possible ground events. Given a Target QEA $\mathcal{Q}$ over alphabet $\mathcal{A}$ using quantified variables X, and a map $\mathcal{D}$ from X to sets of values, let $\mathcal{A}_{\mathcal{D}}$ be the set of ground events $\{\mathtt{e}(v_1,\ldots,v_n) \mid \mathtt{e}(x_1,\ldots,x_n) \in \mathcal{A} \wedge \forall i \leq n : v_i \in \mathcal{D}(x_i)\}$.

The general trace-generation method randomly generates traces over $\mathcal{A}_{\mathcal{D}}$ and checks whether they are accepted by the given Target QEA; we will want to generate both correct and incorrect traces (the latter are used in testing).

Targeting coverage levels

To explore how the mining process is effected by the quality of the data provided we consider two coverage criteria for traces. This is similar to the approach taken by Lo et al. [LMS12], however is updated for our scenario with quantifications and non prefix-closed automata. The coverage levels we consider are as follows:

- State coverage - All non-ultimately failing states are visited at least once by a projection of the trace
- Path coverage - All paths to non-ultimately failing states are visited at least once by a projection of the trace
- None - does not satisfy either coverage condition

A trace with path coverage will have state coverage. Some models only permit path coverage, i.e. a trace is only accepted if it visits every path. By ‘a projection of the trace’ we mean that one projection can visit one half of the sates and another projection visit the other half; all states are visited but not necessarily by one projection.

Targeting trace lengths

We also consider four different lengths of trace: shortest, short, medium and long. For each level we set a maximum trace length and number of bindings and traces are randomly generated within these limits. We do not provide a minimum trace length, otherwise we would find that important information is lost (this was observed in [RBR13b]). The shortest category is set to give the shortest valid trace and the long category is supposed to reflect a realistic scenario.

Test traces

We generate both positive and negative traces for testing. Positive traces for testing are selected equally from different trace levels and coverage categories. Negative traces for testing are restricted so that they only have a few different bindings, although the related trace slices can be short or long. We restrict negative traces in this way so that they contain minimal errors. If we did not do this then we might have a case where there are, say, five different kinds of error and all negative traces capture all five and an extracted QEA only rejects one of them. Then, because all negative testing traces capture this one error, it will look like the extracted QEA can reject all kinds of error.

Refinements

We refine our general generation approach. Firstly, we avoid selecting events that would automatically lead to an undesired strong verdict by verifying the trace incrementally. To do this we compute the set of strong states (Def. 48 on page 121) and ensure that a chosen event would not take us to such a state. Secondly, we note that if the alphabets for different bindings are disjoint (see Sec. 6.5.2 for a discussion of disjoint alphabets) then we can generate a trace per binding and merge these together. Finally, we target different coverage levels in the disjoint alphabet case by generating more per-binding traces than required and selecting a suitable subset to satisfy the given coverage level.

Limitations of generated traces

We note that this trace generation method is limited. Firstly, as it is random it is costly to generate traces with certain lengths and coverage criteria, by randomly selecting a next state (even one that does not immediately lead to failure) the chances of generating a successful trace can be low. The main reason for this is that subtraces *interact* i.e. adding an event to one trace projection may also add that event to another trace projection, causing failure. The traces generated for this study took almost a week to generate (although there was a lot of redundancy in the results).

Model	Number	Length		Number of objects	
		Total	Average	Total	Average
HasNext	6 (72)	66,873,215	928,794	9,912,671	137,675
InputStream	12	2,029,910	169,159	199,887	16,657
OutputStream	10	1,848,196	184,819	2,373	237

Table 10.2: Statistics about the extracted traces.

This is the issue of *observability*. Walkinshaw et al. [WBJ08] note that if traces are constructed randomly there may still be aspects of the automaton that are not explored (as they have *low observability*), meaning that the probability of generating a trace to identify that behaviour by random exploration is very low. To mitigate this we have introduced the different coverage criteria.

10.2.2 The extracted traces

For a subset of the `Java` API models we extract traces from the DaCapo benchmark suite [BGH+06] (see Sec. 7.2.1 for a detailed description). To extract these traces we wrote custom `AspectJ` to detect occurrences of the methods and then output these to a file, using `Identity.hashCode` to record the identity of the callee and parameter objects. In cases where we know there are errors with respect to our ground truth (for example with HasNext) we do not extract traces. Multiple traces are created if the benchmarks run with multiple class loaders (see Sec. 7.2.1).

Table. 10.2 gives details of our extracted traces. These traces are very long (hundreds of thousands to millions of events) and contain many different objects. Therefore, whilst the behaviour is mostly simple we expect mining to take a long time to process all traces. Additionally, if there are any errors in the traces we will fail to extract a specification as our current method cannot cope with imperfect traces (see Chapter. 11).

To deal with the very long HasNext traces we split them up into equivalent shorter traces (around 1M events each) by performing a slicing preprocessing step. This is necessary as the checking part of our framework cannot rely on garbage information and would struggle to deal with this many data values. The preprocessing step is very fast as it does not need to carry out any checking. We indicate the number of traces after preprocessing in brackets and the average lengths and objects for HasNext are given for these preprocessed traces, not the extracted ones.

10.2.3 The generated traces

We aim to generate 10 traces for each training set and 100 traces for each testing set. We choose to generate considerably more testing traces than training traces to reflect the limited information present in test suites whilst maintaining good coverage in the testing set. We gave the generation algorithm a week to generate traces, and any categories that do not have the required number of training traces are removed. Out of a possible 288 categories we have 163 with enough traces. As noted earlier, there are cases where categories are not valid (should be empty) as path coverage is required for successful traces, or the length can only be achieved through path coverage. The number of testing traces varies as in some cases there are only a limited number of unique positive or negative traces.

A table giving the average length of traces is given in Appendix D.3. The average trace length varies as longer traces are produced for simpler specifications as, firstly, this is generally more straightforward and, secondly but more importantly, these specifications are often associated with repeated behaviour i.e. reading from a file or using an iterator. In general positive testing traces are a lot longer than training traces, whilst negative testing traces are typically very short (for reasons discussed earlier). The number of objects used in traces is generally small, ranging from the tens to the low hundreds.

10.3 Results

We use the models and traces from the previous two sections to evaluate our mining framework.

10.3.1 Research questions

We begin by discussing the questions we aim to answer in our evaluation. We are generally concerned with the accuracy and efficiency of our approach. Accuracy is the ability of extracted specifications to accept correct and reject incorrect traces, we discuss how we measure this later. The questions we aim to answer are as follows:

1. Do longer traces lead to more accurate specifications?
2. Do traces with better coverage lead to more accurate specifications?
3. Does our approach scale with trace length?
4. How concise are extracted specifications?
5. Are there structural properties of the models that effect the effectiveness of our technique?
6. How does the pattern library effect the accuracy of extracted specifications, and the efficiency of the extraction process?
7. Does mining in connected mode lead to more accurate specifications? What are the repercussions for efficiency?
8. Can we deal with real world traces i.e. process such traces efficiently?

In the following four sections we will discuss our evaluation setup and present results for generated traces, connected mode and extracted traces respectively.

10.3.2 Evaluation setup

Let us give an overview of our evaluation setup.

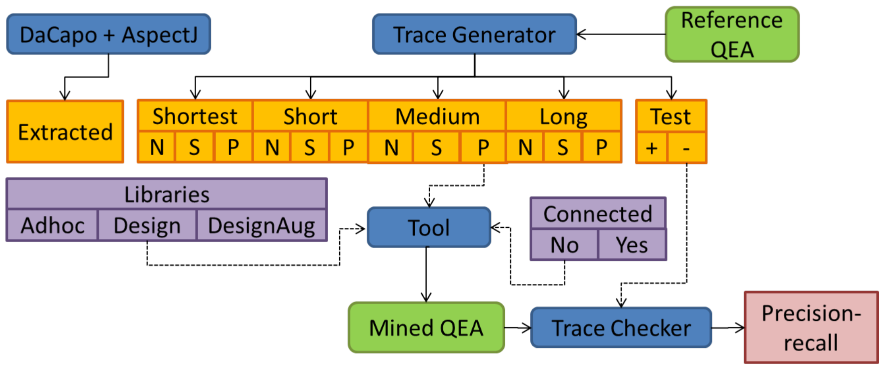


Figure 10.1: Evaluation framework for specification mining tool

Evaluation framework

Figure 10.1 illustrates our evaluation framework. It has the following components:

- **Reference QEA.** This is the model we are attempting to reconstruct, these were presented in Sec. 10.1.
- **Traces.** As discussed in Sec. 10.2 we have generated four sets of training traces (with three coverage levels) and both positive and negative testing traces, and for some models we also extracted a set of traces from DaCapo.
- **Tool.** This is our mining framework introduced in Chapter 8. As well as a set of traces it takes a pattern library (discussed below) and a connectedness mode.
- **Trace checker.** Our runtime verification algorithm developed earlier in this work.
- **Precision-recall.** This is our result, we discuss how this is computed below.

The framework selects a set of training traces, library and connected mode, extracts a QEA and uses it to check the testing traces to produce precision and recall measures.

Pattern libraries

We use three different pattern libraries in this work:

- *Ad-Hoc.* These patterns were developed in a first iteration of our tool. An ad-hoc method was used of writing down variations of patterns seen during development, with no structured approach. Consequently this library is large and allows us to measure whether a structured approach to library design is required.
- *Designed.* This is the library discussed in Section 9.5, designed in Chapter 9.
- *Designed and Augmented.* As explained in Section 9.5, we automatically apply the library augmentation techniques of Sec. 9.3 to the designed library.

The following table summarises the number of patterns in each library with a given number of symbols in their alphabet (not including the $\bullet$ symbol).

Library	1 symbol	2 symbol	3 symbol
Ad-Hoc	50	270	2
Designed	12	34	4
Designed and Augmented	24	596	8

Measuring accuracy

To measure the accuracy of the mining process we use the common precision-recall evaluation measures taken from the field of information retrieval [vR79, MRS08]. We take the refined measures of both positive and negative precision and recall proposed by Walkinshaw et al. [WBJ08]. These measures separate precision and recall for positive and negative behaviour, to account for differing numbers of positive and negative test traces. Precision is a measure of correctness and recall is a measure of completeness.

To compute these measures we test the extracted QEA on the testing traces and add these traces to one or more of four sets as follows:

Extracted QEA	Reference QEA	Ret^+	Rel^+	Ret^-	Rel^-
accept	accept	add	add		
accept	reject	add			add
reject	accept		add	add	
reject	reject			add	add

The positive and negative versions of precision and recall can then be given by:

$$
\begin{aligned}
precision^+ &= \frac{|Ret^+ \cap Rel^+|}{|Ret^+|} & recall^+ &= \frac{|Ret^+ \cap Rel^+|}{|Rel^+|} \\
precision^- &= \frac{|Ret^- \cap Rel^-|}{|Ret^-|} & recall^- &= \frac{|Ret^- \cap Rel^-|}{|Rel^-|}
\end{aligned}
$$

To understand what these measures mean in terms of our specification mining task consider a system that is correct for QEA $\mathcal{A}$ and a QEA $\mathcal{B}$ extracted from traces of that system. In our setup our training traces will be $t^+ \subseteq \mathcal{L}(\mathcal{A})$ and $t^- \subseteq \overline{\mathcal{L}}(\mathcal{A})$ where we use the notation $\overline{\mathcal{L}}$ to represent traces *not* in the language of some QEA. We then have

$$
\begin{aligned}
p^+ &= \frac{|t^+ \cap \mathcal{L}(\mathcal{B})|}{|(t^+ \cap \mathcal{L}(\mathcal{B})) \cup (t^- \cap \mathcal{L}(\mathcal{B}))|} & r^+ &= \frac{|t^+ \cap \mathcal{L}(\mathcal{B})|}{|t^+|} \\
p^- &= \frac{|t^- \cap \overline{\mathcal{L}}(\mathcal{B})|}{|(t^+ \cap \overline{\mathcal{L}}(\mathcal{B})) \cup (t^- \cap \overline{\mathcal{L}}(\mathcal{B}))|} & r^- &= \frac{|t^- \cap \overline{\mathcal{L}}(\mathcal{B})|}{|t^-|}
\end{aligned}
$$

Here we can see why we have introduced the positive and negative versions of precision and recall. If the comparative sizes of t^+ and t^- vary greatly then the measures will be skewed. As patterns have a next semantics the default behaviour is to reject traces, therefore we expect r^- to be very high. If we have high r^- but low p^+ it means that we rejected the majority of the negative test traces (making $t^- \cap \mathcal{L}(\mathcal{B})$ small) but that t^- must be larger than t^+, showing that even a few incorrectly accepted traces effects p^+.

Table 10.3: Overview of success and failure of results, to=timeout.

		AdHoc			Designed			DesignedAug		
		suc	to	no pass	suc	to	no pass	suc	to	no pass
Shortest	N	14	2	1	13	0	4	14	2	1
	S	13	2	2	13	0	4	14	1	2
	P	16	1	0	15	0	2	17	0	0
Short	N	7	3	1	7	0	4	8	2	1
	S	9	1	1	5	0	6	9	2	0
	P	17	3	0	13	0	7	19	1	0
Medium	N	5	1	1	0	1	6	5	1	1
	S	6	3	0	2	1	6	6	3	0
	P	18	2	2	13	0	9	17	3	2
Long	N	4	0	0	0	0	4	4	0	0
	S	5	2	1	1	1	6	6	1	1
	P	17	2	1	11	1	8	17	2	1
Total		131	22	10	93	4	66	136	18	9

We can view these measures in terms of *over-specification* and *under-specification* (discussed in Sec. 9.1). If p^+ and r^- are high but r^+ and p^- are low then we only accept correct traces, but also reject correct traces. This means that the extracted specification is too restrictive and over specifies the behaviour. If r^+ and p^- are high and p^+ and r^- are low then we do not reject traces we are supposed to accept, but we also accept traces we are supposed to reject. This means that the extracted specification is too general and under specifies the behaviour. Whether under or over specification is more of an issue depends on the application.

Experiments

Therefore, in the next three sections we carry out the following experiments. For generated traces we have 24 models in total with 163 non-empty categories, so extracting a specification for each library makes 489 experiments. For connected mode only 5 of the models qualify for connected mode, so with 47 non-empty categories this makes a further 141 experiments. For extracted traces we use three models, making another 9 experiments. Experiments from generated and extracted traces are limited to a maximum of 30 and 120 minutes respectively.

10.3.3 Generated traces

We first examine the results of our evaluation for generated traces (in non-connected mode). The results are summarised in Tables 10.3 to 10.6 giving average pass-rates, precision-recall measures, timings and conciseness measures for each pattern library and trace coverage and length criterion. Appendix D.4 gives the full results i.e. reports all of these measures for each individual experiment. We begin by giving an overview of the results and then address the relevant questions listed above.

Overview

We extract 10 of the 24 specifications perfectly, these are HasNext, InputStream, OutputStream, SatellitesAll, SatellitesLeader, ResLifecycle, IOCallDriver, URL, Formatter, ZwRegistryCreate

Table 10.4: Precision-recall results - reported as positive/negative in each case.

Precision							
		AdHoc		Designed		DesignedAug	
		p^+	r^+	p^+	r^+	p^+	r^+
Shortest	N	0.21 ± 0.41	0.03 ± 0.11	0.15 ± 0.36	0.15 ± 0.36	0.21 ± 0.41	0.00 ± 0.00
	S	0.38 ± 0.48	0.13 ± 0.28	0.44 ± 0.48	0.38 ± 0.48	0.28 ± 0.45	0.01 ± 0.03
	P	0.76 ± 0.40	0.64 ± 0.43	0.75 ± 0.38	0.73 ± 0.44	0.58 ± 0.49	0.43 ± 0.47
Short	N	0.42 ± 0.49	0.07 ± 0.14	0.38 ± 0.39	0.43 ± 0.48	0.25 ± 0.43	0.06 ± 0.15
	S	0.78 ± 0.31	0.27 ± 0.37	0.69 ± 0.35	0.61 ± 0.47	0.66 ± 0.47	0.29 ± 0.40
	P	0.87 ± 0.27	0.65 ± 0.35	0.87 ± 0.26	0.83 ± 0.29	0.92 ± 0.22	0.61 ± 0.33
Medium	N	0.48 ± 0.41	0.23 ± 0.33	-	-	0.6 ± 0.48	0.05 ± 0.08
	S	0.77 ± 0.35	0.57 ± 0.42	0.5 ± 0.5	0.02 ± 0.02	0.83 ± 0.37	0.58 ± 0.43
	P	0.94 ± 0.13	0.75 ± 0.32	0.93 ± 0.09	0.89 ± 0.20	0.85 ± 0.32	0.72 ± 0.36
Long	N	0.12 ± 0.21	0.01 ± 0.02	-	-	0.25 ± 0.43	0.00 ± 0.00
	S	0.76 ± 0.38	0.44 ± 0.46	**1.0** ± 0.0	**1.0** ± 0.0	0.49 ± 0.49	0.22 ± 0.36
	P	0.90 ± 0.25	0.74 ± 0.33	0.94 ± 0.08	0.97 ± 0.09	0.91 ± 0.24	0.77 ± 0.34
Recall							
		AdHoc		Designed		DesignedAug	
		p^-	r^-	p^-	r^-	p^-	r^-
Shortest	N	0.59 ± 0.14	**1.0** ± 0.0	0.64 ± 0.20	**1.0** ± 0.0	0.59 ± 0.13	**1.0** ± 0.0
	S	0.60 ± 0.17	0.98 ± 0.04	0.71 ± 0.24	0.95 ± 0.10	0.58 ± 0.12	0.99 ± 0.01
	P	0.84 ± 0.21	0.99 ± 0.01	0.91 ± 0.15	0.96 ± 0.06	0.77 ± 0.21	**1.0** ± 0.0
Short	N	0.67 ± 0.09	**1.0** ± 0.0	0.78 ± 0.22	0.84 ± 0.21	0.63 ± 0.15	**1.0** ± 0.0
	S	0.70 ± 0.17	0.98 ± 0.01	0.79 ± 0.26	0.89 ± 0.08	0.71 ± 0.19	0.99 ± 0.00
	P	0.82 ± 0.18	0.97 ± 0.06	0.87 ± 0.18	0.96 ± 0.06	0.79 ± 0.19	0.99 ± 0.01
Medium	N	0.71 ± 0.13	0.98 ± 0.01	-	-	0.65 ± 0.08	**1.0** ± 0.0
	S	0.78 ± 0.23	0.97 ± 0.02	0.48 ± 0.15	**1.0** ± 0.0	0.79 ± 0.24	0.99 ± 0.00
	P	0.89 ± 0.15	0.98 ± 0.03	0.90 ± 0.18	0.94 ± 0.08	0.85 ± 0.19	0.99 ± 0.01
Long	N	0.64 ± 0.11	0.99 ± 0.01	-	-	0.64 ± 0.11	**1.0** ± 0.0
	S	0.76 ± 0.19	0.98 ± 0.01	**1.0** ± 0.0	**1.0** ± 0.0	0.64 ± 0.19	0.99 ± 0.00
	P	0.89 ± 0.13	0.97 ± 0.06	0.96 ± 0.10	0.95 ± 0.07	0.90 ± 0.14	0.99 ± 0.00

and KeAcquireSpinLock. All of these required path coverage but did not depend on the length of traces. All are extracted by the DesignedAug pattern library with the exception of URL. A model for James is extracted and reported to be correct (i.e. perfect precision and recall) but on closer inspection it is imprecise i.e. there are incorrect traces it should reject but does not. This is a failing of our randomly generated testing traces, they were not able to capture all negative behaviour. Let us now review the tables of results before addressing the questions set out above.

Firstly, Table 10.3 gives an overview of the successful and failing experiments. Failure is separated into timeout (to) and no patterns passing (no pass). Firstly we note that 360 out 489 experiments were successful i.e. extracted some specification. The AdHoc pattern library saw the most timeouts, with the Designed pattern library seeing the fewest, and the Designed pattern library saw the most cases where no patterns passed. The number of failures seems directly related to the size and diversity of the pattern libraries, we discuss this below. On inspection *all* timeouts occurred during the combination stage and were a result of attempting to combine many patterns, in some cases tens of thousands.

Next let us consider Table 10.4 which gives an overview of the average negative and positive precision and recall for successful experiments. Note that failing experiments are not included and that we also report the standard deviation. Firstly, we see a lot of variance in results due to extracted specifications generally either being highly accurate or highly inaccurate. Overall extracted specifications have greater positive precision (on average 0.66) than positive recall (on average 0.48) and, therefore as expected, higher negative recall (on average 0.99) than negative precision (on average 0.77). Therefore, our technique appears to overspecify, rather than underspecify, the behaviour. We see more accurate specifications extracted with better coverage and longer trace length has a positive effect, but does not appear essential.

In Table 10.4 we give results relating to efficiency. The first thing to note is that, on average, both checking and combining phases complete in a matter of seconds, 8.4 and 5 seconds respectively for successful experiments. The maximum checking time was just under 9 minutes and the maximum combining time was under 8 minutes. Again we see a large variance with a third of experiments completing in under two seconds. We also report on the number of patterns passed and, as expected, we generally see more patterns passing for shorter traces and for pattern libraries containing more patterns.

Finally, Table 10.6 summarises the difference in size and complexity between the original and extracted QEAs. Generally extracted specifications are larger and more complex. The AdHoc pattern library extracted larger models, whereas those extracted by the Designed pattern library are typically smaller than the originals. On average, extracted specifications are roughly 3 times more complex than the original specification, with the most complex extracted specification being roughly 45 times more complex than the original.

Do longer traces lead to more accurate specifications?

To answer this question let us consider the average precision and recall for different trace lengths, as given in the following table.

	$precision^+$	$precision^-$	$recall^+$	$recall^-$	success rate
Shortest	0.44	0.70	0.30	**0.99**	**0.89**
Short	0.72	0.77	0.49	0.97	0.82
Medium	**0.83**	0.83	**0.64**	0.98	0.73
Long	0.78	**0.85**	0.63	0.98	0.75

Shorter traces have a higher success rate as they are more likely to match general patterns, but these will generally lead to inaccurate specifications. Medium traces tend to lead to the most accurate specifications, with the shortest traces not providing enough information to provide specifications of any real accuracy. The positive measures suffer most with shorter traces as behaviour that is not observed in the traces is assumed to be incorrect. The long traces suffer as this set is less likely to contain short traces as well as long traces (whilst the medium category has a higher chance), we illustrate the effect this can have below.

When inspecting the extracted specifications we observe that in some cases we see specifications 'unrolled' as the training sets did not contain any traces short enough to restrict the unrolling. This behaviour will be caused by a pattern capturing the behaviour that an event occurs at least a certain number of times. For example, in Fig. 10.2 we have the simple IO-CallDriver model (on the left) which is perfectly extracted using shortest traces but is unrolled

Table 10.5: Efficiency results for generated traces.

Timing results - times in seconds.							
		AdHoc		Designed		DesignedAug	
		check	combine	check	combine	check	combine
Shortest	N	1.58 ± 0.53	14.9 ± 41.2	11.3 ± 32.8	0.28 ± 0.53	1.88 ± 1.05	4.67 ± 5.23
	S	1.47 ± 0.60	4.05 ± 4.63	1.56 ± 1.19	0.09 ± 0.14	1.73 ± 0.75	2.18 ± 1.92
	P	1.77 ± 0.45	12.0 ± 29.8	1.70 ± 0.45	0.04 ± 0.04	2.11 ± 0.94	2.88 ± 5.30
Short	N	1.37 ± 0.64	2.80 ± 3.03	71.5 ± 167.	0.30 ± 0.43	1.80 ± 1.29	4.43 ± 4.36
	S	1.81 ± 0.74	3.76 ± 3.94	4.44 ± 7.25	0.15 ± 0.14	2.14 ± 1.42	1.66 ± 0.88
	P	1.38 ± 0.92	30.3 ± 105.	1.11 ± 0.63	0.04 ± 0.03	2.11 ± 2.39	1.78 ± 2.45
Medium	N	2.66 ± 2.40	1.93 ± 2.11	-	-	2.86 ± 2.70	1.38 ± 0.88
	S	3.85 ± 3.57	3.44 ± 2.96	14.1 ± 10.5	0.35 ± 0.17	4.30 ± 3.87	2.23 ± 1.35
	P	32.7 ± 119.	9.51 ± 14.7	40.4 ± 130.	0.06 ± 0.04	3.12 ± 2.36	1.35 ± 1.54
Long	N	2.33 ± 0.79	9.26 ± 7.36	-	-	2.90 ± 1.50	1.97 ± 0.86
	S	1.69 ± 0.87	5.73 ± 7.16	0.75 ± 0.0	0.03 ± 0.0	1.58 ± 0.71	2.70 ± 2.74
	P	10.5 ± 20.2	10.8 ± 25.4	22.0 ± 33.0	0.06 ± 0.03	15.9 ± 28.9	1.42 ± 1.37

Patterns passed				
		AdHoc	Designed	DesignedAug
Shortest	N	356 ± 355	65 ± 74	801 ± 752
	S	120 ± 101	29 ± 22	338 ± 260
	P	93 ± 61	24 ± 15	265 ± 157
Short	N	233 ± 278	64 ± 60	611 ± 626
	S	89 ± 75	44 ± 15	278 ± 203
	P	79 ± 57	26 ± 13	224 ± 140
Medium	N	37 ± 13	-	121 ± 30
	S	42 ± 29	62 ± 9	158 ± 111
	P	115 ± 114	30 ± 15	238 ± 141
Long	N	42 ± 19	-	169 ± 69
	S	40 ± 16	18 ± 0	148 ± 70
	P	81 ± 63	28 ± 12	232 ± 146

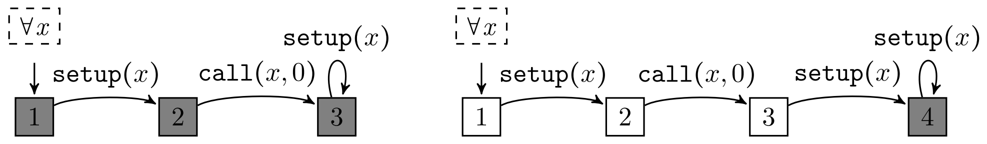


Figure 10.2: An example of loop unrolling.

(on the right) for longer traces as `setup`(x) was observed to occur at least twice. A similar effect occurred with other specifications. In the case of KeAcquireSpinLock the set of long traces contained one very short trace that prevented this pattern from matching.

In the cases where we have more complex looping behaviour between multiple events, such as LockOrdering, MutualExclusion and NestedCommand, we fail to fully generalise these orderings. For example, for NestedCommand we frequently extract models capturing exactly a small number of unrollings of the specification, as shown in Fig. 10.3. These are typically extracted for shorter traces as longer traces contain too many iterations to be captured concisely and general, simplifying, patterns fail to match.

In summary, we need a range of lengths to cover all kinds of behaviour. If traces are too long then we might assume a minimal level of occurrence and if traces are too short we may unroll looping behaviour a limited number of times.

Table 10.6: Conciseness results - Times larger or more complex.

		AdHoc		Designed		DesignedAug	
		$\lvert Q\rvert$	CC	$\lvert Q\rvert$	CC	$\lvert Q\rvert$	CC
Shortest	N	1.71 ± 1.07	1.90 ± 2.45	1.02 ± 0.48	1.02 ± 0.88	1.52 ± 0.78	1.25 ± 1.09
	S	3.33 ± 4.25	4.62 ± 7.83	1.15 ± 0.80	1.07 ± 1.34	2.64 ± 1.91	2.64 ± 2.91
	P	3.00 ± 4.00	4.00 ± 7.48	0.94 ± 0.22	0.72 ± 0.37	3.82 ± 5.83	4.77 ± 10.5
Short	N	1.57 ± 0.71	1.45 ± 1.24	0.73 ± 0.41	0.45 ± 0.26	1.45 ± 0.71	1.24 ± 1.16
	S	3.25 ± 2.93	5.02 ± 5.05	0.72 ± 0.17	0.63 ± 0.24	1.63 ± 0.81	2.06 ± 1.58
	P	4.09 ± 5.58	6.29 ± 11.9	1.12 ± 0.52	0.95 ± 0.85	3.10 ± 3.49	3.67 ± 5.88
Medium	N	1.99 ± 0.80	3.20 ± 1.45	-	-	1.91 ± 1.11	2.21 ± 1.61
	S	3.18 ± 2.01	5.82 ± 4.33	0.92 ± 0.07	0.85 ± 0.10	3.53 ± 4.73	6.94 ± 10.9
	P	3.14 ± 4.56	4.83 ± 9.78	1.05 ± 0.56	0.77 ± 0.62	1.92 ± 3.05	2.49 ± 6.59
Long	N	2.41 ± 0.82	3.75 ± 1.85	-	-	2.55 ± 1.68	3.55 ± 3.73
	S	2.68 ± 2.14	4.04 ± 3.03	0.57 ± 0.0	0.75 ± 0.0	2.00 ± 1.28	2.81 ± 2.25
	P	3.69 ± 4.04	5.38 ± 7.93	0.96 ± 0.32	0.78 ± 0.56	2.27 ± 2.86	2.84 ± 4.86

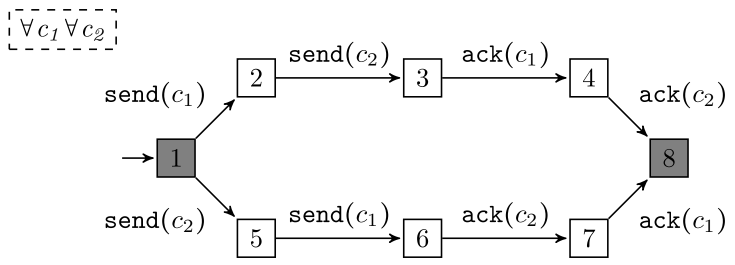


Figure 10.3: One iteration of nested command.

Do traces with better coverage lead to more accurate specifications?

The different coverage categories relate to different methods of trace generation i.e. whether we use techniques to generate traces with certain coverage guarantees. The following table demonstrates that good coverage is not required to extract a specification, but it is necessary to give an accurate specification. As noted earlier, only in the case of traces with path coverage did we manage to perfectly reconstruct the original model.

	$precision^+$	$precision^-$	$recall^+$	$recall^-$	Success rate
None	0.28	0.65	0.1	0.99	0.77
State	0.55	0.69	0.3	0.98	0.75
Path	**0.85**	**0.87**	**0.72**	0.98	**0.86**

We only see reasonable recall with path coverage and both recall and precision are greatly improved with better coverage. Recall is very low with no coverage, suggesting that we have an issue of overspecification. This makes sense as transitions not covered in the traces will not be included in the extracted specification. We also see a better success rate with path coverage, although the success rate for State and No coverage remains reasonable.

In some cases even path coverage failed to give an extracted specification, especially for the larger, more complex models. This may be due to our definition of path coverage, which only requires paths to be covered across trace slices not within trace slices. This seemed reasonable as path coverage within a single trace slice implies an object is used in all possible ways. Further work could explore different coverage criteria further, for example per slice-trace versions of path and state coverage.

In summary, coverage is important and should be considered when collecting traces.

Does our approach scale with trace length?

It is important that our approach scales to large traces. Figure 10.4 plots checking time against trace length for successful experiments for different pattern libraries. Both axes are logarithmic so the relation is linear. We see that the majority (94%) of experiments complete the checking stage in under 10 seconds. The longest checking time is 525.7 seconds and is for a set of training traces with a total length of 14.6k events. As shown in the figure, and the following table, the AdHoc and Designed pattern libraries have the longest running experiments, with the Designed pattern library having the longest checking time on average. But this information is misleading as the Designed pattern library has the largest number of unsuccessful experiments due to no patterns passing and these experiments, generally, had quick checking times for the other libraries.

	Overall	AdHoc	Designed	DesignedAug
Min	0.39	0.5	0.39	0.55
Mean	8.39	7.2	16.45	4.02
Max	525.7	525.7	492.8	98.11

Figure 10.4 shows a weak trend i.e. adding additional events tends to lead to a sublinear increase in checking time. On inspection, the outliers are caused by models with large alphabets such as URL as the checking structures are typically larger, which supports claims made in Sec. 8.5.3.

In summary, our approach scales well with trace length. We did not incorporate all of the monitoring optimisations discussed in our previous work on monitoring so there is scope to increase efficiency further.

How concise are extracted specifications?

Overall extracted specifications are not particularly concise. Table 10.7 captures, for different experimental categories, the minimum, mean, and maximum number of states and transition complexity. We also give the standard deviation in each case and emphasize the most concise.

The transition (cyclomatic) complexity of extracted specifications is generally more inflated than the number of states. This tells us that we are more likely to include extra transitions than extra states. This can be caused by patterns passing with unused transitions, and those transitions being incorporated into the final model leading to underspecification. Alternatively over restrictive patterns can introduce new restrictions on behaviour, inadvertently present in the traces leading to overspecification.

Longer traces typically lead to slightly larger specifications due to the higher chance of different orderings being present. However, shorter traces occasionally lead to very large specifications that fail to generalise from the limited information. For trace coverage we see larger specifications extracted when we have more information and path coverage can lead to very large

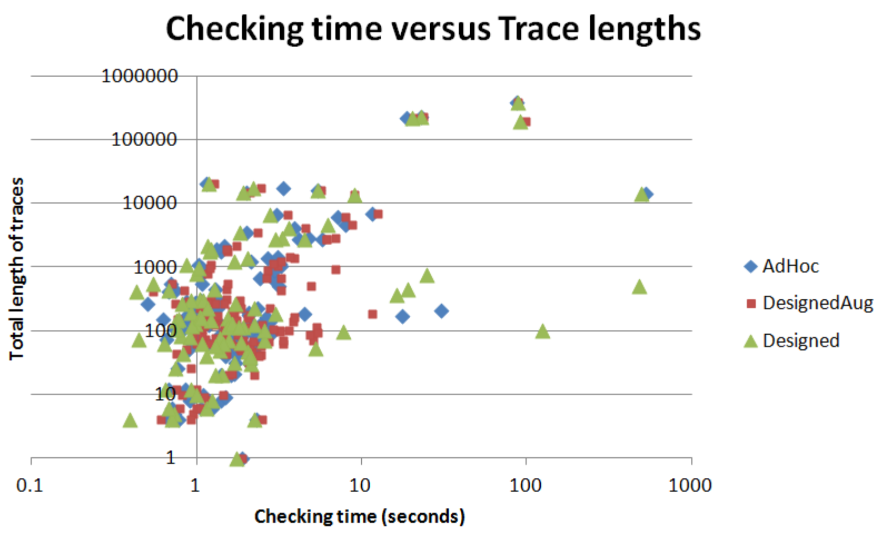


Figure 10.4: Plotting checking time against trace length for different pattern libraries.

Table 10.7: Summary of conciseness results by category.

		Shortest	Short	Medium	Long	None	State	Path	AdHoc	Designed	DesignedAug
|Q|	min	0.41	0.26	0.41	0.57	0.26	0.41	0.41	0.41	0.26	0.41
	mean	**2.18**	2.33	2.29	2.43	**1.55**	2.37	2.56	3.0	**0.99**	2.45
	stdv	3.19	3.27	3.27	2.86	0.99	2.72	3.85	3.83	0.51	3.25
	max	25.4	18.6	16.8	14	5.2	17	25.4	18.6	3.6	25.4
CC	min	0.06	0.15	0.27	0.33	0.06	0.16	0.15	0.06	0.06	0.06
	mean	**2.52**	2.97	3.4	3.32	**1.69**	3.3	3.3	4.41	**0.84**	3.0
	stdv	5.7	6.36	7.11	5.28	1.91	5.23	7.44	7.68	0.79	6.04
	max	45.5	42.2	37.6	26.7	19	30.9	45.5	42.2	5.5	45.5

specifications as we are more likely to capture irrelevant orderings. The largest specification we extract is from the shortest traces with path coverage.

The pattern library used has a large effect on the conciseness of the extracted specification. Patterns extracted with the AdHoc pattern library are the largest and those extracted with the Designed pattern library are the smallest. This is not surprising as the patterns in the AdHoc pattern library are generally larger and are more diverse overall, leading to lots of different patterns of behaviour being captured, on the other hand the Designed pattern library contains many small and similar properties, leading to many transitions or states being missed.

In summary, when we have more information in the traces often fail to generalise, reducing conciseness.

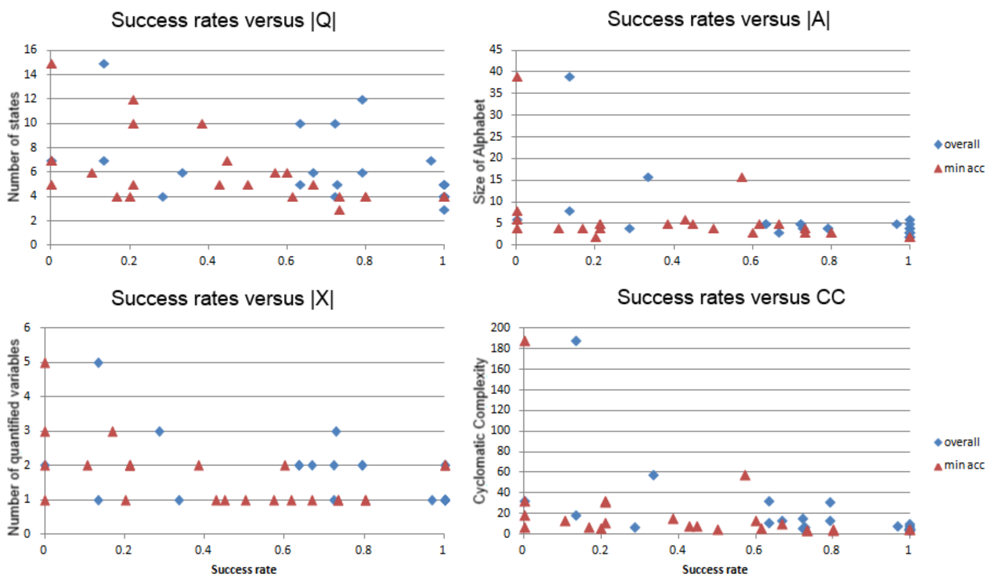


Figure 10.5: Plotting different structural properties against success rate.

Are there structural properties of the models that effect the effectiveness of our technique?

We consider the number of states, size of alphabet, number of quantified variables and the transition complexity of models and ask whether these effect the accuracy of our technique. Figure 10.5 plots these four different structural properties against success rate and the rate of experiments that lead to a specification with at least 0.5 in each accuracy measure, which we call minimum accuracy success rate.

The number of states does not appear to correlate with success rate, but there is a negative correlation with the minimum accuracy success rate suggesting that smaller models are easier to extract accurately. We see the same trends with transition complexity, although the complex Socket model makes this difficult to see on the graph. Models with two or three events in their alphabet typically saw a very high success rate and minimum accuracy success rate. Models with a single quantified variable were easier to mine and we see a negative correlation between the number of quantified variables and minimum accuracy success rate.

Four models have a zero minimum accuracy success rate: PTrace, ReadWrite, ReleaseResource and Socket, all of which have complex structures. On one hand we have Socket, with one quantified variable but a large alphabet and high transition complexity, on the other we have ReleaseResource, which has three quantified variables and a relatively small alphabet and transition complexity. With large alphabets we will produce more patterns and are more likely to timeout. With many quantified variables each trace slice contains less information.

In summary, unsurprisingly, more complex properties are more difficult to mine. However, our technique for generating traces may overly penalise such properties.

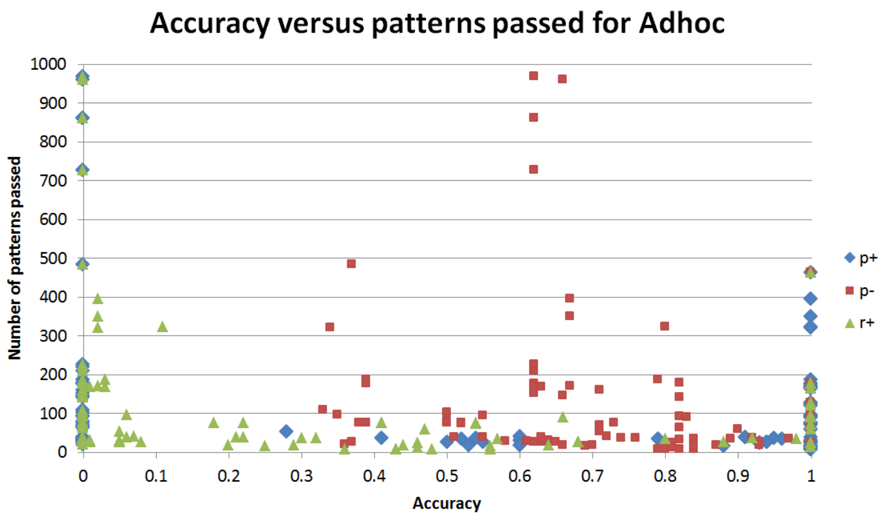


Figure 10.6: Plotting accuracy against patterns passed for different pattern libraries.

How does the pattern library effect the accuracy of extracted specifications, and the efficiency of the extraction process?

The choice of pattern library determines which patterns are successful, which in turn determines whether any specification is returned, and if so, how long it takes to produce and how well it reflects the behaviour in the traces. The following table shows the average accuracy measures (with standard deviation) and success rate broken down by pattern library.

	$precision^+$	$precision^-$	$recall^+$	$recall^-$	success rate
AdHoc	0.68 ± 0.46	0.77 ± 0.2	0.46 ± 0.44	**0.99 ± 0.04**	0.80
Designed	0.66 ± 0.44	**0.82 ± 0.23**	**0.63 ± 0.47**	0.95 ± 0.09	0.57
Des-Aug	0.62 ± 0.48	0.74 ± 0.2	0.40 ± 0.44	**0.99 ± 0.009**	**0.83**

The large variance in results is due to a polarisation; many experiments either have 0 or 1 in each accuracy measure. Variance is lower in the negative cases as specifications tend to be more consistent in the behaviours they reject.

The DesignedAug pattern library sees the most specifications extracted; this can be explained as this library has the most patterns and is most likely to capture some behaviour. No pattern library stands out as giving us more precise specifications, but the Designed pattern library generally extracts specifications with the highest negative precision and positive recall, suggesting underspecification. The Designed pattern library was designed with a particular set of behaviours in mind and contains relatively small and specific patterns, leading to generalisation in some cases and failure to extract a specification in others. For example, if we refer back to Table 10.4 we see that, in the case of long traces with no coverage, the AdHoc and DesignedAug libraries extract specifications when the Designed library does not. The four models in this category are complex and the Designed pattern library did not have the range of patterns to extract any of their behaviour.

Table 10.5 shows that the number of patterns in each library directly relates to the number of patterns extracted. Next we consider whether the number of patterns passed is important,

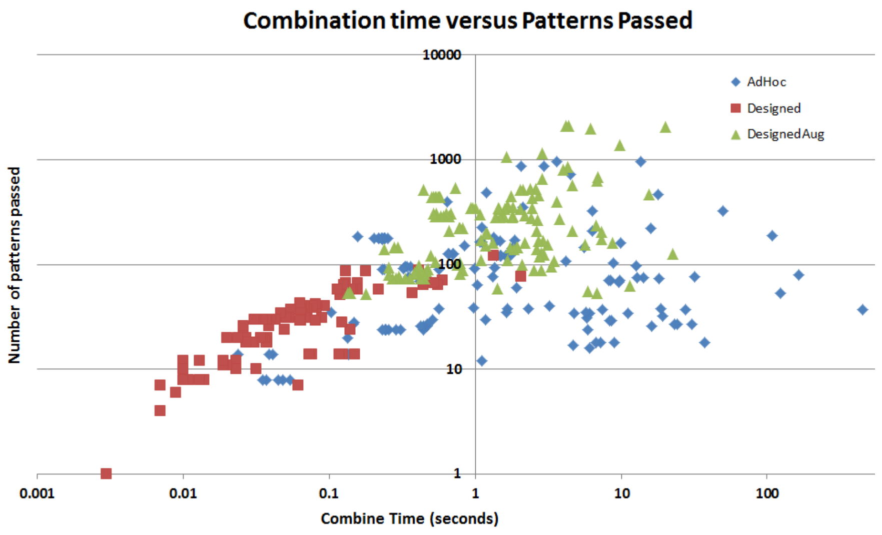


Figure 10.7: Plotting combining time against patterns passed for different pattern libraries.

perhaps a lot of patterns passing indicates a lot of evidence for our specification, or maybe it suggests a lack of clear information. Figure 10.6 plots accuracy measures against the number of patterns passed for the AdHoc library. We do not plot r^- as it is exclusively close to 1. We see a very similar pattern for the other libraries so do not illustrate them here, although the passed patterns are fewer and greater for the Designed and DesignedAug libraries respectively.

For positive precision and recall a large number of extracted patterns seems to imply either complete inaccuracy or complete accuracy, and a very large number (>500) seems to imply inaccuracy. In reference to our previous hypothesis we can see that both cases are supported. In the case of inaccuracy (and also the timeouts caused by combining many patterns) the information is too general and confused to extract an accurate specification. In the case of accuracy we have a large set of patterns supporting our result.

Negative precision is more varied, and we do not see any low (<0.3) results as some negative test traces are always rejected. For all three pattern libraries we see experiments with many passing patterns achieving roughly 0.6 negative precision. These data points are connected to those with very low positive recall and indicate that these specifications based on the general and confused set of patterns also fail to reject behaviours correctly.

Next we consider efficiency. We showed previously that checking time does not vary greatly for different pattern libraries. Figure 10.7 shows the number of patterns passed plotted against the combination time for each pattern library. We see a correlation between the two, as would be expected. This graphs supports the previous observation that fewer patterns are extracted with the Designed pattern library. Combination times vary more with the AdHoc library and tend to take longer with respect to the number of patterns. This can be attributed to the diversity of the pattern library; similar patterns take less time to combine as shared parts will collapse into each other quickly.

Table 10.8: Overview of successful experiments in connected mode

		AdHoc			Designed			DesignedAug		
		suc	to	no pass	suc	to	no pass	suc	to	no pass
Shortest	N	5	0	0	3	0	2	5	0	0
	S	4	0	0	3	0	1	4	0	0
	P	2	0	0	1	0	1	2	0	0
Short	N	3	0	0	1	0	2	3	0	0
	S	4	0	0	1	0	3	4	0	0
	P	4	0	0	0	0	4	4	0	0
Medium	N	4	0	0	2	0	2	4	0	0
	S	5	0	0	1	0	4	5	0	0
	P	4	0	0	0	0	4	4	0	0
Long	N	3	0	0	0	0	3	3	0	0
	S	5	0	0	1	0	4	5	0	0
	P	4	0	0	0	0	4	4	0	0

10.3.4 Connectedness

We now consider the effects of running in connected mode. Our research question asks whether connected mode leads to more accurate specifications and what the repercussions for efficiency are. We address both concerns here.

Overview

Only five of the 24 models are suitable for use with connected mode i.e. they have more than one quantified variable and all variables are transitively connected within the model. The models are ColIter, JFreeChart, SocketOutput, RespectConflicts and ReleaseResource. The first three were taken from settings where connected mode was assumed and the last two do not present connected behaviour. We should therefore expect better results from the first three and poorer results from the last two, which is what we find generally.

TheSocketOutput model was almost perfectly extracted, the only error being not marking the initial state as final. This lead to some test traces not exhibiting any connected behaviour being incorrectly classified and therefore less than perfect accuracy results. We also saw reasonable overspecifications of ColIter and ReleaseResource where some unimportant orderings were captured. In both cases the extracted specifications in connected mode were far more accurate then their counterparts in non-connected mode.

Table 10.8 reports on the number of successful experiments. We avoid timeouts here as the models that saw timeouts previously are not suited to connected mode. Both the AdHoc and DesignedAug pattern libraries completed all experiments successfully because, in general, connected behaviour is simpler than unconnected behaviour. The Designed pattern library did not perform as well; it struggles with RespectConflicts and ReleaseResource in particular. This is because of the simple nature of the patterns in this library. For the selected models the Designed pattern library also failed to extract specifications in the non-connected mode.

Table 10.9: Improvement in precision in connected mode

Precision							
		AdHoc		Designed		DesignedAug	
		p^+	r^+	p^+	r^+	p^+	r^+
Shortest	N	0.73 ± 0.46	0.18 ± 0.13	**1.0** ± 0.0	0.23 ± 0.15	0.51 ± 0.51	0.14 ± 0.12
	S	0.56 ± 0.45	0.17 ± 0.09	0.49 ± 0.36	0.27 ± 0.17	0.33 ± 0.43	0.10 ± 0.07
	P	0.05 ± 0.03	0.0 ± 0.07	-	-	0.0 ± 0.02	0.0 ± 0.06
Short	N	0.16 ± 0.62	0.01 ± 0.04	-	-	0.11 ± 0.68	0.02 ± 0.04
	S	0.08 ± 0.11	-0.1 ± 0.16	-	-	0.18 ± 0.47	*-0.3* ± 0.37
	P	0.02 ± 0.05	-0.2 ± 0.35	-	-	0.0 ± 0.04	0.04 ± 0.11
Medium	N	0.12 ± 0.55	-0.1 ± 0.23	-	-	0.0 ± 0.06	0.10 ± 0.14
	S	0.0 ± 0.03	-0.1 ± 0.22	-	-	0.0 ± 0.04	*-0.2* ± 0.27
	P	0.02 ± 0.02	0.0 ± 0.07	-	-	0.34 ± 0.39	0.10 ± 0.16
Long	N	0.76 ± 0.20	0.22 ± 0.10	-	-	0.58 ± 0.42	0.22 ± 0.11
	S	0.23 ± 0.44	*-0.1* ± 0.24	-	-	0.32 ± 0.49	-0.1 ± 0.22
	P	0.26 ± 0.42	0.10 ± 0.20	-	-	0.32 ± 0.41	0.11 ± 0.20

Accuracy

Table 10.9 reports the average *improvement* in precision i.e. the average amount higher the given measure is when using connected mode. Generally this is positive (we emphasize the three cases where it is not) but in many cases it is not very large. However, in a few cases we see a vast improvement in precision. Notably with SocketOutput where we see a the specification perfectly extracted. We do not report recall as improvement is generally small (at most 0.08).

We see the greatest improvements where we saw the lowest precision previously i.e. in shorter traces and lower coverage. This is because connected mode seems less effected by trace length and coverage criteria. This is most likely due to the fact that traces were not generated in connected mode, and the coverage criteria do not apply to the connected versions of the models.

Efficiency

The main issue we want to address is whether it takes more time to mine in connected mode. Detailed efficiency results are given in Appendix D.5. If we compare these to the results given in Table 10.5 for non-connected mode we can see that checking times are typically shorter and the number of patterns passed typically less. As connected mode only considers trace slices belonging to connected bindings we will generally inspect fewer events and the behaviour will be less complex, leading to fewer patterns being extracted.

Summary

In summary, connected mode can increase precision without negatively effecting efficiency. This mode could be used whenever it is applicable but we are more likely to get positive results if the property concerns objects being created from other objects.

Table 10.10: Precision-recall results for InputStream and OutputStream with extracted traces - together as one and an average of each trace individually.

InputStream				
library	*precision*$^+$	*precision*$^-$	*recall*$^+$	*recall*$^-$
AdHoc	0.87	**1.0**	**1.0**	0.93
Designed	0.80	**1.0**	**1.0**	0.89
DesignedAug	0.87	**1.0**	**1.0**	0.93
AdHoc (each)	0.7 ± 0.31	0.87 ± 0.18	0.6 ± 0.54	0.96 ± 0.02
Designed (each)	0.9 ± 0.07	0.95 ± 0.13	0.84 ± 0.39	0.94 ± 0.05
DesignedAug (each)	0.66 ± 0.3	0.74 ± 0.13	0.2 ± 0.4	0.98 ± 0.01
OutputStream				
library	*precision*$^+$	*precision*$^-$	*recall*$^+$	*recall*$^-$
AdHoc	0.0	0.64	0.0	0.94
Designed	0.86	**1.0**	**1.0**	0.92
DesignedAug	0.0	0.64	0.0	0.94
AdHoc (each)	0 ± 0	0.64 ± 0.005	0 ± 0	0.97 ± 0.02
Designed (each)	0.52 ± 0.48	0.86 ± 0.19	0.6 ± 0.55	0.95 ± 0.05
DesignedAug (each)	0 ± 0	0.64 ± 0.005	0 ± 0	0.98 ± 0.02

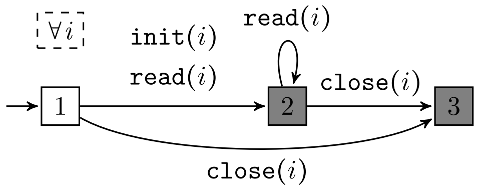
10.3.5 Extracted traces

We now consider the three models that have extracted training traces, we still use the generated traces for testing. The extracted traces are very large (we already preprocess some of them, see Sec.10.2.2) and would take up too much memory if loaded directly. Therefore, we extend our framework so that it can read traces from file incrementally. This is made possible due to our incremental monitoring algorithm introduced earlier in this work.

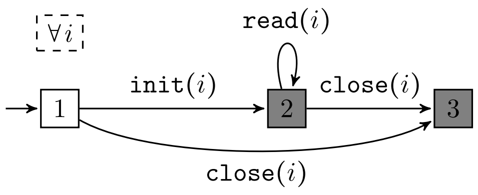
InputStream and OutputStream

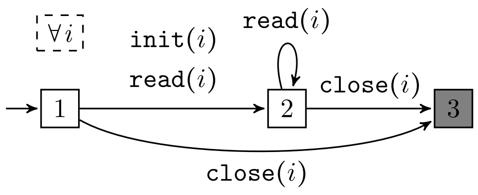
Firstly, let us consider the similar InputStream and OutputStream models. We extract models for the full set of traces and each trace individually. Table. 10.10 reports the precision-recall results.

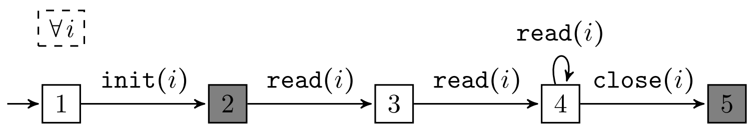
For InputStream precision and recall are high, although the specifications mined from the full set of traces slightly underspecify behaviour. The more precise of these is as follows:



Here we incorrectly allow an InputStream to be read before being created and do not require it to be closed. The specifications mined individually are overall more accurate, although there is some variance. We do not extract the specification exactly but the following three extracted specifications are close:

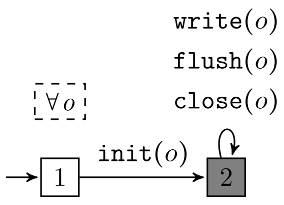




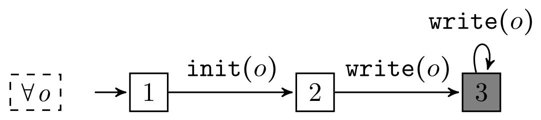


These are all very close to the original model.

For OutputStream the specifications mined from the full set of traces are poor. The specifications for the AdHoc and DesignedAug pattern libraries are far too restrictive, not allowing alternations of `write` and `flush`. The specification for the Designed pattern library is as follows:



This is underspecified, not capturing any ordering between `write`, `flush` and `close`. The specifications extracted from each trace on average did worse but a few traces produced specifications with good accuracy. Two traces do not use `flush` or `close` at all, leading to the following restrictive specification.



We also saw calls of `flush` to `System.out` and `System.err` being intercepted, without associated `init` and `close` calls, leading to many extracted specifications capturing this additional behaviour.

In both cases the Designed pattern library performed the best. The simple patterns were best suited for capturing the simple orderings required here. We note that the DesignedAug pattern library contains the Designed pattern library, yet failed to extract a specification with any precision. This is because additional patterns in this library matched and restricted the behaviour detected by the patterns from the Designed library.

Finally, we verify the extracted specifications against the original extracted traces. The following table gives the percentage of extracted traces accepted by a certain number of the extracted specifications.

InputStream		OutputStream	
Number	% passed	Number	% passed
9	100	9	100
3	50	2	80
2	30	5	20

Table 10.11: Efficiency results for InputStream and OutputStream with extracted traces. Times in seconds.

	InputStream			OutputStream		
library	checking	combination	passed	checking	combination	passed
AdHoc	364.2	5.774	38	490.2	1.182	71
Designed	269.9	1.963	1	480.6	0.024	4
DesignedAug	584.5	14.46	108	507.9	0.974	150
AdHoc (each)	30.16	1.4	67.17	47.4	1.32	296.0
Designed (each)	23.18	0.1725	6	49.49	0.034	22.3
DesignedAug (each)	48.1	2.45	170.3	49.98	1.7	636.2

Table 10.12: Precision-recall results for HasNext with extracted traces .

library	$precision^+$	$precision^-$	$recall^+$	$recall^-$
AdHoc (each)	0.28 ± 0.44	0.78 ± 0.2	0.4 ± 0.55	0.82 ± 0.38
Designed (each)	0.62 ± 0.43	0.9 ± 0.17	0.75 ± 0.46	0.86 ± 0.24
DesignedAug (each)	0.25 ± 0.5	0.72 ± 0.19	0.25 ± 0.5	1.0 ± 0.005

In both cases 9 extracted specifications accept all traces (this is not surprising as the specifications were mined from the trace) but we also see a few specifications that only accept a small number of traces. The traces for OutputStream varied more and some extracted specifications were very restricted. We also verify our ground truth against the extracted traces and find that it fails to hold on any traces as there are some input streams that are not closed in each trace. This shows us that, firstly, we should not have expected to extract the original models as they were not completely present in the traces, and secondly, that specifications extracted from individual traces could generalise to others.

Table 10.11 gives the checking and combination times and number of patterns passed. We see that checking is only slightly more efficient for InputStream than OutputStream even though the traces for OutputStream are far shorter and less complex. This can be explained by the slightly larger alphabet of OutputStream as this determines the number of pattern checkers we must update for each event, again, supporting the claims made in Sec. 8.5.3. Across all experiments we process an average of 4.6k events per second. Generally the number of patterns passing is small; in one case only one pattern passes, and forms the final specification.

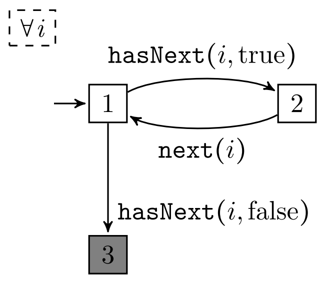
HasNext

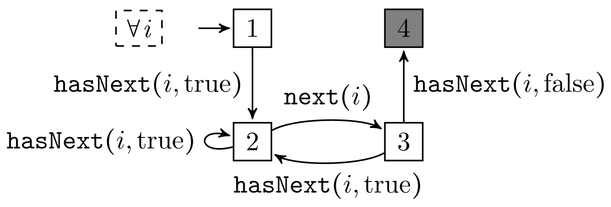
We now consider the HasNext model. Here we have a lot more data, with over 66M events. Table 10.12 gives the precision-recall for the three different pattern libraries. Again, the Designed pattern library extracted the most accurate specifications.

In total 26 specifications are extracted across all of the experiments. We fail to extract the perfect specification, but only because the majority of occurrences of iterators in our traces end with a call to `hasNext`(i,false). Therefore, we extract the following two specifications frequently:

Table 10.13: Efficiency results for Hasnext with extracted traces. Times in seconds.

library	checking	combination	passed
AdHoc	301.94	7.27	62.3
Designed	290.76	2.34	11.96
DesignedAug)	214.81	10.49	220.48





The first is our perfect specification restricted so that every trace must end with `hasNext`(i,false) and `hasNext`(i,true) can only be called once. The second also restricts the perfect specification by including this end condition, as well as unrolling it once.

Much of the inaccuracy reported in Table 10.12 is due to most extracted specifications only allowing traces to end with `hasNext`(i,false) and most positive test traces including bindings that do not. As the results show, these specifications are overspecified but they do capture the most common usage case of iterators i.e. loop until the iterator returns false.

Again we verify the extracted specifications against the original traces. The original HasNext model only accepts 68% of the traces, which means we included some incorrect traces in our training set. The table on the right gives the percentage of extracted traces accepted by a certain number of the extracted specifications.

Number	% passed
5	100
2	90
12	80
13	0

We can see the majority accept over 80% of the training set. The main reason extracted specifications failed to accept any traces was the restriction on the number of iterations allowed. This is also the main reason specifications received zero positive precision and recall.

Table 10.13 reports on the efficiency of these experiments. On average we processed 3.5k events per second. This shows that our technique can extend to the very large traces observed in some real world systems.

10.3.6 Summary of results

In this section we have demonstrated that our mining framework can efficiently extract accurate specifications. We revisit the research questions posed at the beginning of this section and summarise what we have learned.

1. Longer traces do not necessarily lead to more accurate specifications; it is more important to have a range of lengths.
2. Traces with better coverage do lead to more accurate specifications; if a path is not covered in the traces it cannot be extracted.

3. Out approach scales well with trace length and could still be improved further.
4. Extracted specifications are not very concise, if concise specifications are required then further work may be needed. The pattern library determines how concise specifications are; the more varied and complex a pattern library, the larger and more complex the extracted specifications.
5. Generally it is harder to extract specifications with larger alphabets and more quantified variables as a trace of a given length will hold less information, and more patterns may be produced, leading to a higher likelihood of timeout and capturing undesired orderings.
6. A pattern library that has been designed for a certain kind of property is more likely to *overspecify*. A pattern library that is more diverse is more likely to extract a specification (which may not be accurate). Larger pattern libraries tend to lead to more patterns passing and higher combination times.
7. If connected mode is applicable it can lead to higher accuracy with no efficiency implications. Although it may be difficult to distinguish between cases where it is applicable because of connected behaviour or a different transitive relationship between variables.
8. Our technique can process real world traces and extract accurate specifications from them.

An important observation is that when selecting a pattern library it is best to choose a combination of design plus augmentation i.e. select patterns well designed for common cases, and then augment these to capture variants of this common behaviour. As the pattern library can effect whether we tend towards over or under specification, the use case should be considered i.e. whether we would rather ignore behaviours or include extraneous information. Another take away message is that the structure of the training traces is important. Ideally this mining technique should be combined with trace extraction methods that explore the target system systematically. For example, certain automated testing techniques.

These experiments have highlighted a possible improvement. Mined specifications are not concise so further work could explore methods for minimising or 'coring' extracted specifications. We can use approaches that have been suggested elsewhere such as adding weights (probabilities) to transitions during mining and removing transitions with low weight, or just checking the training traces with the extracted specifications and removing any unused transitions.

Reflection on evaluation

Finally, we note that there are ways in which this evaluation could be improved. However, due to time limitations it was not possible to repeat or extend experiments. Firstly, it was noted earlier that it would have been interesting to explore a larger range of coverage levels that specified coverage in a single trace slice. It would also be useful to improve the trace generation methods so that they reflected real programs by, perhaps, adding weights to transitions to indicate frequency. Test traces should be generated to represent all positive or negative behaviours, rather than being generated randomly, perhaps using ideas developed in conformance testing. We noted other issues with the random traces generated; namely that this sometimes led to training sets not covering certain kinds of traces. If we had *cross-validated* our technique (compared the results between different sets of training traces), this issue would not have been observed. Future evaluation of this kind should therefore consider cross-validation.

10.4 Summary

This chapter has evaluated our pattern mining framework for QEA. We introduced a range of models that we aimed to reconstruct and a method for generating traces from these models. We then applied our technique to these models and answered a number of research questions. The results are summarised at the end of the previous section - overall these results were positive and showed that we have achieved what we set out to. As our set of ground truths contained models mined by other tools (JMiner and Tark) we argue that we can extract similar specifications to these tools, although we acknowledge that the setting is different, and that a direct comparison would be preferred.

Chapter 11

Dealing with Imperfect Traces: an initial study

The previous three chapters introduced, explored and evaluated a pattern-based specification mining approach. We identified a particular limitation to this approach; the requirement that the given traces are in some sense *perfect* i.e. belong *exactly* to the conceptual *target* specification. However, this is unrealistic. In this chapter we introduce a technique for dealing with so-called *imperfect* traces.

Let us begin with a short motivating example. Consider the following pattern:

$$\begin{tikzcd} {} \arrow[r] & \boxed{1} \arrow[r, bend left, "a"] & \boxed{2} \arrow[l, bend left, "b"] \end{tikzcd}$$

We can apply this pattern to the following trace by considering six instantiations, with each pair of symbols in the trace instantiating the pattern.

`connect.open.close`

We extract three patterns (1) $[a \mapsto \texttt{connect}, b \mapsto \texttt{open}]$, (2) $[a \mapsto \texttt{connect}, b \mapsto \texttt{close}]$ and (3) $[a \mapsto \texttt{open}, b \mapsto \texttt{close}]$. These can then be combined to form the larger specification:

$$\begin{tikzcd} {} \arrow[r] & \boxed{1} \arrow[r, "\texttt{connect}"] & \boxed{2} \arrow[r, "\texttt{open}"] & \boxed{3} \arrow[ll, bend left, "\texttt{close}"] \end{tikzcd}$$

Now imagine if we had a trace with the above sequence repeated a thousand times followed by the two events `connect` and `open` i.e. missing the final `close`. We would fail to extract the two patterns involving `close` and therefore not extract the above specification. The problem is that this approach assumes *perfect traces* i.e. that the correct behaviour is contained exactly within the given traces. This assumption is not realistic as we would like to be able to deal with cases where there are small errors in traces. The notion is that a programming pattern may hold for the majority of a program but the program may contain one or two bugs.

One approach [GS08b] to dealing with this issue is to reset a pattern being checked to its initial state when an error occurs. However, this technique would not extract the required patterns in our above example . Instead we want to be able to measure how closely a trace matches a pattern. Our approach extends the automata-based pattern mining approach to *imperfect traces* by considering so-called *edit distances* between a trace and a pattern's language.

This work considers *propositional* specification mining only, not the parametric mining we were concerned with in the previous chapters. This is so that we can consider the issue of imperfect traces without the additional mechanisms required for dealing with data values. The ideas introduced can be lifted to the parametric setting as we have seen previously.

11.1 Propositional pattern checking

In this section, we introduce a propositional pattern checking framework. The ideas presented here are similar to those seen in Chapter 8 but we repeat them here within our propositional context. We first describe how patterns are extracted from traces, then we consider how this can be done efficiently, and finally discuss how extracted patterns are combined.

11.1.1 Checking patterns

In this account, a pattern is a regular language over symbols i.e. a set of traces (finite sequences) of symbols. We consider patterns as automata:

Definition 79 (Pattern). *A pattern $p = \langle Q, \Sigma, \delta, q_0, F\rangle$ is an automaton where Q is a finite set of states, Σ is a finite alphabet of symbols, $\delta \in Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of accepting states. The language of a pattern, $\mathcal{L}(p)$ is the set of traces it accepts i.e. $\tau \in \mathcal{L}(p)$ iff there exists a path $q_0 \xrightarrow{\tau} q$ and $q \in F$ where $\rightarrow$ is δ lifted to traces.*

The process of checking a pattern against a trace considers all possible combinations of symbols in the trace as replacements for the pattern's current symbols. To replace a pattern's symbols we *instantiate* it.

Definition 80 (Instantiation). *Given a pattern p and a map φ from $p.\Sigma$ to Σ', the instantiated pattern $\varphi(p)$ has alphabet Σ' and is the result of applying φ to every symbol in p.*

The checking process then checks if each particular instantiation of the pattern holds on the trace. We say an instantiated pattern holds on a trace if the trace appears in the instantiated pattern's language after we remove irrelevant symbols. To remove irrelevant symbols we *project* the trace.

Definition 81 (Projection). *The projection $\tau \downarrow_\Sigma$ of trace τ over alphabet Σ is defined as τ with all elements not in Σ removed.*

Therefore, the extracted instantiated patterns are given as follows.

Definition 82 (Extracted patterns). *Given a pattern p and trace τ the extracted patterns are*

$$\mathsf{extract}(p, \tau) = \{\varphi(p) \mid \mathsf{dom}(\varphi) = p.\Sigma \wedge \forall (a \mapsto s) \in \varphi : s \in \tau \wedge \tau \downarrow_{\varphi(p).\Sigma} \in \mathcal{L}(\varphi(p))\}$$

11.1.2 Checking patterns efficiently

We discuss two approaches that allow us to check patterns efficiently; both techniques were used in Chapter 8, but we describe them here in our propositional setting.

Checking many instantiations

For each pattern we need to check all possible instantiations. Typically we restrict this technique to patterns over 2 or 3 symbols. We can then compute the extracted instantiated patterns for a pattern using a 2 or 3 dimensional grid of reached states. This approach was first used by Yang et al. [YEB$^+$06]. For the introductory example the following matrix would represent the states reached in the pattern after checking the trace.

		a		
		`connect`	`open`	`close`
b	`connect`	2	-	-
	`open`	1	2	-
	`close`	1	1	2

The restriction of patterns to 2 or 3 symbols is for efficiency reasons as this approach is $O(n^m)$ given an alphabet of size n and pattern with m symbols.

Checking many patterns

If we want to check multiple patterns we would currently need to repeat the above process multiple times i.e. for each pattern. However, given a set of patterns with the same set of symbols we can construct a *pattern checker* that checks all these patterns simultaneously by taking the union of the patterns and labelling states with the patterns that are accepting at that state. This general approach was previously presented in Section 8.4.2.

Definition 83 (Propositional pattern checker). *Given an alphabet of symbols Σ and a set of patterns $p_1, \ldots, p_n$ over Σ let the (propositional) pattern checker for these patterns be $\mathsf{C}(p_1, \ldots, p_n) = \langle Q, \Sigma, \Rightarrow, \Gamma \rangle$ where*

$$\begin{array}{lcl} Q & = & p_1.Q \times \ldots \times p_n.Q \\ \Rightarrow (a, (q_1, \ldots, q_n)) & = & (p_1.\delta(a, q_1), \ldots, p_n.\delta(a, q_n)) \\ \Gamma((q_1, \ldots, q_n)) & = & \{p_i \mid q_i \in p_i.F\} \end{array}$$

The patterns extracted by pattern checker C in trace τ are therefore

$$\mathsf{C}(\tau) = \{p \mid q_0 \overset{\tau}{\Rightarrow} q \wedge p \in \Gamma(q)\}$$

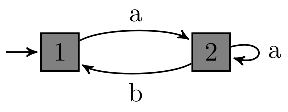
We can extend the notion of instantiation to pattern checkers and define extracted patterns for a pattern checker as follows.

Definition 84 (Pattern checker extracted patterns). *Given a pattern checker C and trace τ*

the extracted patterns are

$$\mathsf{extract}(\mathsf{C},\tau) = \{p \mid \exists\varphi : \mathsf{dom}(\varphi) = p.\Sigma \wedge \forall(a \mapsto s) \in \varphi : s \in \tau \wedge p \in \varphi(\mathsf{C})(\tau \downarrow_{\varphi(p).\Sigma})\}$$

For example, if we call the pattern in the introductory example p_1 and call the following pattern p_2



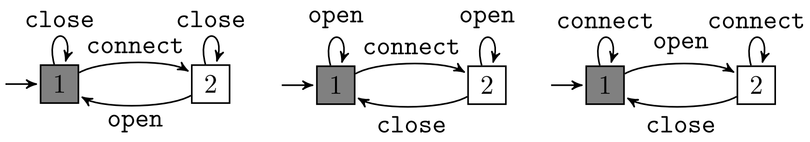
then the pattern checker for p_1 and p_2 would be

$$\begin{tikzcd}
\arrow[r] & \{p_1,p_2\} \arrow[r, bend left, "a"] & \{\} \arrow[l, "b"] \arrow[r, "a"] & \{p_2\} \arrow[loop above, "a"] \arrow[r, bend left, "b"] & \{\} \arrow[l, "a"]
\end{tikzcd}$$

where states are labelled using the output function Γ.

11.1.3 Combining patterns.

Once we have extracted a set of patterns we can *combine* them together using standard automata intersection. However, this operation is only defined when two automata have the same alphabet. To give two automata the same alphabet we can *expand* them by placing self-looping transitions on each state for the missing symbols. For example, the three extract patterns from the introductory example become:



The intersection of these three patterns is the specification given in the introduction. Formally, combination is defined as follows.

Definition 85 (Combination)**.** *Given a set of instantiated patterns $p_1,\ldots,p_n$ with combined alphabet Σ, define their combination as*

$$\mathsf{combine}(p_1,\ldots,p_n) = \mathsf{expand}^{\Sigma\backslash p_1.\Sigma}(p_1) \cap \ldots \cap \mathsf{expand}^{\Sigma\backslash p_n.\Sigma}(p_n)$$

where $\cap$ is automata intersection and $\mathsf{expand}^{\Sigma'}$ *is a function that adds self-looping transitions to a pattern for symbols in Σ'.*

We can either apply this combination operator or directly or use it to define specific combination rules. To use combination directly we can *saturate* the set by repeated application or extract a specification for each alphabet of events in the trace by combining together patterns with the same alphabet.

11.2 Dealing with imperfect traces

The previous framework will only extract a pattern if it matches exactly with an input trace. In this section we consider how it can be extended so that patterns are extracted if they match almost all of the input trace.

11.2.1 What are imperfect traces?

To say that a trace is 'imperfect' we assume that there is an implicit specification that the program that produced the trace follows and there is some bug in the program that deviates from this specification. The process of specification mining is therefore to extract this implicit specification. Alternatively, the program might be correct but the trace recording process may be faulty; either way, identifying a specification and the trace imperfections can aid debugging.

We could view these imperfections as uniform noise, however, in the case of programming bugs, it is likely that these imperfections are introduced by common mistakes such as forgetting to close a resource or check a condition, or accidentally calling the wrong method. We can therefore think of imperfections as small edits that involve the removal, addition or substitution of events from a 'perfect' trace.

In Sec. 7.2.2 we saw that real systems often contain few (repeating) errors and the general idea here is to allow for those few errors in the mining process.

11.2.2 The restart approach

Previous approaches deal with imperfect traces by 'restarting' the pattern and counting the number of such restarts. With small patterns such as the simple alternation pattern this can be effective. Let us consider the following common 3-symbol resource usage pattern.

$$\begin{tikzcd} {} \arrow[r] & \boxed{1} \arrow[r, bend left, "a"] & \boxed{2} \arrow[l, bend left, "c"] \arrow[loop right, "b"] \end{tikzcd}$$

Consider checking the following (imperfect) trace for the instantiation $[a \mapsto \texttt{open}, b \mapsto \texttt{use}, c \mapsto \texttt{close}]$. The checking would fail after the fifth event as an `open` event is omitted. If we restart here then we immediately fail again.

```
open.use.use.close.use.close.open.use.close
```

Instead, we would like to detect that the `open` event is missing and flag this as a potential bug.

11.2.3 Edit distance

As an alternative to the restart approach we consider replacing our previous condition that a trace must exactly match a pattern with the requirement that the edit-distance between the trace and any trace in the language of the pattern must be below some limit.

The edit-distance we consider uses the following "edit" operations: inserting a new symbol; deleting an existing symbol; and substituting an existing symbol for a new symbol. The edit-distance between two traces is then given by the (minimum) number of edits that transform one trace into the other. This is sometimes called the Levenshtein distance [Lev66]. Formally, this distance is given as follows.

Definition 86 (Levenshtein distance)**.** *The Levenshtein distance between traces* τ_1 *and* τ_2 *is* $\mathsf{distance}(\tau_1, \tau_2)$*, defined as*

$$\begin{aligned} \mathsf{distance}(\tau_1, \epsilon) &= |\tau_1| \\ \mathsf{distance}(\epsilon, \tau_2) &= |\tau_2| \end{aligned} \qquad \mathsf{distance}(a\tau_1, b\tau_2) = \min \begin{cases} \mathsf{distance}(\tau_1, b\tau_2) + 1 & \\ \mathsf{distance}(a\tau_1, \tau_2) + 1 & \\ \mathsf{distance}(\tau_1, \tau_2) + 1 & \text{if } a \neq b \\ \mathsf{distance}(\tau_1, \tau_2) & \text{if } a = b \end{cases}$$

We define an updated notion of extracted patterns using this metric.

Definition 87 (Imperfect extracted patterns)**.** *Given a pattern* p*, trace* τ *and integer* $\gamma > 0$*, which we call the tolerance, the imperfect extracted patterns are*

$$\mathsf{imperfect\_extract}(p, \tau, \gamma) = \left\{ \begin{array}{l|l} \varphi(p) & \mathsf{dom}(\varphi) = p.\Sigma \wedge \forall (a \mapsto s) \in \varphi : s \in \tau \wedge \\ & \exists \tau' \in \mathcal{L}(\varphi(p)) : \mathsf{distance}(\tau', \tau \downarrow_{\varphi(p).\Sigma}) < \gamma \end{array} \right\}$$

We extend this definition for pattern checkers as we did before (Sec. 11.1.2).

11.2.4 Detecting bugs

So far our approach has been abstract, considering traces of symbols generated by a program. But our motivation has been to extract specifications that allow us to detect potential bugs. To do so we need to be able to access information about the part of a program that generates a trace; we assume this is contained in a so-called *program trace.*

Definition 88 (Program trace)**.** *A program trace is a finite sequence of pairs of the form* (*code_point*, *event*) *where code_point identifies the point in the program that generates the event.*

It is easy to extend our previous constructions to work on these program traces by ignoring the code point information. Our goal is to identify points in the program trace that should be 'edited' for a mined specification to hold. These edits will follow those described above i.e. the removal of an event, addition of an event between two existing events or replacement of one event with another. The solutions we describe in the following two sections will produce so-called *rewrites.*

Definition 89 (Rewrite)**.** *A rewrite* ρ *is a finite sequence of indexes and rewrite operations that can be applied to a program trace to produce an 'edited' version.*

A rewrite can then be used to identify the code points that may contain bugs, and suggest potential solutions i.e. edits.

11.3 Editing on failure

We first consider an approach that does not use the true edit-distance, but introduces a new 'restart' operation inspired by the metric. The idea is to introduce edit operations only when a trace fails to match a pattern.

11.3.1 Failing-edit-distance

In the following we introduce an alternative formulation of the edit-distance that only applies edits when we fail. We say a pattern fails for a trace if no extensions of the trace can satisfy the pattern.

For pattern p and trace τ let $\tau = \mathsf{good}(\tau).a.\mathsf{rest}(\tau)$ where $\mathsf{good}(\tau)$ is longest prefix of τ such that

$$(\exists \tau'.\mathsf{good}(\tau).\tau' \in \mathcal{L}(p)) \wedge (\forall \tau'.\mathsf{good}(\tau).a.\tau' \notin \mathcal{L}(p)$$

Let edit be a function on symbols that non-deterministically replaces the symbol by the empty trace, a trace consisting of another symbol from the trace followed by the original symbol or another symbol in the trace i.e. it can pick one of the three edit operations discussed above.

An edited trace is defined recursively as

$$\mathsf{edited}(\tau) = \begin{array}{ll} \mathsf{edited}(\mathsf{good}(\tau).\mathsf{edit}(a).\mathsf{rest}(\tau)) & \text{if } \tau \notin \mathcal{L}(\tau) \\ \tau & \text{otherwise} \end{array}$$

i.e. the repeated application of the edit function to the event causing failure. As edit is non-deterministic the failing-edit-distance is given as the minimum number of times the edited function must be applied to a trace. This is still an edit-distance, but not necessarily minimal.

11.3.2 Computing the failing-edit-distance

To compute the failing-edit-distance we explore the non-deterministic edit operations by maintaining a number of possible *configurations* of the instantiated pattern. A configuration is a pair consisting of a rewrite (Def. 89) and state. We say that a trace *reaches* a configuration $\langle \rho, q \rangle$ for pattern p iff $q_0 \xrightarrow{\rho(\tau)} q$ where q_0 and $\rightarrow$ are the initial state and transition relation of p.

Algorithm 16 decides failing-edit-distance by computing the set of configurations reached by a trace. The algorithm uses a tolerance γ to restrict the size of rewrites and therefore the algorithm will only find the edit-distance if it is below this tolerance. The algorithm uses a function $\mathsf{failing}$ that returns true if a final (accepting) state is not reachable from the given state.

The use of γ helps restrict the exponential blow-up introduced by the non-determinism of edit functions. Other optimisations that can reduce this blow-up include restricting the number of edits allowed in a row and combining similar rewrites together.

Algorithm 16 Computing the failing-edit-distance with tolerance γ for pattern $p = \langle Q, \Sigma, \delta, q_0, F\rangle$ and trace τ.

```
$C \leftarrow \{\langle [], q_0\rangle\}$
**for** $i$ in 1 to $|\tau|$ **do**
    a $\leftarrow \tau(i)$, C' $\leftarrow$ {}
    **for** $\langle \rho, q\rangle$ in C **do**
        $q' \leftarrow \delta(q, a)$
        **if** $\mathsf{failing}(q')$ **then**
            **if** $|\rho| < \gamma$ **then**
                $C' \leftarrow C' \cup \{\langle (i,-).\rho, q\rangle, \langle (i,+b).\rho, \delta(a, \delta(b,q))\rangle, \langle (i,\%b).\rho, \delta(b,q)\rangle \mid b \in \Sigma\}$
        **else**
            $C' \leftarrow C' \cup \{\langle \rho, q'\rangle\}$
    $C \leftarrow \{\langle \rho, q\rangle \in C' \mid \neg\mathsf{failing}(q)\}$
**return** $\mathsf{min}(\{|\rho| \mid \langle \rho, q\rangle \in C \wedge q \in F\})$
```

11.3.3 Example of computing failing-edit-distance

Let us take the resource usage pattern introduced in Sec. 11.2.2 and consider the trace

$$\texttt{open.open.use.close.use}$$

for the instantiation $[a \mapsto \texttt{open}, b \mapsto \texttt{use}, c \mapsto \texttt{close}]$. Checking this pattern will fail on the second event as there is no a transition from the second state. Two edit operations can be applied here: removal of the second event or addition of a `close` event immediately before the second `open`. This leads to two alternative configurations:

$$\{\langle [(1,-)], 2\rangle, \langle [(1, +\texttt{close})], 2\rangle\}$$

We continue checking and fail again on the fifth event, the final `use`. Here there are also three edit operations that can be applied: removal of the event, addition of a `open` event or substitution of the `use` event with an `open` event. This leaves us with six final configurations:

$$\left\{ \begin{array}{c} \langle [(1,-),(5,-)], 1\rangle, \langle [(1,-),(5,+\texttt{open})], 2\rangle, \\ \langle [(1,-),(5,\%\texttt{open})], 2\rangle, \langle [(1,+\texttt{close}),(5,-)], 1\rangle, \\ \langle [(1,+\texttt{close}),(5,+\texttt{open})], 2\rangle, \langle [(1,+\texttt{close}),(5,\%\texttt{open})], 2\rangle \end{array} \right\}$$

Therefore, the instantiated pattern matches with failing-edit-distance 2.

11.4 Using the true edit distance

We now consider an approach that uses the true edit distance between the trace and language. We consider a technique that uses weighted transducers to compute the edit-distance between a trace and a finite automaton [AM09]. The general idea is that we model the trace and pattern as weighted transducers T and P and model the edit operations as a transducer X.

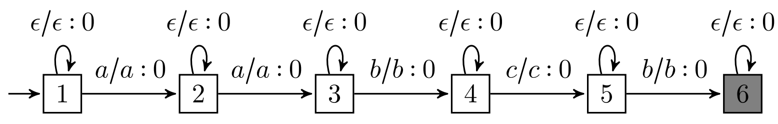
The composition $T \circ X \circ P$ will capture the different ways that the trace can be rewritten to match the pattern and the minimal edit-distance is the shortest path to an accepting state.

11.4.1 Weighted transducers

A weighted transducer has transitions labelled with an input symbol, output symbol and weight; for this application we take weights as being 0 or 1. We allow ϵ input and output transitions that can be taken without consuming or producing a symbol.

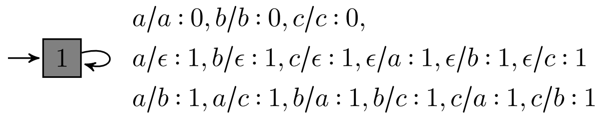
Definition 90 (Weighted transducer)**.** *A weighted transducer is a 5-tuple* $T = \langle Q, \Sigma, \Delta, \delta, F\rangle$ *where* Q *is a finite set of states,* Σ *is a finite input alphabet of symbols,* Δ *is a finite output alphabet of symbols,* $\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \{0,1\} \times Q$ *is a finite set of transitions and* $F \subseteq Q$ *is a set of final states.*

We translate traces into weighted transducers by creating a transition to a new state per event, adding self-looping ϵ transitions and only making the last state final. For example, the trace $a.a.b.c.b$ would become the following weighted transducer where transitions are written *input*/*output* : *weight*. Note that we use a weight of 0 as there is no cost associated with following the trace.



Patterns are translated by keeping the structure and labelling transitions with the same input and output symbols using a weight of 0, and adding self-looping ϵ transitions

The edit transducer consists of a single state and looping transitions for each of the edit operations it can perform; for an alphabet of $\{a, b, c\}$ this would be as follows. Note how ϵ is used to model deletions and additions and all edit operations have a weight of 1.


1
$a/a : 0, b/b : 0, c/c : 0,$
$a/\epsilon : 1, b/\epsilon : 1, c/\epsilon : 1, \epsilon/a : 1, \epsilon/b : 1, \epsilon/c : 1$
$a/b : 1, a/c : 1, b/a : 1, b/c : 1, c/a : 1, c/b : 1$

11.4.2 Composition

The composition $T \circ X$ of two transducers T and X considers all transitive matchings between strings of T and strings X i.e. if $a/b.a/c$ is a string of T and $b/d.c/a$ is a string of X then $a/d.a/a$ is a string of $T \circ X$. Here we consider a three-way composition i.e. $T \circ X \circ P$. We compute as a single operation for efficiency reasons; if we computed $T \circ X$ and then $(T \circ X) \circ P$ it is likely that $(T \circ X)$ would contain many superfluous transitions. An approach for doing this is presented by Allauzen and Mohri [AM08]. Algorithm. 17 gives an algorithm for three-way composition.

Algorithm 17 Computing the three-way composition of transducers T, X and P with the same input and output alphabets Σ and Δ.

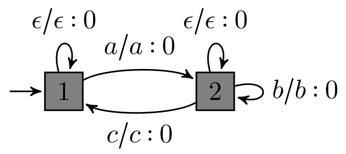
```
Enqueue($S, (T.q_0, X.q_0, P.q_0)$)
$Q \leftarrow \{(T.q_0, X.q_0, P.q_0)\}$
$\delta, F \leftarrow \varnothing$
while $\neg$isEmpty($S$) do
    $(q_1, q_2, q_3) \leftarrow$ Dequeue($S$)
    if $(q_1, q_2, q_3) \in T.F \times X.F \times P.F$ then
        $F \leftarrow F \cup \{(q_1, q_2, q_3)\}$
    for $(q_1, i_1, o_1, w_1, q_1') \in T.\delta$ and $(q_3, i_3, o_3, w_3, q_3') \in P.\delta$ do
        for $(q_2, i_2, o_2, w_2, q_2') \in X.\delta$ where $i_2 = o_1 \land o_2 = i_3$ do
            if $(q_1, q_2, q_3) \notin Q$ then
                $Q \leftarrow Q \cup \{(q_1, q_2, q_3)\}$
                Enqueue($S, (q_1, q_2, q_3)$)
            $\delta \leftarrow \delta \cup ((q_1, q_2, q_3), i_1, o_3, w_1 + w_2 + w_3, (q_1', q_2', q_3'))$
return $\langle Q, \Sigma, \Delta, \delta, F \rangle$
```

11.4.3 An example of computing true edit-distance

Let us take the same example we used for the failing edit-distance i.e. the trace

`open.open.use.close.use`

and the resource usage pattern introduced in Sec. 11.2.2. For ease of presentation we translate the trace using a for `open`, b for `use` and c for `close`. This gives us the trace used as an example in Sec. 11.4.1 above. We therefore already have our weighted transducer T. We then compute the weighted transducer P for the resource usage pattern as follows.



We now compute $T \circ X \circ P$, using the edit transducer X presented in Sec. 11.4.1 above. This gives us the weighted transducer in Figure 11.1. We then use Dijkstra's shortest path algorithm to find a shortest path between the initial state and an accepting trace. We indicate one such shortest path with a dashed line, this corresponds to the string a/a.**a/b**.b/b.c/c.**b/a** with a weight of 2. This gives two edits to our string: replacing the second `open` event with a `use` event and the last `use` event with an `open` event. Note that there are multiple paths with a weight of 2 here, and therefore multiple ways we can rewrite our trace.

A shortest path through the composition will always be at least as long as the trace and will give a rewrite by relating the projected trace back to the original trace. If a pattern checker is used then, instead of computing the shortest distance to an accepting state, for each pattern we compute the shortest distance to an accepting state labelled with that pattern.

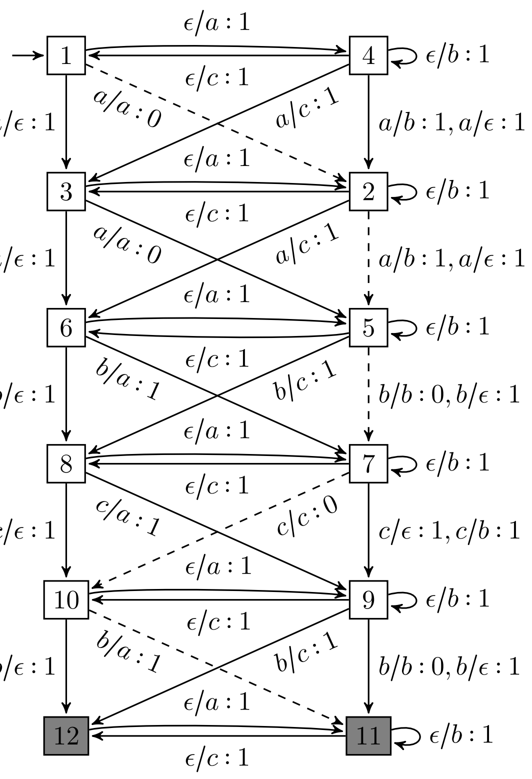


Figure 11.1: An example of the composition $T \circ X \circ P$

11.5 Combining imperfect patterns

The previous two sections presented two different techniques for extracting 'imperfect' patterns from imperfect traces. Each pattern is given a set of rewrites that tell us how to edit the input trace to make it match the pattern. When combining patterns we now need to consider these rewrites. In this section we present an approach for combining a set of imperfect patterns that are *compatible* i.e. have a set of rewrites that do not clash. We then discuss a *saturation* approach to producing a set of pattern combinations.

11.5.1 The approach

We first define what we mean by imperfect pattern. If we took an imperfect pattern as a pair of a pattern and its shortest rewrite then when combining two patterns we might find that these shortest rewrites are incompatible, but that if we had chosen, say, the second shortest rewrite we would be able to combine the two patterns. Therefore, we consider all rewrites up to a certain size for a pattern.

An imperfect pattern is a pair $\langle p, R\rangle$ where p is a pattern and R is a set of rewrites. In the case of the failing edit-distance approach R is given by the reached configurations. In the case of true edit-distance approach R is given by the language of the composition, therefore can be infinite, but in practice we use a breadth-first search to select the k-shortest paths.

A set of imperfect patterns $\{\ldots\langle p_i, R_i\rangle\ldots\}$ is *compatible* if there exists a set of rewrites

Algorithm 18 Computing the minimum compatibility between sets of rewrites R_1 and R_2 from imperfect patterns extracted from trace τ where R_i relates to a pattern with alphabet Σ_i.

```
$G \leftarrow \{R_1 \cup R_2\}$
**for** $i$ from 1 to $|\tau|$ **do**
    $G' \leftarrow \emptyset$
    **for** $g \in G$ **do**
        $D \leftarrow \{\rho \mid \rho(i) \text{ is defined }\}$
        $M \leftarrow [e \mapsto \{\rho \in D \mid \rho(i) = e\}]$
        $M \leftarrow M \cup [\tau(i) \mapsto \{\rho \in g \backslash D \mid \tau(i) \in \Sigma_i \wedge \rho \in R_i]$
        **if** $D = \emptyset$ **then** $G' \leftarrow G' \cup g$
        **else** $G' \leftarrow G' \cup \{(g \backslash D) \cup d \mid (e \mapsto d) \in M\}$
    $G \leftarrow G'$
$G_{okay} \leftarrow \{g \in G \mid \exists \rho_1 \in R_1, \rho_2 \in R_2 : \rho_1, \rho_2 \in g\}$
**if** $G_{okay} = \emptyset$ **then** **return** "incompatible"
**else** **return** $\mathsf{min}(\{|\bigcup g| \mid g \in G_{okay}\})$
```

$\{\dots\rho_i\dots \mid \rho_i \in R_i\}$ such that every pair of rewrites is compatible. Two rewrites are compatible if they do not attempt to make different rewrites at the same *relevant* points in a trace. We interpret no edit as an identity edit. A point in the trace is relevant to a rewrite if it is in the alphabet of the associated pattern. The edit-distance of $\langle p_n, R_n\rangle \cap \dots \cap \langle p_n, R_n\rangle$ is $|\rho_1 \cup \dots \cup \rho_n|$ i.e. the number of (non identity) edits when all rewrites are combined. Therefore, given a set of compatible patterns we want to find the set of rewrites that minimizes this distance.

11.5.2 Computing compatibility

We compute the compatibility between two sets of rewrites R_1 and R_2 by taking the the set $R_1 \cup R_2$ and repeatedly splitting it based on conflicts between rewrites and then checking that there is a set of rewrites with a rewrite in R_1 and R_2. An algorithm for computing compatibility between two rewrites is given in Algorithm 18. This can be extended to a set of sets of rewrites.

The algorithm will return "incompatible" if the two sets of rewrites are incompatible and the smallest number of edits that makes them compatible otherwise. Let min be the function that returns this minimum distance and is undefined otherwise.

11.5.3 Saturating the set of patterns

Given a set of imperfect patterns P_0 extracted from a trace we compute the ith saturation of P_0 as follows, recalling that $\mathsf{min}(R_1, R_2)$ is only defined if R_1 and R_2 are compatible.

$$P_{i+1} = \{\langle p_1 \cap p_2, \mathsf{min}(R_1, R_2)\rangle \mid \langle p_1, R_1\rangle\langle p_2, R_2\rangle \in P_i\}$$

In general, $|P_i| = \frac{1}{2}|P_{i-1}|(|P_{i-1}| - 1)$. However, many combinations in P_{i_1} will be trivial and can be removed. However, the saturation can grow exponentially. Let P_∞ be the fixed-point of P_i i.e. the set P_i such that $P_{i+1} = P_i$. To make saturation practical we take the following steps:

- **Limit** - We place an upper limit on the saturation set i.e. P_3

```
send("serverA",new String[]{"start","45"});
send("serverB", null);
send("serverC",new String[]{"end","23"});

void send(String address, String[] lines){
  Connection C = connect(address);
  Stream S = C.open();
  try{
    for(String line : lines) S.send(line);
  }
  catch(NullPointerException e){
    send("empty");    C.close();
  }
  C.close();
}
```

Figure 11.2: A hypothetical piece of `Java` code.

- **Prune** - We filter patterns if:
  - **Subsumption** - they are subsumed by another pattern.
  - **Maximal alphabet** - they do not use all symbols.
  - **Minimum distance** - their edit distances is $> 1.5x$ the minimum.
- **Rank** - We rank patterns by edit-distance and size.

11.6 Experiments

In this section we explore our new technique by first applying it to a hypothetical code snippet and then carrying out an experiment to evaluate accuracy where we attempt to recreate a known specification from imperfect traces.

11.6.1 Application to example code

Consider the `Java` code in Figure 11.2. This gives a hypothetical method for sending an array of lines to an address by first connecting to that address, opening a stream, sending the lines and then closing the stream. This example contains a bug; in the case where a null array of lines is given the connection is closed twice.

Let us assume we execute the above code, which calls the method three times with different inputs, recording the occurrences of the `connect`, `open`, `send` and `close` events. The resulting trace would be as follows.

```
connect.open.send.send.close.connect.open.send.close.close.
connect.open.send.send.close.
```

We now consider mining this trace with two patterns; the alternating pattern given in the introduction and the resource usage pattern given in Section 11.2.2. We take the alternating pattern first.

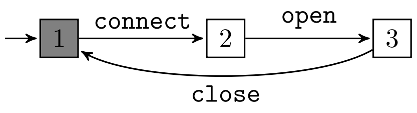
The following table gives the failing and true edit-distances (failing/true) for the above trace and the different instantiations of the alternating pattern; a '-' represents that no distance should be given (we do not consider the case where $a = b$) and an 'x' represents that no distance is returned.

		a			
		connect	open	send	close
b	connect	-	x/2	3/3	4/3
	open	0/0	-	3	4/3
	send	2/2	2/2	-	4/3
	close	1/1	1/1	3/3	-

The instantiation $[a \mapsto$ `open`$, b \mapsto$ `connect`$]$ does not have a failing edit-distance as it finishes in a non-final state that can be extended to a final state; this is one drawback of the failing edit-distance approach. For $[a \mapsto$ `close`$, b \mapsto$ `connect`$]$ and $[a \mapsto$ `close`$, b \mapsto$ `open`$]$ there is a shorter true edit-distance as this approach is allowed to make edits without failure, here removing the last event to bring the pattern into an accepting state. Note that all other distances are the same, this shows that in failing edit-distance can be a good approximation of true edit-distance.

For one case, $[a \mapsto$ `connect`$, b \mapsto$ `open`$]$ there is a distance of 0 because this instantiated pattern matches the trace exactly. If we consider the two cases where there is an edit-distance of 1 and look at the rewrite generated we see that all of these produce the same rewrite; the removal of the ninth event (the second `close`).

Combining the three instantiated patterns with an edit distance of 0 or 1 we get the following pattern.



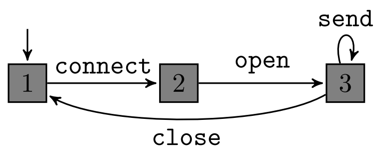
Now let us consider the resource usage pattern. The following table gives the failing and true edit distances as before, with each entry in the table representing the c dimension using a 4-tuple. Here, again, computed distances are the same but the true edit-distance approach generates some distances where the failing edit-distance approach does not.

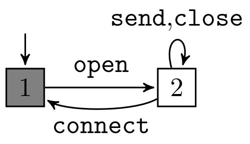
		a			
		connect	open	send	close
b	connect	(-,-,-,-)	(-,-,5/5,4/4)	(-,3/3,-,5/5)	(-,2/2,4/4,-)
	open	(-,-,2/2,1/1)	(-,-,-,-)	(5,-,-,5)	(5/5,-,4/4,-)
	send	(-,4/4,-,1/1)	(1/1,-,-,1/1)	(-,-,-,-)	(x/6,x/6,-,-)
	close	(-,4/4,5/5,-)	(1/1,-,5/5,-)	(3/3,3/3,-,-)	(-,-,-,-)

There are five instantiations with an edit-distance of 1, but they represent different rewritings of the trace. One set removes the ninth event (as before) and one set removes the first `connect` event, therefore they are incompatible. When combined they give the following respectively:

Table 11.1: Results from accuracy experiment. A=accuracy. E = edits. Time gives checking and saturation time separately in seconds.

Trace	Noise	Failing-2				Failing-5			
length	level	A	E	Time	P_3	A	E	Time	P_3
10	0.0	1.0	0	0.06, 0.84	36	1.0	0	0.06, 34.7	34
10	0.05	0.48	1	0.01, 0.96	11	0.83	3	0.01, 195	1
10	0.1	0.58	2	0.01, 2.22	2	0.80	3	0.01, 172	14
100	0.0	1.0	0	0.09, 1.64	35	1.0	0	0.09, 1.47	36
100	0.05	0.33	1	0.01, 4.96	1	0.53	4	0.01, 476	2
1000	0.0	1.0	0	0.17, 11.5	35	1.0	0	0.17, 12.4	32
1000	0.01	0.0	-	0.16, 2.48	0	0.33	2	0.16, 1382	1
Trace	Noise	Perfect-2				Perfect-20			
length	level	A	E	Time	P_3	A	E	Time	P_3
10	0.0	1.0	0	0.11, 0.13	36	1.0	0	0.05, 0.11	34
10	0.05	0.86	1	0.04, 0.05	4	0.78	1	0.03, 0.03	4
10	0.1	0.68	2	0.03, 0.06	13	0.87	2	0.03, 0.04	4
100	0.0	1.0	0	0.46, 0.27	33	1.0	0	0.26, 0.16	34
100	0.05	0.66	1	0.27, 0.17	1	0.0	-	0.29, 0.12	0
1000	0.0	1.0	0	3.29, 5.13	33	1.0	0	3.09, 0.70	35
1000	0.01	0.0	-	3.48, 3.67	0	0.16	1	3.11, 3.5	2





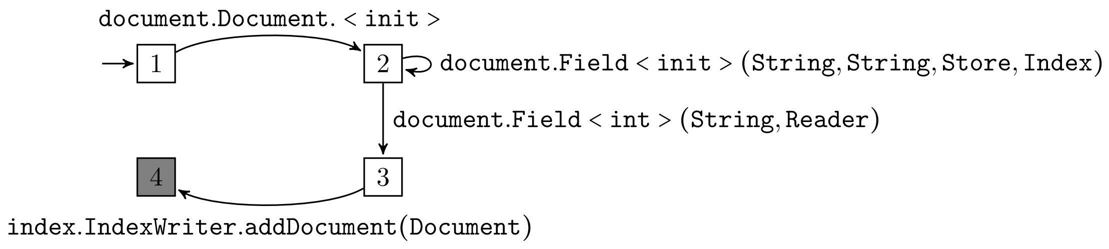
The rewrite for the first pattern here is compatible with the rewrite for pattern extracted using the alternation pattern and we can combine these patterns to form a final specification, which is the same as the one on the left above, but with only the initial state accepting.

Therefore, we can extract informative specifications that suggest alternative edits. One of these identifies the error in our code snippet, and is the specification we expected. The fact that we have two alternative specifications with the same edit distance suggests that manual inspection is required.

11.6.2 An accuracy experiment

Before we begin we should note that this experiment is not fully measuring the expected usage of this technique as there is no manual inspection of the produced specifications. We evaluate the *accuracy* of our approach by generating traces from the specification in Fig. 11.3.

We generate imperfect traces by first generating perfect traces and then randomly editing events according to some noise level (probability). We then pass these traces to our techniques and test the resulting patterns for accuracy using a set of perfect traces generated from the specification. Table 11.1 gives the *average* results over three runs. For each approach it reports the average accuracy, the minimum edit required to produce a pattern with maximum accuracy, the time taken for checking and then saturation and the size of the pruned 3-saturated set. Experiments were carried out with a range of trace lengths and noise levels and different γ for



document.Document. < init >
1
2
document.Field < init > (String, String, Store, Index)
document.Field < int > (String, Reader)
3
4
index.IndexWriter.addDocument(Document)

Figure 11.3: A specification for the Lucene tool from [GS08b]

failing and k-shortest paths for perfect.

Every experiment with a non-empty P_3 produced at least one pattern with perfect accuracy. Therefore, if we manually inspected the set P_3 we would always be able to find this specification. However, this set will also contain other specifications with the same edit distance. Therefore, we report the *average* accuracy for this set.

As expected, with zero noise we achieve perfect accuracy. In some cases we see empty P_3 for Failing as the tolerance γ is not high enough. In some cases we see empty P_3 for Perfect as the relevant rewrites are not in the top k-shortest rewrites. These parameters can be increased, but they currently have a high impact on running times. As expected, as noise increases accuracy generally decreases and in general the larger γ or k the better accuracy. A noise level of 0.1 represents a case where 10% of relevant lines of code contain bugs and, as most lines of code in the program will not be relevant, this represents a very buggy system. Therefore, obtaining reasonable results at this level suggests we could handle normal levels of bug occurrence.

Generally checking times are very fast, with saturation dominating the process. The main cost in saturation is the computation of comparability between rewrites, which is why we saw the highest saturation times with Failing-5. Further work should consider methods for optimising this process and trimming the set of rewrites considered.

It is clear that saturation of this kind is too costly. If we combine this approach with the open automata of Chapter 8 we would combine together compatible patterns directly, without saturation. Alternatively, the introduction of specific combination rules would reduce the cost of saturation.

11.7 How related work deals with imperfect traces

Here we consider how alternative techniques discussed in Chapter 2 deal with imperfect traces, we replicate a summary of the relevant techniques here for completeness.

Ammons et al. [ABL02] developed an early approach that used a probabilistic finite automata learner from the field of grammar inference and requires the alphabet of the inferred specification to be known beforehand. Imperfect traces require human experts to check violations of the inferred specification in a coring phase. Lo et al. [LK06b] extend this approach; one extension that is relevant here is the introduction of a stage that attempts to filter out erroneous traces before learning. In contrast we attempt to use this information to extract a specification and identify the error.

Techniques that use *frequent-itemset mining* (i.e. [LZ05a]) and *closed frequent sequential pattern mining* (i.e. [LM08]) rely on computing *support* and *confidence* values where support reflects the level of imperfection, and therefore can handle imperfect traces. However, the properties extracted are not as strict as automata-based specifications as in the first case symbols are only related by frequent association, not order, and in the second the ordering relation is simple. These techniques scale very well and require minimal information to be provided.

The automata-based pattern-mining technique was first used by Engler et al. [ECH+01]. They focus on the alternating pattern $(ab)^*$ and deal with imperfect traces by counting the number of times that a and b occur together in order, and a occurs without b and compute the likelihood that they form a specification. Goues and Weimer [GW09] extend this approach with techniques for pruning false positives by examining the source code.

Yang et al. [YEB+06] introduced a template-based technique focusing on extracting specifications from imperfect traces. They use the alternating pattern and deal with imperfect traces by partitioning a trace into sequences of one event followed by another, i.e. a^+b^+, performing mining on each subtrace and then counting the number of subtraces the pattern holds for. This is similar to restarting the pattern on failure but allows for a larger range of failures. They also introduce a chaining heuristic for combining their alternating patterns.

Gabel and Su. [GS08b, GS08a] extend this approach by introducing a symbolic method for specification mining using binary decision diagrams and the Javert tool that uses two patterns $(ab)^*$ and $(ab^*c)^*$ and combination rules based on automata combination to extract large patterns. They deal with imperfect traces by restarting a pattern to the initial state on failure. Later [GS10] they extend this approach to infer and enforce temporal properties at runtime over a finite window, thus detecting potential bugs at runtime.

Li et al. [LFS10] extend this approach to mine specifications with timing bounds and more complex pattern combination rules, but cannot handle imperfect traces. Instead their focus is on mining specifications from perfect traces and using these to detect bugs in imperfect ones.

Finally, recent techniques [LCR11, LRRV12] consider the *parametric* case, including our own work in previous chapters. JMiner [LCR11] extends the approach taken by Ammons et al. [ABL02] and therefore use the same coring technique to deal with imperfect traces and Tark [LRRV12] uses the notions of *support* and *confidence* from data mining.

11.8 Summary

We have introduced a new approach for mining specifications from imperfect traces. Two techniques are introduced that use the notion of edit-distance to compute the number of changes that would have to be made to a trace for a pattern to hold. We then formalise when it is safe to combine two imperfect patterns and the process is explored by first applying it to a small code snippet to demonstrate how it works and then attempting to measure the accuracy of the approach using traces generated from a known specification.

This technique not only produces specifications, but also a description of how a program should be updated to make the specification hold. This would be useful in bug detection and location but a case study is required to establish applicability.

Further work is required to improve the efficiency and applicability of the approach. This may involve the combination of this approach with an existing technique, for example the symbolic mining technique of Gabel and Su [GS08a], and combination rules explored previously [GS08b, LFS10]. The lifting of this technique to the parametric approach should also be formalised and implemented (discussed in Sec. 12.2.9). As this approach uses open automata all extracted patterns can be soundly combined to form a specification i.e. we would use pattern combination directly, rather than introducing pattern composition rules.

Chapter 12

Conclusion

This work has explored the automata-based monitoring and mining of execution traces by developing a new specification formalism, quantified event automata, and applying it in the fields of runtime monitoring and specification mining. We have developed two associated tools and, through extensive evaluation, shown that they perform well.

In the following we revisit and summarise the contributions of this thesis and then discuss further avenues of research.

12.1 Overview of work

In Section 1.1 we outlined the aim of this work as developing an expressive formalism with an efficient trace checking mechanism suitable for use in monitoring and mining. We have developed quantified event automata by combining the concept of extended finite automata with the logical notion of quantification to extend the efficient trace slicing approach. We have then implemented and optimised a monitoring algorithm, showing that it can outperform the more expressive techniques and compete with the more efficient techniques. Finally, we introduce a mining technique that makes use of this efficient trace checking procedure to efficiently extract accurate specifications from traces. We have, therefore, achieved our original aim.

In the following we revisit these three contributions and discuss the results in each area.

12.1.1 Expressive and elegant specification

We have introduced quantified event automata (Chapter 3) as a parametric trace specification formalism. These capture both a quantified and free view on data. Event automata are used to specify properties using free variables; a general notion of guards and assignments labelling transitions creates a Turing-complete language that allows us to express common properties elegantly. Quantifications are added to create quantified event automata, which specify families of event automata based on the domains of quantified variables (given by the trace).

We have introduced many examples of using QEA to represent properties from a variety of domains, and more are given in the Appendices. We have explored (Chapter 4) the complexity of the trace checking process for QEA in both the big-step and small-step semantics. This showed that this is, at worst, exponential in the number of quantified variables and dependent on the *reuse* of values in the trace. This was validated during evaluation. Finally, we have considered the expressiveness of QEA, comparing it with other languages.

12.1.2 Efficient runtime monitoring

We then turn to the issue of monitoring QEA. The problem here was that QEA were defined using a big-step style semantics i.e. to decide if a trace satisfies a QEA we needed the whole trace. This is not suitable for incremental monitoring, so we introduced a small-step style semantics and proved it equivalent to the big-step version (Chapter 5). This allowed us to give a basic incremental monitoring algorithm.

This incremental approach is an improvement on the big-step semantics, but is still not suitable for efficient runtime monitoring. Therefore, we consider a collection of techniques to improve the effectiveness and efficiency of our approach (Chapter 6). Firstly, we introduce a refined notion of verdict, allowing us to detect success or failure earlier than we could before. We then consider redundancies in the monitoring algorithm, indexing strategies and methods for dealing with pragmatic issues such as object equality and garbage collection.

The novel contributions in this part of work are a full account of the incremental trace slicing approach extended with full quantification and free variables, an incremental method for checking that is extended to the five-valued verdict domain via a notion of strong states, an extension of the value-based indexing strategy to this setting, an introduction of an effective symbol-based indexing strategy that incorporates the notion of maximality necessary for trace slicing, and an algorithm selection mechanism based on the structure of the monitored QEA.

Finally, we showed that our monitoring techniques were efficient (Chapter 7) for both a hypothetical case study and a real-world benchmark suite. We did not beat the world-leader (JavaMOP) on their home turf but remained competitive. We also vastly outperformed a very expressive system (RuleR) in a number of cases. We show that removing garbage redundant bindings is very important (and more work is required in this area). We also concluded that the number of quantified variables has a large impact on monitoring overhead for any trace-slicing technique, so should be reduced where possible.

12.1.3 Accurate specification mining

There are many different methods used for specification mining, as seen in Chapter 2, and we chose a generate-and-check approach that leveraged our efficient monitoring algorithm. Our framework (Chapter 8) introduces the novel contributions of *pattern checkers* and *open automata*. Pattern checkers allow us to check many patterns at once, by combining them together in a single structure. Open automata tackle a previously identified issue with *pattern combination*: patterns that do not capture their context lead to imprecise combination. We explore the notion of pattern combination and show that the set of patterns identifiable with open automata is strictly larger than that identifiable with finite state automata.

The performance of our framework is dependent on the *pattern library* used. Therefore, we explore the kinds of patterns we might want to include in a pattern library (Chapter 9).

We review previous pattern used and methods for automatically augmenting a set of existing patterns. Lastly, we take a set of common specification patterns and *de-combine* them to see what patterns are required to identify them.

We show that our framework can accurately and efficiently extract specifications (Chapter 10). As expected, more accurate specifications are extracted from traces containing more information (greater coverage, with a range of lengths) and our technique scales well to long traces, including those extracted from real world programs. The connectedness mode is effective where we see connected behaviour. The pattern library has a large effect on the kinds of patterns mined; their accuracy and conciseness. In general, extracted specifications were large, and future work is suggested that could reduce this if it were an issue.

During development and evaluation of our mining framework we noticed that it was highly sensitive to imperfections, or bugs, in the extracted traces. In the last part of this work we introduce a new technique for dealing with these imperfections in a propositional setting (Chapter 11). This idea of this approach is to measure the *edit-distance* between the language of a pattern (or pattern-checker) and an extracted trace. Each imperfection, or bug, in the trace is represented as an *edit* and the (minimal) sequence of edits can suggest fixes to the code to make the specification true.

12.2 Future directions

Here we discuss, in detail, possible extensions to the work described in this thesis. Some of this work has already been partly developed but is not discussed in the main text as it has not been fully implemented and evaluated.

12.2.1 Explore guard and assignment languages

We have (informally) parameterised the QEA specification language with a guard and assignment language. Future work could consider formalising this by adding a formal theory $\mathcal{T}$ to the definition of a QEA and restricting guards and assignments to those drawn from this theory. This would then allow complexity and expressiveness issues to draw on information about this theory. Furthermore, operations on QEA, such as reachability, could make use of this theory to ask certain questions, such as where one guard implies another. An extension of this idea would be to embed QEA into some logic also able to express our theory $\mathcal{T}$.

12.2.2 Completing QEA as a specification formalism

At the end of Chapter 4 on page 90 we discussed further work required to make QEA a well-rounded specification formalism. These can be split into two concerns. The first area is that of different decision problems we might wish to address with QEA, for example the empty language problem. The second area is that of language operations such as negation and intersection. Defining and exploring these further aspects of QEA would mean that it could be used as a general purpose specification language. These actions may also give further insights into how QEA compares with other well-defined languages.

12.2.3 Discrimination tree indexing

Our symbol-based indexing technique bears some similarities to the tree-based term-indexing method used in automated theorem proving [SRV01, Gra95]. In general, tree-based indexing organises terms into a single tree and uses the structure of the tree to carry out queries. This further direction briefly considers an alternative implementation of the symbol-based indexing idea that makes use of these tree-based indexes.

Currently the symbol-based indexing technique makes use of a (hash)map to store an index from events to lists of bindings (see Sec. 6.5.4). The idea is to replace this map with a tree-based indexing structure such that the list of bindings for an event can be found deterministically without hashing and dealing with collisions. We demonstrate this approach using an example.

Consider a QEA with alphabet $\{\mathtt{f}(x), \mathtt{g}(x, y)\}$. After observing the events $\mathtt{f}(5).\mathtt{g}(6,1).\mathtt{g}(6,2)$ we would have domains $[x \mapsto \{5,6\}, y \mapsto \{1,2\}]$ and an indexing map of

$$\left[\begin{array}{lcl@{\qquad\qquad}lcl}
\mathtt{f}(\_) & \mapsto & [\,] & \mathtt{g}(6,\_) & \mapsto & [x \mapsto 6] \\
\mathtt{g}(\_,\_) & \mapsto & [\,] & \mathtt{g}(5,1) & \mapsto & [x \mapsto 5, y \mapsto 1] \\
\mathtt{f}(5) & \mapsto & [x \mapsto 5, y \mapsto 1].[x \mapsto 5, y \mapsto 2].[x \mapsto 5] & \mathtt{g}(5,2) & \mapsto & [x \mapsto 5, y \mapsto 2] \\
\mathtt{f}(6) & \mapsto & [x \mapsto 6, y \mapsto 1].[x \mapsto 6, y \mapsto 2].[x \mapsto 6] & \mathtt{g}(6,1) & \mapsto & [x \mapsto 6, y \mapsto 1] \\
\mathtt{g}(5,\_) & \mapsto & [x \mapsto 5] & \mathtt{g}(6,2) & \mapsto & [x \mapsto 6, y \mapsto 2]
\end{array}\right]$$

this indexing map can be captured in the following tree:

$$\begin{tikzcd}
 & & & \epsilon \arrow[dll, "\mathtt{f}(\_)"'] \arrow[drrr, "\mathtt{g}(\_,\_)"] & & & & & \\
 & {[\,]} \arrow[dl, "\mathtt{f}(5)"'] \arrow[dr, "\mathtt{f}(6)"] & & & & & {[\,]} \arrow[dll, "\mathtt{g}(5,\_)"'] \arrow[dr, "\mathtt{g}(6,\_)"] & & \\
\begin{array}{c}[x \mapsto 5, y \mapsto 1].\\ {[x \mapsto 5, y \mapsto 2]}.\\ {[x \mapsto 5]}\end{array} & & \begin{array}{c}[x \mapsto 6, y \mapsto 1].\\ {[x \mapsto 6, y \mapsto 2]}.\\ {[x \mapsto 6]}\end{array} & & {[x \mapsto 5]} \arrow[dl, "\mathtt{g}(5,1)"'] \arrow[dr, "\mathtt{g}(6,1)"] & & & {[x \mapsto 6]} \arrow[dl, "\mathtt{g}(5,2)"'] \arrow[dr, "\mathtt{g}(6,2)"] & \\
 & & & {[x \mapsto 5, y \mapsto 1]} & & {[x \mapsto 6, y \mapsto 1]} & {[x \mapsto 5, y \mapsto 2]} & & {[x \mapsto 6, y \mapsto 2]}
\end{tikzcd}$$

Previously to gather the bindings to use for an incoming event such as $\mathtt{g}(5,3)$ we would query the index structure for the events $\mathtt{g}(5,3)$, $\mathtt{g}(5,\_)$ and $\mathtt{g}(\_,\_)$ in that order, appending the results together (to maintain maximality). Now we can gather this list with one simple traversal of the indexing tree instead of three separate lookups in the (hash)map; we start at the root and take the $\mathtt{g}(\_,\_)$ branch then the $\mathtt{g}(5,\_)$ branch before finally taking the $\mathtt{g}(5,3)$ branch, appending the results in reverse order. In some cases there will be alternate routes through the tree, but this should still result in a faster lookup. Updating the tree still requires an insertion per new symbol, but as we saw previously updates happen infrequently in comparison to lookups.

12.2.4 State based indexing

Dwyer et al. [PDE12] have described three forms of indexing for runtime monitoring: value-based, symbol-based and state-based. In this work we have explored value-based and symbol-based indexing but not state-based. We note that the TRACEMATCHES tool uses state-based indexing.

A note

This previous work ([PDE12]) captures events as being of the type $2^O \times \Sigma$ and monitors as being of type $2^O \times S$ where O are the objects in the trace, and Σ and S are the alphabet and sates of the monitored finite state property respectively. They then categorise indexing schemes as different strategies for implementing the transition relation $(2^O \times S \times \Sigma) \rightarrow S$. In this scheme value (or object) based monitoring is given as $2^O \rightarrow ((S \times \Sigma) \rightarrow S)$, symbol-based monitoring is given as $\Sigma \rightarrow ((2^O \times \Sigma) \rightarrow S)$ and state-based monitoring is given as $S \rightarrow ((2^O \times \Sigma) \rightarrow S)$. Dwyer et al. were the first to discuss symbol-based indexing.

However, this presentation is flawed in its treatment of values; by only discussing sets of values they make the same error JAVAMOP makes in not allowing $\texttt{f}(x)$ and $\texttt{f}(y)$ to both appear in a specification's alphabet. Therefore, we should consider bindings instead of sets of objects.

Advantages and disadvantages

The advantage of value and symbol based indexing is that they can use the incoming event to immediately lookup the relevant information to update. At first it appears that state-based indexing does not allow this, as an event does not give any information about states. However, if we construct, for each state, the set of event names that are *enabled* at that state (i.e. have an outgoing transition) then we can use the event name (symbol) to identify the states that we need to inspect. One disadvantage of the state-based approach is that when a monitor (binding) changes state then the indexing structures need to be modified.

It would be interesting to explore a state-based indexing approach for QEA, but it is unlikely to be more efficient than either the value or symbol based approaches.

12.2.5 Parallel monitoring

We have previously explored the notion of parallel runtime monitoring within the context of the RULER runtime verification tool [Reg10]. In this work we concluded that parallelising the internals of a monitor will not gain a significant reduction in overhead as the work carried out in each step is generally not significant.

One important conclusion was that it is important to *desynchronise* the monitor from the monitored application, allowing the monitor to make use of time between events. To do this we can identify synchronisation events on which we must allow the monitor to catch up; intuitively these events are those that could lead to a verdict we care about. If we want pure synchronous monitoring then all events are synchronisation events, alternatively if we want pure asynchronous monitoring no events are synchronisation events.

Given a set of interesting verdicts we can automatically detect synchronisation events in a QEA as those events labelling a transition into a state that can give that verdict i.e. a strong failure state if the verdict is failure. Alternatively we could add a notion of such states being *live* i.e. we either produce a static over-approximation or we dynamically track such states.

One approach that seems promising is to create a network of communicating nodes running in parallel with each node processing a subset of events. We can use the two event orderings previously discussed to organise nodes. The first being the $\sqsubseteq$ relation on bindings and the second being the lexicographical ordering on events used in tree-based indexing. Each node would check to see if an event is relevant to it and then decide whether it should pass the event to nodes containing bindings lower in the given order. This parallelism can be seen as an alternative to indexing; instead of identifying the exact part of the monitor space relevant to an incoming event we carry out a directed parallel search of the space.

In the case where we have a single quantified variable this scheme would not provide any advantage. Therefore, like indexing, we would select an appropriate parallel scheme depending on the structure of the monitored QEA.

12.2.6 Replacing event automata

As discussed in Sec. 4.4.2 we could replace event automata with an alternative event formalism and maintain the 'quantified' part of our semantics. There are two kinds of formalism we could consider replacing event automata with: so-called propositional formalisms where events are treated as symbols, possibly with free variables, and parameterised formalisms where specifications can take parameters determining the monitored states, as in RuleR.

We can consider regular expressions, context-free grammars, temporal logics and general rewriting systems. But to effectively deal with free variables we need to be able to define a rewrite system, which may not be as intuitive for some languages such as temporal logic. We cannot combine free variables with any form of past time reasoning.

12.2.7 Identifying likely alphabets

As discussed in Sec. 8.9.1 a limitation of our mining approach is that we must know the alphabet of the mined specification in order to extract the relevant traces. JMiner [LCR11] uses a technique called *event specification mining* to identify likely alphabets through source code heuristics i.e. two methods are likely to occur in the same alphabet if they are used in the same method together. However, this approach would fail to detect intra-procedural properties. They also make use of the concept of *connectedness* to ensure that the extracted alphabet is connected (as they only mine in this mode). One approach might be to extract traces from a much larger alphabet and then identify alphabets that lead to non-trivial subtraces.

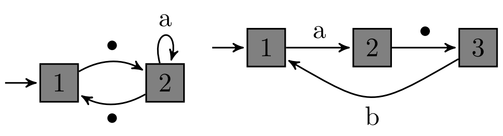
12.2.8 Free variables in mining

The presented mining technique does not allow us to extract QEAs that make use of free variables. Here we firstly demonstrate why we cannot immediately extend our current approach to free variables, and then explore possible methods that allow us to incorporate free variables, guards and assignments into the mining process.

Limitation of open automata combination

We show that the current notion of open automata combination is not compatible with free variables.

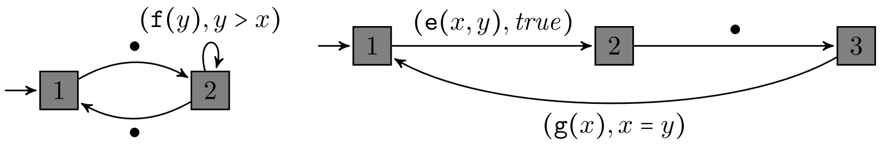
Let us consider one possible extension of our current framework where patterns can be instantiated with events containing free variables associated with guards. Now consider the following two patterns, which we will call p_1 and p_2 respectively.



Let our 'event-guard alphabet' be $\{(\texttt{e}(x,y), \mathit{true}), (\texttt{f}(y), y > x), (\texttt{g}(x), x = y)\}$ where x and y are free variables. In this setting we would then instantiate p_1 with each event-guard pair, and p_2 with each pair of event-guard pairs, giving us nine pattern instances. Now, the following trace

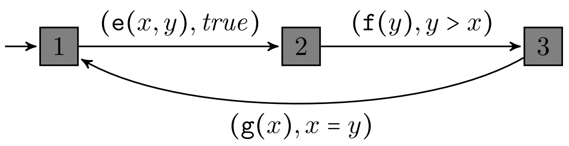
$$\texttt{e}(0,0).\texttt{f}(1).\texttt{g}(0)$$

will be successful for (only) the following two pattern instances:



We assume uninstantiated variables are mapped to zero. The first pattern instance accepts the trace as the $\texttt{f}(y)$ matches $\texttt{f}(1)$ and $1 > 0$. The second pattern instance accepts the trace as $\texttt{e}(0,0)$ matches $\texttt{e}(x,y)$, saving 0 into x and y, and $\texttt{g}(0)$ matches $\texttt{g}(x)$ and $0 = 0$. At this point the reader should be able to see where the error will occur; in the second pattern the event matching $\bullet$ updated the value of y.

Using our current notion of pattern combination we would get the following specification when combining these two pattern instances:



As noted earlier, this does not accept our original trace and the mining process is no longer sound. Our problem is that the combination process does not consider how free variables may be updated by events replacing hole symbols.

Free variable extension

Before presenting our two alternative solutions we give an overview of the general free variable extension.

The main difference is that we can include guards in the input alphabet, can specify some variables as free and may have guards in the extracted QEA. Firstly, we introduce the notion of a guard alphabet that associates a set of guards with each event, this set may consist only of the single guard *true*.

Definition 91 (Guard Alphabets)**.** *A guard alphabet $\mathcal{A}_G$ is a set of pairs $\langle \mathbf{a}, G\rangle$ where $\mathbf{a}$ is an event and G a set of guards.*

Our notion of Target QEA is extended and we can update our notion of instantiation accordingly.

The following two sections introduce two solutions to the combination problem discussed earlier. The first introduces local guards that do not allow guards to consider variables introduced previously, effectively removing the outlined issue. The second introduces an updated notion of open automata combination that labels holes with the variables they are allowed to update.

Local guards

One notion of guard that would work with our current definition of combination is a *local guard*, as used recently by Walkinshaw et al. [WTD13], who used classifiers to construct local transition guards when learning extended finite state machines.

A local guard for an event is allowed to refer to any quantified variable and the free variables of that event.

Definition 92 (Local Guard)**.** *A guard g is* local *to an event $e(\overline{x})$ for quantified variables X iff the set of variables Y that g refers to is a subset of $\overline{x} \cup X$. Let $LGuard(\mathbf{a}, X)$ be the set of such local guards.*

A guard alphabet $\mathcal{A}_G$ is local for set of quantified variables X if and only if $\forall \langle \mathbf{a}, G\rangle \in \mathcal{A}_G . G \subseteq LGuard(\mathbf{a}, X)$.

The influential Daikon tool [ECGN01, NE02] extracts invariants containing parameters that can be set during the mining process. For example, they use the invariant $x > c$ where the constant c is inferred such that the invariant holds. To allow for this behaviour we introduce a third kind of variable (beyond quantified and free); a *constant variable.* This contradictory term captures the idea that the value can only vary during the mining process and will be constant in the final specification.

To clarify, if we use the local guard $x > c$ then every time we see it we might assign c to be 1 less than the value of x so that at the end of the mining process the guard will always have been true. For more complex guards it may not be possible to relax the constant variables to ensure that the guard passes, in this case the guard must be false.

Definition 93 (Guard with Constants)**.** *A guard with constants is a function from bindings to pairs of boolean values and bindings, i.,e. a member of $Binding \rightarrow \mathbb{B} \times Binding$. The boolean* *value indicates whether the guard is true and the resultant binding gives the values for* constant variables *necessary to make the guard true.*

We can now extend our notion of trace checking to include guards with constants by using the binding in a configuration to store the value of constant variables and allowing guards to update this.

When we combine successful patterns we can either treat guards as symbols that cannot be merged or attempt to combine them in some way, by either taking the conjunction, or checking that one is subsumed in the other.

We can consider a number of optimisations, such as ordering guards (via logical implication) and only introducing a more general guard when a more specific one fails. We could also combine many guards together, in a similar method to our pattern checker technique.

Labeling holes

Before we discuss a new form of combination we note that one straightforward approach would be to *name apart* all free and constant variables, so that the variables from the two patterns could not interact. This would achieve correctness, but may lead to a very large number of variables in the extracted specification.

Alternatively, we can consider a new notion of combination that labels each hole with the variables it is allowed to update and only allows the hole to be replaced with events that have a subset of those variables. Consequently, combination is no longer deterministic as some combinations are disallowed.

The general idea is to define the *updated* variables of an event (or assignment) and the *used* variables of a guard and use these to define the *used* and *updated* variables for each transition. We then define combination such that an inserted transition does not *update* anything the next transitions *use.*

Guard and Assignment library

Both of the above approaches would work with a library of guards (and in the second case assignments). Here we discuss suitable candidates for such a library.

Following from notions in register automata we can introduce guards based on *equality.* For a set of n free variables we generate a set of $n(n-1)$ guards of either form $x = y$ or $x \neq y$. However, if we introduce the notion of *typed variables* we can reduce this set (equality between variables of the same type) and include type-specific guards. For example, if we have integer variables i and j we can introduce the guards $i = c$, $i < c$, $i > j$ or $i = cj$ for some parameter c. If we have string variables s and t we can introduce the guards $s.\texttt{subString}(t)$ or $s.\texttt{isEmpty}$.

A set of events can automatically be expanded with guards, if the set of free variables are identified. If we consider only local guards then the set of variables to consider are those used in each event, otherwise all backward-reachable free-variables are used.

Assignments would have to be highly domain specific. One kind of assignment we can introduce is one that saves the previous value of a variable in an auxiliary variable, which can be added to the set of variables used in guards. We may also wish to design particular patterns to capture notions such as counting.

Learning guards

An alternative approach would be to *learn* guards rather than use patterns as we have here. This has been explored previously. Xiao et al. [XSL⁺13] use support vector machines (SVM) to learn a linear function that separates two sets of data; each distinguishing some different behaviour. Walkinshaw et al. [WTD13] use classifiers to generate local guards. In both cases, the general idea is to *train* the classifiers on the input traces and then use the classifier as a guard.

12.2.9 Extending imperfect work to the parametric setting

Ideally we would extend the work done for imperfect traces in Chapter 11 to the parametric setting. One question to answer is how we compute the final edit-distance if we have trace slices with different edit distances i.e. if we have one slice with an edit distance of 5 and one with an edit distance of 0 what should the overall edit distance be? One solution would be to take the maximum edit distance for universal quantification, and the minimum for existential.

Bibliography

[AAC$^+$05] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Not.*, 40:345–364, October 2005.

[ABL02] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[AHK$^+$12] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F. W. Vaandrager. Automata learning through counterexample guided abstraction refinement. In *FM*, pages 10–27, 2012.

[AM08] C. Allauzen and M. Mohri. 3-way composition of weighted finite-state transducers. In *Proceedings of the 13th international conference on Implementation and Applications of Automata*, CIAA '08, pages 262–273, Berlin, Heidelberg, 2008. Springer-Verlag.

[AM09] C. Allauzen and M. Mohri. Linear-space computation of the edit-distance between a string and a finite automaton. *CoRR*, abs/0904.4686, 2009.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 106:87–106, 1987.

[Ang88] D. Angluin. Identifying languages from stochastic examples. *IEEE Transactions on Software Engineering*, 1988.

[Ang92] D. 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. ACM, 1992.

[AS83] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15:237–269, September 1983.

[BBC$^+$10] J. Berstel, L. Boasson, O. Carton, J.-E. Pin, and A. Restivo. The expressive power of the shuffle product. *Inf. Comput.*, 208(11):1258–1272, November 2010.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, 1972.

[BFH$^+$12] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In *FM*, pages 68–84, 2012.

[BFMW01] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BGHS04] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *VMCAI*, pages 44–57, 2004.

[BGHS10] H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *Journal of Aerospace Computing, Information, and Communication*, 2010.

[BH11a] H. Barringer and K. Havelund. Internal versus external DSLs for trace analysis. In *Proc. of the 2nd Int. Conference on Runtime Verification (RV'11)*, volume 7186 of *LNCS*, pages 1–3. Springer, 2011.

[BH11b] H. Barringer and K. Havelund. Tracecontract: a Scala DSL for trace analysis. In *Proc. of the 17th international conference on Formal methods*, pages 57–72, Berlin, Heidelberg, 2011.

[BHL+07] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative runtime verification with tracematches. In *Proceedings of the 7th International Conference on Runtime Verification*, RV'07, pages 22–37, Berlin, Heidelberg, 2007. Springer-Verlag.

[BHLM13] B. Bollig, P. Habermehl, M. Leucker, and B. Monmege. A fresh approach to learning register automata. In *Developments in Language Theory*, pages 118–130, 2013.

[BHRG09] H. Barringer, K. Havelund, D. Rydeheard, and A. Groce. Rule systems for runtime verification: A short tutorial. *Runtime Verification*, 2009.

[BJLS03] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin's learning. Technical report, In International Workshop on Software Verification and Validation (SVV, 2003.

[BJR06] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In *FASE*, pages 107–121, 2006.

[BKA11] N. E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 2–26. Springer-Verlag, 2011.

[BL05] J. Bongard and H. Lipson. Active coevolutionary learning of deterministic finite automata. *J. Mach. Learn. Res.*, 6:1651–1678, December 2005.

[BLH10] E. Bodden, P. Lam, and L. Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 183–197, Berlin, Heidelberg, 2010. Springer-Verlag.

[BLS07] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 126–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[BLS10] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *J. Log. and Comput.*, 20(3):651–674, June 2010.

[Bod05] E. Bodden. J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University, November 2005.

[BRH08] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from EAGLE to RULER. *J Logic Computation*, November 2008.

[BW08] R. P. Buse and W. R. Weimer. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 121–130, New York, NY, USA, 2008. ACM.

[CGP10] C. Colombo, A. Gauci, and G. J. Pace. Larvastat: Monitoring of statistical properties. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 480–484. Springer-Verlag, 2010.

[CHJ$^+$11] S. Cassel, F. Howar, B. Jonsson, M. Merten, and B. Steffen. A succinct canonical register automaton model. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, pages 366–380, Berlin, Heidelberg, 2011. Springer-Verlag.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, October 1971.

[CJH13] Y.-H. Choe, C.-Y. Jong, and S. Han. Software cognitive information measure based on relation between structures. *CoRR*, abs/1304.0374, 2013.

[CJK07] M. Christodorescu, S. Jha, and C. 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.

[CK02] O. Cicchello and S. 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.

[CK10] N. Choubey and M. 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.

[CL11] A. Clark and S. Lappin. Computational learning theory and language acquisition. In R. Kempson, N. Asher, and T. Fernando, editors, *Handbook of Philosophy of Linguistics*. Elsevier, 2011. forthcoming.

[Cla96] M. Clavel. Principles of Maude. *Electronic Notes in Theoretical Computer Science*, 4:65–89, 1996.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second ed.)*, chapter 21: Data structures for Disjoint Sets, pages 498–524. MIT Press, 2001.

[CP99] M. Chechik and D. Păun. Events in property patterns. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 154–167. Springer Berlin Heidelberg, 1999.

[CPS09] C. Colombo, G. J. Pace, and G. Schneider. LARVA — safer monitoring of real-time Java programs (tool paper). In *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 33–37. IEEE Computer Society, November 2009.

[CR08] F. Chen and G. Roşu. Mining parametric state-based specifications from executions. Technical Report UIUCDCS-R-2008-3000, University of Illinois at Urbana-Champaign, 2008.

[CR09] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 246–261, Berlin, Heidelberg, 2009. Springer-Verlag.

[CSR08] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A predictive runtime analysis tool for Java. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 221–230. ACM, 2008.

[DAC99] M. B. Dwyer, G. S. Avrunin, and J. 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.

[DGR04] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Trans. Softw. Eng.*, 30(12):859–872, December 2004.

[dH05] M. d'Amorim and K. Havelund. Event-based runtime verification of Java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[DKM+10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 85–96. ACM, 2010.

[dlH05] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recogn.*, 38:1332–1348, September 2005.

[dlH10] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars.* Cambridge University Press, New York, NY, USA, 2010.

[DLWZ06] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. *Proceedings of the 2006 international workshop on Dynamic systems analysis - WODA '06*, page 17, 2006.

[Dru00] D. Drusinsky. The temporal rover and the ATG rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[Dup94] P. 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.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198 – 208, mar 1983.

[ECGN01] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27:99–123, February 2001.

[ECH+01] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[EFB01] T. Elrad, R. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[FGMP94] M. Frazier, S. Goldman, N. Mishra, and L. Pitt. Learning from a consistently ignorant teacher. In *Proceedings of the seventh annual conference on Computational learning theory*, COLT '94, pages 328–339. ACM, 1994.

[FHR13] Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In M. Broy and D. Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear.* IOS Press, 2013.

[For82] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. volume 19, pages 17–37. 1982.

[FOW66] L. Fogel, A. Owens, and M. Walsh. *Artificial intelligence through simulated evolution.* Wiley, Chichester, WS, UK, 1966.

[GDPT13] R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 260–276, Berlin, Heidelberg, 2013. Springer-Verlag.

[GJL10] O. Grinchtein, B. Jonsson, and M. Leucker. Learning of event-recording automata. *Theor. Comput. Sci.*, 411:4029–4054, October 2010.

[GLO08] J. Goubault-Larrecq and J. Olivain. Runtime verification. chapter A Smell of Orchids, pages 1–20. Springer-Verlag, Berlin, Heidelberg, 2008.

[GMS00] A. K. Ghosh, C. Michael, and M. 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.

[GOA05] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. *SIGPLAN Not.*, 40(10):385–402, October 2005.

[Gol67] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978.

[GPD11] R. Grigore, R. L. Petersen, and D. Distefano. TOPL: A language for specifying safety temporal properties of object-oriented programs. In *FOOL 2011*, 2011.

[Gra95] P. Graf. Substitution tree indexing. In J. Hsiang, editor, *Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin Heidelberg, 1995.

[GRS04] D. Gao, M. K. Reiter, and D. 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.

[GS08a] M. Gabel and Z. 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.

[GS08b] M. Gabel and Z. Su. Symbolic mining of temporal specifications. *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 51, 2008.

[GS10] M. Gabel and Z. 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.

[GS12] M. Gabel and Z. Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 4:1–4:11, New York, NY, USA, 2012. ACM.

[GW09] C. L. Goues and W. Weimer. Specification mining with few false positives. *Tools and Algorithms for the Construction and*, pages 1–15, 2009.

[GZ03] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[Hav13a] K. Havelund. Rule-based runtime verification revisited. *International Journal on Software Tools for Technology Transfer (STTT). (submitted)*, 2013.

[Hav13b] K. Havelund. A scala DSL for rete-based runtime verification. In A. Legay and S. Bensalem, editors, *RV*, volume 8174 of *Lecture Notes in Computer Science*, pages 322–327. Springer, 2013.

[HBPS11] M. Horridge, S. Bail, B. Parsia, and U. Sattler. The cognitive complexity of OWL justifications. In *Proceedings of ISWC-11*, 2011.

[HFH$^+$09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[HFS98] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6:151–180, August 1998.

[Hin01] P. Hingston. A genetic algorithm for regular inference. In *Genetic and Evolutionary Computation Conference*, 2001.

[HR01] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 135–, Washington, DC, USA, 2001. IEEE Computer Society.

[HR04] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *STTT*, 6(2):158–173, 2004.

[HSJC12] F. Howar, B. Steffen, B. Jonsson, and S. Cassel. Inferring canonical register automata. In *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'12, pages 251–266, Berlin, Heidelberg, 2012. Springer-Verlag.

[ISBF07] K. L. Ingham, A. Somayaji, J. Burge, and S. Forrest. Learning DFA representations of HTTP for protecting web applications. *Comput. Netw.*, 51:1239–1255, April 2007.

[JMGR11] D. Jin, P. O. Meredith, D. Griffith, and G. Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, 2011. to appear.

[Joa98] T. Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods – Support Vector Learning*, pages 169–185. MIT Press, 1998.

[Kar72] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KC05] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, NY, USA, 2005. ACM.

[KHB99] M. Karaorman, U. Hölzle, and J. L. Bruno. jContractor: A reflective Java library to support design by contract. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 175–196, London, UK, UK, 1999. Springer-Verlag.

[KL97] B. Keller and R. 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.

[KMT95] T. Koshiba, E. Mäkinen, and Y. 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.

[KTB+06] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. 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.

[KV94] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM*, 41:67–95, January 1994.

[Lad03] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[Lan92] K. 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.

[Lan95] 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.

[LBaK+98] I. Lee, H. Ben-abdallah, S. Kannan, M. Kim, I. Sokolsky, and M. Viswanathan. A monitoring and checking framework for run-time correctness assurance. In *In Proceedings of the 1998 Korea-U.S. Technical Conference on Strategic Technologies*, 1998.

[LBR98] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *In Formal Underpinnings of Java Workshop (at OOPSLA'98*, 1998.

[LCH+09] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th international conference on Knowledge discovery and data mining*, pages 557–566, New York, NY, USA, 2009. ACM.

[LCR11] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 591–600. ACM, 2011.

[Leu07] M. 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.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

[LFS10] W. Li, A. Forin, and S. 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.

[LJMR12] C. Lee, D. Jin, P. O. Meredith, and G. Roşu. Towards categorizing and formalizing the JDK API. Technical Report http://hdl.handle.net/2142/30006, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2012.

[LK06a] D. Lo and S.-C. 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.

[LK06b] D. Lo and S.-C. Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 265–275, New York, NY, USA, 2006. ACM.

[LK08] D. Lo and S. Khoo. Mining patterns and rules for software specification discovery. *Proceedings of the VLDB Endowment*, 1(2):1609–1616, 2008.

[LKHL11] D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining Software Specifications: Methodologies and Applications*. CRC Press, May 2011.

[LKL07] D. Lo, S.-C. Khoo, and C. 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.

[LKL08a] D. Lo, S. Khoo, and C. Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[LKL08b] D. Lo, S.-C. Khoo, and C. 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.

[LM08] D. Lo and S. Maoz. Specification mining of symbolic scenario-based models. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 29–35. ACM, 2008.

[LM12] D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. *Autom. Softw. Eng.*, 19(4):423–458, 2012.

[LMP08] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 501–510, NY, USA, 2008. ACM.

[LMP09] D. Lo, L. Mariani, and M. 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.

[LMS12] D. Lo, L. Mariani, and M. Santoro. Learning extended FSA from software: An empirical assessment. *J. Syst. Softw.*, 85(9):2063–2076, September 2012.

[LPH+07] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, and RA. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. *ACM SIGOPS*, 2007.

[LPP98] K. J. Lang, B. A. Pearlmutter, and R. 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.

[LR03] S. M. Lucas and T. J. Reynolds. Learning DFA: evolution versus evidence driven state merging. In *IEEE Congress on Evolutionary Computation (1)*, pages 351–358, 2003.

[LR05] S. M. Lucas and T. J. Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27:1063–1074, July 2005.

[LRRV09] D. Lo, G. Ramalingam, V. 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.

[LRRV12] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. *Sci. Comput. Program.*, 77(6):743–759, 2012.

[LZ05a] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, 30(5):306–315, 2005.

[LZ05b] B. Livshits and T. 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.

[M9̈6] E. Mäkinen. A note on the grammatical inference problem for even linear languages. *Fundam. Inf.*, 25:175–181, February 1996.

[MA08] S. Misra and I. Akman. A model for measuring cognitive complexity of software. In I. Lovrek, R. Howlett, and L. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 5178 of *Lecture Notes in Computer Science*, pages 879–886. Springer Berlin Heidelberg, 2008.

[McC76] T. J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[Mey92a] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Mey92b] B. Meyer. *Eiffel: The Language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[MGR] O. Mondragon, A. Q. Gates, and S. Roach. Prospec: Support for elicitation and formal specification of software properties. In *in Proc. of Runtime Verification Workshop, ENTCS*, page 2004.

[MJG+11] P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *J Software Tools for Technology Transfer*, pages 1–41, 2011.

[MLL05] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. *SIGPLAN Not.*, 40(10):365–383, October 2005.

[MP95] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118:316–326, May 1995.

[MP08] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 117–126, Washington, DC, USA, 2008. IEEE Computer Society.

[MR10] P. Meredith and G. Roşu. Runtime verification with the RV system. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 136–152, Berlin, Heidelberg, 2010. Springer-Verlag.

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[MTP05] F. Masseglia, M. Teisseire, and P. Poncelet. Sequential pattern mining: A survey on issues and approaches. *Encyclopedia of Data Warehousing and Mining*, 2005.

[Mur96] K. Murphy. Passively learning finite automata. Technical report, 1996.

[NC02] N. Niparnan and P. 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.

[NE02] J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. *SIGSOFT Softw. Eng. Notes*, 27:229–239, July 2002.

[NP07] A. Naidoo and N. 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.

[NSV04] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, July 2004.

[O11] B. A. O. A novel metric for measuring the readability of software source code. In *Journal of Emerging Trends in Computing and Information Sciences*, 2011.

[Obj09] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.2, 2009.

[Oeh05] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security and Privacy*, 3:58–62, March 2005.

[OG91] J. Oncina and P. Garcia. *Inferring regular languages in polynomial update time*, chapter -. World Scientific Publishing, 1991.

[OGV93] 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.

[OSV08] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[Pan10] H. Pandey. Context free grammar induction library using genetic algorithms. In *Computer and Communication Technology (ICCCT), 2010 International Conference on*, pages 752 –758, sept. 2010.

[PBG10] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[PDE12] R. Purandare, M. B. Dwyer, and S. Elbaum. Monitoring finite state properties: Algorithmic approaches and their relative strengths. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 381–395, Berlin, Heidelberg, 2012. Springer-Verlag.

[PG09] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[PG12] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *International Conference on Software Engineering (ICSE)*, 2012.

[PGB$^+$08] C. S. Păsăreanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Form. Methods Syst. Des.*, 32:175–205, June 2008.

[Pit89] L. Pitt. Inductive inference, DFAs, and computational complexity. In K. Jantke, editor, *Analogical and Inductive Inference*, volume 397 of *Lecture Notes in Computer Science*, pages 18–44. Springer Berlin / Heidelberg, 1989.

[PN02] P. 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.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[PVY01] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7:225–246, November 2001.

[Rau96] M. Rauterberg. How to measure cognitive complexity in human-computer interaction, 1996.

[RBK$^+$13] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[RBR13a] G. Reger, H. Barringer, and D. Rydeheard. Automata-based pattern mining from imperfect traces. In *The 2nd International Conference on Software Mining*, 2013.

[RBR13b] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *The 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, pages 658–663. IEEE, 2013.

[RC12] G. Roşu and F. Chen. Semantics and algorithms for parametric monitoring. *Logical Methods in Computer Science*, 8(1):1–47, 2012. Short version presented at TACAS 2009.

[Reg10] G. Reger. Rule-based runtime verification in a multicore system setting. Master's thesis, School of Computer Science, University of Manchester, 2010.

[RGJ07] M. K. Ramanathan, A. Grama, and S. 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.

[RP97] A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In *International Conference on Machine Learning*, 1997.

[RSM08] H. Raffelt, B. Steffen, and T. 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.

[Sak97] Y. Sakakibara. Recent advances of grammatical inference. *Theor. Comput. Sci.*, 185:15–45, October 1997.

[SB06] V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Int. Workshop on Runtime Verification (RV'05)*, volume 144(4) of *ENTCS*, pages 109–124. Elsevier, 2006.

[SBDB01] 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.

[SBS+12] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 193–207, Berlin, Heidelberg, 2012. Springer-Verlag.

[SDV] Windows Driver Development. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff551714\%28v=vs.85\%29.aspx`.

[SG09] M. Shahbaz and R. Groz. Inferring mealy machines. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 207–222, Berlin, Heidelberg, 2009. Springer-Verlag.

[SGKR07] S. Salamah, A. Q. Gates, V. Kreinovich, and S. Roach. Using patterns and composite propositions to automate the generation of LTL specifications. In *Proceedings of the 5th International Conference on Automated Technology for Verification and Analysis*, pages 533–542, Berlin, Heidelberg, 2007. Springer-Verlag.

[SHL08] G. Shu, Y. Hsu, and D. 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.

[SK99] Y. Sakakibara and M. 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.

[SL07] G. 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.

[SLG07a] M. Shahbaz, K. Li, and R. Groz. Learning and integration of parameterized components through testing. In A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, editors, *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 319–334. Springer Berlin / Heidelberg, 2007.

[SLG07b] M. Shahbaz, K. Li, and R. 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.

[SM00] Y. Sakakibara and H. 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.

[SO95] M. Schwehm and A. 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.

[SRV01] R. Sekar, I. V. Ramakrishnan, and A. Voronkov. Handbook of automated reasoning. chapter Term Indexing, pages 1853–1964. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001.

[Sto10] V. Stolz. Temporal assertions with parametrized propositions*. *J. Log. and Comput.*, 20:743–757, June 2010.

[Str92] G. Strube. The role of cognitive science in knowledge engineering. In *Proceedings of the First Joint Workshop on Contemporary Knowledge Engineering and Cognition*, pages 161–174, London, UK, UK, 1992. Springer-Verlag.

[SY86] R. Strom and S. Yemini. Typestate : A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 1986.

[SYFP07] S. Shoham, E. Yahav, S. Fink, and M. 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.

[Tak88] Y. Takada. Grammatical inference for even linear languages based on control sets. *Information Processing Letters*, 28(4):193 – 199, 1988.

[TH08] K. Tu and V. 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.

[TMs] abc compiler website. `http://www.sable.mcgill.ca/abc/`.

[Val84] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27:1134–1142, November 1984.

[vR79] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, London, 2nd edition, 1979.

[WB68] C. S. Wallace and D. M. Boulton. An information measure for classification. *Computer Journal*, 11(2):185–194, August 1968.

[WB08] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 248–257, Washington, DC, USA, 2008. IEEE Computer Society.

[WBHS08] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Improving dynamic software analysis by applying grammar inference principles. *J. Softw. Maint. Evol.*, 20:269–290, July 2008.

[WBJ08] N. Walkinshaw, K. Bogdanov, and K. 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.

[WML02] 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.

[WN05] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In N. Halbwachs and L. 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.

[WTD13] N. Walkinshaw, R. Taylor, and J. Derrick. Inferring extended finite state machine models from software executions. In R. Lämmel, R. Oliveto, and R. Robbes, editors, *WCRE*, pages 301–310. IEEE, 2013.

[Wya93] 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.

[WZL07] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *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. ACM, 2007.

[XSL$^+$13] H. Xiao, J. Sun, Y. Liu, S.-W. Lin, and C. Sun. Tzuyu: Learning stateful typestates. *The 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, 2013.

[YE04a] J. Yang and D. 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.

[YE04b] J. Yang and D. 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.

[YEB$^+$06] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. 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.

[ZSLL09] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. Runtime verification. chapter DMaC: Distributed Monitoring and Checking, pages 184–201. Springer-Verlag, Berlin, Heidelberg, 2009.

AUTOMATA BASED MONITORING AND MINING OF EXECUTION TRACES

VOLUME II OF II

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2014

By
Giles Reger
School of Computer Science

Contents

Appendix A

Further Examples

Here we give further examples of QEAs to support the description given in Chapter 3. We begin in A.1 by giving further examples that demonstrate the definition of QEA. We then give a number of examples that demonstrate different trace checking processes in A.2. next we introduce sets of QEAs from two different domains - the `Java` Standard library (in A.3) and a hypothetical planetary rover case study (in A.4). Finally, we use an example related to file usage to comment on different styles of writing QEA in A.5.

A.1 Building up to QEA

These examples support those given in Chapter 3.

A.1.1 SQSEA

SQSEA are defined in Section 3.3. We give a worked example for simple file usage.

In this example we consider the proper usage of files - which is slightly more involved than the alternating open/close example given in Section 3.3. Here file f is used correctly if

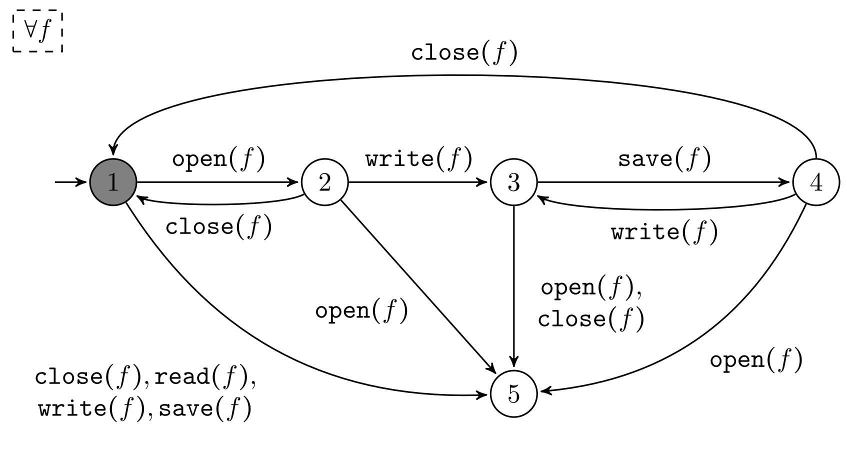


Figure A.1: SQSEA for proper usage of files

1. f is only used when open
2. f is not closed when open or open when closed and starts closed
3. If f is opened then it must be closed before the end of the program
4. If f is written to then it must be saved before being closed

The SQSEA $\mathcal{QF} = \langle\{f\}, \mathcal{F}\rangle$ for this property is given in Fig. A.1. The first thing to note is that state 5 is a sink failure state i.e., it has no outgoing transitions and is non-final and therefore if we get to state 5 we cannot succeed. Let us consider the following trace where files are identified by integers:

$$\begin{aligned}\tau = \;& \texttt{open}(1).\texttt{read}(1).\texttt{read}(1).\texttt{open}(2).\texttt{write}(2).\texttt{save}(2).\texttt{read}(2).\texttt{write}(1).\\ & \texttt{save}(1).\texttt{close}(1).\texttt{open}(3).\texttt{read}(3).\texttt{write}(2).\texttt{close}(3).\texttt{close}(2)\end{aligned}$$

We want to decide whether $\tau \in \mathcal{L}(\mathcal{QF})$. The first step is to construct $\textsf{relevant}(\tau, \{f\})$, in this case it is easy to see that $\textsf{Dom}(\tau) = [f \mapsto \{1,2,3\}]$ and therefore $\textsf{relevant}(\tau, \{f\}) = \{[f \mapsto 1], [f \mapsto 2], [f \mapsto 3]\}$. We can now evaluate each binding as described in Def. 17 i.e., we must check $\tau \in \mathcal{L}(\mathcal{F}[f \mapsto i])$ for $i \in \{1,2,3\}$. To do this we must look at the general language of each instantiated SEA.

For each binding $[f \mapsto 1]$, $[f \mapsto 2]$ and $[f \mapsto 3]$ we need to compute the projections of τ. This simply involves removing events that do not refer to the value of interest, as described in Def. 6. In this case we get the following projections:

$$\begin{aligned}\tau\downarrow_{\mathcal{F}([f\mapsto 1])} &= \texttt{open}(1).\texttt{read}(1).\texttt{read}(1).\texttt{write}(1).\texttt{save}(1).\texttt{close}(1)\\ \tau\downarrow_{\mathcal{F}([f\mapsto 2])} &= \texttt{open}(2).\texttt{write}(2).\texttt{save}(2).\texttt{read}(2).\texttt{write}(2).\texttt{close}(2)\\ \tau\downarrow_{\mathcal{F}([f\mapsto 3])} &= \texttt{open}(3).\texttt{read}(3).\texttt{close}(3)\end{aligned}$$

The next question is whether these projected traces have a sequence of transitions ending in a final state. Let us consider $[f \mapsto 1]$ first. Firstly, $\mathcal{F}([f \mapsto 1])$ is constructed by replacing instances of f in $\mathcal{F}$ with 1. Then, for a trace to be in the language of $\mathcal{F}([f \mapsto 1])$ there needs to be a sequence of transitions using $\rightarrow_{\mathcal{F}([f\mapsto 1])}$ that ends in a final state. We find the following sequence of transitions that achieves this:

$$\begin{aligned}1 &\xrightarrow{\texttt{open}(1)}_{\mathcal{F}([f\mapsto 1])} 2 \xrightarrow{\texttt{read}(1)}_{\mathcal{F}([f\mapsto 1])} 2 \xrightarrow{\texttt{read}(1)}_{\mathcal{F}([f\mapsto 1])} 2 \xrightarrow{\texttt{write}(1)}_{\mathcal{F}([f\mapsto 1])} 3\\ &\xrightarrow{\texttt{save}(1)}_{\mathcal{F}([f\mapsto 1])} 4 \xrightarrow{\texttt{close}(1)}_{\mathcal{F}([f\mapsto 1])} 1\end{aligned}$$

Therefore $\tau\downarrow_{\mathcal{F}([f\mapsto 1])} \in \mathcal{L}(\mathcal{F}([f \mapsto 1]))$. Next let us consider $[f \mapsto 2]$. As $\mathcal{F}$ is deterministic there is only one sequence of transitions for a given trace - for our projected trace this is as follows:

$$\begin{aligned}1 &\xrightarrow{\texttt{open}(2)}_{\mathcal{F}([f\mapsto 2])} 2 \xrightarrow{\texttt{write}(2)}_{\mathcal{F}([f\mapsto 2])} 3 \xrightarrow{\texttt{save}(2)}_{\mathcal{F}([f\mapsto 2])} 2 \xrightarrow{\texttt{read}(2)}_{\mathcal{F}([f\mapsto 2])} 2\\ &\xrightarrow{\texttt{write}(2)}_{\mathcal{F}([f\mapsto 2])} 3 \xrightarrow{\texttt{close}(2)}_{\mathcal{F}([f\mapsto 2])} 5\end{aligned}$$

As this does not finish in a final state we have that $\tau\downarrow_{\mathcal{F}([f\mapsto 2])} \notin \mathcal{L}(\mathcal{F}([f \mapsto 2]))$ and therefore our original trace τ is not correct. For completeness we can see that the projected trace for

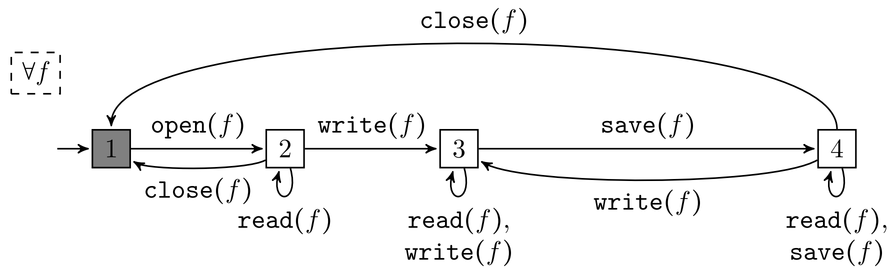


Figure A.2: SQSEA for proper usage of files using next states

$[f \mapsto 3]$ is accepted using this sequence of transitions:

$$1 \xrightarrow{\texttt{open(3)}}_{\mathcal{F}([f\mapsto 3])} 2 \xrightarrow{\texttt{read(3)}}_{\mathcal{F}([f\mapsto 3])} 2 \xrightarrow{\texttt{close(3)}}_{\mathcal{F}([f\mapsto 3])} 1$$

The trace τ is not accepted as the file with identifier 2 is written to and then closed without being saved.

Earlier we noted that it can be useful to use a next-style semantics and that this would be introduced using the graphical notation of a rectangular state. The SQSEA in Fig. A.2 gives an equivalent SQSEA for the simple file usage property using next states - note that this requires us to introduce some looping transitions for when the previous SQSEA would stay in the same state. This alternative way of representing SQSEA can be more concise.

A.1.2 EA

EA are defined in Section 3.4. We give a worked example for SQL Injection and two further, short examples.

SQL Injection

This example demonstrates how the non-determinism of EA can be used, along with free variables, to capture a property about connectedness in time.

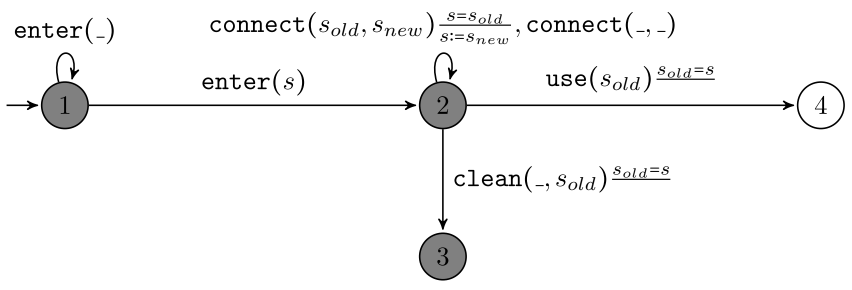
The setting is that of SQL injection. Consider a program that runs the following SQL script given a user input $EMAIL.

```
SELECT fieldlist
FROM table
WHERE field ='$EMAIL';
```

This is open to an SQL injection attack if the input is not sanitized - consider an input[1] of "sometext' OR 'x'='x". This leads to the following command being executed.

```
SELECT fieldlist
FROM table
WHERE field ='sometext' OR 'x'='x';
```

[1]Or perhaps consider the input "x'; DROP TABLE table; –" as a more dangerous example.



enter(_)
connect(s_{old}, s_{new}) $\frac{s=s_{old}}{s:=s_{new}}$, connect(_, _)
enter(s)
use(s_{old}) $\frac{s_{old}=s}{}$
clean(_, s_{old}) $\frac{s_{old}=s}{}$
1
2
3
4

Figure A.3: An EA capturing an SQL injection property that makes use of non-determinism.

The WHERE clause will now always evaluate to true, giving access to all fields and not just the fields associated with a given email address. The normal defence against this kind of attack is to pass all user inputs through a `clean` function that sanitizes inputs by removing problematic characters. Therefore, a property we want to check is that every user input passes through this function before being used. Let us consider a system that produces the following trace where four different kinds of events are monitored - those that introduce a string through user input (`enter`), those that create a new string from an old (`connect`), those that sanitize a string (`clean`) and those that use a string (`use`).

`enter`(“xxx' OR $'x' = 'x$ ”).
`connect`(“xxx' OR $'x' = 'x$”, “xxx' or $'x' = 'x$”).
`enter`(“$REGERG@cs.man.ac.uk$”).
`connect`(“$REGERG@cs.man.ac.uk$”, “$regerg@cs.man.ac.uk$”).
`clean`(“$regerg@cs.man.ac.uk$”, “$regerg@cs.man.ac.uk$”).
`use`(“$REGERG@cs.man.ac.uk$”).
`use`(“xxx' OR $'x' = 'x$”)

Two strings are entered into the system and both are passed through a lowercasing function (here noted with `connect`) producing new strings. Neither string that is used is previously cleaned; the system evidently having some control path that avoids this step. Note that by passing through the lowercasing function we are not tracking the original string, but strings created from them.

An event automaton for this property is given in Fig. A.3. Let us consider how the above trace is processed by this EA. We begin with the initial configuration, as usual:

$$\left\{ \ \langle 1, [\ ] \rangle \ \right\}$$

Then after receiving `enter`(“xxx' OR $'x' = 'x$ ”) we match two transitions from state 1. The first returns to state 1, we use the notation _ to represent a variable that can be ignored, ensuring that the initial configuration is retained. The second takes us to state 2, binding the entered string.

$$\left\{ \begin{array}{l} \langle 1, [\ ] \rangle, \\ \langle 2, [s \mapsto \text{“}xxx' \ OR \ 'x' = 'x \text{ ”}] \rangle \end{array} \right\}$$

The next event, `connect`(“xxx' OR $'x'$ $='$ x”, “xxx' or $'x'$ $='$ x”) also matches two transitions. The first, again, retains the configuration and the second binds the new value to s through the use of two auxiliary variables.

$$\left\{\begin{array}{l}\langle 1, [\,]\rangle,\\ \langle 2, [s \mapsto \text{“}xxx'\ OR\ 'x'\ ='\ x\text{”}]\rangle\\ \left\langle 2, \left[\begin{array}{lcl} s_{old} & \mapsto & \text{“}xxx'\ OR\ 'x'\ ='\ x\text{”}]\\ s_{new} & \mapsto & \text{“}xxx'\ or\ 'x'\ ='\ x\text{”}]\\ s & \mapsto & \text{“}xxx'\ or\ 'x'\ ='\ x\ \text{”}]\end{array}\right]\right\rangle\end{array}\right\}$$

As s_{old} and s_{new} are only used as auxiliary variables we will omit them from now on. The next event, `enter`(“*REGERG@cs.man.ac.uk*”), introduces a new string and the following event, `connect`(“*REGERG@cs.man.ac.uk*”, “*regerg@cs.man.ac.uk*”), introduces a string created from this. Again, self-loops are used to retain the starting configurations whilst introducing new ones.

$$\left\{\begin{array}{l}\langle 1, [\,]\rangle,\\ \langle 2, [s \mapsto \text{“}xxx'\ OR\ 'x'\ ='\ x\ \text{”}]\rangle\\ \langle 2, [s \mapsto \text{“}xxx'\ or\ 'x'\ ='\ x\ \text{”}]\rangle\\ \langle 2, [s \mapsto \text{“}REGERG@cs.man.ac.uk\text{”}]\rangle\\ \langle 2, [s \mapsto \text{“}regerg@cs.man.ac.uk\text{”}]\rangle\end{array}\right\}$$

The next event, `clean`(“*regerg@cs.man.ac.uk*”, “*regerg@cs.man.ac.uk*”) causes the configuration for that string to be moved to state 3. As this is a skip state any future usage of this string would keep it in this accepting state.

$$\left\{\begin{array}{l}\langle 1, [\,]\rangle,\\ \langle 2, [s \mapsto \text{“}xxx'\ OR\ 'x'\ ='\ x\ \text{”}]\rangle\\ \langle 2, [s \mapsto \text{“}xxx'\ or\ 'x'\ ='\ x\ \text{”}]\rangle\\ \langle 2, [s \mapsto \text{“}REGERG@cs.man.ac.uk\text{”}]\rangle\\ \langle 3, [s \mapsto \text{“}regerg@cs.man.ac.uk\text{”}]\rangle\end{array}\right\}$$

The events `use`(“*REGERG@cs.man.ac.uk*”) and `use`(“xxx' OR $'x'$ $='$ x”) then both use a string that has not been cleaned, causing their respective configurations to move to state 4. As this is not accepting the trace fails.

$$\left\{\begin{array}{l}\langle 1, [\,]\rangle,\\ \langle 2, [s \mapsto \text{“}xxx'\ OR\ 'x'\ ='\ x\ \text{”}]\rangle\\ \langle 4, [s \mapsto \text{“}xxx'\ or\ 'x'\ ='\ x\ \text{”}]\rangle\\ \langle 4, [s \mapsto \text{“}REGERG@cs.man.ac.uk\ \ \text{”}]\rangle\\ \langle 3, [s \mapsto \text{“}regerg@cs.man.ac.uk\text{”}]\rangle\end{array}\right\}$$

In this example, we have captured our property entirely without quantification. This demonstrates that there are some properties where the quantification setup is required and some where it is not. In general, quantifications are needed when describing a general property for some set of well-defined objects.

We could have used quantification here and the self-loop on the initial state is effectively what the universal quantification is achieving in SQSEA - every time a new 'quantified' variable is met a new configuration is created. In fact, EA can encode SQSEA, although not necessarily concisely as multiple quantifications can become complex. Additionally, modelling quantification in this way would not allow us to perform the monitoring optimisations discussed in Chapter. 6.

Acknowledging messages in time

Let us revisit the EA described as motivation earlier in Fig. 3.4. Recall that the property is that a message m must be acknowledged with 100 units of time or resent and can only be resent up to 10 times. Note how a is introduced as an auxiliary variable in an assignment rather than through matching with an event - this demonstrates that assignments can be used to keep track of derived values, not just those seen in the trace. This property could be simplified if we assume that the guard/assignment language has access to the current time as we could use this function directly and not need to contain the information in the trace. However, care would need to be taken when monitoring especially if this were offline.

Talking philosophers

The next example is another property that requires only free variables to capture the desired behaviour. Consider a 'talking philosophers' problem (as an allegory for mutual exclusion) where there are a number of philosophers wishing to speak but only a single philosopher may be speaking at any one time. The event automaton in Fig. A.4 captures this property where we have two events $\texttt{start}(p)$ indicates that philosopher p begins speaking and $\texttt{stop}(p)$ indicated that philosopher p stops speaking. The EA states that if philosopher p has begun speaking then the next event that must occur is for philosopher p to stop talking, and then some other (possibly the same) philosopher can start speaking.

Consider the following (incorrect) trace:

$$\tau = \texttt{start}(1).\texttt{stop}(1).\texttt{start}(2).\texttt{start}(1)$$

Let $\mathcal{TP}$ be the EA in Fig. A.4. The (only) transition sequence for this trace is

$$\begin{aligned}
\langle 1, [\ ]\rangle \quad &\xrightarrow{\texttt{start}(1)}_{\mathcal{TP}} \quad \langle 2, [p \mapsto 1]\rangle \\
&\xrightarrow{\texttt{stop}(1)}_{\mathcal{TP}} \quad \langle 1, [p \mapsto 1, q \mapsto 1]\rangle \\
&\xrightarrow{\texttt{start}(2)}_{\mathcal{TP}} \quad \langle 2, [p \mapsto 2, q \mapsto 1]\rangle \\
&\xrightarrow{\texttt{stop}(1)}_{\mathcal{TP}} \quad \langle q_\perp, [p \mapsto 2, q \mapsto 1]\rangle
\end{aligned}$$

as this ends in the implicit failure state $q_\perp$ (which is added when using next-style states) the trace is not in the language of $\mathcal{TP}$. Recall that we assume that the implicit closure for next-style states uses assignments such that the binding remains unchanged - therefore, to record the philosopher that spoke out of turn we would need to add a transition $2 \xrightarrow{\texttt{stop}(q)\ \underline{p \neq q}} 3$ where 3 is a non-final state.

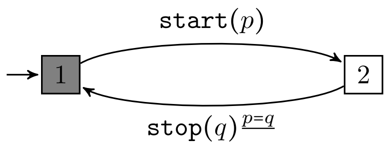


Figure A.4: An EA capturing the property that only one philosopher (p) can be talking at any time.

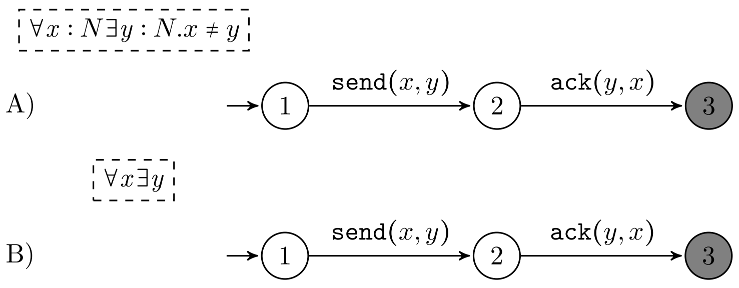


Figure A.5: Two possible QEAs for describing acknowledgements

Note that, as free variables are rebound whenever they are encountered we need the guard on the transition from state 2 to state 1 - without it the EA would accept any trace of alternating `start` and `stop` events regardless of philosopher.

A.1.3 QEA

QEA are introduced in Section 3.5 we give further examples of different kinds of QEA here.

Acknowledgements

The property described here is similar to the motivational property described in Section 3.5.1 and also demonstrates the need for type variables. Let us return to our general scenario where we have nodes (physical or virtual) sending messages to each other. A new property we may wish to specify is that every node sends a message and has it acknowledged i.e. it communicates with another node at some point. This is, of course, a very weak property, but suffices to demonstrate the point at hand.

Fig A.5 gives two QEA that present two different interpretations of this property - let us call them $\mathcal{C}_1 = \langle \forall x : N, \exists y : N . x \neq y, \mathcal{E}_1, [\ ] \rangle$ and $\mathcal{C}_2 = \langle \forall x \exists y, \mathcal{E}_2, [\ ] \rangle$. To understand the difference between these QEA let us consider the following trace:

$$\tau = \texttt{send}(A,B).\texttt{send}(B,A).\texttt{send}(A,C).\texttt{ack}(C,A).\texttt{ack}(A,B)$$

Neither QEA accepts τ. To see why let us consider the domains of x and y for each QEA:

$$\begin{array}{rcl} \mathsf{Dom}^{\mathcal{E}_1}(\tau) & = & [N \mapsto \{A,B,C\}] \\ \mathsf{Dom}^{\mathcal{E}_2}(\tau) & = & [x \mapsto \{A,B\}, y \mapsto \{A,B,C\}] \end{array}$$

For $\mathcal{C}_2$ the node C is not in the domain of x as it never sends and therefore it does not need to have a message acknowledged. This may be what we intended - it is not clear from the informal description above what constitutes a node i.e. whether it is only the sending node or also the nodes that acknowledge messages. Additionally, it is not clear whether a node must send a message to itself - the global guard $x \neq y$ is used in $\mathcal{C}_1$ to enforce this but not in $\mathcal{C}_2$. To see that $\mathcal{C}_1$ does not accept τ note that C never sends a message. To see that $\mathcal{C}_2$ does not accept τ note that no node sends a message to itself.

Here we have seen that an informal description can seem to match a formal specification whilst the specification does not have the intended effect. It is important that descriptions are precise and do not leave behaviours undefined or assumed. It should be noted that the alphabet of a QEA and the type variables can have a large, perhaps expected, impact on the language of the QEA.

Mutual Exclusion

We capture the property of mutual exclusion i.e. that an object may be locked by at most one thread. Fig. A.6 give two equivalent QEA for this property that use different levels of quantification - which will have an effect on how efficiently they can be monitored, as discussed later in Sec. 4.3. Interestingly, the less efficient QEA is a SQSEA - another motivation for introducing guards and assignments.

As we have stated that these QEA are equivalent we should demonstrate this. Let us refer to these two QEA here as $\mathcal{A}$ and $\mathcal{B}$. We want to show that

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$$

and will do so by showing that if a trace is in the language of one then it is in the language of the other. Let us begin by showing that for any trace τ

$$\tau \notin \mathcal{L}(\mathcal{A}) \Rightarrow \tau \notin \mathcal{L}(\mathcal{B}).$$

As $\tau \notin \mathcal{L}(\mathcal{A})$ there must exist an object O and thread T such that $\tau \downarrow_{\theta_1} \notin \mathcal{E}_{\mathcal{A}}(\theta_1)$ where $\theta_1 = [o \mapsto \text{O}, t_1 \mapsto \text{T}]$. The sequence of states passed through by $\tau \downarrow_{\theta_1}$ must end in state 3 and will be of the form $(12)^+3^+$. Let $\mathbf{a}_1.\sigma.\mathbf{a}_2$ be the subtrace of τ covering the transition from state 1 to state 3 passing through state 2 only i.e., passing through states 12^+3. By definition $\mathbf{a}_1$ must match with $\texttt{lock}(o, t_1)$ and $\mathbf{a}_2$ must match with $\texttt{lock}(o, t_2)$ such that $t_1 \neq t_2$ - let $\mathbf{a}_2 = \texttt{lock}(\text{O,S})$. To demonstrate that $\tau \notin \mathcal{L}(\mathcal{B})$ we must find a binding θ_2 such that $\tau \downarrow_{\theta_2} \notin \mathcal{E}_{\mathcal{B}}(\theta_2)$. Let $\theta_2 = [o \mapsto \text{O}, t_1 \mapsto \text{T}, t_2 \mapsto \text{S}]$. $\tau \downarrow_{\theta_2}$ must contain $\mathbf{a}_1.\sigma.\mathbf{a}_2$ as the ground alphabet of $\mathcal{E}_A(\theta_1)$ is a subset of the ground alphabet of $\mathcal{E}_B(\theta_2)$. As $\mathbf{a}_1 = \texttt{lock}(\text{O,T})$, $\mathbf{a_2} = \texttt{lock}(\text{O,S})$ and σ does not contain $\texttt{unlock}(\text{O,T})$ the sequence of transitions must visit the implicit failure state - and therefore τ is not accepted by $\mathcal{B}$.

The argument that

$$\tau \notin \mathcal{L}(\mathcal{B}) \Rightarrow \tau \notin \mathcal{L}(\mathcal{A})$$

follows in a symmetric manner i.e., there must be a binding θ_2 such that $\tau \downarrow_{\theta_2} \notin \mathcal{E}_{\mathcal{B}}(\theta_2)$ from which we can construct a binding θ_1 such that $\tau \downarrow_{\theta_1} \notin \mathcal{E}_{\mathcal{A}}(\theta_1)$. Note that θ_2 must contain

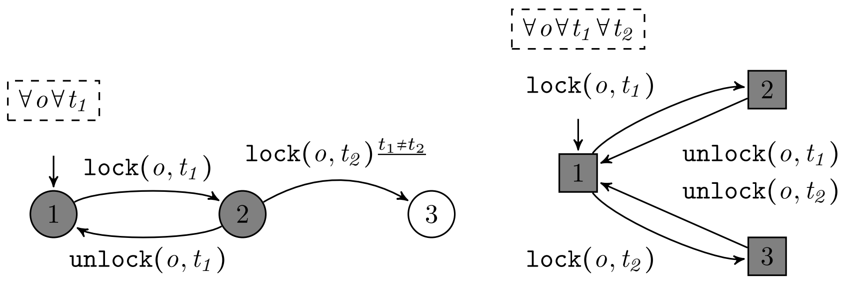


Figure A.6: Two QEA expressing the mutual exclusion property

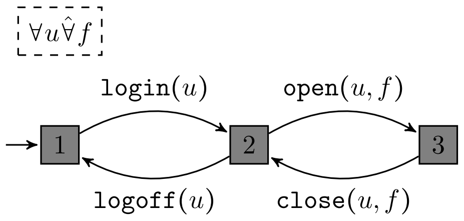


Figure A.7: A user logging in and out and opening and closing files.

different values for t_1 and t_2 as due to the non-determinism of $\mathcal{B}$ in the special case where $t_1 = t_2$ the trace must be accepted.

Another example of partial binding

The following example makes use of this partial quantifier. Consider a property about users and files that states that:

1. A user starts logged off and can only log in if logged off and only log off if logged in,
2. A file starts closed and can only be opened if closed and closed if opened,
3. A user can only open a file if logged in.

The QEA in Fig. A.7 captures this property using a partial quantifier for f. The following trace demonstrates a trace that can violate the property without introducing a value for f.

$$\texttt{login}(\mathit{Giles}).\texttt{login}(\mathit{Giles})$$

Without the partial quantifier this trace would be accepted even though it violates the property we wanted to capture. As the domain of f is empty the partial quantifier ensures a dummy value is added, allowing a binding to be constructed.

A.1.4 Quantifier stripping

The QEA at the top of Fig. A.8 describes a property about channels and messages. The property is that after a channel has been closed we should not send a message to it. The QEAs at the bottom of Fig. A.8 give a simple example of how we can construct a QEA to capture

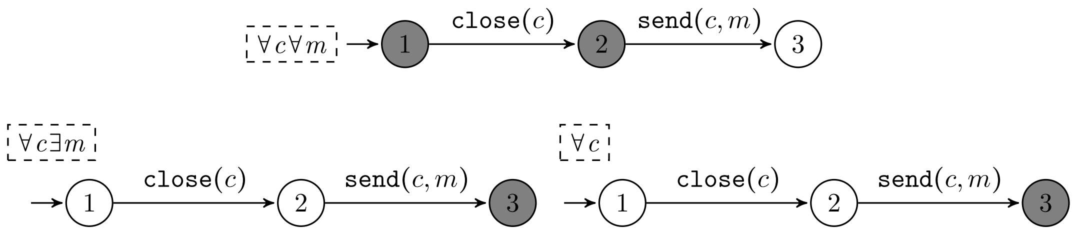


Figure A.8: A simple example of quantifier stripping.

this property using one quantifier - we first take the complement of the QEA (and therefore would monitor for success instead of failure) and then strip the final existential quantification.

A.2 Further examples related to trace checking

In this section we give a number of examples that demonstrate concepts related to trace checking for QEA.

A.2.1 Binding extensions further example

We give an example that demonstrates the binding extensions definition given in Section 5.2.3.

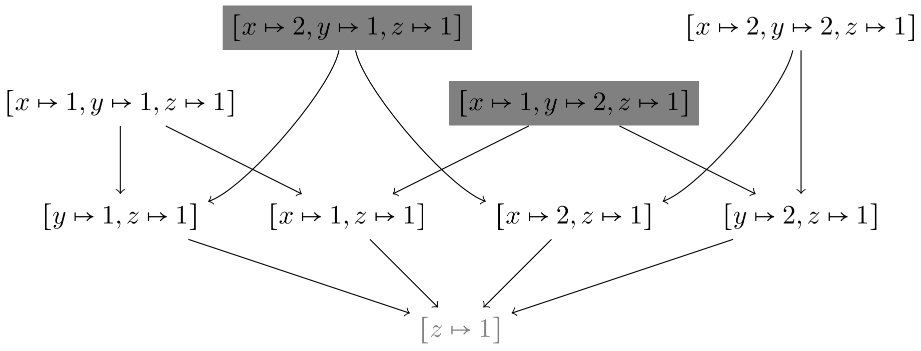
To help us understand what the binding extensions function is doing let us consider an alphabet $\mathcal{A} = \{\mathsf{a}(x,x,y), \mathsf{a}(z,y,x)\}$ where x, y and z are quantified, a ground event $\mathbf{a} = \mathsf{a}(1,1,2)$ and a binding $\theta = [z \mapsto 1]$ and compute $\texttt{extensions}(\theta, \mathbf{a}, \mathcal{A})$. The first thing we need to do is construct $\mathsf{from}(\theta, \mathbf{a}, \mathcal{A})$ by matching $\mathsf{a}(1,1,2)$ with $\mathsf{a}(x,x,y)$ and $\mathsf{a}(z,y,x)$ to get

$$\mathsf{from}(\theta, \mathbf{a}, \mathcal{A}) = \{[x \mapsto 1, y \mapsto 2], [x \mapsto 2, y \mapsto 1, z \mapsto 1]\}$$

We then close this set by first adding all submaps and taking the lub-closure:

$$\begin{aligned}
\mathsf{all\_from}(\theta, \mathbf{a}, \mathcal{A}) &= \mathsf{close}_{\sqcup} \left\{ \begin{array}{l} [x \mapsto 1], [x \mapsto 2], [y \mapsto 1], [y \mapsto 2], [z \mapsto 1], \\ [x \mapsto 1, z \mapsto 1], [y \mapsto 1, z \mapsto 1], [x \mapsto 1, y \mapsto 2], \\ [x \mapsto 2, y \mapsto 1, z \mapsto 1] \end{array} \right\} \\
&= \left\{ \begin{array}{l} [x \mapsto 1], [x \mapsto 2], [y \mapsto 1], [y \mapsto 2], [z \mapsto 1], \\ [x \mapsto 1, z \mapsto 1], [y \mapsto 1, z \mapsto 1], [x \mapsto 1, y \mapsto 2], \\ [x \mapsto 2, y \mapsto 1, z \mapsto 1], [x \mapsto 2, z \mapsto 1], \\ [x \mapsto 1, y \mapsto 1], [y \mapsto 2, z \mapsto 1], [x \mapsto 1, y \mapsto 1, z \mapsto 1], \\ [x \mapsto 2, y \mapsto 2, z \mapsto 1], [x \mapsto 1, y \mapsto 2, z \mapsto 1] \end{array} \right\}
\end{aligned}$$

The set of extensions is the set of bindings we get by adding θ ($[z \mapsto 1]$) to every binding in



$$\begin{tikzcd}
 & & {[x \mapsto 2, y \mapsto 1, z \mapsto 1]} \arrow[ddll] \arrow[ddrr] & & & & {[x \mapsto 2, y \mapsto 2, z \mapsto 1]} \arrow[ddll] \arrow[dd] \\
{[x \mapsto 1, y \mapsto 1, z \mapsto 1]} \arrow[d] \arrow[drr] & & & & {[x \mapsto 1, y \mapsto 2, z \mapsto 1]} \arrow[dll] \arrow[drr] & & \\
{[y \mapsto 1, z \mapsto 1]} \arrow[drrr] & & {[x \mapsto 1, z \mapsto 1]} \arrow[dr] & & {[x \mapsto 2, z \mapsto 1]} \arrow[dl] & & {[y \mapsto 2, z \mapsto 1]} \arrow[dlll] \\
 & & & {[z \mapsto 1]} & & &
\end{tikzcd}$$

Figure A.9: Lattice of binding extensions

$\mathsf{all\_from}(\theta, \mathbf{a}, \mathcal{A})$:

$$\mathsf{extensions}(\theta, \mathbf{a}, \mathcal{A}) = \left\{ \begin{array}{l} [x \mapsto 1, z \mapsto 1], [y \mapsto 1, z \mapsto 1] \\ \;[x \mapsto 2, y \mapsto 1, z \mapsto 1], [x \mapsto 2, z \mapsto 1], \\ \;[y \mapsto 2, z \mapsto 1], [x \mapsto 1, y \mapsto 1, z \mapsto 1], \\ \;[x \mapsto 2, y \mapsto 2, z \mapsto 1], [x \mapsto 1, y \mapsto 2, z \mapsto 1] \end{array} \right\}$$

The subtle part of this definition is how, in $\mathsf{all\_from}$ we take submaps of bindings in from rather than taking the full bindings only. If we had instead defined $\mathsf{all\_from}$ as the lub-closure of from then in our example case we would not have constructed the binding $[x \mapsto 1, y \mapsto 2, z \mapsto 1]$, which is necessary to ensure that no information is lost as the event $\mathbf{a}$ is relevant to this binding:

$$\begin{array}{l} \mathsf{relevant}(\mathsf{a}(1,1,2), [x \mapsto 1, y \mapsto 2, z \mapsto 1], \{\mathsf{a}(x,x,y), \mathsf{a}(z,y,x)\}) \text{ as} \\ \qquad\qquad \mathsf{matches}(\mathsf{a}(1,1,2), \mathsf{a}(x,x,y)) \wedge [x \mapsto 1, y \mapsto 2] \sqsubseteq [x \mapsto 1, y \mapsto 2, z \mapsto 1] \end{array}$$

Sec. 6.4 shows that there are circumstances where we can reduce the number of bindings that need to be created.

A.2.2 Demonstrating small-step semantics

We give a worked example for the small step semantics, as defined in Section 5.3.

As a worked example let us consider the lock ordering property (see page 346). Adjusting this QEA for type variables leaves it the same. Let us consider the following trace

$$\tau = \texttt{lock}(1).\texttt{lock}(2).\texttt{unlock}(2).\texttt{lock}(2).\texttt{unlock}(1).\texttt{lock}(3).\texttt{lock}(1).$$

We will first build up the monitor lookup for each trace prefix and then consider the acceptance of each monitor lookup.

Using τ_i to denote the prefix of τ of length i, we first construct $L_{\tau_1} = \texttt{lock}(1) * [[\ ] \mapsto \{\langle 1, [\ ]\rangle\}]$. As the domain of the initial monitor lookup consists only of $[\ ]$ this collapse to

$$L_{\tau_1} = [\ ] \dagger\ \mathsf{Add}_1 \dagger\ [[\ ] \mapsto \mathsf{next}([\ ], \texttt{lock}(1), \{\langle 1, [\ ]\rangle\})]$$

We will now construct this first monitor lookup by considering every step involved. We will then omit most of these details when constructing the remainder of the monitor lookups.

Firstly, let us compute the extending monitor lookup Add_1 as

$$\mathsf{Add}_1 = [(\theta' \mapsto C') \in \mathsf{extend}(\mathtt{lock}(1), [\ ], \{\langle 1, [\ ]\rangle\}) \mid \theta' \notin \mathsf{dom}([\ ])].$$

To construct this we must first compute $\mathsf{extend}(\mathtt{lock}(1), [\ ], \{\langle 1, [\ ]\rangle\})$, which is given as

$$[\theta' \mapsto \mathsf{next}(\theta', \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\}) \mid \theta' \in \mathsf{extensions}([\ ], \mathtt{lock}(1))]$$

i.e. the mapping of extending bindings to their next configurations. Let us compute the extending bindings

$$\mathsf{extensions}([\ ], \mathtt{lock}(1) = \{\theta \dagger \theta' \mid \theta' \in \mathsf{all\_from}([\ ], \mathtt{lock}(1) \wedge \theta' \neq [\ ]\}$$

This requires us to construct the the closed set of from:

$$\begin{aligned}\mathsf{from}([\ ], \mathtt{lock}(1)) \ &= \{\mathsf{quantified}(\mathsf{match}(\mathtt{lock}(1), \mathbf{b})) \\ &\qquad\qquad \mid \mathbf{b} \in \mathcal{A}([\ ]) \wedge \mathsf{matches}(\mathtt{lock}(1), \mathbf{b})\end{aligned}$$

As

$$\mathcal{A}([\ ]) = \{\mathtt{lock}(l_1), \mathtt{lock}(l_2), \mathtt{unlock}(l_1), \mathtt{unlock}(l_2)\}$$

we can construct the extending bindings as follows:

$$\begin{array}{lcl}\mathsf{from}([\ ], \mathtt{lock}(1)) & = & \{[l_1 \mapsto 1], [l_2 \mapsto 1]\} \\ \mathsf{all\_from}([\ ], \mathtt{lock}(1)) & = & \{[l_1 \mapsto 1], [l_2 \mapsto 1], [l_1 \mapsto 1, l_2 \mapsto 1]\} \\ \mathsf{extensions}([\ ], \mathtt{lock}(1)) & = & \{[l_1 \mapsto 1], [l_2 \mapsto 1], [l_1 \mapsto 1, l_2 \mapsto 1]\}\end{array}$$

Note that the closure of $\mathsf{from}([\ ], \mathtt{lock}(1))$ includes the binding $[l_1 \mapsto 1, l_2 \mapsto 1]$ - this is necessary as without it we would not have every total binding required for deciding acceptance.

The map $\mathsf{extend}(\mathtt{lock}(1), [\ ], \{\langle 1, [\ ]\rangle\}$ is therefore

$$\left[\begin{array}{lcl}[l_1 \mapsto 1] & \mapsto & \mathsf{next}([l_1 \mapsto 1, \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\}) \\ [l_2 \mapsto 1] & \mapsto & \mathsf{next}([l_2 \mapsto 1, \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\}) \\ [l_1 \mapsto 1, l_2 \mapsto 1] & \mapsto & \mathsf{next}([l_1 \mapsto 1, l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\})\end{array}\right]$$

Let us first consider how to construct $\mathsf{next}([l_1 \mapsto 1, l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\})$, this next set of configurations is given by

$$\mathsf{move}([l_1 \mapsto 1, l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\} \cup \mathsf{stay}([l_1 \mapsto 1, l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\}.$$

$$\begin{array}{ll}\mathsf{move}([l_1 \mapsto 1, l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\} = & \{\langle 2, [\ ]\rangle\} \\ \mathsf{stay}([l_1 \mapsto 1, l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\} = & \{\}\end{array}$$

The set $\mathsf{next}([l_1 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\})$ is constructed in a similar fashion. We now consider how to construct $\mathsf{next}([l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\})$, as before we construct the sets of moving and staying configurations.

$$\begin{array}{ll}\mathsf{move}([l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\} = & \{\} \\ \mathsf{stay}([l_2 \mapsto 1], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\} = & \{\langle 1, [\ ]\rangle\}\end{array}$$

After computing next configurations the map $\mathsf{extend}(\mathtt{lock}(1), [\ ], \{\langle 1, [\ ]\rangle\}$, and therefore the map Add_1, is

$$\begin{bmatrix} [l_1 \mapsto 1] & \mapsto & \{\langle 2, [\ ]\rangle\} \\ [l_2 \mapsto 1] & \mapsto & \{\langle 1, [\ ]\rangle\} \\ [l_1 \mapsto 1, l_2 \mapsto 1] & \mapsto & \{\langle 2, [\ ]\rangle\} \end{bmatrix}$$

We must now construct $\mathsf{next}([\ ], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\})$. As the event is not relevant to $[\ ]$ we *stay* in the same configuration.

$$\begin{array}{ll}\mathsf{move}([\ ], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\} = & \{\} \\ \mathsf{stay}([\ ], \mathtt{lock}(1), \{\langle 1, [\ ]\rangle\} = & \{\langle 1, [\ ]\rangle\}\end{array}$$

This finally gives

$$L_{\tau_1} = \begin{bmatrix} [\ ] & \mapsto & \{\langle 1, [\ ]\rangle\} \\ [l_1 \mapsto 1] & \mapsto & \{\langle 2, [\ ]\rangle\} \\ [l_2 \mapsto 1] & \mapsto & \{\langle 1, [\ ]\rangle\} \\ [l_1 \mapsto 1, l_2 \mapsto 1] & \mapsto & \{\langle 2, [\ ]\rangle\} \end{bmatrix}$$

We will now construct the rest of the monitor lookups for τ, but will not take such a detailed approach. The monitor lookup $L_{\tau_2} = \mathtt{lock}(2) * L_{\tau_1}$ is constructed by linearising the domain of L_{τ_1} and building a new monitor lookup. As the domain of L_{τ_1} is partially ordered as follows

$$\begin{tikzcd} & {[l_1 \mapsto 1, l_2 \mapsto 1]} \arrow[dl, dash] \arrow[dr, dash] & \\ {[l_1 \mapsto 1]} \arrow[dr, dash] & & {[l_2 \mapsto 1])} \arrow[dl, dash] \\ & {[\ ]} & \end{tikzcd}$$

there are two possible linerisations and it does not matter which we pick. Let us choose

$$[l_1 \mapsto 1, l_2 \mapsto 1], [l_1 \mapsto 1], [l_2 \mapsto 1], [\ ].$$

and build up a new monitor lookup by extending each binding in turn. The incoming event is not relevant to any of these bindings so we do not consider the value of next as it will not change the configurations - but we do look at binding extensions.

- As $[l_1 \mapsto 1, l_2 \mapsto 1]$ is total there are no binding extensions and $\mathsf{extend}(\mathtt{lock}(2), [l_1 \mapsto 1, l_2 \mapsto 1], \{\langle 2, [\ ]\rangle\})$ is empty.
- There is one binding extension of $[l_1 \mapsto 1]$, which is $[l_1 \mapsto 1, l_2 \mapsto 2]$, we then compute $\mathsf{next}([l_1 \mapsto 1, l_2 \mapsto 2], \mathtt{lock}(2), \{\langle 2, [\ ],\rangle\}) = \{\langle 3, [\ ]\rangle\}$
- There is one binding extension of $[l_1 \mapsto 2]$, which is $[l_1 \mapsto 2, l_2 \mapsto 1]$, we then compute $\mathsf{next}([l_1 \mapsto 2, l_2 \mapsto 1], \mathtt{lock}(2), \{\langle 1, [\ ],\rangle\}) = \{\langle 2, [\ ]\rangle\}$

- There are three binding extensions of [ ] - $[l_1 \mapsto 2]$, $[l_2 \mapsto 2]$ and $[l_1 \mapsto 2, l_2 \mapsto 2]$, each of them having a `next` configuration ending in state 2.

This gives us

$$L_{\tau_2} = \begin{bmatrix} [\,] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_1 \mapsto 1] & \mapsto & \{\langle 2, [\,]\rangle\} \\ [l_2 \mapsto 1] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_1 \mapsto 1, l_2 \mapsto 1] & \mapsto & \{\langle 2, [\,]\rangle\} \\ [l_1 \mapsto 2] & \mapsto & \{\langle 2, [\,]\rangle\} \\ [l_2 \mapsto 2] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_1 \mapsto 2, l_2 \mapsto 2] & \mapsto & \{\langle 2, [\,]\rangle\} \\ [l_1 \mapsto 1, l_2 \mapsto 2] & \mapsto & \{\langle 3, [\,]\rangle\} \\ [l_1 \mapsto 2, l_2 \mapsto 1] & \mapsto & \{\langle 2, [\,]\rangle\} \end{bmatrix}$$

We now compute $L_{\tau_3} = \texttt{unlock}(2) * L_{\tau_2}$. In this case there are no binding extensions but the event is relevant to the bindings that map l_1 or l_2 to 2. Out of these it is only those that map l_1 to 2 that are updated - as they are in state 2 and the `unlock` event takes them back to state 1. This gives us

$$L_{\tau_3} = \begin{bmatrix} [\,] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_1 \mapsto 1] & \mapsto & \{\langle 2, [\,]\rangle\} \\ [l_2 \mapsto 1] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_1 \mapsto 1, l_2 \mapsto 1] & \mapsto & \{\langle 2, [\,]\rangle\} \\ [l_1 \mapsto 2] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_2 \mapsto 2] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_1 \mapsto 2, l_2 \mapsto 2] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [l_1 \mapsto 1, l_2 \mapsto 2] & \mapsto & \{\langle 3, [\,]\rangle\} \\ [l_1 \mapsto 2, l_2 \mapsto 1] & \mapsto & \{\langle 1, [\,]\rangle\} \end{bmatrix}$$

The last four monitoring states are computed in a similar way to give the following configurations for bindings. Note that when we introduce a third lock the binding extensions for the first two locks extend existing configurations, for example when introducing $[l_1 \mapsto 2, l_2 \mapsto 3]$ the configurations for $[l_1 \mapsto 2]$ are used, in this case $\{\langle 2, [\,]\rangle\}$.

Binding	L_{τ_4} `lock(2)` $* L_{\tau_3}$	L_{τ_5} `unlock(1)` $* L_{\tau_4}$	L_{τ_6} `lock(3)` $* L_{\tau_5}$	L_{τ_7} `lock(1)` $* L_{\tau_6}$
$[\ ]$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$
$[l_1 \mapsto 1]$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$
$[l_2 \mapsto 1]$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$
$[l_1 \mapsto 1, l_2 \mapsto 1]$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$
$[l_1 \mapsto 2]$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$
$[l_2 \mapsto 2]$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$
$[l_1 \mapsto 2, l_2 \mapsto 2]$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$
$[l_1 \mapsto 1, l_2 \mapsto 2]$	$\{\langle 4,[\ ]\rangle\}$	$\{\langle 4,[\ ]\rangle\}$	$\{\langle 4,[\ ]\rangle\}$	$\{\langle 5,[\ ]\rangle\}$
$[l_1 \mapsto 2, l_2 \mapsto 1]$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 3,[\ ]\rangle\}$
$[l_1 \mapsto 3]$	-	-	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$
$[l_2 \mapsto 3]$	-	-	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 1,[\ ]\rangle\}$
$[l_1 \mapsto 3, l_2 \mapsto 3]$	-	-	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$
$[l_1 \mapsto 1, l_2 \mapsto 3]$	-	-	$\{\langle 1,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$
$[l_1 \mapsto 3, l_2 \mapsto 1]$	-	-	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 3,[\ ]\rangle\}$
$[l_1 \mapsto 2, l_2 \mapsto 3]$	-	-	$\{\langle 3,[\ ]\rangle\}$	$\{\langle 3,[\ ]\rangle\}$
$[l_1 \mapsto 3, l_2 \mapsto 2]$	-	-	$\{\langle 2,[\ ]\rangle\}$	$\{\langle 2,[\ ]\rangle\}$

Let us now consider the acceptance of monitor lookups L_{τ_1} and L_{τ_7} - the process is the same for the lookups in-between. Firstly we are asking if

$$L_{\tau_1} \vDash_{[\ ]} (\forall l_1 : true)(\forall l_2 : l_1 \neq l_2)$$

to decide this we must construct $D_{L_{\tau_1}} = [l_1 \mapsto \{1\}, l_2 \mapsto \{1\}]$. We can then unfold Def. 43 to read

$$\textit{for all } d \textit{ in } \{1\} \textit{ if } (true)([\ ]\dagger[l_1 \mapsto d]) \textit{ then } L_{\tau_1} \vDash_{[l_1 \mapsto 1]} (\forall l_2 : l_1 \neq l_2)$$

which becomes $L_{\tau_1} \vDash_{[l_1 \mapsto 1]} (\forall l_2 : l_1 \neq l_2)$, this itself unfolds to read

$$\textit{for all } d \textit{ in } \{1\} \textit{ if } (l_1 \neq l_2)([l_1 \mapsto 1]\dagger[l_2 \mapsto d]) \textit{ then } L_{\tau_1} \vDash_{[l_1 \mapsto 1]} (\forall l_2 : l_1 \neq l_2)$$

and as the guard is false the universal quantification is empty and therefore true and the verdict returned is true.

Now let us ask

$$L_{\tau_7} \vDash_{[\ ]} (\forall l_1 : true)(\forall l_2 : l_1 \neq l_2)$$

again we construct $D_{L_{\tau_7}} = [l_1 \mapsto \{1,2,3\}, l_2 \mapsto \{1,2,3\}]$ and this time we unfold Def. 43 once so that our statement is true iff all of the below hold:

$$\begin{array}{l} L_{\tau_7} \vDash_{[l_1 \mapsto 1]} (\forall l_2 : l_1 \neq l_2) \\ L_{\tau_7} \vDash_{[l_1 \mapsto 2]} (\forall l_2 : l_1 \neq l_2) \\ L_{\tau_7} \vDash_{[l_1 \mapsto 3]} (\forall l_2 : l_1 \neq l_2) \end{array}$$

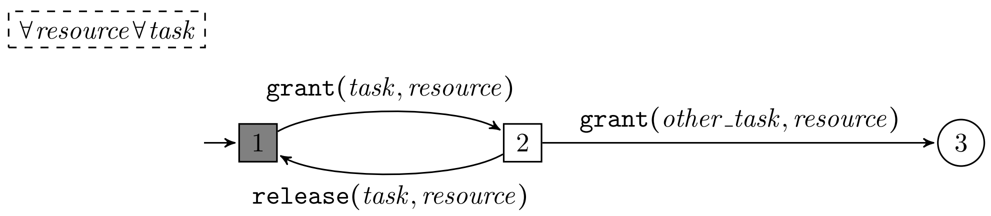


Figure A.10: QEA specifying that a task is granted a resource must release that resource and no other task may be granted the resource before this.

which can be expanded to become the following:

$$\begin{array}{ll}
L_{\tau_7} \models_{[l_1\mapsto 1, l_2\mapsto 2]} \epsilon & \text{iff } \exists\langle q,\varphi\rangle \in L_{\tau_7}([l_1 \mapsto 1, l_2 \mapsto 2]) = \{\langle 5, [\ ]\rangle\} : q \in \{1,2,3,4\} \\
L_{\tau_7} \models_{[l_1\mapsto 1, l_2\mapsto 3]} \epsilon & \text{iff } \exists\langle q,\varphi\rangle \in L_{\tau_7}([l_1 \mapsto 1, l_2 \mapsto 3]) = \{\langle 2, [\ ]\rangle\} : q \in \{1,2,3,4\} \\
L_{\tau_7} \models_{[l_1\mapsto 2, l_2\mapsto 1]} \epsilon & \text{iff } \exists\langle q,\varphi\rangle \in L_{\tau_7}([l_1 \mapsto 2, l_2 \mapsto 1]) = \{\langle 3, [\ ]\rangle\} : q \in \{1,2,3,4\} \\
L_{\tau_7} \models_{[l_1\mapsto 2, l_2\mapsto 3]} \epsilon & \text{iff } \exists\langle q,\varphi\rangle \in L_{\tau_7}([l_1 \mapsto 2, l_2 \mapsto 3]) = \{\langle 3, [\ ]\rangle\} : q \in \{1,2,3,4\} \\
L_{\tau_7} \models_{[l_1\mapsto 3, l_2\mapsto 1]} \epsilon & \text{iff } \exists\langle q,\varphi\rangle \in L_{\tau_7}([l_1 \mapsto 3, l_2 \mapsto 1]) = \{\langle 3, [\ ]\rangle\} : q \in \{1,2,3,4\} \\
L_{\tau_7} \models_{[l_1\mapsto 3, l_2\mapsto 2]} \epsilon & \text{iff } \exists\langle q,\varphi\rangle \in L_{\tau_7}([l_1 \mapsto 3, l_2 \mapsto 2]) = \{\langle 2, [\ ]\rangle\} : q \in \{1,2,3,4\}
\end{array}$$

All of which but the first are true, but as the first is false the verdict is false as the universal quantification requires this to hold for all total bindings.

A.2.3 A worked example for the basic algorithm

We demonstrate the application of the basic algorithm (defined in Setion 5.6) to a QEA and a trace.

Let us use the GrantRelease rover property, which we will refer to as $\mathcal{GR}$, introduced in Sec. A.4 and reproduced in Fig. A.10. Consider the following (incorrect) trace

$$\texttt{grant}(1, A).\texttt{grant}(2, B).\texttt{grant}(1, A).\texttt{grant}(1, B)$$

We will begin by considering steps in detail, but will focus only on the interesting points as we go on.

The first step is to run INIT in Algorithm 2 that will store the QEA and initialise the monitor lookup L to be

$$L = \left[\ [\ ] \ \mapsto\ \{\langle 1, [\ ]\rangle\}\ \right].$$

Next is a call of STEP(grant(1,A)), which leads to a call of UPDATE(grant(1,A)) in Algorithm 4. The first step of this function is a call of MATCHING(grant(1,A)), which computes the closed set of bindings that match grant(1,A) with the alphabet of $\mathcal{GR}$, giving

$$\text{B} = \{[resource \mapsto A], [task \mapsto 1], [task \mapsto 1, resource \mapsto A]\}.$$

Next we step through the domain of L, which currently only contains the empty binding, and attempt to extend it with each binding in B and add it to L, using NEXT to compute the appropriate configurations. This leads to the following monitor lookup.

$$L = \begin{bmatrix} [\ ] & \mapsto & \{\langle 1, [\ ]\rangle\} \\ [\mathit{resource} \mapsto A] & \mapsto & \{\langle 1, [\ ]\rangle\} \\ [\mathit{task} \mapsto 1] & \mapsto & \{\langle 1, [\ ]\rangle\} \\ [\mathit{task} \mapsto 1, \mathit{resource} \mapsto A] & \mapsto & \{\langle 2, [\ ]\rangle\} \end{bmatrix}$$

Let us briefly consider how $\textsc{Next}([\mathit{task} \mapsto 1, \mathit{resource} \mapsto A], \mathtt{grant}(1, A), \{\langle 1, [\ ]\rangle\})$ computed the configuration $\{\langle 2, [\ ]\rangle\}$. This concerns the $\textsc{Next}$ function in Algorithm 1 on page 100.

First the set next is initialised to the empty set, then we consider each configuration in $C = \{\langle 1, [\ ]\rangle\}$ in turn - here there is only one configuration. Then we consider each transition in $\mathcal{GR}$ that begins at the relevant state. For state 1 there are two such transitions - the explicit $(1, \mathtt{grant}(\mathit{task}, \mathit{resource}), \mathit{true}, \mathit{id}, 2)$ and the implicit $(1, \mathtt{release}(\mathit{task}, \mathit{resource}), \mathit{true}, \mathit{id}, q_\perp)$. Only the event of the first transition matches, and as we apply $[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]$ as a substitution to $\mathtt{grant}(\mathit{task}, \mathit{resource})$ the binding φ' is empty. As $\mathsf{quantified}(\varphi') = [\ ]$ and the true guard is always true we add $\langle 2, [\ ]\rangle$ to the next set. As all transitions are considered, and a transition has been taken, this is the set of configurations we return.

Now let us consider what happens when we apply $\textsc{Next}$ in a case where the event is not relevant to the binding, for example $\textsc{Next}([\mathit{task} \mapsto 1],\ \mathtt{grant}(1, A),\ \{\langle 1, [\ ]\rangle\})$. We follow the same process as for above but in this case we only apply the substitution $[\mathit{task} \mapsto 1]$ to $\mathtt{grant}(\mathit{task}, \mathit{resource})$, making $\varphi' = [\mathit{resource} \mapsto A]$. In this case $\mathsf{quantified}(\varphi') = [\mathit{resource} \mapsto A]$ and no transitions are taken and therefore the configuration $\langle q, [\ ]\rangle$ is added back into the next set.

We then run the Check function. This first computes the domain to be $[\mathit{task} \mapsto \{1\}, \mathit{resource} \mapsto \{A\}]$ and then calls $\textsc{Checking}(\forall \mathit{resource}, \forall \mathit{task}, [\ ])$. This produces two recursive calls of $\textsc{Checking}(\forall \mathit{task}, [\mathit{resource} \mapsto A])$ and $\textsc{Checking}(\epsilon,\ [\mathit{task} \mapsto 1, \mathit{resource} \mapsto A])$ before returning False as state 2 is not an accepting state.

This completes the processing of the first event by returning False. We then process the next event with a call of $\textsc{Step}(\mathtt{grant}(2,\mathrm{B}))$. This proceeds in a similar way as with the first event by first computing

$$\mathrm{B} = \{[\mathit{resource} \mapsto B], [\mathit{task} \mapsto 2], [\mathit{task} \mapsto 2, \mathit{resource} \mapsto B]\}$$

and then stepping through the bindings of L from biggest to smallest again. This time there are four bindings in L, that lead to the following values for B′ - note that if we had iterated over B directly then we would have produced these bindings multiple times.

$\theta \in \mathsf{dom}(L)$	B′
$[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]$	$\{\}$
$[\mathit{task} \mapsto 1]$	$\{[\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]\}$
$[\mathit{resource} \mapsto A]$	$\{[\mathit{task} \mapsto 2, \mathit{resource} \mapsto A]\}$
$[\ ]$	$\{[\mathit{resource} \mapsto B], [\mathit{task} \mapsto 2], [\mathit{task} \mapsto 2, \mathit{resource} \mapsto B]\}$

The $\textsc{Next}$ function is called appropriately, i.e. with the configurations of the binding that is being extended, to add the following entries to L.

$$\begin{bmatrix} [\mathit{resource} \mapsto B] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [\mathit{task} \mapsto 2] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [\mathit{task} \mapsto 2, \mathit{resource} \mapsto B] & \mapsto & \{\langle 2, [\,]\rangle\} \\ [\mathit{task} \mapsto 1, \mathit{resource} \mapsto B] & \mapsto & \{\langle 1, [\,]\rangle\} \\ [\mathit{task} \mapsto 2, \mathit{resource} \mapsto A] & \mapsto & \{\langle 1, [\,]\rangle\} \end{bmatrix}$$

The CHECK function is called to give a verdict of False again as of the four total bindings only two are in an accepting state.

The next event is then processed with a call of STEP(`grant`$(1, A)$). As with the first event, we have

$$\mathrm{B} = \{[\mathit{resource} \mapsto A], [\mathit{task} \mapsto 1], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]\}$$

and the we get the following B′ when iterating over the domain of L:

$\theta \in \mathsf{dom}(L)$	B′
$[\mathit{task} \mapsto 2, \mathit{resource} \mapsto B]$	$\{\}$
$[\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]$	$\{[\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]\}$
$[\mathit{task} \mapsto 2, \mathit{resource} \mapsto A]$	$\{[\mathit{task} \mapsto 2, \mathit{resource} \mapsto A\}$
$[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]$	$\{[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]\}$
$[\mathit{task} \mapsto 1]$	$\{[\mathit{task} \mapsto 1], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]\}$
$[\mathit{task} \mapsto 2]$	$\{\}$
$[\mathit{resource} \mapsto A]$	$\{[\mathit{resource} \mapsto A], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]\}$
$[\mathit{resource} \mapsto B]$	$\{\}$
$[\,]$	$\{[\mathit{resource} \mapsto A], [\mathit{task} \mapsto 1], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]\}$

Only existing bindings are generated but we see that the binding $[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]$ is generated many times. However, as it is an existing binding we only update its entry when we are considering that existing binding when iterating over the domain of L.

There is only one interesting call to NEXT here - the call NEXT($[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]$, `grant`$(1, A)$, $\{\langle 2, [\,]\rangle\}$). Two of the outgoing transitions of state 2 match the event `grant`$(1, A)$ leading to the configurations $\langle q_\bot, [\,]\rangle$ and $\langle 3, [\mathit{other\_task} \mapsto 1]\rangle$ being returned. After processing this event only one entry in L is updated, that is

$$\begin{bmatrix} [\mathit{task} \mapsto 1, \mathit{resource} \mapsto A] & \mapsto & \{\langle q_\bot, [\,]\rangle, \langle 3, [\mathit{other\_task} \mapsto 1]\rangle\} \end{bmatrix}$$

The final event is processed with a call to STEP(`grant`$(1, B)$). This leads to the matching bindings

$$\mathrm{B} = \{[\mathit{resource} \mapsto B], [\mathit{task} \mapsto 1], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]\}$$

and the following extending bindings:

$\theta \in \mathsf{dom}(L)$	B$'$
$[\mathit{task} \mapsto 2, \mathit{resource} \mapsto B]$	$\{[\mathit{task} \mapsto 2, \mathit{resource} \mapsto B]\}$
$[\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]$	$\{[\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]\}$
$[\mathit{task} \mapsto 2, \mathit{resource} \mapsto A]$	$\{\}$
$[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A]$	$\{[\mathit{task} \mapsto 1, \mathit{resource} \mapsto A\}$
$[\mathit{task} \mapsto 1]$	$\{\}$
$[\mathit{task} \mapsto 2]$	$\{[\mathit{task} \mapsto 1], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]\}$
$[\mathit{resource} \mapsto A]$	$\{\}$
$[\mathit{resource} \mapsto B]$	$\{[\mathit{resource} \mapsto B], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]\}$
$[\ ]$	$\{[\mathit{resource} \mapsto B], [\mathit{task} \mapsto 1], [\mathit{task} \mapsto 1, \mathit{resource} \mapsto B]\}$

Now the interesting case is $\textsc{Next}([\mathit{task} \mapsto 2, \mathit{resource} \mapsto B], \texttt{grant}(1, B), \{\langle 2, [\ ]\rangle\})$ as this is a case where task 1 is granted a resource currently held by task 2. Consequently we make the following update to L:

$$\Big[\quad [\mathit{task} \mapsto 2, \mathit{resource} \mapsto B] \quad \mapsto \quad \{\langle 3, [\mathit{other\_task} \mapsto 1]\rangle\} \quad \Big]$$

The final call of Check will now return False as when it creates the binding $[\mathit{task} \mapsto 2, \mathit{resource} \mapsto B]$ in Checking it will find that there are no configurations in $M([\mathit{task} \mapsto 2, \mathit{resource} \mapsto B])$ with an accepting state.

A.2.4 Demonstrating incremental checking

To demonstrate how the incremental checking process (introduced in Section 6.2) works consider the quantifier list $\forall x.x > 0, \forall y, \exists z$ - the QEA it is attached to does not matter. Let us start in the situation where the domain is $[x \mapsto \{1,2,3\}, y \mapsto \{4,5,6\}, z \mapsto \{7,8\}]$ and enough bindings accepting for the trace to be accepting. A checking structure for this situation appears at the top of Figure A.11, let us call this Γ_1.

Now let us update the binding $\theta_1 = [x \mapsto 1, y \mapsto 4, z \mapsto 7]$ with a set of states S_1 that is not accepting. We evaluate $\textsc{update}(\Gamma_1, 0, \theta_1, S_1)$. The first two checks fail as $0 \neq 3$ and $1 > 0$ so we extract the leftmost subtree of Γ_1 as Γ_2 and recursively call $\textsc{update}(\Gamma_2, 1, \theta_1, S_1)$. We repeat this process twice to label the leftmost subtree of Γ_2 as Γ_3 and recursively call $\textsc{update}(\Gamma_3, 2, \theta_1, S_1)$ then $\textsc{update}(\mathit{Leaf}\text{ of }\mathit{true}, 3, \theta_1, S_1)$. Now as we have reached the bottom of the tree we rebuild the checking state by first returning *Leaf* of *false*. At the previous level ($\exists z$) we add this leaf to the map appropriately and return it in a new node with *false* as there are now no subtree with a true value. This continues back up the Γ_1 to produce a checking structure identical to Γ_1, except with the highlighted result (in Fig. A.11) changed to $\bot$.

Now consider adding the binding $\theta_2 = [x \mapsto 0, y \mapsto 4, z \mapsto 7]$. As the global guard $x > 0$ does not hold we do not update the checking structure and the binding θ_2 is ignored - this is okay as this is what would happen in the normal evaluation of the acceptance relation.

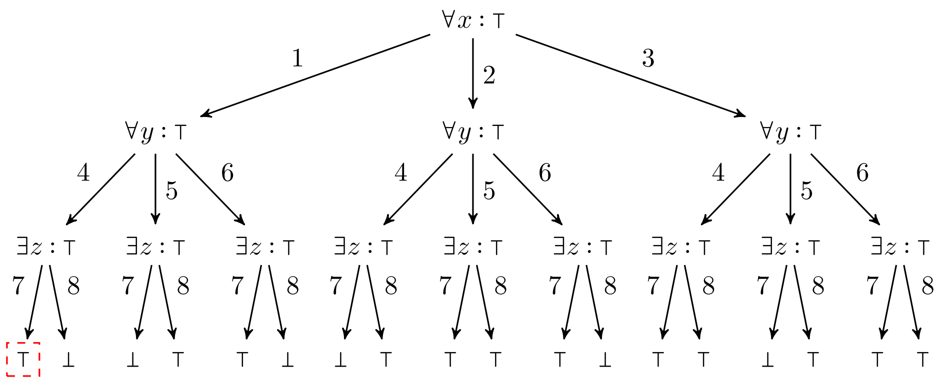


Figure A.11: An example of an incremental checking data structure.

A.3 Java Library Properties

Here we present QEAs describing properties of the `Java` standard library i.e. ordering constraints on method calls found in the library. In some cases we will use method names as events directly and in others we will map sets of method names to a single event name to make specifications more concise.

Most of these properties have been described within the context of runtime verification elsewhere, and later we will make a comparison with these alternative specification languages.

We call this class of properties API properties as they describe the correct usage of an API .This is a useful class of properties as it reflects properties that can be used without full knowledge of the program using the API.

A.3.1 Communicating

Let us first consider properties about classes in `java.io` and `java.net` i.e., those used for communication via writing to and reading from files or sockets.

Reader and Writer.

This is a very straight-forward property related to the usage of `java.io.Writer` or `java.io.Reader` - or any class that subclass them. The property is that a writer or reader should not be used after it has been closed. To specify this property we map the following events to methods of `Writer` and `Reader`.

Event	Methods
`new`(f)	`Writer.new(..)` and `Reader+.new(..)`
`use`(f)	`Writer.write(..)`, `Writer+.flush()`, `Reader.read()`
`close`(f)	`Writer.close(..)` and `Reader.close(..)`
`garbage`(f)	`Writer.finalize(..)` and `Reader.finalize(..)`

The QEA for this property is given in Fig. A.12. Note that a writer or reader cannot be reused once closed. In this QEA we have included garbage collection of the object as an event (but do not later). This leads to an accepting skip state with no outgoing transitions, which can be

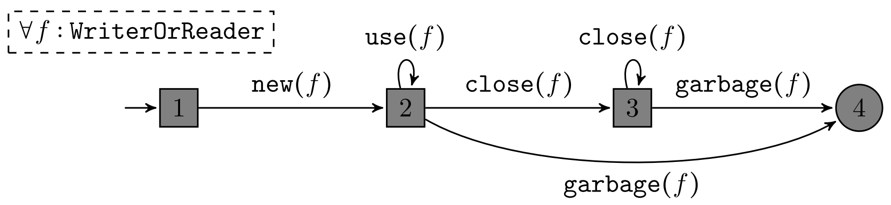


Figure A.12: QEA for proper usage of `java.io.Writer` and `java.io.Reader` (SafeWriter-Reader)

thought of as a special state that indicates that failure cannot occur. Later (Sec 6.6) we will discuss an optimization that will allow us to 'forget' about bindings when they reach such a state - tools monitoring `Java` objects generally do this implicitly.

Socket Usage.

This property is about using a socket, or any input/output streams connected to it, after it has been closed. Again, we begin by relating methods in the `Java` standard library to events.

Event	Methods	Description
`output`(s, o)	`SocketImpl.getOutputStream`	Creating an `OutputStream` o from a `SocketImpl` s
`close`(s)	`SocketImpl.close` and `SocketImpl.shutdownOutput`	Closing the socket s or indicating we should close its connected `OutputStream`
`use`(o)	`OutputStream.*`	Use `OutputStream` o
`connect`(s)	`SocketImpl.connect` and `SocketImpl.bind`	Connects or binds a socket to an address

There are two properties we want to capture - that an output stream connected to a socket is not used after the socket is closed and that a socket is not connected to or bound to an address after it is closed. Fig A.13 gives a QEA for this property. There is a symmetric property for `InputStream` that does not relate the method `shutdownOutput` with the `close` event. Note the use of a partial quantifier for the output stream as we want to detect a failure even if a socket is not connected to an output stream. Note also that if the initial state were not a skip state then an unconnected socket and output stream could lead to failure - consider the following trace.

$$\texttt{use}(M).\texttt{connect}(A, N).\texttt{use}(N).\texttt{close}(A)$$

According to the quantifiers, the binding $[s \mapsto A, o \mapsto M]$ must be considered. The demonstrates that care should be taken when using multiple quantifications to ensure that the correct relation between them is captured. An alternative, and preferred, approach would be to use the connected global guard.

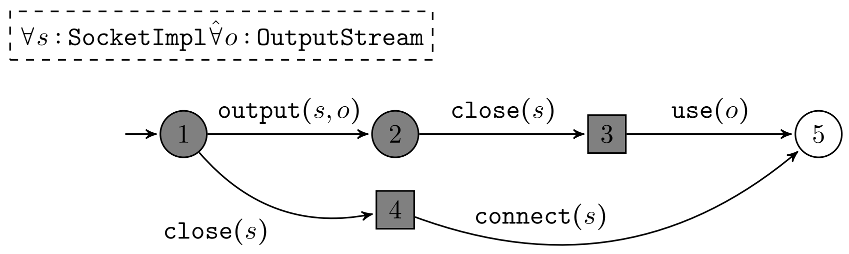


Figure A.13: QEA for proper usage of `java.net.SocketImpl` and `java.io.OutputStream` (SocketUsage)

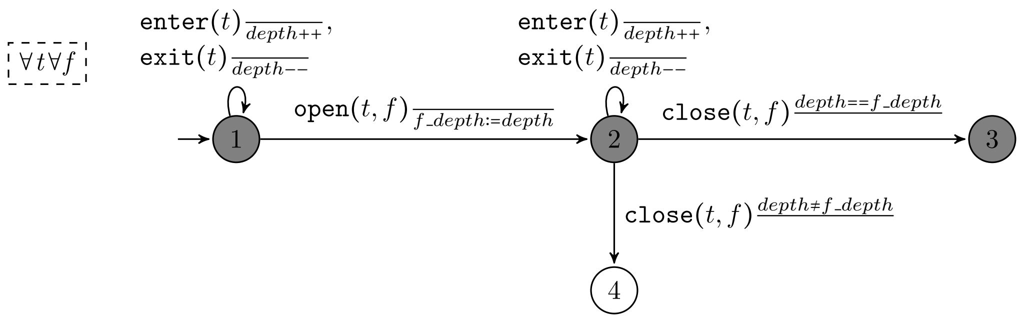

∀t∀f
enter(t) depth++ ,
exit(t) depth−−
enter(t) depth++ ,
exit(t) depth−−
open(t, f) f_depth:=depth
close(t, f) depth==f_depth
close(t, f) depth≠f_depth
1
2
3
4

Figure A.14: A QEA for the safe opening and closing of files within the same method.

CloseFiles.

This property is about files and represents a good software design style that says that if a resource is opened in a method it should also be closed in that method. Here we consider the use of files as a resource and capture the property that if a file is opened for reading or writing in a method then it should be closed in that method. To do this we need to capture not only when a file is 'opened' and 'closed' but also the start and end of method calls - here we track the method calls by each thread separately. Therefore, the events are as follows.

Event	Methods
enter(t)	Start of any method by thread t
exit(t)	End of any method by thread t
open(t, f)	`Writer.new(..)` and `Reader.new(..)` by t
close(t, f)	`Writer.close(..)` and `Reader.close(..)` by t

Note that we capture the opening of a file as the creation of a reader or writer. A QEA for this property is given in Fig. A.14. This is a context-free property captured by the following grammar.

$$\begin{array}{rcl} S & \rightarrow & SA \mid \epsilon \\ A & \rightarrow & A\texttt{enter}(t)A\texttt{exit}(t) \mid A\texttt{open}(t,f)A\texttt{close}(t,f) \mid \epsilon \end{array}$$

We capture this in QEA by keeping track of the depth of the current method call stack using the *depth* variable and storing the depth at which the file is opened in the *f_depth* variable.

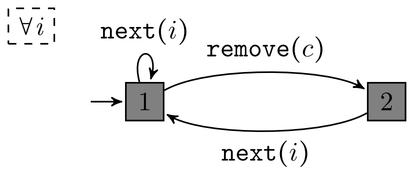


Figure A.15: A QEA for the safe removal of objects using iterators.

A.3.2 Collections usage

Here we consider properties about collections in `java.util`. These are the most common form of API specifications to consider as collections often have some inherent state attached to them that precludes certain actions being performed on them. Here we also see a set of properties considering multiple objects where one collection is constructed from another.

Iterator Usage.

Iterators are objects which walk over a collection. The following events map to methods in `java.util.Collection`, `java.util.Map` and `java.util.Iterator`:

Event	Methods	Description
`create`(c)	`Collection.new`	Constructs a collection
`iterator`(c, i)	`Collection.iterator`	Constructs on iterator from a collection
`connect`(c_1, c_2)	`Map.entrySet`, `Map.keySet`, `Map.values`	Constructs a collection from a map
`update`(c)		Updates the contents of a collection
`hasNext`(i, b)	`Iterator.hasNext`	Returns $false$ if the iterator is finished and $true$ otherwise
`next`(i)	`Iterator.next`	Returns the next object
`remove`(i)	`Iterator.remove`	Removes the current object from the underlying collection

We have chosen not to include the objects added to or removed from the collection or returned by next as they are not interesting to the specifications considered here. The properties considered are:

- **HasNext** - This property states that every call to `next` should be 'guarded' by a call to `hasNext` to ensure that a next object exists. We have already seen this in Sec. 3.3.4 in Fig. 3.2 on page 61.
- **RemoveOnce** - This property states that for every iterator we can only have one remove per call to next. A QEA for this property is given in Fig. A.15.
- **UnsafeIter** - This property states that if an iterator is created from a collection then it should not be used after that collection has been updated. A QEA for the property is given in Fig. A.16.

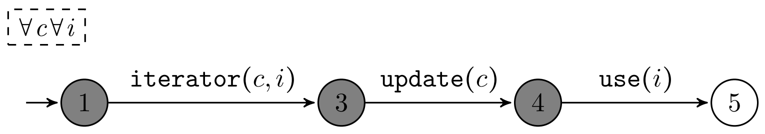


Figure A.16: A QEA for the safe usage of iterators created from collections.

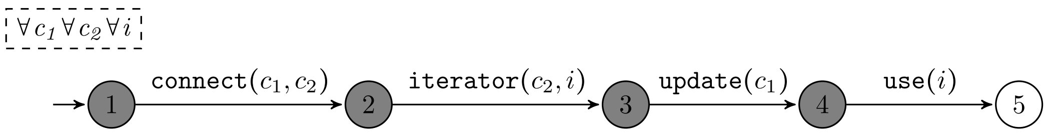


Figure A.17: A QEA for the safe usage of iterators created from collections created from maps.

- **UnsafeMapIter** - This property states that if a set is created from a map (of keys or values) and an iterator is created from this set then if the map is later updated then the iterator should no longer be used. A QEA for this property is given in Fig. A.17.

Hash Persistence.

Next we consider data structures that use hashing i.e. `HashSet` and `HashMap`. These datastructures work by *hashing* some input for efficient storage using the object's `hashCode`, a value that every object in `Java` has. However, hashcodes are computed by a method and are not necessarily consistent. Therefore, a property we would like to capture about datastructures that using hashing is that an object's hashcode remains persistent whilst it is in the collection. Therefore, the events we monitor are as follows.

Event	Methods
`add`(d, o)	Add to datastructure i.e. `HashMap.put`
`observe`(d, o)	Observe in datastructure i.e. `HashMap.get`
`remove`(d, o)	Remove from datastructure i.e. `HashMap.remove`

A QEA for this property is given in Fig. A.18, this assumes that our guard and assignment languages have access to the `Java hashCode` function - otherwise we would need to compute this prior to constructing the event and include it in the event as a free variable.

A.3.3 Concurrency

Let us now consider properties about classes that can be found in `java.util.concurrent`.

Synchronized collections.

Firstly let us consider the problem of unsynchronized access to synchronized collections - this can lead to potential problems such as inconsistent internal state of these collections. To monitor properties related to such access we monitor the following events.

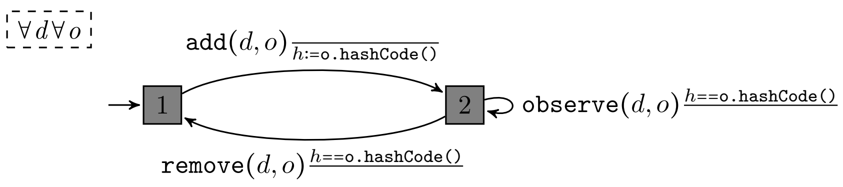

∀d∀o
add(d, o) h:=o.hashCode()
1
2
observe(d, o) h==o.hashCode()
remove(d, o) h==o.hashCode()

Figure A.18: A QEA for the safe usage of hashing data structures.

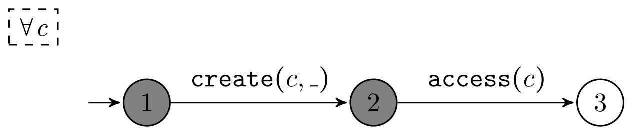


Figure A.19: A QEA for the safe usage of synchronized collections.

Event	Methods
`create`(c_1, c_2)	Create a synchronized collection from an unsynchronized one i.e. `Collections.synchronizedCollection`
`access`(c)	Use a collection i.e. `Collection.add` but also specific methods such as `List.get`
`iterator`(c, i)	`Collection.Iterator`
`use`(i)	Use an iterator i.e. `Iterator.next`, `Iterator.next`

The properties we consider are as follows.

- **LeakingSync** - This property states that If a synchronized collection is created then the original one should not be used. The notion is that the synchronized collection was created to protect accesses to the original collection. A QEA for this property is given in Fig. A.19. Note that we do not need to record the synchronized collection, only the fact that a synchronized collection was created.

- **UnsafeSyncCollection** - This property states that we should not iterate over a synchronized collection without holding the lock for that collection. A QEA for this property is given in Fig. A.20. This property assumes that the guard language has access to the `Java` method `Thread.holdsLock` - if it did not then this information would have to be included in the event. A similar property can be created for maps, as we did with iterators previously, as given in Fig. A.21.

Locks.

We consider two properties concerning the locking and unlocking of object monitors. We monitor two events - `lock`(t, o) and `unlock`(t, o) when an object o is locked or unlocked by a thread t.

- **Mutual Exclusion** - This property states that an object can be locked by at most one thread at any point. A QEA for this property has been given previously in Sec. A.1.3 in Fig. A.6 on page 329.

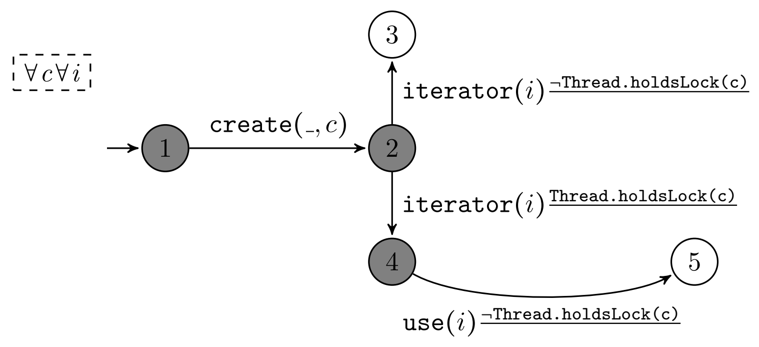

∀c∀i
create(_, c)
iterator(i) ¬Thread.holdsLock(c)
iterator(i) Thread.holdsLock(c)
use(i) ¬Thread.holdsLock(c)
1
2
3
4
5

Figure A.20: A QEA for the safe iteration over synchronized collections.

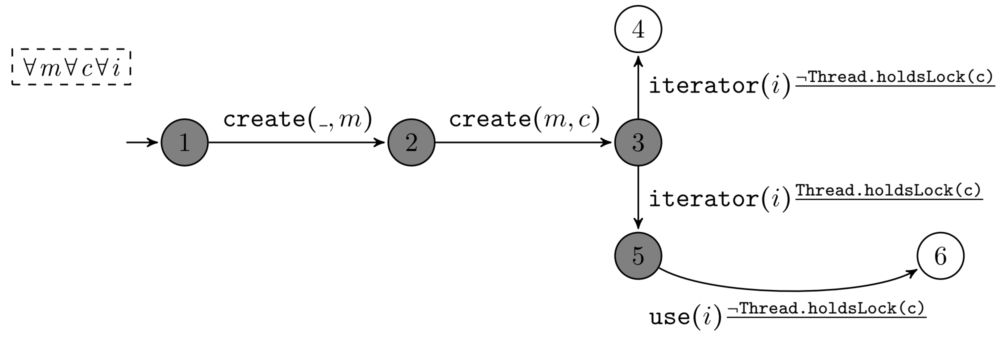

∀m∀c∀i
create(_, m)
create(m, c)
iterator(i) ¬Thread.holdsLock(c)
iterator(i) Thread.holdsLock(c)
use(i) ¬Thread.holdsLock(c)
1
2
3
4
5
6

Figure A.21: A QEA for the safe iteration over synchronized maps.

- **Lock Ordering** - This property states that if two locks are locked in a given order at some point in the program's execution then they should not be locked in the reverse order at any other point in the program's execution. A QEA for this property is given in Fig. A.22.

A.4 Planetary Rover System

We build up a hypothetical system that involves the coordination of a set of planetary rovers and then show how we can use QEA to write certain properties of this system. Many of the specifications in this section are taken from work carried out in collaboration with Klaus Havelund on a tutorial on Runtime Verification at Marktoberdorf 2012 and some can be found in the related book chapter [FHR13]. We have already discussed some example in this setting - see page 61.

A.4.1 Description of system

In this section we briefly introduce the hypothetical planetary rover platform. In this system we have a number of inter-communicating autonomous rovers operating on a remote planet surface (for example, Mars), a number of satellites orbiting the planet used to relay messages from a base station on the home planet, Earth.

We consider two forms of behaviour:

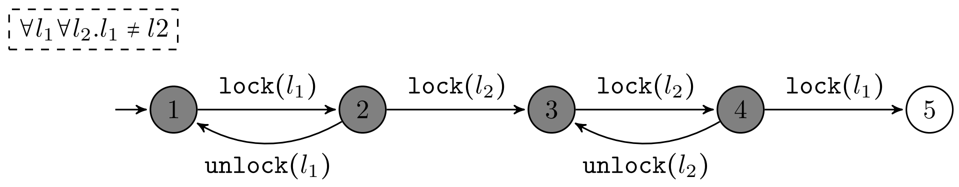


Figure A.22: A QEA specifying that locks should be taken in a consistent order, from [Sto10]

- The **internal** operations carried out by a rover to achieve given tasks.
- The **external** communication between rovers, orbitals and the ground station.

We discuss these two forms in the following.

Internal.

The rover runs *tasks* that use *resources*. A *scheduler* manages the task execution and resource allocation. A task may handle an instrument on board of the rover, such as a camera. Tasks are commanded by the scheduler and report back whether they succeed or fail. In addition, the scheduler manages resources (the antenna for example), which can be requested by tasks, and granted if available and are not in conflict with other granted resources. Tasks must eventually cancel (hand back) resources they have been granted. Conflicting resources can have different priorities. A request for a resource with a higher priority than an already granted resource causes the lower priority resource to be rescinded (the task owning it is asked to cancel it).

A command is scheduled onto a task, and is only ran once with a unique identifier. A task can be granted a resource, or asked to cancel it again (the resource is rescinded). Tasks report success or failure back to the scheduler, as well as request and cancel resources. Two resources can be declared as being in conflict with each other. Furthermore, one resource r_1 can be declared as having higher priority than another conflicting resource r_2. Finally, the scheduler can be put to sleep during the dark nights on the foreign planet. The message sequence diagram in Figure A.23 illustrates a possible sequence of communication events: the scheduler commands a task to perform a job, the task then requests a resource r_1 which has higher priority than a conflicting resource r_2 already granted to another task, the other task is therefore asked to rescind r_2, which it does, after which the resource r_1 is granted to the first task, which subsequently after done job then hands back the resource r_1, and reports success.

Therefore, the events are interest are:

- `schedule`(t, c) - command c is scheduled on task t.
- `finish`(c) - command c successfully completes.
- `request`(t, r) - task t requests resource r.
- `grant`(t, r) - task t is granted resource r.
- `deny`(t, r) - a request for resource r by task t is denied.
- `cancel`(t, r) - task t releases the granted resource r.

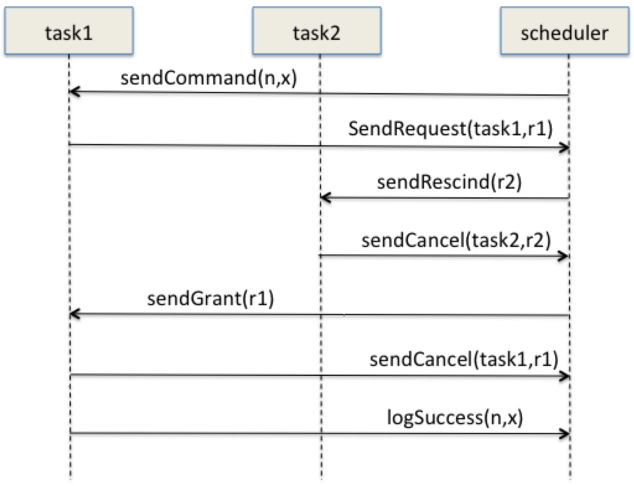


Figure A.23: Message sequence diagram illustrating a possible interaction between scheduler and two tasks.

- `rescind`(t, r) - task t is asked to cancel resource r.
- `conflict`(r_1, r_2) - resources r_1 and r_2 are in conflict.
- `priority`(r_1, r_2) - resource r_1 has higher priority than resource r_2. Note that this implies r_1 is in conflict with r_2, without the need for this to be given explicitly.

External.

Now let us consider external behaviour. The different entities need to communicate with each other to achieve different tasks. We consider two types of communication - that between planetary rovers and the base station, and the intercommunication between rovers and satellites. Fig. A.24 outlines this communication.

The primary form of communication is that between the base station and the planetary rovers (directed via the satellites), this involves the communication of tasks to individual rovers in the form of commands. We consider the point-to-point communication and therefore elide information about the individual rover being targeted i.e. we assume that these low-level mechanisms function correctly. Rovers and the base station communicate using the following messages:

- `com`$(id, name, payload, t)$ - command id given with $name$ and $payload$
- `ack`(id, t) - command id acknowledged
- `suc`$(id, result, t)$ - command id succeeds with $result$
- `fail`$(id, reason, t)$ - command id fails with $reason$

Every messages is given a timestamp t there is an additional message that occurs at startup which is `set_ack_timeout`(t) that sets the required acknowledgment window to t units of time.

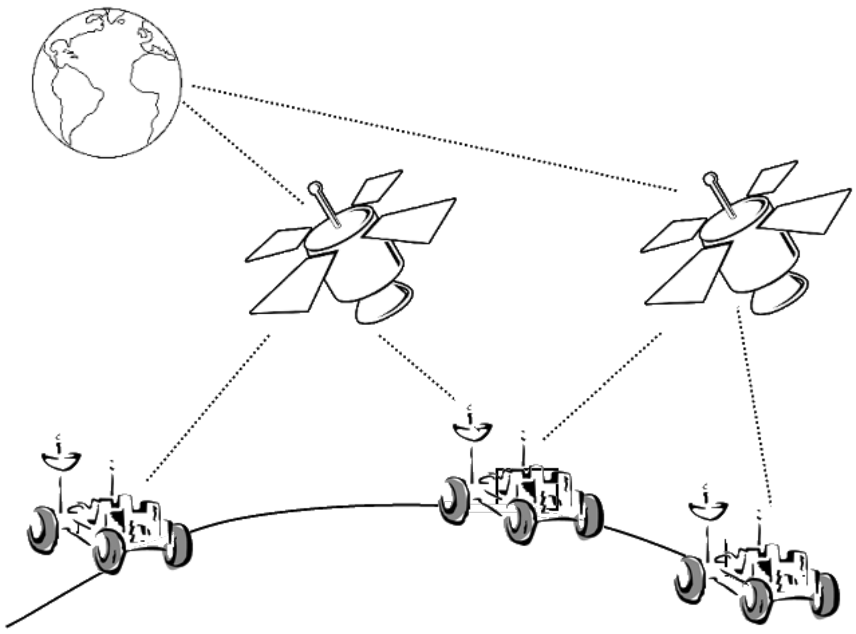


Figure A.24: The different communication

As well as commands sent between the base station and rovers we also consider short-range intercommunication between rovers and satellites. Later we draw distinctions between whether a rover or satellite is sending the message, but these messages all take the following form:

- `ping`(x_1, x_2) - entity x_1 sends a ping request to entity x_2
- `ack`(x_1, x_2) - entity x_1 responds to the ping request of entity x_2
- `send`(x_1, x_2, msg) - entity x_1 sends entity x_2 the message msg
- `ack`(x_1, x_2, h) - entity x_1 acknowledges a message sent by entity x_2 using a hash value h of the sent message

A.4.2 Properties of internal behaviour

We define the following properties of a rover's internal behaviour.

- **GrantCancel.** Grant and cancel actions on a resource by a task should alternate, beginning with a grant and ending with a cancel and a resource should only be granted to a single task at any one time. Fig. A.25 gives two equivalent QEAs for this property - the first using two quantified variables, and the second using one.
- **Resource lifecycle.** This property describes the allowing actions on a resource. A resource can be requested and then either denied or granted. When granted it can be rescinded multiple times before it is canceled. A QEA for this property is given in Fig. A.26.

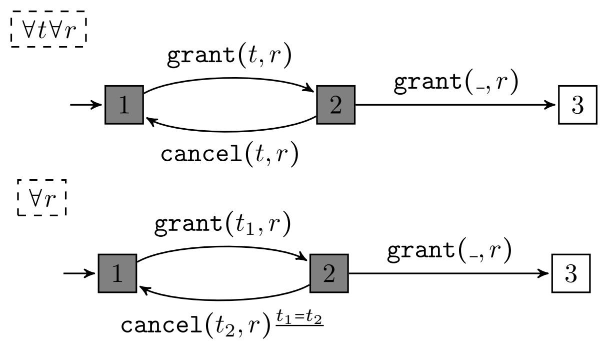


Figure A.25: Two equivalent QEAs for the GrantCancel property.

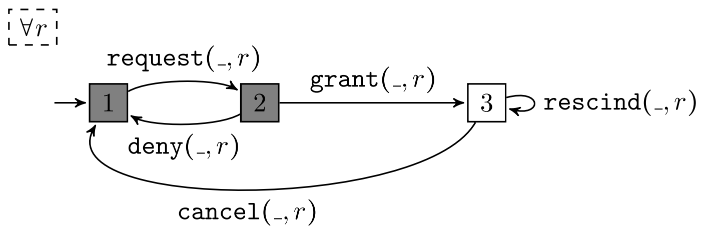


Figure A.26: A QEA for the ResourceLifecycle property.

- **ReleaseResource.** If a resource is granted during the execution of a command then it should be released before the command succeeds. A QEA for this property is given in Fig. A.27.

- **Respecting resource conflicts.** This property states that conflicting resources cannot be granted at the same time. We present two alternative QEA for capturing this property. The first is given in Fig. A.28 and states that for every pair of resources if they are in conflict then if the first is granted the second cannot be granted before the first is canceled. Note that for every pair of resources A and B there will be two instantiations of this QEA - one for $r_1 = A, r_2 = B$ and one for $r_1 = B, r_2 = A$. The second QEA is given in Fig. A.29 and uses a single quantified variable. This is quantified over a single resource r_1 and first collects a set of conflicting resources and then whenever a grant occurs whilst r_1 is granted it checks to ensure the newly granted resource is not conflicting. The second QEA has a lower monitoring complexity as it produces far fewer bindings - as discussed later.

- **Respecting resource priorities.** Let us begin by noting that this is a very complex property to specify. The property is that if a resource r_2 has higher priority than resource

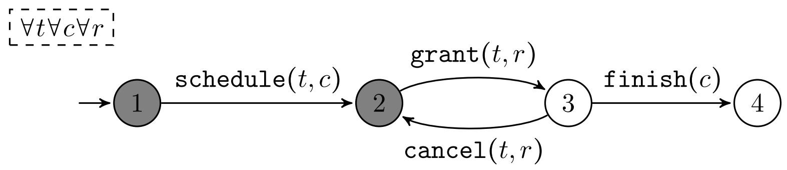


Figure A.27: A QEA for the ReleaseResource property.

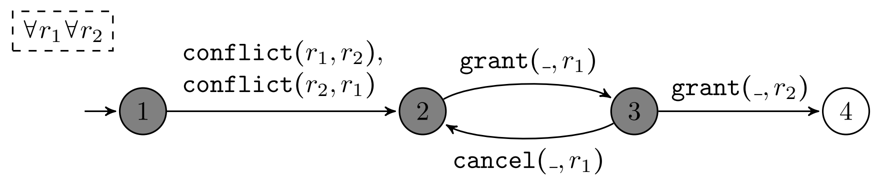

$\forall r_1 \forall r_2$
conflict(r_1, r_2), conflict(r_2, r_1)
grant($\_, r_1$)
cancel($\_, r_1$)
grant($\_, r_2$)
1
2
3
4

Figure A.28: The RespectConflicts property using two quantified variables.

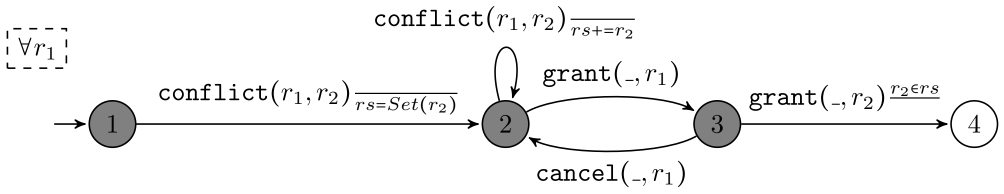

$\forall r_1$
conflict(r_1, r_2) $\overline{rs{+}{=}r_2}$
conflict(r_1, r_2) $\overline{rs{=}Set(r_2)}$
grant($\_, r_1$)
cancel($\_, r_1$)
grant($\_, r_2$) $\overline{r_2 \in rs}$
1
2
3
4

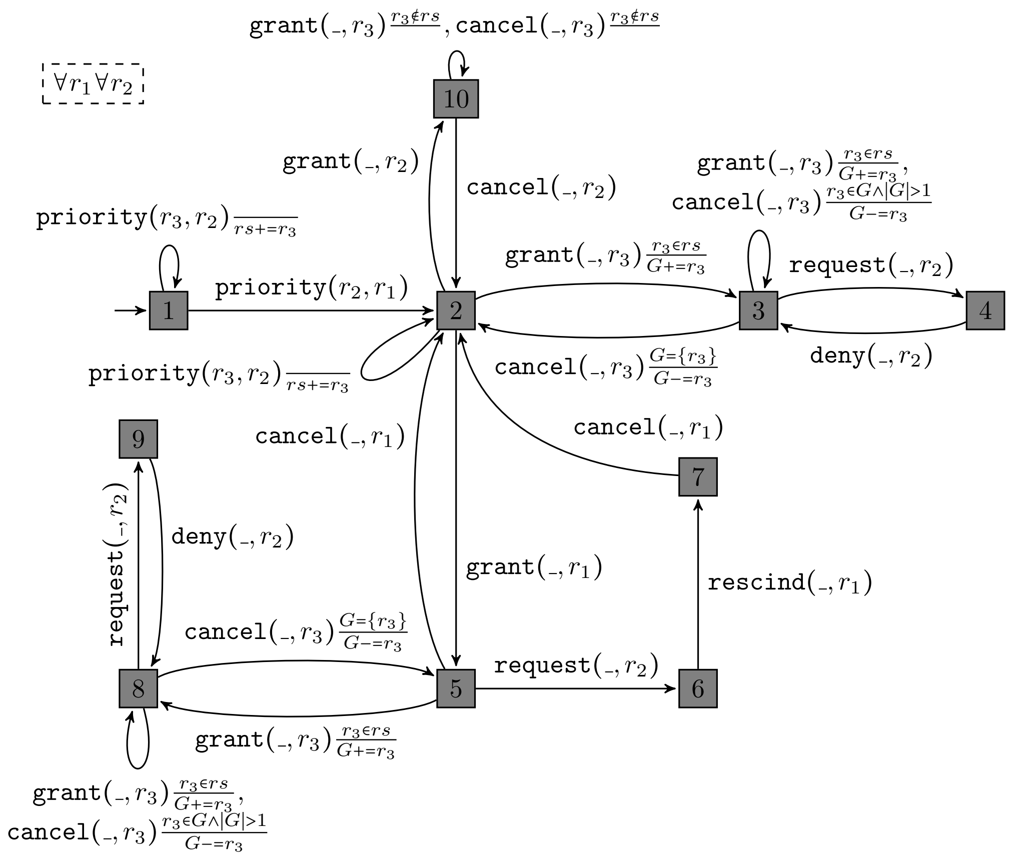
Figure A.29: The RespectConflicts property using one quantified variable.

r_1 and r_2 is requested whilst r_1 is granted then r_1 should be rescinded *but only if* there is no resource r_3 with higher priority than r_2 that is also currently granted. It is this second aspect that introduces the complexity, as we need to keep track of the granted status of all resources with higher priority than resource r_2. Furthermore, note that priority is not a total order, and is not even transitive i.e. $r_3 > r_2$ and $r_2 > r_1$ *does not* imply $r3 > r_1$. The QEA for this property is given in Fig. A.30. This works by building a set rs of resources with higher priority than r_2 and using this to check whether such a resource is currently granted. The transitions between states 2 and 3 (and 5 and 8) track a set G of granted resources and ensure that a request for resource r_2 is denied if this set is not empty. The expected behaviour is then given by the path through states 2, 5, 6 and 7. Note that state 10 is used to record the fact that when resource r_2 is granted neither r_1 or any resource in rs can be granted. Finally, note that we do not consider the case where r_2 should be rescinded, as this will be covered in the case where the value for r_2 is bound to r_1.

A.4.3 Properties of external behaviour

Here we consider properties of external behaviour. These describe the accepted communication protocols between different entities. We first consider the communication between the base station and planetary rovers, many of these have been considered previously:

- **Exactly one success.** Each command succeeds exactly once. A QEA for this property is given in Fig. A.31. Note that the use of next states preclude the behaviour with two `suc` events.
- **Nested commands.** This property was described in Sec. 3.3.4 and a QEA for the property was given in Fig. 3.3 on page 61. Note that this QEA only uses the command identifier part of the `com` and `suc` events.
- **Increasing command identifiers.** To ensure that every command is given a unique



$\forall r_1 \forall r_2$
grant$(_, r_3)\frac{r_3 \notin rs}{}$, cancel$(_, r_3)\frac{r_3 \notin rs}{}$
grant$(_, r_2)$
cancel$(_, r_2)$
priority$(r_3, r_2)\frac{}{rs{+}{=}r_3}$
priority(r_2, r_1)
grant$(_, r_3)\frac{r_3 \in rs}{G{+}{=}r_3}$
grant$(_, r_3)\frac{r_3 \in rs}{G{+}{=}r_3}$, cancel$(_, r_3)\frac{r_3 \in G \wedge |G|>1}{G{-}{=}r_3}$
request$(_, r_2)$
deny$(_, r_2)$
cancel$(_, r_3)\frac{G=\{r_3\}}{G{-}{=}r_3}$
cancel$(_, r_1)$
cancel$(_, r_1)$
request$(_, r_2)$
deny$(_, r_2)$
grant$(_, r_1)$
rescind$(_, r_1)$
cancel$(_, r_3)\frac{G=\{r_3\}}{G{-}{=}r_3}$
request$(_, r_2)$
grant$(_, r_3)\frac{r_3 \in rs}{G{+}{=}r_3}$
grant$(_, r_3)\frac{r_3 \in rs}{G{+}{=}r_3}$, cancel$(_, r_3)\frac{r_3 \in G \wedge |G|>1}{G{-}{=}r_3}$
1
2
3
4
5
6
7
8
9
10

Figure A.30: A QEA for the RespectPriorities property.

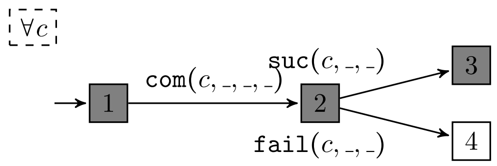


Figure A.31: A QEA for the ExactlyOnceSuccess property.

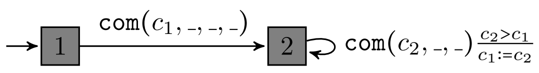


Figure A.32: A QEA for the IncreasingIdentifiers property.

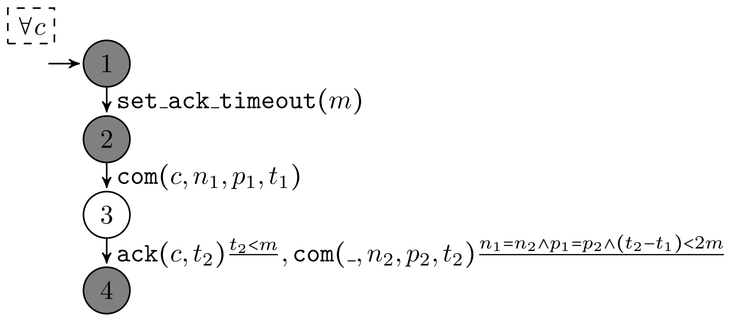

$\forall c$
1
set_ack_timeout(m)
2
com(c, n_1, p_1, t_1)
3
ack(c, t_2) $\frac{t_2<m}{}$, com(_, n_2, p_2, t_2) $\frac{n_1=n_2 \wedge p_1=p_2 \wedge (t_2-t_1)<2m}{}$
4

Figure A.33: A QEA for the CommandAcknowledgements property.

identifier we enforce the stricter policy that identifiers are strictly increasing. A quantifier-free QEA for this property is given in Fig. A.32.

- **CommandAcknowledgements**. Every command should be acknowledged within a given amount of time m, otherwise a command with the same name and payload should be sent within $2m$. A QEA for this property is given in Fig. A.33. Note that this QEA has one disadvantage to the event-driven approach of QEA. If we pass $2m$ time units for a command c and have not seen an acknowledgment or resend we know that the trace cannot succeed, however, this QEA will only detect failure at the end of the trace i.e. it cannot differentiate between failure because the condition *has not* been met and failure because the condition *cannot* be met. One possible extension that could address this issue is the addition of *timers* that produced an event after a given amount of time.

Next we consider the short-range intercommunication between satellites and planetary rovers.

- **Exists satellite.** This property states that every rover can communicate with at least one satellite. Communication is established via `ping` and `ack` messages. A QEA for this property is given in Fig. A.34 - the assumption is that all `ping` requests come from rovers, and all `ack` replies come from satellites. We can also strip the innermost existential quantifier using the technique described in Sec. 4.3.2 - giving the QEA in Fig. A.35.

- **Exists rover leader.** This is an implicit property. We want to state that after a certain amount of time there is a rover who has successfully communicated with all known rovers.

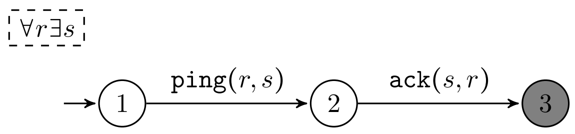


Figure A.34: A QEA for the ExistsSatellite property.

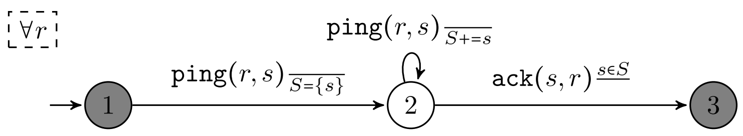

$\forall r$
$\texttt{ping}(r,s)\,\frac{}{S=\{s\}}$
$\texttt{ping}(r,s)\,\frac{}{S{+}{=}s}$
$\texttt{ack}(s,r)\,\frac{s\in S}{}$
1
2
3

Figure A.35: A QEA for the ExistsSatellite property with the innermost existential quantifier stripped.

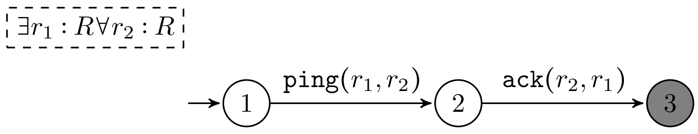


Figure A.36: A QEA for the ExistsLeader property.

Again, communication is established via `ping` and `ack` messages. A QEA for this property is given in Fig. A.36 - here type variables are used to ensure that the set of rover considers consists of all rovers that take part in `ping` or `ack` messages. This property was previously explored in Sec. 3.5.1.

- **MessageHashCorrect.** The hash of a message in an acknowledgment should match the actual hash of the message. A QEA for this property is given in Fig. A.37. Here we quantify over communication between entity x_1 and entity x_2 (in that direction) and keep track of the set of sent messages. This approach is necessary as we may have two messages sent before either is acknowledged.

A.5 Comments on style

In this section we use a file system example to explore the different specification styles that can be used when writing a QEA this builds on the previous two examples taken from such a setting by introducing the idea of a file usage mode and version number. Consider a file system where users can open, read, writer, save and close files. Let us assume the system has been instrumented to generate the following events:

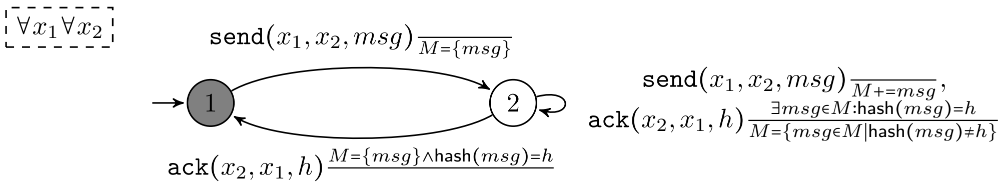

$\forall x_1 \forall x_2$
$\texttt{send}(x_1, x_2, msg)\,\frac{}{M=\{msg\}}$
$\texttt{send}(x_1, x_2, msg)\,\frac{}{M{+}{=}msg}$,
$\texttt{ack}(x_2, x_1, h)\,\frac{\exists msg\in M:\mathsf{hash}(msg)=h}{M=\{msg\in M \mid \mathsf{hash}(msg)\neq h\}}$
$\texttt{ack}(x_2, x_1, h)\,\frac{M=\{msg\}\wedge\mathsf{hash}(msg)=h}{}$
1
2

Figure A.37: A QEA for the MessageHashCorrect property.

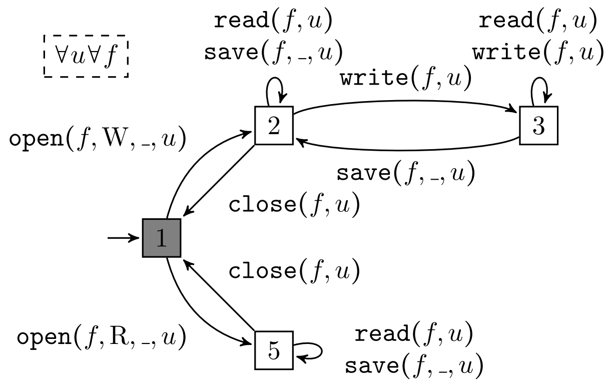


Figure A.38: A QEA describing how files should be opened,closed and saved.

Event	Description
open(f,r,n,u)	User u opens file f in mode r with version number n
close(f,u)	User u closes file f
read(f,u)	User u reads file f
write(f,u)	User u writes to file f
save(f,n,u)	User u saves file f with version number n

We consider and specify the following properties of this system.

Using a file.

A file must be opened to be used or closed and cannot be opened if already open. A file can be opened in read or write mode. If it is opened in read mode it can only be read or saved but if it is opened in write mode it can also be written. If it is written then it must be saved before being closed. A QEA for this property is given in Fig. A.38. Note that this is similar to the simple file usage property introduced in Sec. A.1.1. This has the hallmarks of a typical *safety* property - universal quantification and the use of next states throughout, however as only the initial state is accepting we have a *response* property i.e. it is always the case that given some action (the file is opened) then we expect some response (it is used before being closed). As the QEA is relatively large it is useful to use next states, as otherwise we would need to include a failing state and many additional transitions.

Access control.

If a file is opened by one user it cannot be edited by another. Three equivalent QEA for this property have been given in Fig. A.39. These represent three different *styles* of specifying properties. The first (top left) uses universal quantification with a global guard to capture the behaviour that u_2 writes to a file opened by u_1. Note that skip states are used as we are only interested in events that take us towards the failing condition - this is a typical style for describing a property via reachability of some undesirable state. The second (bottom left) replaces the quantification of u_2 with a free variable - this is referred to as the quantifier stripping trick in Sec. 4.3.2 and achieves the same effect as the first style as we are asserting

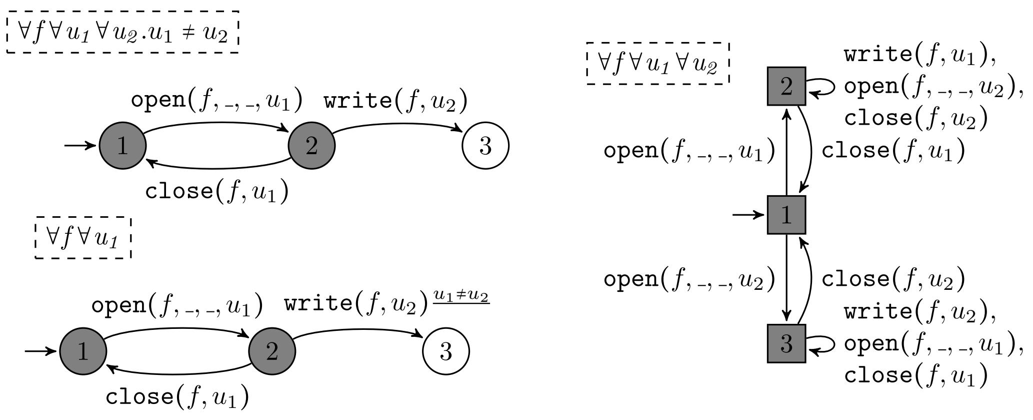

$\forall f \forall u_1 \forall u_2 . u_1 \neq u_2$
open$(f,_,_,u_1)$
write(f,u_2)
close(f,u_1)
$\forall f \forall u_1$
open$(f,_,_,u_1)$
write(f,u_2) $\frac{u_1 \neq u_2}{}$
close(f,u_1)
$\forall f \forall u_1 \forall u_2$
write(f,u_1), open$(f,_,_,u_2)$, close(f,u_2)
open$(f,_,_,u_1)$
close(f,u_1)
open$(f,_,_,u_2)$
close(f,u_2)
write(f,u_2), open$(f,_,_,u_1)$, close(f,u_1)

Figure A.39: Three equivalent QEA for proper file access.

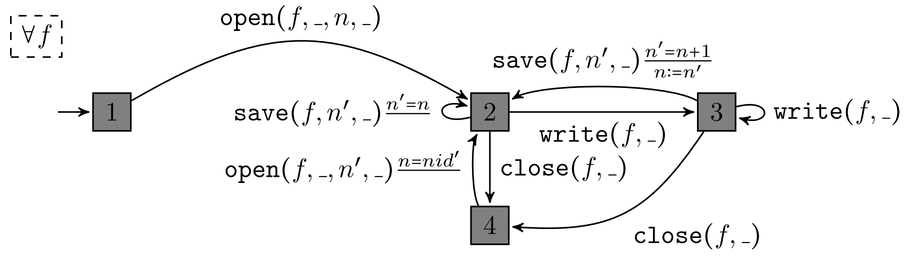

$\forall f$
open$(f,_,n,_)$
save$(f,n',_)$ $\frac{n'=n+1}{n:=n'}$
save$(f,n',_)$ $\frac{n'=n}{}$
write$(f,_)$
write$(f,_)$
open$(f,_,n',_)$ $\frac{n=nid'}{}$
close$(f,_)$
close$(f,_)$

Figure A.40: A QEA describing the property that a files version number should increase by one very time it is saved after a write.

the existence of a counterexample to the universal quantification. The third (right) uses next states to enumerate all possible acceptable behaviours. Note that in this case the QEA is more complicated when using next states rather than skip states. This is because our property is best (i.e. most concisely) described by the undesired, rather than desired, behaviour.

Version numbers.

A files version number should increase by one every time it is saved after being written to. If a file is closed then the next time it is opened it should have the same version number. A QEA for this property is given in Fig. A.40. Here we see a typical safety property - universal quantification, next states all accepting - describing the exact desired behaviour. Note the way that we save the previous version number so that we can refer to it at a later stage.

Appendix B

Proofs

This appendix contains proofs omitted in Chapter 5.

B.1 Computing the same bindings

We begin with proofs related to showing that the big and small step semantics compute the same bindings. The first lemma was first given on page 106.

Lemma 9. *Given QEA* $\mathcal{Q} = \langle \Lambda, \mathcal{E}, \mathcal{D} \rangle$ *and trace* τ *let* $\mathsf{adjusted}(\mathcal{Q}) = \langle \Lambda, \mathcal{E}', \mathcal{D} \rangle$ *then*

$$\forall x \in \mathsf{vars}(\Lambda) : x \notin \mathsf{dom}(\mathcal{D}) \Rightarrow \mathsf{Dom}^{\mathcal{E}}(\tau)(\mathsf{typeOf}(x)) = D_{L_\tau}(x)$$

where L_τ *is based on* $\mathsf{adjusted}(\mathcal{Q})$.

Proof. Recall that given the alphabet of $\mathcal{E}$ as $\mathcal{A}$ and $\Lambda = Q_1 x_1 : X_1.g_1, \ldots, Q_n x_n : X_n.g_n$ for $Q \in \{\forall, \exists\}$ the alphabet $\mathcal{A}'$ of $\mathcal{E}'$ is

$$\mathcal{A}' = \mathcal{A} \cup \{e(y_1, \ldots, y_n) \mid \exists e(x_1, \ldots, x_n) \in \mathcal{A}, \forall i \in [1, n] : y_i \in \mathsf{varsOf}(\mathsf{typeOf}(x_i))\}$$

Also recall the definition of Dom given in Def. 26

$$\begin{aligned}\mathsf{Dom}^{\mathcal{E}}(\tau)(X) = \{\mathtt{match}(\mathbf{a}, \mathbf{b})(x) \mid \quad & x \in \mathsf{varsOf}(X) \wedge \mathbf{b} = e(..., x, ...) \in \mathcal{A} \wedge \\ & \mathbf{a} \in \tau \wedge \mathtt{matches}(\mathbf{a}, \mathbf{b})\}.\end{aligned}$$

and the definition of D_L given in Def. 43

$$D_L(x) = \begin{array}{ll} \mathcal{D}(\mathsf{typeOf}(x)) & \text{if } \mathsf{typeOf}(x) \in \mathsf{dom}(\mathcal{D}) \\ \{\theta(x) \mid \theta \in \mathsf{dom}(L) \wedge x \in \mathsf{dom}(\theta)\} & \text{otherwise} \end{array}$$

We are aiming to show that

$$\forall x \in \mathsf{vars}(\Lambda) : \mathsf{Dom}^{\mathcal{E}}(\tau)(\mathsf{typeOf}(x)) = \{\theta(x) \mid \theta \in \mathsf{dom}(L_\tau) \wedge x \in \mathsf{dom}(\theta)\} \qquad \text{(B.1)}$$

as we can remove the $\notin \mathcal{D}$ condition by applying it to both sides of the equivalence. By definition (of $\mathsf{extensions}$) the domain of L_τ includes, for any $\mathbf{a} \in \mathcal{A}$ and any $\mathbf{b} \in \tau$ the binding $\mathsf{match}(\mathbf{a},\mathbf{b})$. Therefore we can write (B.1) as

$$\begin{aligned}\forall x \in \mathsf{vars}(\Lambda) : \mathsf{Dom}^{\mathcal{E}}(\tau)(\mathsf{typeOf}(x)) = \quad & \{\mathtt{match}(\mathbf{a},\mathbf{b})(x) \mid \wedge \mathbf{a} \in \mathcal{A} \wedge \mathbf{b} \in \tau \wedge \\ & x \in \mathsf{dom}(\theta) \wedge \mathtt{matches}(\mathbf{a},\mathbf{b})\}\end{aligned}$$

The only syntactic difference between the definitions of each side is the use of $x \in \mathsf{varsOf}(X)$ on the left. Therefore, we are just required to show that every $y \in \mathsf{varsOf}(\mathsf{typeOf}(x))$ is treated in the same way as x for $\mathcal{E}'$. By the definition of $\mathcal{A}'$ the variables y and x have appear equivalently in the alphabet, and therefore must be treated in the same way. □

This next lemma was first given on page 106.

Lemma 10. *For all traces τ and all QEA $\langle \Lambda, \mathcal{E}, \mathcal{D}\rangle$*

$$\mathsf{build}(\tau) = \mathsf{construct}(\tau)$$

Proof. By structural induction on τ. For the base case, $\tau = \epsilon$, $\mathtt{build}(\epsilon) = \{\,[\,]\}$ and $\mathsf{Dom}(\epsilon) = [\,]$ therefore $\mathsf{construct}(\epsilon) = \{[\,\,]\}$. For the inductive step, $\tau = \sigma.\mathbf{a}$, we show that

$$\mathsf{build}(\sigma.\mathbf{a}) = \mathsf{construct}(\sigma.\mathbf{a}) \tag{B.2}$$

by assuming $\mathsf{build}(\sigma) = \mathsf{construct}(\sigma)$ and showing the additions are equivalent. Let the bindings derivable from the new event $\mathbf{a}$ be given as

$$\begin{aligned}\mathsf{direct}(\mathbf{a},\mathcal{A}) \quad &= \{\theta \mid \exists \mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \theta \sqsubseteq \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b}))\} \\ \mathsf{derived}(\mathbf{a}) \quad &= \mathsf{close}_{\sqcup}\ \mathsf{direct}(\mathbf{a})\end{aligned}$$

Rewrite $\mathsf{construct}(\sigma.\mathbf{a})$ as follows:

$$\begin{aligned}\mathsf{construct}(\sigma.\mathbf{a}) \quad &= \quad \{\theta \in \mathit{Bind} \mid \forall (x \mapsto v) \in \theta : v \in \mathsf{Dom}(\sigma.\mathbf{a})(\mathsf{typeOf}(x))\} \\ &= \quad \mathsf{construct}(\sigma) \cup \textstyle\bigcup_{\theta \in \mathsf{construct}(\sigma)} \{\theta \sqcup \theta' \mid \theta' \in \mathsf{derived}(\mathbf{a})\} \\ &= \quad \mathsf{build}(\sigma) \cup \textstyle\bigcup_{\theta \in \mathsf{build}(\sigma)} \{\theta \sqcup \theta' \mid \theta' \in \mathsf{derived}(\mathbf{a})\}\end{aligned}$$

The last step uses our induction hypothesis $\mathsf{construct}(\sigma) = \mathsf{build}(\sigma)$. By expanding $\mathsf{build}(\sigma.\mathbf{a})$(Def 46), equation (B.2) becomes

$$\mathsf{build}(\sigma) \cup \bigcup_{\theta \in \mathsf{build}(\sigma)} \mathsf{extensions}(\theta,\mathbf{a}) = \mathsf{build}(\sigma) \cup \bigcup_{\theta \in \mathsf{build}(\sigma)} \{\theta \sqcup \theta' \mid \theta' \in \mathsf{derived}(\mathbf{a})\} \tag{B.3}$$

It is sufficient to show that the following holds for a given θ in $\mathsf{build}(\sigma)$

$$\mathsf{extensions}(\theta,\mathbf{a}) = \{\theta \sqcup \theta' \mid \theta' \in \mathsf{derived}(\mathbf{a})\}. \tag{B.4}$$

By expanding $\mathsf{extensions}(\mathbf{a},\theta)$ and $\mathsf{derived}(\mathbf{a})$ equation (B.4) becomes

$$\{\theta\dagger\theta' \mid \theta' \in \mathsf{all\_from}(\theta,\mathbf{a},\mathcal{A}) \wedge \theta' \neq [\;]\} = \{\theta \sqcup \theta' \mid \theta' \in \mathsf{close}_{\sqcup}\ \mathsf{direct}(\mathbf{a})\} \tag{B.5}$$

We can drop the $\theta' = [\;]$ condition as we have that $\theta = \theta\dagger[\;]$, which is already contained in $\mathsf{build}(\sigma)$ by our inductive hypothesis. We can also replace the $\dagger$ with $\sqcup$ on the left hand side as it is guaranteed that θ and θ' are consistent and in this case $\dagger$ and $\sqcup$ are equivalent.

$$\{\theta \sqcup \theta' \mid \theta' \in \mathsf{all\_from}(\theta,\mathbf{a},\mathcal{A})\} = \{\theta \sqcup \theta' \mid \theta' \in \mathsf{close}_{\sqcup}\ \mathsf{direct}(\mathbf{a})\} \tag{B.6}$$

Let us write this as

$$A = B$$

and expand the formulas A and B to show that they are equivalent. Firstly, in A expand $\mathsf{all\_from}(\theta,\mathbf{a},\mathcal{A})$, then push the lub operator through the lub-closure, expand $\mathsf{from}(\theta,\mathbf{a})$ and finally remove the redundant substitution $\mathcal{A}(\theta)$. Not that $\mathcal{A}(\theta)$ is redundant as we take the lub with θ.

$$\begin{aligned}
A &= \{\theta \sqcup \theta' \mid \theta' \in \mathsf{close}_{\sqcup}\ \{\theta'' \mid \exists\theta''' \in \mathsf{from}(\theta,\mathbf{a}).\theta'' \sqsubseteq \theta'''\}\}\\
&= \mathsf{close}_{\sqcup}\ \{\theta \sqcup \theta' \mid \exists\theta'' \in \mathsf{from}(\theta,\mathbf{a}).\theta' \sqsubseteq \theta''\}\\
&= \mathsf{close}_{\sqcup}\ \{\theta \sqcup \theta' \mid \exists\mathbf{b} \in \mathcal{A}(\theta) : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \theta' \sqsubseteq \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b}))\}\\
&= \mathsf{close}_{\sqcup}\ \{\theta \sqcup \theta' \mid \exists\mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \theta' \sqsubseteq \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b}))\}
\end{aligned}$$

Next, in B expand $\mathsf{direct}(\mathbf{a})$ and then push the lub operator through the lub-closure.

$$\begin{aligned}
B &= \{\theta \sqcup \theta' \mid \theta' \in \mathsf{close}_{\sqcup}\ \{\theta'' \mid \exists\mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \theta'' \sqsubseteq \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b}))\}\}\\
&= \mathsf{close}_{\sqcup}\ \{\theta \sqcup \theta' \mid \exists\mathbf{b} \in \mathcal{A} : \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge \theta' \sqsubseteq \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b}))\}
\end{aligned}$$

Therefore, we have shown that $A = B$ and have completed our proof. □

B.2 Computed configurations match computed projections

We now consider the parts of the proof related to showing that the small step semantics for computing configurations is equivalent to that for computing projections. The lemma we consider was first given on page 110.

Lemma 18. *For binding θ, event $\mathbf{a}$ relevant to θ, and set of configurations C*

$$\mathsf{next}(\mathbf{a},\theta,C) = \{c' \mid \exists c \in C : c \overset{\mathbf{a}}{\rightsquigarrow}_{\theta,\mathcal{E}(\theta)} c'\}$$

Proof. We will derive the left-hand side of this equation from the right-hand side. We do this by rewriting the set

$$\{c' \mid \exists c \in C : c \overset{\mathbf{a}}{\rightsquigarrow}_{\theta,\mathcal{E}(\theta)} c'\}$$

as two separate sets by considering the relation $\rightsquigarrow_{\theta,\mathcal{E}(\theta)}$, which describes two sets of next configurations - those that are the result of *moving*, captured by $\rightsquigarrow_\theta$, and those that are the result of *staying*, captured by the closure to $\rightsquigarrow_{\theta,\mathcal{E}(\theta)}$. Therefore

$$\{c' \mid \exists c \in C : c \overset{\mathbf{a}}{\rightsquigarrow}_{\theta,\mathcal{E}(\theta)} c'\} = \{c' \mid \exists c \in C : c \overset{\mathbf{a}}{\rightsquigarrow}_\theta c'\} \cup \{c \in C \mid \nexists\, c' \in \mathit{Config} : c \overset{\mathbf{a}}{\rightsquigarrow}_\theta c'\}.$$

Let us consider the configurations which are the result of moving, that is the configurations

$$\mathsf{move}(\mathbf{a},\theta,C) = \{c' \mid \exists c \in C : c \overset{\mathbf{a}}{\rightsquigarrow}_\theta c'\}.$$

The relation $\rightsquigarrow_\theta$ is given by

$$\langle q,\varphi\rangle \rightsquigarrow_\theta \langle q',\varphi'\rangle \text{ iff } \begin{array}{l} \exists \mathbf{b} \in \mathcal{A}(\theta), \exists g \in \mathit{Guard}, \exists \gamma \in \mathit{Assign} : (q,\mathbf{b},g,\gamma,q') \in \delta(\theta)\ \wedge \\ \quad \mathsf{matches}(\mathbf{a},\mathbf{b}) \wedge g(\varphi\dagger\mathsf{match}(\mathbf{a},\mathbf{b})) \wedge \varphi' = \gamma(\varphi\dagger\mathsf{match}(\mathbf{a},\mathbf{b}))\wedge \\ \quad \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\ ] \end{array}$$

and therefore, we can rewrite the set $\mathsf{move}(\mathbf{a},\theta,C)$ as follows

$$\begin{array}{rl} \mathsf{move}(\mathbf{a},\theta,C) = & \{\langle q',\gamma(\varphi\dagger\mathsf{match}(\mathbf{a},\mathbf{b}))\rangle \mid \exists\langle q,\varphi\rangle \in C, \exists g \in \mathit{Guard} : \\ & \qquad (q,\mathbf{b},g,\gamma,q') \in \delta(\theta) \wedge \mathsf{matches}(\mathbf{a},\mathbf{b})\wedge \\ & \qquad g(\varphi\dagger\mathsf{match}(\mathbf{a},\mathbf{b})) \wedge \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\ ] \end{array}$$

Now let us consider the configurations which are the result of staying, that is the configurations

$$\mathsf{stay}(\mathbf{a},\theta,C) = \{c \in C : \nexists\, c' \in \mathit{Config} \mid c \overset{\mathbf{a}}{\rightsquigarrow}_\theta c'\}.$$

Again, we can use the definition of $\rightsquigarrow_\theta$ to rewrite the set $\mathsf{stay}(\mathbf{a},\theta,C)$:

$$\begin{array}{rl} \mathsf{stay}(\mathbf{a},\theta,C) = & \{\langle q,\varphi\rangle \in C \mid \nexists\, \mathbf{a} \in \mathcal{A}(\theta) : \exists q' \in Q, \exists g \in \mathit{Guard} \exists \gamma \in \mathit{Assign} : \\ & \qquad (q,\mathbf{b},g,\gamma,q') \in \delta(\theta) \wedge \mathsf{matches}(\mathbf{a},\mathbf{b})\wedge \\ & \qquad g(\varphi\dagger\mathsf{match}(\mathbf{a},\mathbf{b})) \wedge \mathsf{quantified}(\mathsf{match}(\mathbf{a},\mathbf{b})) = [\ ] \end{array}$$

We have now used the definition of the partial transition relation to derive

$$\mathsf{move}(\mathbf{a},\theta,C) \cup \mathsf{stay}(\mathbf{a},\theta,C) = \mathsf{next}(\mathbf{a},\theta,C)$$

□

B.3 Basic algorithm

Finally we give the proofs for the propositions of correctness for the basic monitoring algorithm given on page 116.

Proposition 2. *When processing event* $\mathbf{a}$ *for each binding* $\theta \in \mathsf{dom}(M)$ *the computed set* B' *is equivalent to the set* $\mathsf{extensions}(\theta,\mathbf{a}) \cup \{\theta \mid \mathsf{relevant}(\theta,\mathbf{a})\}$ *i.e. it adds* θ *to the set of extensions*

Algorithm 19 A refactored matching function.

```
function MATCHING_R($\mathbf{a}$ : GEvent) : Set[Binding]
    B : Set[Binding] $\leftarrow$ [ ]
    for $\mathbf{b} \in \mathcal{Q}.\mathcal{E}.\mathcal{A}$ do
        if $\mathsf{matches}(\mathbf{a}, \mathbf{b})$ then
            $\theta \leftarrow \mathsf{quantified}(\mathsf{match}(\mathbf{a}, \mathbf{b}))$
            B += $\theta$
    for $\theta \in$B do
        for $\theta_1 \sqsubseteq \theta$ do
            B += $\theta_1$
    max $\leftarrow \|$B$\|$
    for 1 to max do
        for $\theta_1, \theta_2 \in$ B do
            if $\theta_1$ compatible with $\theta_2$ then B += $\theta_1 \dagger \theta_2$
    return B
```

if and only if θ *is relevant to* $\mathbf{a}$.

Proof. Firstly we note that

$$\mathrm{B}' = \{\theta\dagger\theta' \mid \theta' \in \textsc{Matching}(\mathbf{a}) \wedge \mathsf{compatible}(\theta, \theta')\} \tag{B.7}$$

as B′ is constructed by iterating over the bindings in B. Recall that

$$\mathsf{extensions}(\theta, \mathbf{a}) = \{\theta\dagger\theta' \mid \theta' \in \mathsf{all\_from}(\theta, \mathbf{a}, \mathcal{A}) \wedge \theta' \neq [\ ]\}$$

and that, due to the definition of all_from we have

$$[\ ] \in \mathsf{extensions}(\theta, \mathbf{a}) \Leftrightarrow \mathsf{relevant}(\theta, \mathbf{a})$$

therefore, we are trying to show that

$$\mathrm{B}' = \{\theta\dagger\theta' \mid \theta' \in \mathsf{all\_from}(\theta, \mathbf{a}, \mathcal{A})\}$$

Let us note that we can refactor the Matching function so that it operates in two steps, as shown in Algorithm 19. The first loop is equivalent to the computation of

$$\mathsf{from}([\ ], \mathbf{a}, \mathcal{A}) = \{\mathsf{quantified}(\mathsf{match}(\mathbf{a}, \mathbf{b})) \mid \mathbf{b} \in \mathcal{A}(\theta) \wedge \mathsf{matches}(\mathbf{a}, \mathbf{b})\}$$

this is easy to verify by noting that we consider all events in $\mathcal{A}$. The second two loops computes

$$\mathsf{all\_from}([\ ], \mathbf{a}, \mathcal{A}) = \mathsf{close}_{\sqcup}\ \{\theta' \mid \exists\theta'' \in \mathsf{from}([\ ], \mathbf{a}).\theta' \sqsubseteq \theta''\}$$

this is slightly less straightforward to verify. Note that the first of these loops constructs the set

$$\{\theta' \mid \exists\theta'' \in \mathsf{from}(\theta, \mathbf{a}).\theta' \sqsubseteq \theta''\}$$

by adding to B every submap of any binding already in B, and the second of these loops closes this set under the $\sqcup$ operator. Therefore we can rewrite equation (B.7) as

$$\mathrm{B}' = \{\theta \dagger \theta' \mid \theta' \in \mathsf{all\_from}([\ ], \mathbf{a}, \mathcal{A}) \wedge \mathsf{compatible}(\theta, \theta')\} \tag{B.8}$$

Note that if $\theta_1 \sqsubseteq \theta_2$ then $\mathsf{all\_from}(\theta_2, \mathbf{a}, \mathcal{A}) \subseteq \mathsf{all\_from}(\theta_1, \mathbf{a}, \mathcal{A})$ as the additional information in θ_2 will lead to events in $\mathcal{A}$ being partially instantiated. Therefore, we have $\mathsf{all\_from}(\theta, \mathbf{a}, \mathcal{A}) \subseteq \mathsf{all\_from}([\ ], \mathbf{a}, \mathcal{A})$ and furthermore $\mathsf{all\_from}(\theta, \mathbf{a}, \mathcal{A}) = \{\theta' \in \mathsf{all\_from}([\ ], \mathbf{a}, \mathcal{A}) \mid \mathsf{compatible}(\theta, \theta')\}$. Therefore we have

$$\mathrm{B}' = \{\theta \dagger \theta' \mid \theta' \in \mathsf{all\_from}(\theta, \mathbf{a}, \mathcal{A})\}$$

□

Proposition 4. *The* CHECK *function produces the same output as the judgement given in Def. 43.*

Proof. We argue that the CHECK function sets up D_L appropriately and that CHECKING mirrors the structure of $\vDash$.

Recall that D_L is defined as follows.

$$D_L(x) = \begin{array}{ll} \mathcal{D}(\mathsf{typeOf}(x)) & \text{if } \mathsf{typeOf}(x) \in \mathsf{dom}(\mathcal{D}) \\ \{\theta(x) \mid \theta \in \mathsf{dom}(L) \wedge x \in \mathsf{dom}(\theta)\} & \text{otherwise} \end{array}$$

It is easy to see that the nested for-loop in CHECK computes $\{\theta(x) \mid \theta \in \mathsf{dom}(L) \wedge x \in \mathsf{dom}(\theta)\}$ for each x and we also override this with $\mathcal{D}$.

The function CHECKING follows the same recursive structure as $\vDash$. Firstly, we have the base case of $\Lambda = \epsilon$ where we take the two cases of $\theta \in \mathsf{dom}(L)$ and $\theta \notin \mathsf{dom}(L)$ as we did in Def. 43. Secondly, we explore the case where Λ is non-empty and iterate over the values in the domain of the quantified variables, carrying out the appropriate universal or existential search. □

Appendix C

Further details from monitoring evaluation

This appendix gives supplementary information for the evaluation of our monitoring techniques given in Chapter 7. We begin by giving the RuleR and JavaMOP specifications for the properties monitored in C.1. We then give additional results for the planetary rover case study in C.2 and the DaCapo benchmark suite in C.3.

C.1 Specifications

Here we give the specifications for the other tools.

C.1.1 RuleR

We give the specifications for RuleR. All monitors are defined using

```
monitor{
   uses M : <property-name>;
   run M .
}
```

The following table gives the specifications divided into those for the rover case study and the `Java` API.

Rover case study

```
ruler GrantCancel{
  observes
    grant(obj,obj),cancel(obj,obj);

  always Start(){
    grant(t:obj,r:obj) -> G(t,r);
  }
  state G(t:obj,r:obj){
    cancel(t,r) -> Ok;
    grant(s:obj,r) -> Fail;
  }
  initials Start;
}
```

```
ruler HashCorrect{
  observes send(x:obj,y:obj,m:int),
        ack(x:obj,y:obj,h:int);

  always Start(){
    send(x:obj,y:obj,m:int) -> A(x,y,m);
  }
  state A(x:obj,y:obj,m:int){
    ack(y,x,h:int), h = -m -> Ok;
  }
  initials Start;
  forbidden A;
}
```

```
ruler ReleaseResource{
  observes
    schedule(obj,obj), grant(obj,obj),
    cancel(obj,obj), finish(obj);

  always Start(){
    schedule(t:obj,c:obj) -> S(t,c);
  }
  state S(t:obj,c:obj){
    grant(t,r:obj) -> G(t,c,r), S(t,c);
    finish(c) -> Ok;
  }
  state G(t:obj,c:obj,r:obj){
    cancel(t,r) -> Ok;
  }
  assert Start, S, G;
  initials Start;
}
```

```
ruler RespectConflicts{
  observes
    conflict(obj,obj),
    grant(obj), cancel(obj);

  always Start(){
    conflict(x:obj,y:obj) -> C(x,y);
  }
  always C(x:obj,y:obj){
    grant(x) -> N(x,y);
    grant(y) -> N(y,x);
  }
  state N(x:obj,y:obj){
    cancel(x) -> Ok;
    grant(y) -> Fail;
  }
  initials Start;
}
```

```
ruler RespectPriorities{
  observes
    priority(obj,obj),
    request(obj), grant(obj),
    cancel(obj), rescind(obj), blank;

  always Start(){
    priority(x:obj,y:obj) -> P(x,y);
    grant(x:obj) -> G(x);
  }
  state P(x:obj,y:obj){ blank -> Ok;}
  always G(x:obj){
    cancel(x) -> !G(x);
    request(y:obj), P(y,x)
    {:
      P(z:obj,y),G(z) -> Ok;
      default -> Res(x,y);
    :}
  }
  state Res(x:obj,y:obj){
   rescind(x) -> Ok;
   grant(y) -> Fail;
  }
  initials Start;
}
```

```
ruler CommandAcks{
  observes
    set_ack_timeout(int), ack(obj,int),
   com(obj,obj,obj,int);

  step Start(){
    set_ack_timeout(t:int) -> T(t);
  }
  always T(t:int){
    com(c:obj,n:obj,p:obj,st:int)
             -> A(c,n,p,st+t);
  }
  state A(c:obj,n:obj,p:obj,t:int){
    ack(c,rt:int), rt < t -> Ok;
    com(c2:obj,n,p,st:int), st > t -> Ok;
  }
  initials Start;
  forbidden A;
}
```

```
ruler IncreasingIdentifiers{
  observes command(int);

  state C(x:int){
    command(y:int), y>x -> C(y);
  }
  assert C;
  initials C(0);
}
```

```
ruler ExactlyOneSuccess{
  observes
    com(obj),  suc(obj),  fail(obj);

  always Start(){
    com(x:obj) -> A(x);
  }
  state A(x:obj){
    suc(x) -> D(x);
    fail(x) -> Fail;
  }
  state D(x:obj){
    suc(x) -> Fail;
  }
  initials Start;
  forbidden A;
}
```

```
ruler NestedCommand{
  observes
    com(obj),suc(obj);

  always Start(){
    com(x:obj), !C(x) -> C(x);
  }
  state C(x:obj){
    com(y:obj), !D(x,y) -> D(x,y), C(x);
    suc(x), !D(x,y:obj) -> Ok;
  }
  state D(x:obj,y:obj){
    suc(y) -> Ok;
  }
  assert Start, C, D;
  initials Start;
  forbidden C;
}
```

```
ruler ExistsSatellite{
  observes
    ping(obj,obj), ack(obj,obj);

  always Start(){
    ping(r:obj,s:obj) -> P(r,s);
  }
  state P(r:obj,s:obj){
    ack(s,r) -> Ok;
    ack(t:obj,r)-> Ok;
  }
  initials Start;
  forbidden P;
}
```

```
ruler ExistsLeader{
  observes ping(obj,obj), ack(obj,obj), d;

  always Start(){
    ping(r:obj,s:obj), !A(r,s) -> P(r,s);
    ping(r:obj,s:obj), !R(r) -> R(r), NL;
    ping(r:obj,s:obj), !R(s) -> R(s), NL;
  }
  state NL(){
    ack(s:obj,r:obj), R(t:obj), t!=s,
          !A(r,t) -> NL;
    ack(s:obj,r:obj) -> Ok;
  }
  always R(r:obj){
    d -> Ok;
  }
  state P(r:obj,s:obj){
    ack(s,r) -> A(r,s);
  }
  always A(r:obj,s:obj){
    d -> Ok;
  }
  initials Start, NL;
  forbidden NL;
}
```

```
ruler ResourceLifecycle{
  observes
    request(obj), deny(obj),
    grant(obj), cancel(obj),
    rescind(obj);

  always Start(){
    request(r:obj) -> R(r);
  }
  state R(r:obj){
    deny(r)  -> Ok;
    grant(r) -> G(r);
  }
  state G(r:obj){
    rescind(r) -> G(r);
    cancel(r)  -> Ok;
  }
  assert Start, R, G;
  initials Start;
  forbidden G;
}
```

Java API

```
ruler HasNext{
 observes  hasNext(obj), next(obj),
               end(obj);

    always Start {
      hasNext(i:obj) -> Next(i);
      end(i:obj) -> !Next(i);
    }
    state Next(i:obj) {
      next(i) -> Ok;
    }
    assert Start, Next;
    initials  Start;
}
```

```
ruler UnsafeIter{
 observes  iterator(obj,obj),
            update(obj), use(obj);

    always Start {
      iterator(c:obj,i:obj) -> I(c,i);
    }
    state I(c:obj,i:obj){
      update(c) -> U(i);
    }
    state U(i:obj) {
      use(i) -> Fail;
    }
    initials  Start;
}
```

```
ruler UnsafeMapIter{
 observes  connect(obj,obj),  update(obj),
         iterator(obj,obj), next(obj);

    always Start {
      connect(c1:obj,c2:obj) -> C(c1,c2);
    }
    always C(c1:obj,c2:obj){
      iterator(c2,i:obj) -> I(c1,c2,i);
    }
    state I(c1:obj,c2:obj,i:obj){
      update(c1) -> U(i);
    }
    state U(i:obj) {
      next(i) -> Fail;
    }
    initials  Start;
}
```

```
ruler UnsafeSyncCollection{
 observes  async_iterator(obj,obj),
    sync_iterator(obj,obj), create(obj),
    async_use(obj,obj);

    always Start {
      create(c:obj) -> C(c);
    }
    always C(c:obj) {
      async_iterator(c,i:obj) -> Fail;
      sync_iterator(c,i:obj) -> S(c,i);
    }
    state S(c:obj,i:obj){
      async_use(c,i) -> Fail;
    }
    initials  Start;
}
```

```
ruler UnsafeSyncMap{
 observes  create(obj), create_set(obj,obj),
         async_iterator(obj,obj),
         sync_iterator(obj,obj),
         async_use(obj,obj,obj);

    always Start {
      create(m:obj) -> M(m);
    }
    always M(m:obj){
      create_set(m,c:obj) -> C(m,c);
    }
    always C(m:obj,c:obj) {
      async_iterator(m,c,i:obj) -> Fail;
      sync_iterator(m,c,i:obj) -> S(m,c,i);
    }
    state S(m:obj,c:obj,i:obj){
      async_use(m,c,i) -> Fail;
    }
    initials  Start;
}
```

```
ruler ConsistentHashes{

 observes  add(obj,obj,long),
     observe(obj,obj,long), remove(obj,obj);

    always Start(){
add(c:obj,o:obj,h:long) -> C(c,o,h);
    }
    state C(c:obj,o:obj,h:long){
        remove(c,o) -> Ok;
        observe(c,o,t:long), h!=t -> Fail;
    }
    assert State, C;
    initials Start;
}
```

```
ruler LockOrdering{
 observes  lock(obj), unlock(obj), dum;

    always Start(){
        lock(l:obj) -> L(l);
    }
    state L(l1:obj){
  unlock(l1) -> Ok;
  lock(l2:obj) -> Order(l1,l2);
  lock(l2:obj), Order(l2,l1) -> Fail;
    }
    always Order(l1:obj,l2:obj){
        dum -> Ok;
    }
    initials Start;
}
```

```
ruler CloseFiles{

 observes enter(obj),exit(obj),
              open(obj),close(obj) ;

    always Start(){
       enter(t:obj), !In(t,d:int) ->
                     In(t,1);
    }
    state In(t:obj,d:int){
   enter(t) -> In(t,d+1);
   exit(t) -> In(t,d-1);
           open(t,f:obj) -> Open(t,f,d);
    }
    state Open(t:obj,f:obj,d:int){
           close(t,f), In(t,d) -> Ok;
           close(t,f), !In(t,d) -> Fail;
    }
    assert Start, In, Open;
    initials Start;
}
```

```
ruler UnsafeFileWriter{

 observes open(obj), write(obj),
               close(obj);

    always Start {
      open(f:obj), !Open(f) -> Open(f);
    }
    state Open(f:obj) {
      close(f) -> Ok;
      write(f) -> Open(f);
    }
    assert Start, Open;
    initials  Start;
}
```

C.1.2 JavaMOP

We now consider the same specifications for JavaMOP. Note that not all specifications from the rover case study can be captured with JavaMOP.

Rover case study

```
GrantReleaseMOP(Object t1, Object t2,
                     Object r) {

// event specification

   fsm:
   start [
      grant_t1 -> granted_t1
      grant_t2 -> granted_t2
      cancel_t1 -> error
      cancel_t2 -> error
   ]
   granted_t1 [
      cancel_t1 -> start
      grant_t2 -> error
   ]
   granted_t2 [
      cancel_t2 -> start
      grant_t1 -> error
   ]
   error [ ]

   @error{ logError(); }
}
```

```
ExactlyOneSuccessMOP(Object c) {

// event specification

   fsm:
   start [
      command -> sent
   ]
   sent [
      succeed -> succeeded
      failure -> error
   ]
   succeeded [
      succeed -> error
   ]
   error [

   ]

   @fail{ logError(); }
   @error{ logError("A command
      failed or succeeded twice"); }
}
```

```
NestedCommandMOP(Object x, Object y) {

// event specification

   fsm:
   start [
      com_x -> xcom
      com_y -> ycom
   ]
   xcom [
      com_y -> xycom
      suc_x -> start
   ]
   ycom [
      com_x -> yxcom
      suc_y -> start
   ]
   xycom [
      suc_y -> xcom
   ]
   yxcom [
      suc_x -> ycom
   ]

   @fail{
      if(saved_x!=saved_y) logError();
   }
}
```

```
ReleaseResourceMOP(Object t, Object c,
                          Object r) {

\\ event specification

   fsm:
   start [
      schedule -> scheduled
   ]
   scheduled [
      grant -> granted
      finish -> end
   ]
   granted [
      cancel -> scheduled
      finish -> error
   ]
   error [

   ]
   end [

   ]

   @fail{ logError(); }
   @error{ logError("finished with
         resource granted"); }
}
```

```
ResourceLifecycleMOP(Object r) {

\\ event specification

   fsm:
   start [
      request -> requested
   ]
   requested [
      deny -> start
      grant -> granted
   ]
   granted [
      rescind -> granted
      cancel -> start
   ]

   @fail{ logError(); }
}
```

```
RespectConflictsMOP(Object r1, Object r2) {

\\ event specification

   fsm:
   start [
      conflict -> conflicted
   ]
   conflicted [
      conflict -> conflicted
      grant_r1 -> r1_granted
      grant_r2 -> r2_granted
   ]
   r1_granted [
      release_r1 -> conflicted
      grant_r2 -> error
   ]
   r2_granted [
      release_r2 -> conflicted
      grant_r1 -> error
   ]
   error [ ]

   @error{ logError(); }
}
```

Java API

```
HasNext(Iterator i) {
   event hasnext after(Iterator i) : call(* Iterator.hasNext()) && target(i) {}
   event next before(Iterator i) :  call(* Iterator.next()) && target(i) {}

   ere : (hasnext+ next)* next

   @match { logError(); }
}
```

```
UnsafeIter(Collection c, Iterator i) {
   event create after(Collection c) returning(Iterator i) :
         call(Iterator Collection+.iterator()) && target(c) {}
   event updatesource after(Collection c) : (call(* Collection+.remove*(..))
         || call(* Collection+.add*(..)) ) && target(c) {}
   event next before(Iterator i) : call(* Iterator+.next()) && target(i) {}

   ere : create next* updatesource updatesource* next

   @match { logError(); }
}
```

```
full-binding UnsafeMapIter(Map map, Collection c, Iterator i){
   event createColl after(Map map)  returning(Collection c) :
         (call(* Map+.values()) || call(* Map+.keySet())) && target(map) {}
   event createIter after(Collection c)  returning(Iterator i) :
         call(* Collection+.iterator())  && target(c) {}
   event useIter before(Iterator i) : call(* Iterator+.next()) && target(i) {}
   event updateMap after(Map map) :
         (call(* Map+.put*(..)) || call(* Map+.putAll*(..))
         || call(* Map+.clear()) || call(* Map+.remove*(..)))
         && target(map) {}

   ere : createColl updateMap* createIter useIter* updateMap updateMap* useIter

   @match{ logError(); }
}
```

```
UnsafeSyncCollection(Object c, Iterator iter) {
   Object c;

   creation event sync after() returning(Object c) :
         call(* Collections.synchr*(..)) {this.c = c;}
   event syncCreateIter after(Object c) returning(Iterator iter) :
         call(* Collection+.iterator()) && target(c) && if(Thread.holdsLock(c)){}
   event asyncCreateIter after(Object c) returning(Iterator iter) :
         call(* Collection+.iterator()) && target(c)  && if(!Thread.holdsLock(c)){}
   event accessIter before(Iterator iter) :
         (call(* Iterator+.*(..)) || call(* Iterator+.*())) && target(iter)
         && condition(!Thread.holdsLock(this.c)) {}

   ere : (sync asyncCreateIter) | (sync syncCreateIter accessIter)

   @match{ logError(); }
}
```

```
UnsafeSyncMap(Map syncMap, Set+ mapSet, Iterator iter) {
   Map c;

   creation event sync after() returning(Map syncMap) :
      call(* Collections.synchr*(..)) {
         this.c = syncMap;
      }
   event createSet after(Map syncMap) returning(Set+ mapSet) :
      call(* Map+.keySet()) && target(syncMap) {}
   event syncCreateIter after(Set+ mapSet) returning(Iterator iter) : target(mapSet) &&
      call(* Collection+.iterator()) &&  condition(Thread.holdsLock(c)){}
   event asyncCreateIter after(Set mapSet) returning(Iterator iter) : target(mapSet) &&
      call(* Collection+.iterator()) && condition(!Thread.holdsLock(c)) { }
   event accessIter before(Iterator iter) :  target(iter) &&
      (call(* Iterator+.*(..)) || call(* Iterator+.*()))
      && condition(!Thread.holdsLock(c)) {}

   ere : sync createSet (asyncCreateIter | (syncCreateIter accessIter))

   @match{ logError(); }
}
```

```
UnsafeHashSet(HashSet t, Object o) {
   int hashcode;

   event add after(HashSet t, Object o) : call(* Collection+.add(Object)) &&
    target(t) && args(o) {
      hashcode = o.hashCode();
    }
   event unsafe_contains before(HashSet t, Object o) : target(t) && args(o) &&
     call(* Collection+.contains(Object)) &&  condition(hashcode != o.hashCode()) {}
  event remove after(HashSet t, Object o) : target(t) && args(o) &&
     call(* Collection+.remove(Object)) {}

  ere: add unsafe_contains unsafe_contains*

  @match{ logError(); }
}
```

```
UnSafeHashMap(HashMap m, Object o) {
  int hashcode;

  event add after(HashMap m, Object o) :
    call(* Map+.put(Object,Object)) && target(m) && args(o,*) {
      hashcode = o.hashCode();
    }
  event unsafe_contains_key  before(HashMap m, Object o) :
    call(* Map+.containsKey(Object)) && target(m) && args(o)
    && condition(hashcode != o.hashCode()) {}
  event unsafe_get_key before(HashMap m, Object o) :
    call(* Map+.get(Object)) && target(m) && args(o)
    && condition(hashcode != o.hashCode()) {}

  event remove after(HashMap m, Object o) :
    call(* Map+.remove(Object))
    && target(m) && args(o){}

  ere: add (unsafe_contains | unsafe_get_key) (unsafe_contains | unsafe_get_key)*

  @match{ logError(); }
}
```

```
CloseFiles(FileReader f, Thread t) {
   event open after(Thread t) returning(FileReader f) :
      call(FileReader.new(..)) && thread(t){}
   event close after(FileReader f, Thread t) :
      call(* FileReader.close(..)) && target(f) && thread(t){}
   event begin before(Thread t) : execution(* *.*(..)) && thread(t) {}
   event end after(Thread t) : execution(* *.*(..)) && thread(t) {}

   cfg :
      S -> A S | epsilon,
      A -> A begin A end | A open A close | epsilon

   @fail { logError();}
}
```

```
UnsafeFileWriter(FileWriter f) {
   event open after() returning(FileWriter f) : call(FileWriter+.new(..)) {}
   event write before(FileWriter f) : call(* FileWriter+.write(..)) && target(f) {}
   event close after(FileWriter f) : call(* FileWriter+.close()) && target(f) {}

   ere : (open write write* close)*

   @fail { logError(); }
}
```

Table C.4: A summary of the workloads used for each property.

Name	Parameters	$\lvert\tau\rvert$	bindings
GrantCancelS	t tasks, r resources and e events	e	r
GrantCancelD	t tasks, r resources and e events	e	tr
ResourceLifecycle	r resources and e events	e	r
ReleaseResource	t tasks, r resources and e events	e	$\frac{rte}{4}$
RespectConflictsS	g resource groups, r reps of e events	re	g^2
RespectConflictsD	g resource groups, r reps of e events	re	g^4
RespectPriorities	r resources and e events	e	r^2
ExactlyOneSuccess	c commands	$2c$	c
IncreasingIdentifiers	e events	e	1
CommandAcks	c commands	$2c$	$\frac{3c}{2}$
NestedCommands	r reps nested d deep and b wide	$\frac{2rb(1-d^d)}{1-d}$	$\frac{\frac{\lvert\tau\rvert}{2}\left(1-\left(\frac{\lvert\tau\rvert}{2}\right)^d\right)}{1-\frac{\lvert\tau\rvert}{2}}$
NestedCommandsG	r reps nested d deep and b wide	$\frac{2rb(1-d^d)}{1-d}$	$\frac{\frac{\lvert\tau\rvert}{2}\left(1-\left(\frac{\lvert\tau\rvert}{2}\right)^d\right)}{2-\lvert\tau\rvert}$
ExistsSatellite	s satellites and r rovers	sr	$\frac{sr}{2}$
ExistsSatelliteS	s satellites and r rovers	sr	r
ExistsLeader	r rovers	$r+r^2$	$r+r^2$
HashCorrect	x entities and m messages	m	$\frac{m}{2}$

C.2 Additional rover case study monitoring results

This section gives additional details and results supporting the evaluation in Section 7.1. We begin by giving extra details about the workloads and properties used before giving additional results for each research question.

C.2.1 Workload and property details

Table C.4 gives an overview of the workload used for each property, along with the parameters for that workload and the average trace length and number of bindings in terms of those parameters. The trace lengths and number of bindings are average due to random choices made in the workloads. The table accounts for the binding redundancy elimination methods of Section 6.4. The trace length and number of bindings is particularly complicated for the NestedCommands workloads due the nested structure.

Table. C.5 gives the trivial events (and reminds us of the total number of events) for each property. In many cases these trivial events match with nontrivial events, i.e. $\texttt{cancel}(t_2, r)$ matches with $\texttt{cancel}(t_1, r)$. This means that, even though these events are trivial, they can never be used in redundancy elimination.

C.2.2 RQ1

For the single indexing technique evaluated in Section 7.1.3 we left out results for some properties for space reasons. We give these here. The properties we consider are ExactlyOneSuccess, ExistsSatelliteS, GrantCancelS and RespectConflictS.

Let us consider the ExactlyOneSuccess property. The workload for this property consists of randomly creating commands, up to some limit c, and randomly recording them successful. The graphs in Fig. C.1 give the results for different values of c, i.e. the number of commands, note the log scale. The single quantifier and symbol-based indexing strategies perform similarly, with the value-based approach performing significantly worse with many commands. Again, this is due to the large number of bindings

Table C.5: Trivial events per property

Property	Trivial events	$\|\mathcal{A}\|$
GrantCancelS	$\{\texttt{cancel}(t_2, r)\}$	3
GrantCancelD	$\{\texttt{grant}(t_2, r)\}$	2
ResourceLifecycle	$\{\}$	5
ReleaseResource	$\{\texttt{finish}(c)\}$	4
RespectConflictsS	$\{\texttt{cancel}(r_1), \texttt{grant}(r_2)\}$	5
RespectConflictsD	$\{\texttt{grant}(r_1), \texttt{cancel}(r_1), \texttt{grant}(r_2)\}$	5
RespectPriorities	$\{\texttt{request}(r_2), \texttt{deny}(r_2), \texttt{grant}(r_2), \texttt{cancel}(r_2),$ $\texttt{cancel}(\_), \texttt{request}(\_), \texttt{grant}(\_)\}$	10
ExactlyOneSuccess	$\{\texttt{succeed}(c), \texttt{fail}(c)\}$	3
IncreasingIdentifiers	$\{\texttt{command}(c_2)\}$	2
CommandAcks	$\{\texttt{command}(c_2, n_2, p_2, t_2)\}$	4
NestedCommands	$\{\}$	4
ExistsSatellite	$\{\texttt{ack}(s, r)\}$	3
ExistsSatelliteS	$\{\texttt{ack}(s, r)\}$	3
ExistsLeader	$\{\}$	2
HashCorrect	$\{\texttt{ack}(x_2, x_1, h)\}$	2

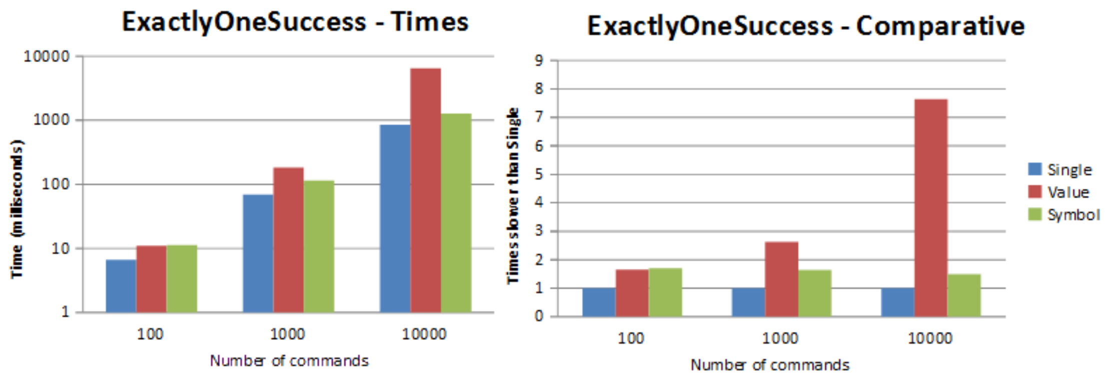


Figure C.1: The results of RQ1 with single quantifiers for ExactlyOneSuccess.

raising the cost of adding a new binding. Not shown here, the basic monitoring approach takes about 6.6 seconds to evaluate a trace with 1k commands, 95 times slower than the single-quantifier approach. On average compared to the basic monitoring strategy, the single quantifier strategy performed 37 times faster, the value-based approach ran 15 times faster and the symbol-based approach ran 22 times faster.

Let us now consider the ExistsSatelliteS property. The workload for this property consists of randomly selecting a non-empty set of satellites to acknowledge each rover and producing the appropriate `ping` and `ack` events. The graphs in Fig. C.2 give the results for different values of r and s. Again, the basic monitoring approach performs very badly with respect to the single quantifier indexing approach - although with only 10 rovers and satellites we see comparable performance. In this case the value-based indexing strategy outperforms the symbol-based one, this is due to the relatively small number of bindings combined and high reuse rate. On average compared to the basic monitoring strategy, the single quantifier strategy performed 42 times faster, the value-based approach ran 30 times faster and the symbol-based approach ran 24 times faster.

Next we consider the GrantCancelS property. The workload for this property consists of t tasks granting and canceling r resources over e events. The results for different values of $t/r/e$ are given

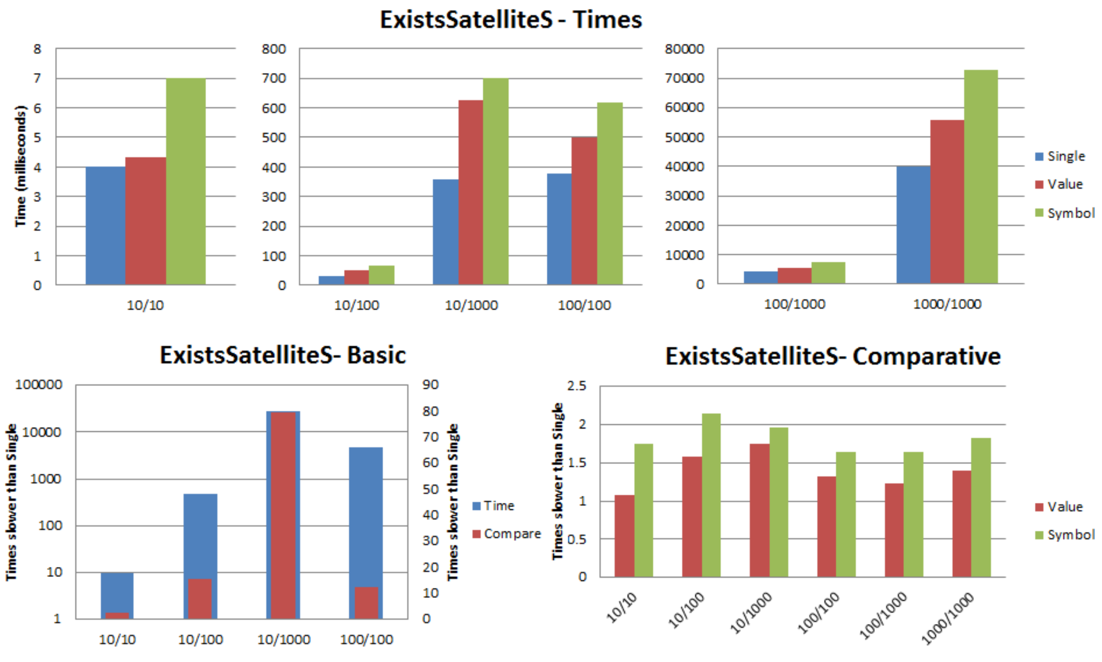


Figure C.2: The results of RQ1 with single quantifiers for ExistsSatelliteS.

in Fig. C.4. Here we have another case of the value-based approach outperforming the symbol-based one, however the single quantifier algorithm still performs 2-3 times faster than either approach. On average compared to the basic monitoring strategy, the single quantifier strategy performed 46 times faster and both the value-based and symbol-based approaches ran 15 times faster.

The last single-quantifier property is RespectConflictsS. The workload for this property consists of r conflict groups containing r resources (giving r^2 resources in total) exercised for e events. The results for different values of r^2/e and both single and double versions of the property are given in Fig. C.4. The single quantifier strategy is the most efficient with the symbol-based approach being consistently less than twice as slow. The value-based approach does not do very well at all. This is because the value-based approach is more effected by the use of free-variables as it will need to iterate through the set of all entries, making it dependent on the number of bindings. As we can see, running time increases uniformly with the number of bindings with both the value-based and basic approaches. On average compared to the basic monitoring strategy, the single quantifier strategy performed 15 times faster, the value-based approach ran 1.5 times faster and the symbol-based approach ran 9 times faster.

We also omitted the HashCorrect property for Disjoint alphabet indexing. Let us consider this property now. The workload for this property produces x entities and m messages (and the appropriate acknowledgments) between random entities. The results for different values of x/m are given in Fig. C.5. Again, the disjoint alphabet approach outperforms all others considerably. For small numbers of entities and messages the value-based and symbol-based approaches perform similarly, with the symbol-based approach slightly outperforming the value-based one. But, when we increase the number of messages we see the performance of the value-based approach degrade heavily. Again, we see the disjoint alphabet approach achieving hundreds of times speedup on the basic monitoring approach. On average compared to the basic monitoring strategy, the single quantifier strategy performed 247 times faster, the value-based approach ran 30 times faster and the symbol-based approach ran 71 times faster.

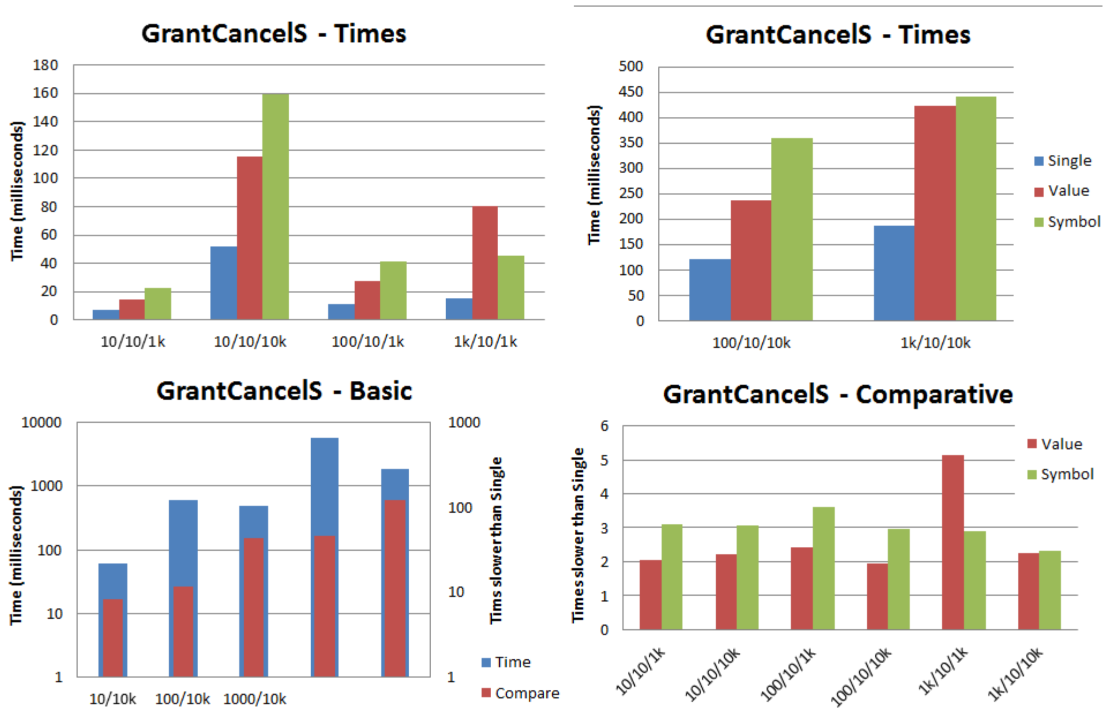


Figure C.3: The results of RQ1 with single quantifiers for GrantCancelS.

C.2.3 RQ2

Table C.6 gives the average milliseconds per event and events processed per second for each property using the Value and Symbol strategies. This table agrees with the previous table over which strategy performed the best, with the exception of GrantCancelS - here we see the Value strategy achieving higher average throughput. Looking at the results we see that in general the Value strategy gave the best speedup, but in one experiment the reuse rate is very low and the Symbol strategy ran twice as fast, brining its average speedup above that of Value.

The rest of the properties

We give detailed evaluation results for the rest of the properties. These are the GrantCancelD, ReleaseResource, RespectConflictsD, RespectPriorities, NestedCommands, CommandAcks, and ExistsLeader.

First, let us consider the GrantCancelD property. We use the same workload as with GrantCancelS and the results for different values of $t/r/e$ are given in Fig. C.6. The symbol-based indexing strategy far outperforms the value-based approach due to the significant number of bindings being produced. The speedup increases with the number of bindings. As we can see from the 10/10/1k and 10/10/10k cases the basic and value-based approaches take more time to process each event when there are more bindings.

Next we consider the ReleaseResource property. The workload for this property consists of t tasks dealing with r resources over e events. The results for different values o $t/r/e$ are given in Fig. C.7. Note that we have a logarithmic y-axis for time. Again, we see the symbol-based approach performing best, due to the large number of bindings being produced and manipulated for the basic and value-based approaches.

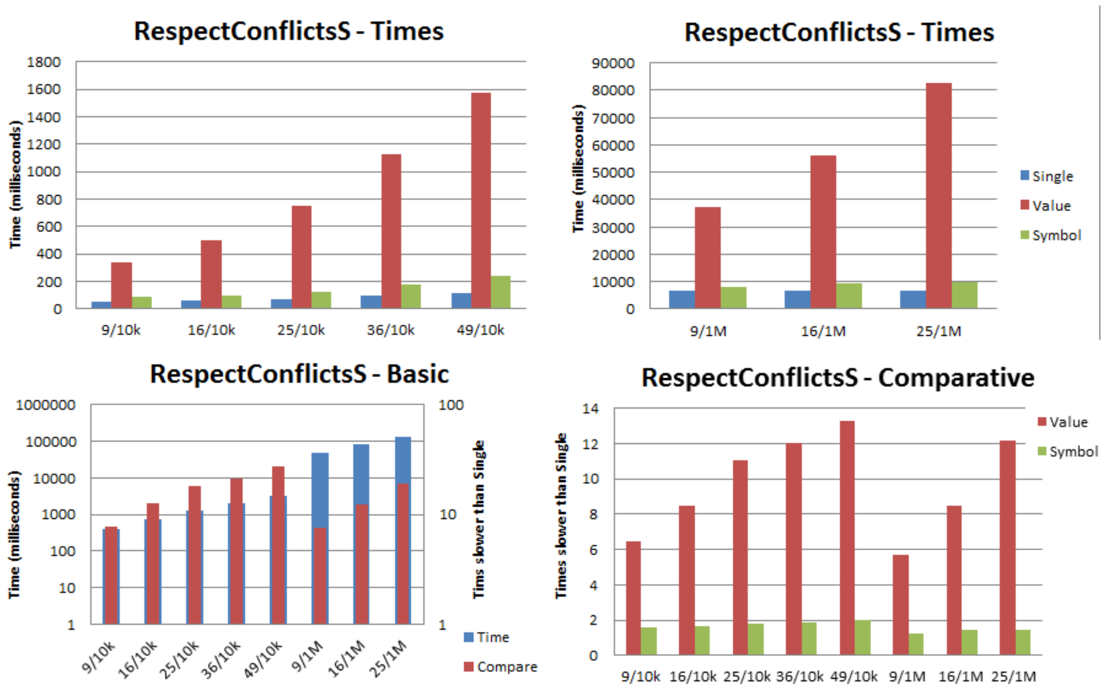


Figure C.4: The results of RQ1 with single quantifiers for RespectConflictsS.

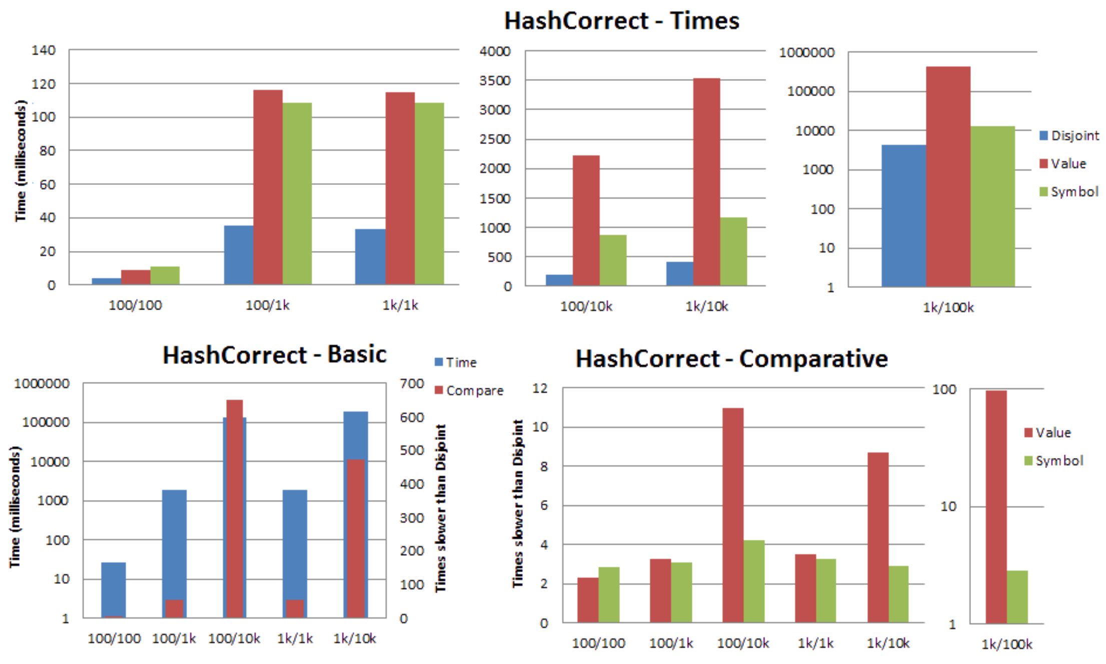


Figure C.5: The results of RQ1 for HashCorrect.

Now let us consider the RespectConflictsD property. We use the same workload as with RespectConflictS and the results for different values of r^2/e and both single and double versions of the property are given in Fig. C.8. We see the symbol-based approach outperforming the value-based approach. The speedup achieved with 1M events is relatively small compared with 10k events, this is because the base

Table C.6: Average throughput for Value and Symbol.

Property	Value		Symbol	
	ms per event	events per sec	ms per event	events per sec
IncreasingIdentifiers	**0.03**	**31.5k**	0.04	26k
ResourceLifeCycle	0.047	32k	**0.03**	**37k**
ExactlyOneSuccess	0.13	13k	**0.05**	**20k**
ExistsSatelliteS	**0.05**	**19k**	0.07	14.5k
GrantCancelS	**0.033**	**44.8k**	0.034	34k
RespectConflictS	0.08	17k	**0.01**	**91k**
ExistsSatellite	101	487	**3.99**	**787**
HashCorrect	0.87	6k	**0.2**	**9.2k**
GrantCancelD	0.17	10k	**0.07**	**22k**
ReleaseResource	8.7	1.1k	**0.77**	**2.4k**
RespectConflictsD	0.16	8.6k	**0.14**	**10k**
RespectPriorities	1.45	2.2k	**1.31**	**5.2k**
NestedCommands	6.1	380	**4.7**	**490**
CommandsAcks	**3.9**	**1.5k**	13	1.1k
ExistsLeader	**0.1**	**9.4k**	0.11	8.7k

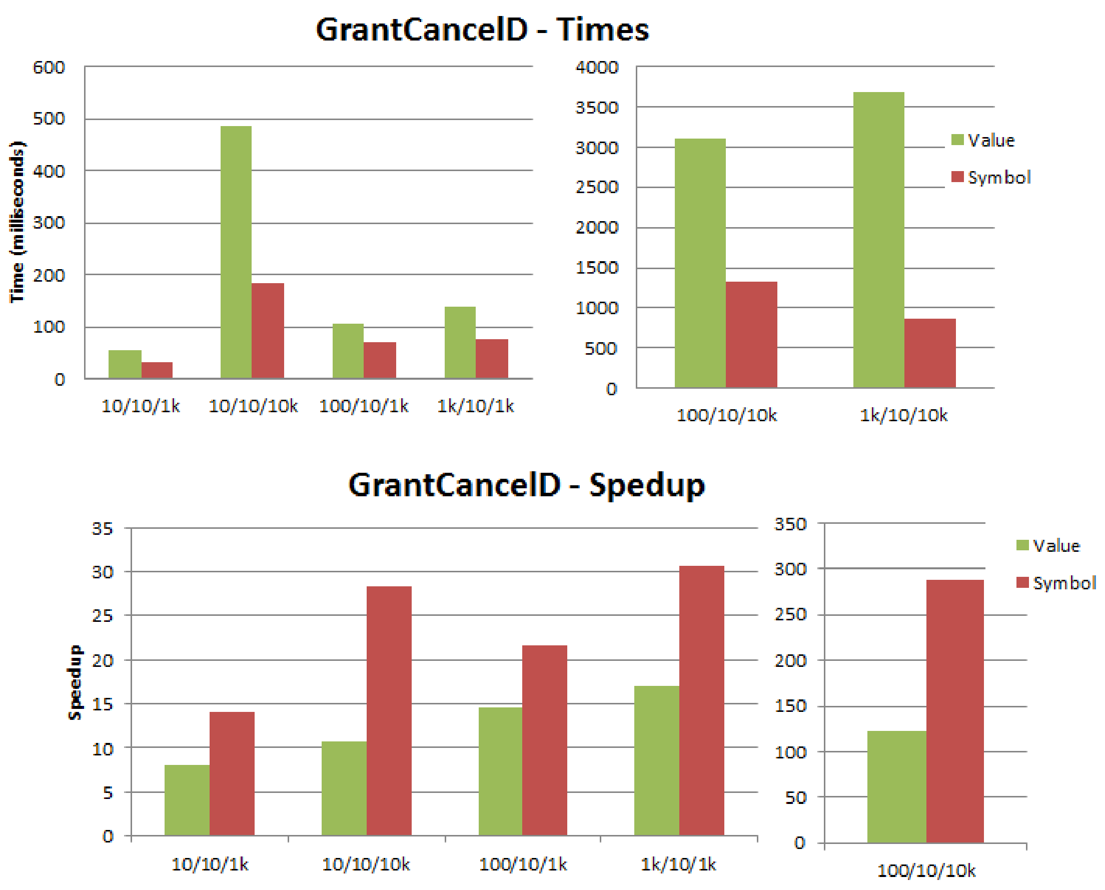
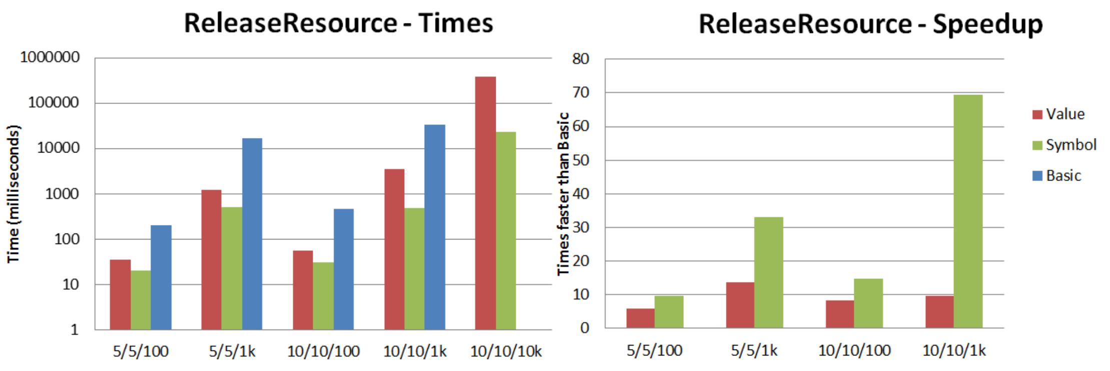


Figure C.6: The results of RQ1 for GrantCancelD.

monitoring approach scales badly with the number of bindings, which is small here.



Graph here

Figure C.7: The results of RQ1 for ReleaseResource.

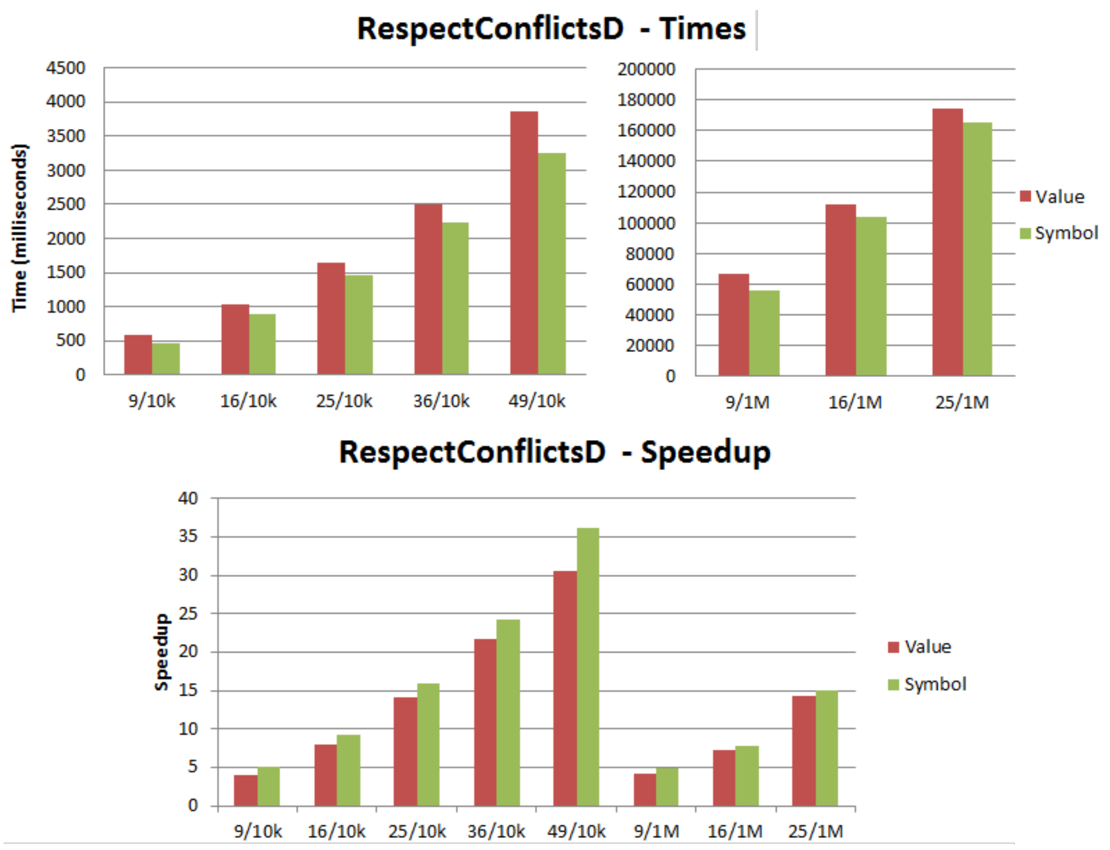


Figure C.8: The results of RQ1 for RespectConflictsD.

We now consider the RespectPriorities property - the most complex property in our set. The workload for this property considers r resources organised in a binary tree of proprieties, randomly exercised over e events. The results for different values of r/e are given in Fig. C.10. Note the logarithmic scale for times. We see little speedup with the value-based, with the symbol-based approach performing best in all but one case. In the case of 10 resources and 100k events we see that the value-based approach performs 15 times better than the symbol-based approach, yet with 20 resources and 100k events the symbol-based approach performs 1.6 times better. The number of bindings is quadratic

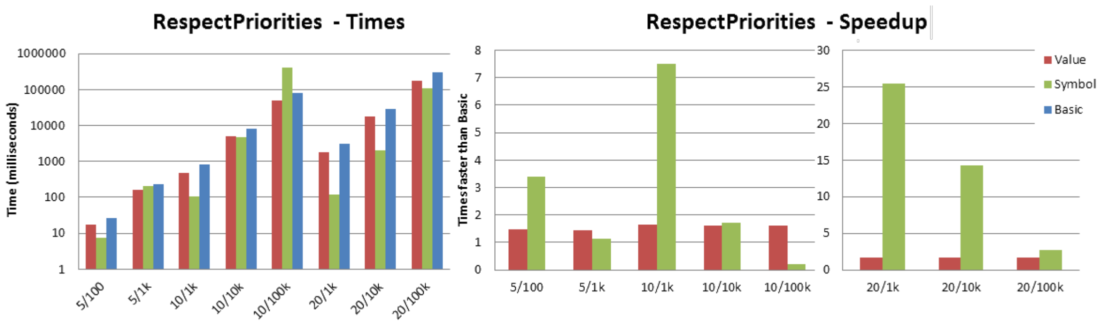


Figure C.9: The results of RQ1 for RespectPriorities.

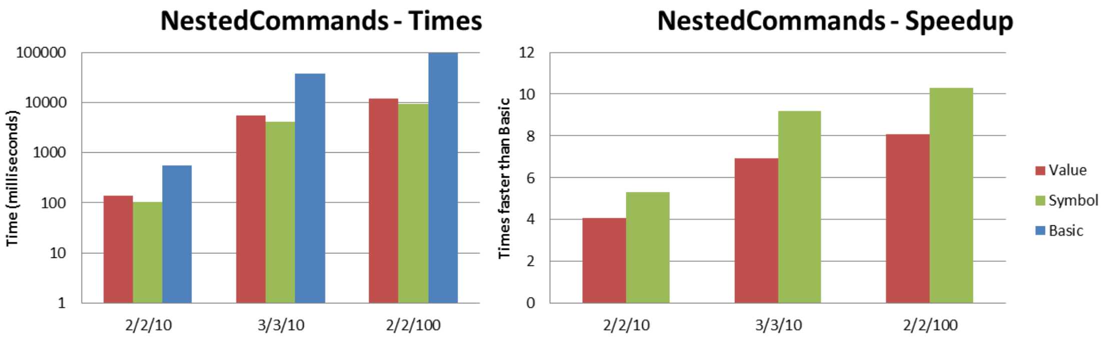


Figure C.10: The results of RQ1 for NestedCommands.

in r so there are 4 times as many bindings in the second case, which appears to make the difference.

We also see that speedup decreases significantly as the length of the trace increases for the symbol-based approach, but remains reasonably constant for the value-based approach. This is because the basic and value-based are effected by the increase in trace length in the same way, and that the trace length is more important than the number of bindings for the symbol-based approach.

Next we consider the NestedCommands property. The workload for this property consists of a command structure nested d deep and b wide, repeated r times. The results for different values of $d/b/r$ are given in Fig. C.10. Note the logarithmic scale for times, and recall that garbage collection is relevant for this example (this is discussed further in Sec. 7.1.5). The symbol-based approach performs the best, with speedup increasing with additional bindings and events.

Next we consider the ExistsLeader property. The workload for this property consists of r rovers communicating with (on average) half of the other rovers, with one selected as leader - this gives roughly X events. The results for different values of r are given in Fig. C.11. The value-based approach performs best here and both indexing strategies achieve good speedup. The value-based approach works well as there are relevantly few bindings.

C.2.4 RQ3

We give the omitted analysis of the ResourceLifecycle garbage frequency experiment.

Figure C.12 present the results of for ResourceLifecycle - we plot against *running time* because running without garbage collection led to timeouts. We test garbage frequency of every 1, 10, 100 and 1000 events.

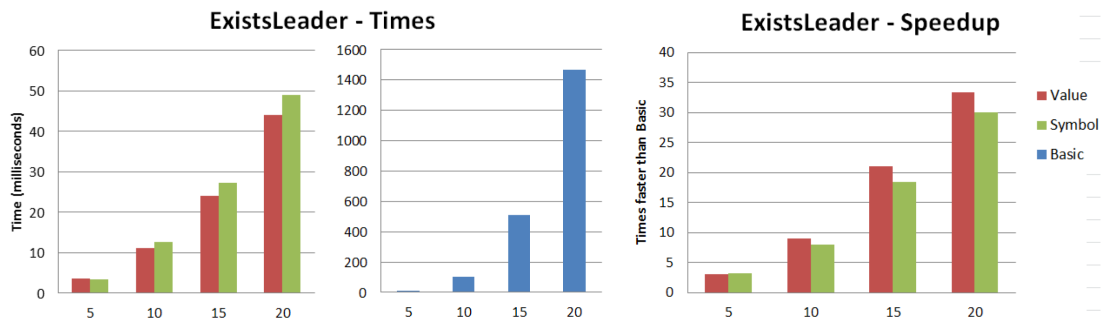


Figure C.11: The results of RQ1 for ExistsLeader.

The ResourceLifecyle workload produces a less garbage than NestedCommands as it uses one resource object at a time and then lets it go out of scope. In Figure C.12 we see a frequency of 100 performing well across both strategies for the less complex workloads. However, with the more complex workloads we see a frequency of 1 working best for Value again and a frequency of 10 performing well for Symbol. The most likely explanation here is that with longer running experiments the number of resources that become garbage over the lifetime of the experiment increases, therefore the likelihood that there will be garbage to remove increases.

C.3 Additional DaCapo monitoring results

We now give additional information about the evaluation carried out in Section 7.2. We first give additional information about the traces observed, and then report monitoring times.

C.3.1 Trace statistics

We report on the number of events present in a *single run* of each benchmark for each property. If a benchmark has no events for a property it is not included in tables. Note that we monitor multiple runs as we wait for convergence of running time.

UnsafeIter and UnsafeMapIter

Tables C.8 and C.8 gives statistics for the UnsafeIter and UnsafeMapIter properties. For UnsafeIter we also consider the number of collections that have more than one iterator created from them.

The first thing an observant reader will note is the discrepancy between the number of iterator creation events and the number of iterators reported for the HasNext property for some benchmarks. This is most notable in fop where we have over 70k iterators for HasNext and only 7k for UnsafeIter. This is because of custom iterator objects that are created using methods other than the standard `iterator` method. To catch these we should target constructors of objects inheriting `Iterator` - however, it may not, in general, be possible to identify the underlying collection. Note however, that the number of `use` events is the same - therefore we have many `use` events without an associated iterator creation event.

Note that the collections being updated are not necessarily those being iterated over - these are just method calls that update a collection. Also note that the `update` event for UnsafeMapIter is for maps, not collections as it is for UnsafeIter, hence the discrepancy. It is necessary to capture all `update` events as we cannot tell whether they are related to an iterator from the method call.

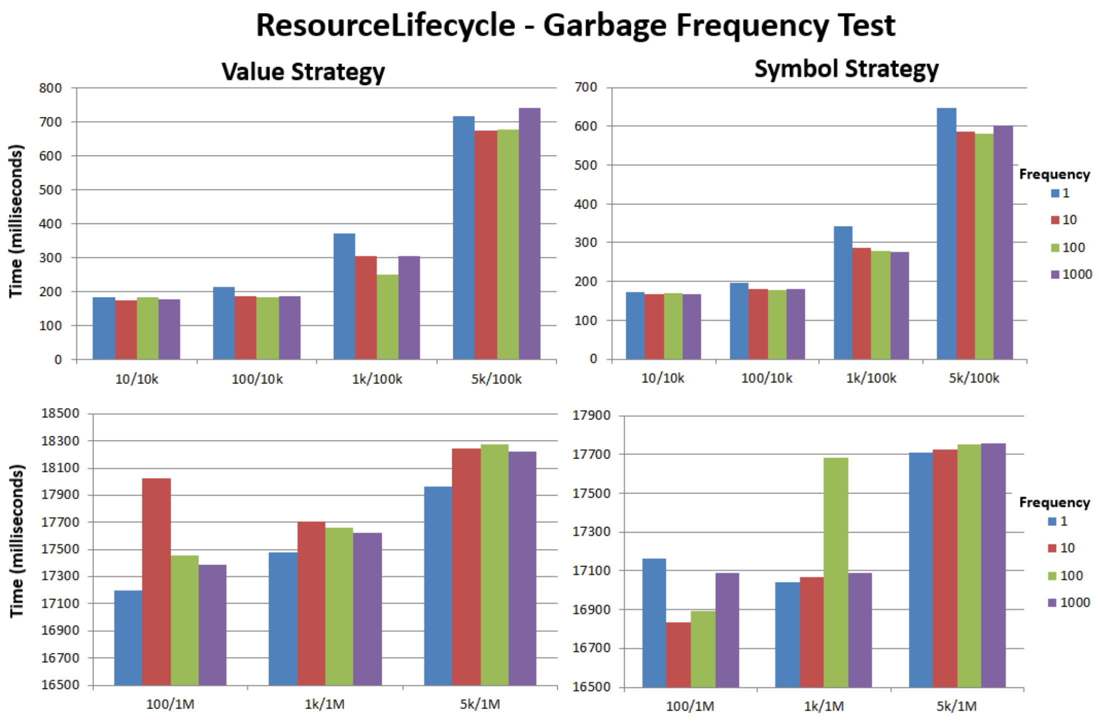


Figure C.12: Garbage frequency test with ResourceLifecycle - lower is better.

For both properties we have three benchmarks without any relevant events - tomcat, tradebeans and tradesoap - and one benchmark with only `update` events so no bindings will be created (due to the optimisation described in Sec. 6.4.5).

The UnsafeSyncCollection and UnsafeSyncMap properties

Table. C.9 gives the trace statistics for these two properties. We see many use events and relatively few `sync` and `create` events. This means that there is not much relevant behaviour associated with either property.

The ConsistentHash property

Table C.10 gives the number of events relevant to the ConsistentHash property, this combines information about `HashSet` and `HashMap`. In general, objects added to these structures are observed and removed frequently i.e. the ratio of `add` events to other events is low.

The LockOrdering property

Table. C.11 gives the number of each events, the minimum, maximum and mean number of nested locks (i.e. how many are held at once) and the minimum, maximum and mean lock lifetime. These are all due to `synchronized` keyword, thus mutual exclusion is guaranteed by the language. None of the benchmarks use explicit `Lock` objects.

Table C.7: The trace statistics for the UnsafeIterator property.

	Events				Collections with
	`iterator`	`update`	`use`	**`total`**	> 1 Iterator
avrora	908,977	1,015,243	1,507,546	3,431,766	531
batik	24,354	112,864	48,755	185,973	46
eclipse	3,894	24,659	146,138	174691	246
fop	7,744	252,108	1,800,197	2,060,049	308
h2	2,872	2,377,203	26,546,017	28,926,092	5
jython					
luindex	64	4,293	365	4,722	9
lusearch	128	373,540	640	374,308	16
pmd	589,753	3,258,673	8,612,254	12,460,680	10554
sunflow	0	134	2,655,751	2,655,885	0
xalan	0	1,309,143	0	1,309,143	0

Table C.8: The trace statistics for the UnsafeMapIter property.

	Events				
	`connect`	`iterator`	`update`	`next`	**`total`**
avrora	22	908,976	1,794	350,595	1,261,387
batik	42	24,354	17,560	12,559	54,515
eclipse	1,051	3,891	11,689	60,472	77,103
fop	60	7,744	29,788	461,585	499,177
h2	34	2,811	1,634,141	10,004,450	11,641,436
jython	29	63,174	18,516	256,439	338,158
luindex	2	64	136	151	353
lusearch	64	128	9,177	256	9,625
pmd	110,113	593,863	43,707	3,640,109	4,387,792
sunflow	0	0	0	1,276,938	1,276,938
xalan	0	0	130,142	0	130,142

Table C.9: The trace statistics for the UnsafeSyncCollection and UnsafeSyncMap properties.

	Events				
	`create`	`async`	`use`	`sync`	**total**
avrora	0	5,453,810	9,040,956	0	14,494,766
batik	27	218,975	434,348	27	653,377
eclipse	23	15,107	277,392	1	292,523
fop	30	54,172	13,773,965	0	13,828,167
h2	1	15,243	40,141,937	0	40,157,181
jython	104	81,531	4,343,841	0	4,425,476
luindex	0	896	5,110	0	6,006
lusearch	0	1,280	6,400	0	7,860
pmd	≥				
sunflow			15,934,506		15,934,506

	Events					
	`create`	`create_set`	`async`	`use`	`sync`	**total**
avrora	0	72	62	9,041,190	62	9,041,386
batik	0	336	3,704	386,150	3704	393,894
eclipse	1	336	8,833	222,726	8,833	240,729
fop	26	480	986	15,741,666	968	15,744,126
h2	≥					
jython	0	41	4,618	4,343,841	4,618	4,353,118
luindex	0	28	574	5,110	574	6,286
lusearch	0	448	448	4480	448	5,824
pmd	≥					
sunflow	0	0	0	15,934,506	0	15,934,506

Table C.10: The trace statistics for the ConsistentHash property.

	Events			
	`add`	`observe`	`remove`	**total**
avrora	1,984	713,709	0	715,693
batik	10,872	23,130	459	34,461
eclipse	7,865	25,367	308	33,540
fop	29,735	359,969	0	389,704
h2				
jython	10,626	3,146,865	4	3,157,495
luindex	218	124,565	30	124,813
lusearch	9,202	1,049,474	1	1,058,677
pmd	149,039	764,939	0	913,978
xalan	29,534	462,500	0	492,034

Table C.11: The trace statistics for the LockOrdering property.

	Events		Potential Nestings			Lock Lifetimes		
	`lock`	`unlock`	min	mean	max	min	mean	max
avrora	1677965	1677965	0	6.19	12	-	-	-
batik	52441	52441	0	0.52	4	-	-	-
eclipse	118866	118864	2	4.25	10	345	5174	15672
fop	3553	3553	0	0.5	2	-	-	-
h2	≥21746120	≥21746088	1	32.45	35	-	-	-
jython	16550615	16550615	-	-	-	-	-	-
luindex	214304	214304	0	2.85	6	2514	4287	9332
lusearch	1644850	1644850	0	8.42	61	1245	1880	2803
pmd	8857	8857	0	8.76	17	567	1366	2476
sunflow	824	824	0	8.63	17	-	-	-
xalan	4467560	4467560	0	37.84	65	4788	287106	500409

C.3.2 Results

We now report the monitoring times for each tool for each property.

HasNext times

Table C.12 gives the average times reported for each tool.

The UnsafeIter and UnsafeMapIter properties

Table. C.13 gives the slowdown introduced by each monitoring approach. The first thing we note is that QEA did not perform very well at all. JavaMOP performed the best in all non-trivial cases and RuleR performed relatively well. The reason that QEA performed badly is related to the garbage collection process, as discussed previously.

The UnsafeSyncCollection and UnsafeSyncMap properties

Table C.14 gives the slowdown results for these two properties. QEA performed very badly as it was not able to remove redundant bindings properly, as we saw previously.

The ConsistentHash property

Table C.15 gives the timing for the ConsistentHash property. Note that for JavaMOP this involves monitoring two separate properties - one for `HashSet` and one for `HashMap`.

CloseFiles

Table C.16 gives the timing for the CloseFiles property. Many experiments failed to complete as we are tracking every method call.

The LockOrdering property

Table C.17 gives the timing for the LockOrdering property. This cannot be captured using JavaMOP. Many experiments fail to complete as (in many cases) we are comparing every object that is locked to every other object that is locked - as seen above, there can be many locks.

Table C.12: Timing results for HasNext

	Times (milliseconds)				
	QEA-Single	QEA-Symbol	QEA-Value	JavaMOP	RuleR
avrora	11,583	1,183,528	-	36,657	23,936
batik	1,711	3,912	4,841	2,126	3,460
eclipse	27,225	28,240	27,678	27,137	27,686
fop	1,832	617,536	-	75,747	507,564
h2	10,917	-	-	7,653	-
jython	3,203	9,201	8,918	2,958	4,492
luindex	948	1,018	1,035	925	958
lusearch	1,409	1,508	1,481	1,366	1,481
pmd	14,066	280,450	847,471	6,182	-
sunflow	5,230	45,310	43,759	2,215	22,925
tomcat	1,941	1,950	1,909	1,930	1,971
tradebeans	10,472	9,803	9,826	10,518	9,838
tradesoap	14,398	14,685	14,732	14,305	14,852
xalan	929	954	1,274	916	977

Table C.13: Slowdown results for UnsafeIter and UnsafeMapIter

Slowdown for UnsafeIter				
	JavaMOP	QEA-Symbol	QEA-Value	RuleR
avrora	**1.79**	575.01	-	2.14
batik	**1.06**	82.87	426.76	1.18
eclipse	**0.99**	1.00	0.99	1.01
fop	**2.08**	567.16	772.39	6.79
h2	**1.11**	11.51	33.39	1.79
jython	**0.59**	-	-	0.81
luindex	**1.01**	1.23	1.19	1.05
lusearch	**1.97**	34.21	31.98	2.87
pmd	**3.37**	-	-	11.40
sunflow	**1.05**	1.30	1.30	3.98
tomcat	1.12	1.02	1.03	1.03
tradebeans	1.02	1.02	1.02	1.02
tradesoap	1.05	0.86	0.86	0.84
xalan	3.00	2.69	2.36	11.20

Slowdown for UnsafeMapIter				
	JavaMOP	QEA-Symbol	QEA-Value	RuleR
avrora	**1.64**	-	-	1.59
batik	**1.06**	-	-	1.07
eclipse	**0.98**	-	-	1.01
fop	**1.74**	-	-	2.56
h2	**1.18**	-	-	1.56
jython	**0.52**	-	-	0.56
luindex	1.12	34.08	655.19	**1.09**
lusearch	**0.97**	325.28	539.64	1.02
pmd	**2.99**	-	-	4.76
sunflow	**1.03**	180.65	500.13	2.25
tomcat	1.14	1.05	1.03	1.05
tradebeans	1.03	1.03	1.01	1.00
tradesoap	0.86	0.84	0.84	0.82
xalan	1.00	1.13	1.54	1.47

Table C.14: Slowdown results for UnsafeSyncCollection and UnsafeSyncMap

UnsafeSyncCollection			
	JavaMOP	QEA-Symbol	QEA-Value
avrora	1.01	-	-
batik	1.03	324.47	-
eclipse	1.01	1.37	-
fop	1.52	-	-
h2	1.08	-	-
jython	0.53	-	-
luindex	1.14	1.11	27.78
lusearch	0.97	1.06	181.61
pmd	1.87	-	-
sunflow	1.03	191.96	842.74
UnsafeSyncMap			
avrora	1.02	-	-
batik	0.99	429	-
eclipse	0.98	-	-
fop	1.36	-	-
h2	0.96	-	-
jython	0.52	-	-
luindex	1.09	3.28	-
lusearch	0.97	30	-
pmd	1.68	-	-
sunflow	1.03	191	833

Table C.15: Timing results for ConsistentHash

	JavaMOP	QEA-Symbol	QEA-Value	RuleR
avrora	5,972	44,996	32,623	-
batik	1,614	3,542	4,938	1,396,870
eclipse	27,660	28,678	27,403	182
fop	701	21,206	255,924	1,194,726
h2	-	-	-	-
jython	2,995	62,617	46,821	-
luindex	874	4,714	3,318	385,983
lusearch	2,852	172,787	185,196	-
pmd	1,614	-	-	-
sunflow	2,260	2,423	2,357	2,393
tomcat	1,951	1,982	1,946	1,947
tradebeans	10,380	10,417	10,377	10,296
tradesoap	14,510	14,872	14,594	14,307
xalan	1,241	24,027	94,564	-

Table C.16: Timing results for CloseFiles

	JavaMOP	QEA-Symbol	QEA-Value	RuleR
avrora	-	-	-	-
batik	-	174,457	149,638	299,805
eclipse	-	27,842	28,312	33,527
fop	324,228	222,520	209,024	236,557
h2	394,970	-	-	-
jython	-	-	-	-
luindex	-	-	-	871,331
lusearch	-	-	-	-
pmd	-	1,258,037	1,209,092	-
sunflow	1,227,390	-	-	-
tomcat	-	1,901	1,967	1,878
tradebeans	-	10,455	9,985	10,405
tradesoap	-	14,670	14,257	14,322
xalan	-	-	-	-

Table C.17: Timing results for LockOrdering

	QEA-Symbol	QEA-Value	RuleR
avrora	555,147	535,619	-
batik	849,817	1,097,101	2,136
eclipse	-	-	-
fop	4,041	3,698	500
h2	-	-	-
jython	-	-	-
luindex	548,227	572,990	6,083
lusearch	-	-	-
pmd	355,474	371,699	-
sunflow	3,261	2,830	2,468
tomcat	1,991	1,958	1,954
tradebeans	9,797	9,761	9,775
tradesoap	14,472	14,459	18,188
xalan	240	252	1,625,149

Appendix D

Further details from mining evaluation

Here we describe the Designed pattern library, give the models used in the mining evaluation and the detailed results split into precision-recall, efficiency and details about extracted specifications.

D.1 The Designed Library

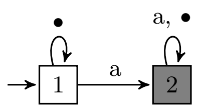
We introduce our framework of patterns, extending and refining the Specification Pattern System (SPS) [DAC99]. Note that our patterns are event-based and the categories have been chosen with the mining method in mind i.e. the fact that we will use combination. We introduce the different categories using patterns discussed previously as examples. The four categories we consider here are Occurrence, Ordering, Scope and Compound.

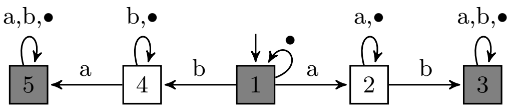
D.1.1 Occurrence

Simply, one or more event occurs (or does not) in no specified order. In SPS this category contained the absence and universal subcategories. We do not feel that these are useful categories in our setting. With respect to absence, the single event version would not informative and absence with respect to other events will be captured by Scope. The notion of event occurring universally is not useful to us.

Existence

Patterns in this subcategory capture events existing within the trace i.e. occurring one or more times. We only include two patterns here, illustrated below. The first captures that event a occurs at least once. The second captures that the events a and b either both occur or neither occur.

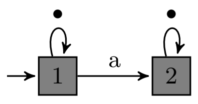


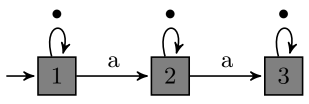


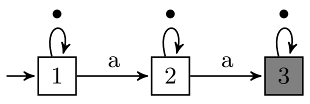
Bounded Existence

Patterns in this category refine the notion of existence, capturing events occurring at least, or at most, a given number of times. These will generally be useful in restricting other patterns. For example, we show below that the occurs at most once pattern can be used to unroll a looped pattern once.

The following patterns capture that event a can occur at most once or twice and at least twice respectively - combinations can specify an exact number of times.





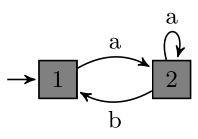


D.1.2 Ordering

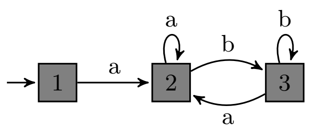
Here we consider orderings between events. As has been noted in the SPS framework, and by previous mining tools, these relationships often take the form of a preceding or causing b, or b being a response to or effect of a. We split ordering into three categories - the three being these two forms of relationship and the third being alternation, where neither cause nor effect take prominence.

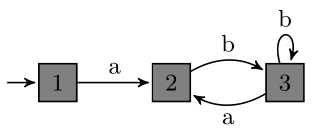
Precedence or Cause

An example of this ordering relationship would be that `hasNext` is a *precondition* for `next` when dealing with iterators, or that `press_start` *causes* `run_main`. The idea is that the effect cannot occur without the cause, but the cause can occur without the effect. The general notion of precedence given by SPS is as follows - if b occurs then a must have occurred previously.

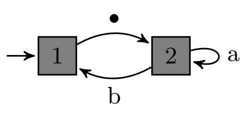


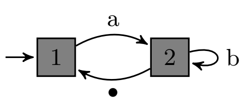
We can also allow multiple effects and single causes, as used in the Perracotta [YEB+06] mining tool.





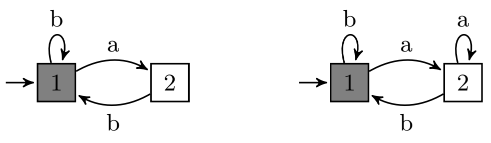
The above patterns are hole-free and, as well as considering opening extensions (Sec. 9.3.2), we can consider patterns where events are caused by external events (represented by holes). For example, the following capture that an external symbol causes possibly some as followed by b, and a causes possibly some bs followed by an external symbol.



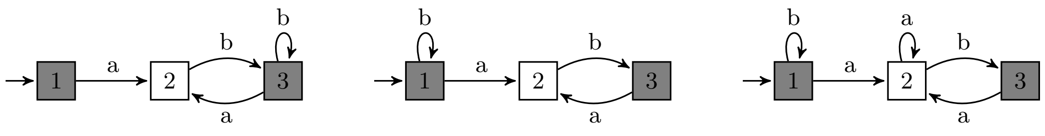


Response or Effect

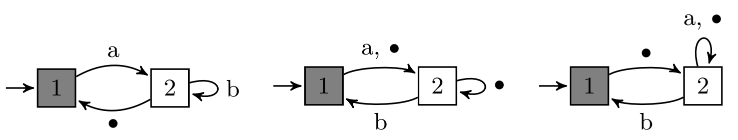
An example of this ordering relationship would be that the event `send` must always be followed by the *response* `acknowledge`. The idea is that if the cause occurs the effect *must* happen. The general notion of response given by SPS is the following pattern on the left - if a occurs then b must happen next, another notion of response used by Perracotta is that if a occurs then b must eventually occur - given on the right.



Again, the Perracotta tool introduces versions with multiple causes and single effects.

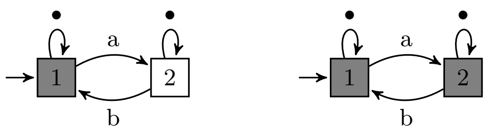


As before we can open these patterns and introduce new patterns with external symbols explicitly being cause or effect - for example the following identified in our decombination exercise.

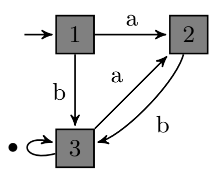


Alternation

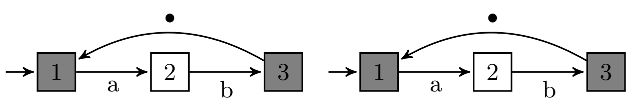
The most common pattern used in previous specification mining tools is that of alternation, either strict (i.e. we must finish with the second event) or not.



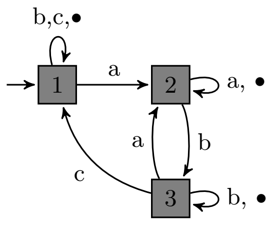
We can also alternate with no sense of start event, as follows - note that we only loop with • on state 3 to allow for chaining of these (as seen at the end of Sec. 9.4.4).



As before, we can consider alternation with an external symbol.



Finally, when adding a third symbol the choice of alternations grows. The following pattern was identified in the decombination exercise and captures aternation between a and b, started by a and finished by c i.e. between a and c.

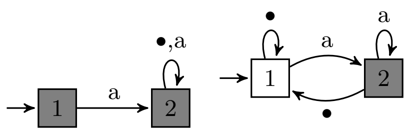


D.1.3 Scope

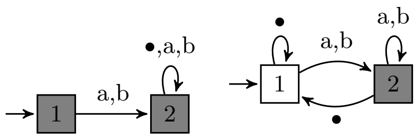
These patterns restrict the scope of events - and therefore can be combined with other events to restrict their behaviour. We consider scope in terms of restrictions on the ends of the trace, after or before events, and between events or not.

Ends of trace

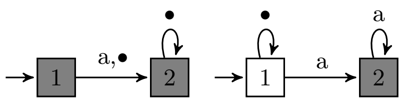
We first consider capturing the fact that an event *must* occur at the beginning or end of a trace - captured by the following two patterns respectively.



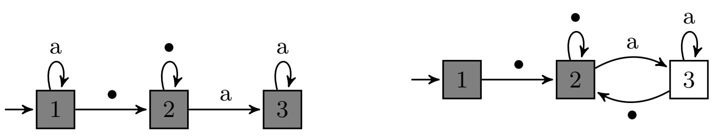
We also extend these patterns to allow the choice of two events to occur at either the beginning or end of the trace:



An alternative restriction is to say *if* an event occurs then it *must* occur *only* at either the beginning or end of the trace - note that in combination with above patterns we can capture the property that an event must occur only at the beginning or end of the trace.

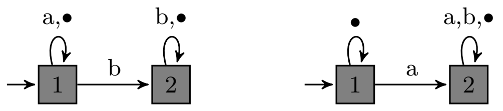


One last restriction is that that an event can only occur at the beginning or end of a trace, or cannot occur at either end.



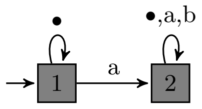
After

This subcategory of patterns capture the notion of events appearing after other events. The most simple form is all occurrences of b appear after all occurrences of b. Another pattern might be that all occurrences of b occur after the *first* occurrence of a. These are given below.



Before

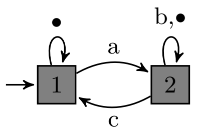
This subcategory of patterns capture the general notion of one or more events appearing before one or more other events. For example, a appears before any b:



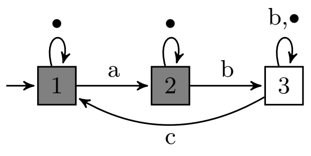
Note that if we want to say that no a's appear after any b's we can say that all a's appear before any b's.

Between

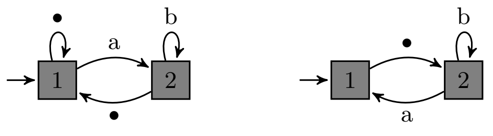
Here we restrict certain events to occur only between other events. The most general form of this pattern is that event b can only occur between events a and c, as follows:



The SPS framework had a notion of between scope where the event occurring between two other events can only occur if the second event is present. This can be captured by the following pattern:

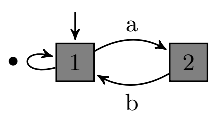


Another notion of between is to use the external hole symbol as one side of the guarding events - as in the following.

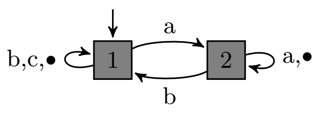


Exclusion or Not Between

We can also exclude events from appearing between other events. This is most notable in the pattern identified for the mutual exclusion specification in the decombinatino exercise - capturing that no other symbol can occur between a and b.

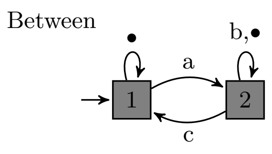


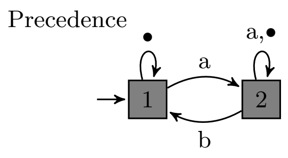
The following pattern captures the more general notion of exclusion i.e. that an event c cannot occur between events a and b.



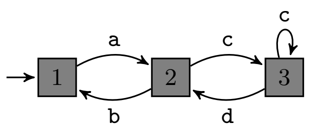
An example

Let us demonstrate how the scope patterns will work. If we want to capture the property that between `a` and `b` we can see `d` only if we first see `c` then we would use the following two patterns:





If we take the combination of the Between pattern with bindings $[a \mapsto \texttt{a}, b \mapsto \texttt{c}, c \mapsto \texttt{b}]$ and $[a \mapsto \texttt{a}, b \mapsto \texttt{d}, c \mapsto \texttt{b}]$ and the Precedence pattern with binding $[a \mapsto \texttt{c}, b \mapsto \texttt{d}]$ then we would get the following (after removing holes):



Which captures the intended behaviour.

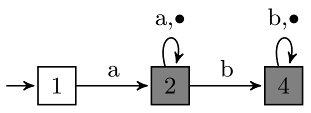
D.1.4 Compound

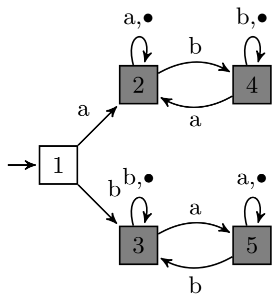
Our final category is concerned solely with combining patterns. This was a consideration when selecting previous patterns, so there is limited work to do there.

Chain

The SPS framework consider chaining precedence and response orderings separately, but these are already captured by those patterns, given above. Here we consider the general chaining or arbitrary patterns.

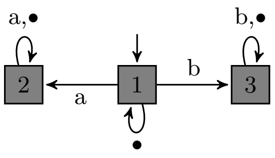
For example, if patterns have disjoint symbols and allow chaining (for example, they loop and contain hole symbols) then we can introduce patterns that take the sequence of a pattern involving a (at least once) followed by a pattern involving b (at least once). If we do not wish to specify an ordering between the patterns, just that one happens then the other, we can also capture this choice, as well as the possibility of this looping.



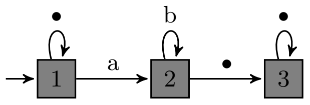


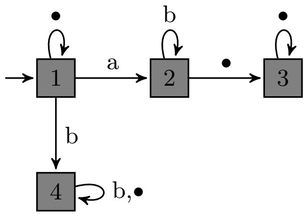
Disjunction

Disjunction of patterns with disjoint symbols is straightforward as long as the original patterns allow disjunction i.e. they have a hole looping an accepting initial state. This disjunction would be captured using the following pattern.



It is more complicated where patterns share symbols. If the alternative behaviour is chosen by the first symbol then we can extend a pattern to allow for this by adding additional states - to create an equivalent condition to the self-looping hole transitions on the initial state described above. For example, if we had the following pattern on the left we would add an additional state as given on the right.

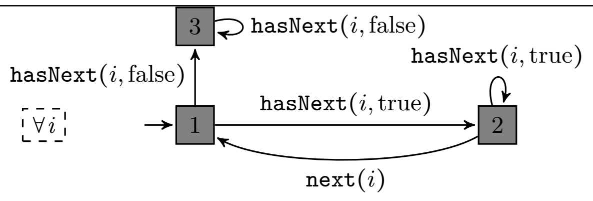


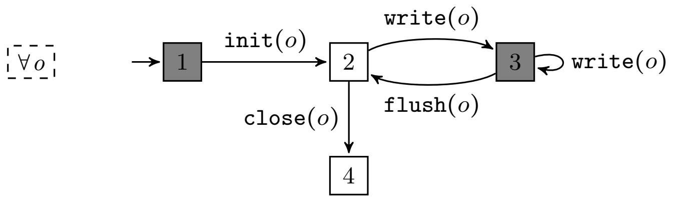


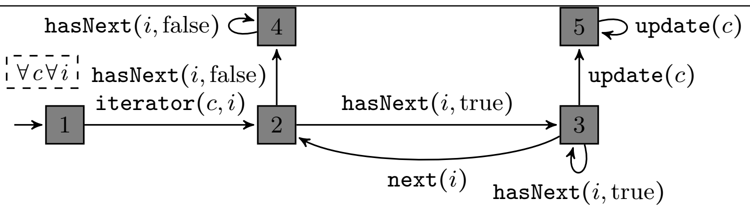
D.2 Models

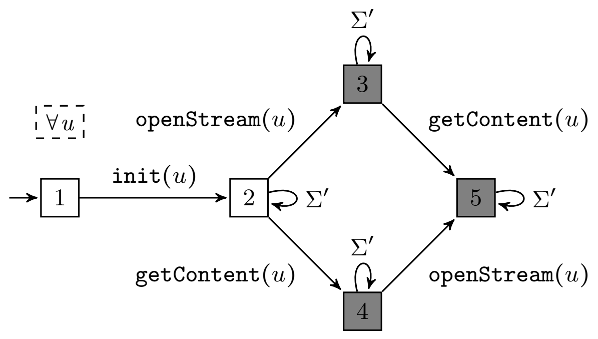
The following table is a companion to the table found on page 243 in Sec. 10.1.2 detailing the models used in the specification mining evaluation. Some of these models are also reported elsewhere, but we reproduce all models here for completeness. We also report the source of each model

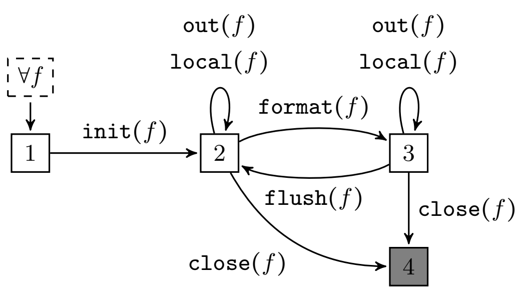
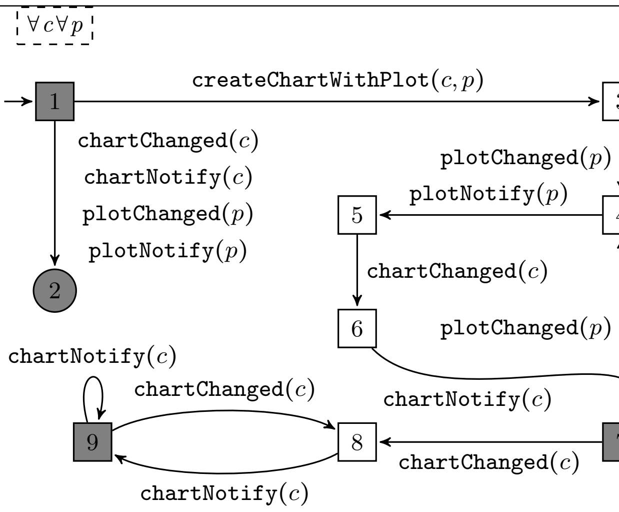
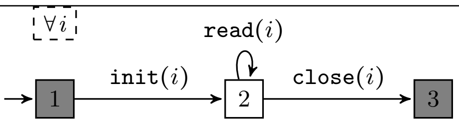
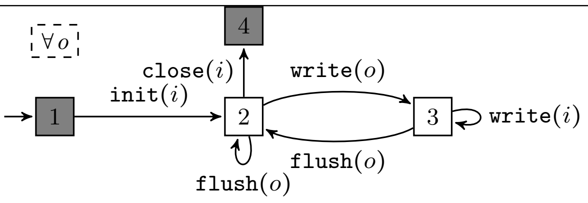
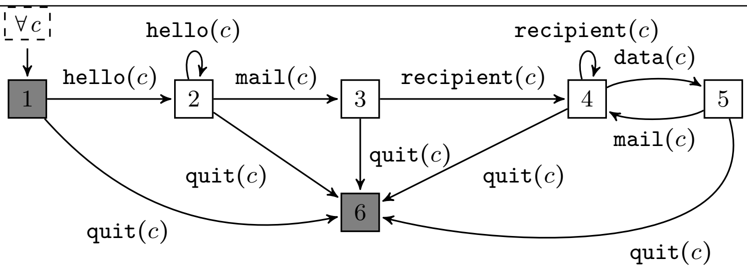
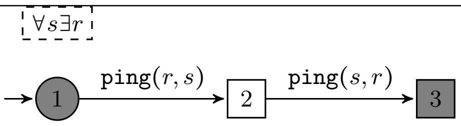
Name	Source	Model
		Java API

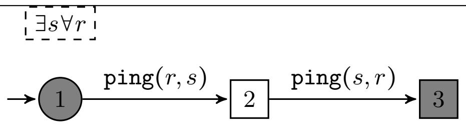
HasNext	[LCR11]	 ∀i
		1
		2
		3
		hasNext(i, false)
		hasNext(i, false)
		hasNext(i, true)
		hasNext(i, true)
		next(i)

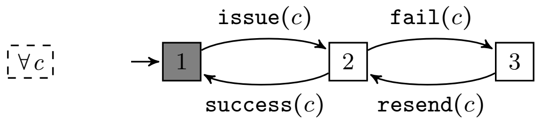
SocketOutput	[LJMR12]	 ∀o
		1
		2
		3
		4
		init(o)
		write(o)
		write(o)
		flush(o)
		close(o)

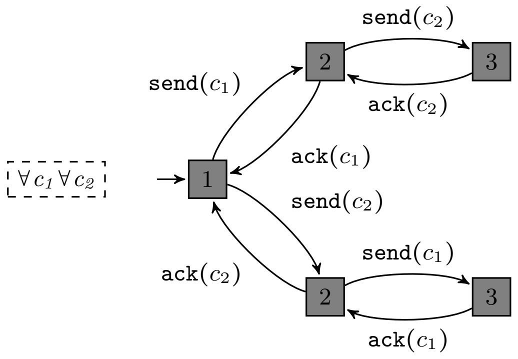
ColIter	[LCR11]	 ∀c∀i
		1
		2
		3
		4
		5
		hasNext(i, false)
		hasNext(i, false)
		iterator(c, i)
		hasNext(i, true)
		next(i)
		hasNext(i, true)
		update(c)
		update(c)

Socket	[PBG10]	See Fig. D.1 below - too large for the table.
URL	[PBG10]	 Σ′ 3 ∀u openStream(u) getContent(u) init(u) 1 2 Σ′ 5 Σ′ Σ′ getContent(u) openStream(u) 4 Where Σ′ = { `getUserInfo`(u), `toURI`(u), `getProtocol`(u), `toExternalForm`(u), `getRef`(u), `getPort`(u), `getFile`(u), `getPath`(u), `getQuery`(u), `getDefaultPort`(u), `getHost`(u), `getAuthority`(u), `sameFile`(u) }.

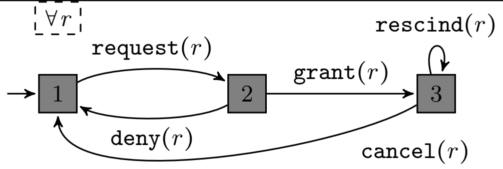
Formatter	[PBG10]	 ∀f out(f) local(f) out(f) local(f) init(f) format(f) flush(f) close(f) close(f) 1 2 3 4
JFreeChart	[CR08]	 ∀c∀p createChartWithPlot(c,p) chartChanged(c) chartNotify(c) plotChanged(p) plotNotify(p) plotChanged(p) plotNotify(p) chartChanged(c) plotChanged(p) chartNotify(c) chartChanged(c) chartNotify(c) chartChanged(c) chartNotify(c) 1 2 3 4 5 6 7 8 9
InputStream	[LCR11]	
OutputStream	[LCR11]	 ∀o close(i) init(i) write(o) write(i) flush(o) flush(o) 1 2 3 4
	Communication	
James	[CR08]	 ∀c hello(c) hello(c) mail(c) recipient(c) recipient(c) data(c) mail(c) quit(c) quit(c) quit(c) quit(c) quit(c) 1 2 3 4 5 6
Satellites- All	[FHR13, BH11b]	

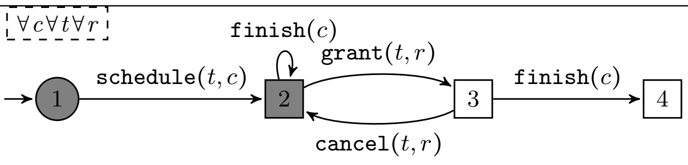
Satellites-Leader	[FHR13, BH11b]	 $\exists s \forall r$ ping(r, s) ping(s, r) 1 2 3

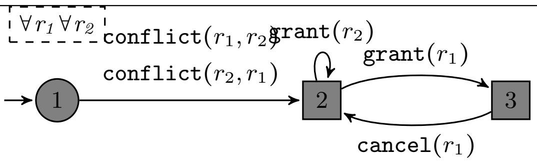
Commands	[FHR13, BH11b]	 $\forall c$
		issue(c)
		fail(c)
		success(c)
		resend(c)
		1
		2
		3

NestedCommand	[FHR13, BH11b]	 $\forall c_1 \forall c_2$
		send(c_2)
		send(c_1)
		ack(c_2)
		ack(c_1)
		send(c_2)
		ack(c_2)
		send(c_1)
		ack(c_1)
		1
		2
		3
		2
		3

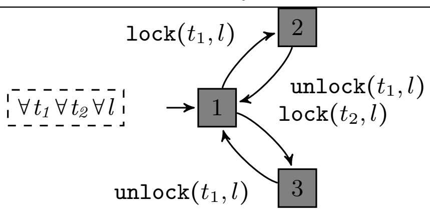
Rovers - see Appendix A.4		

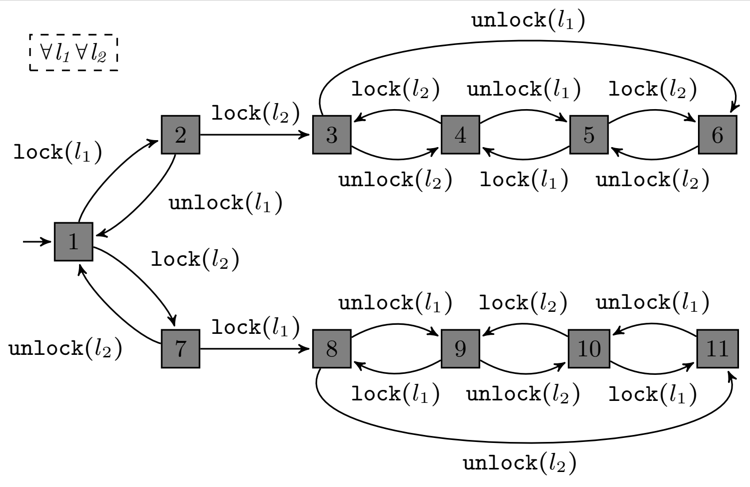
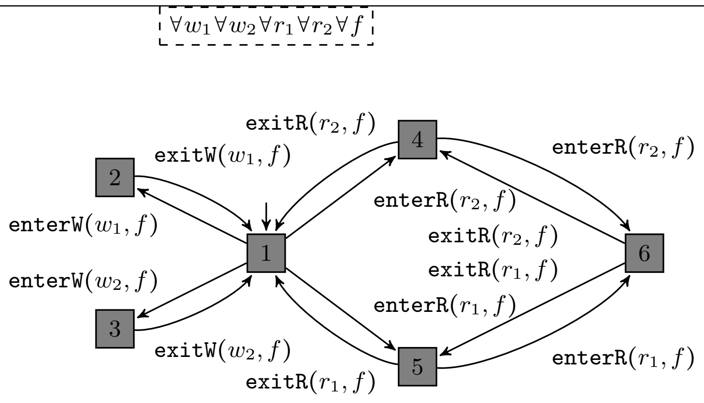
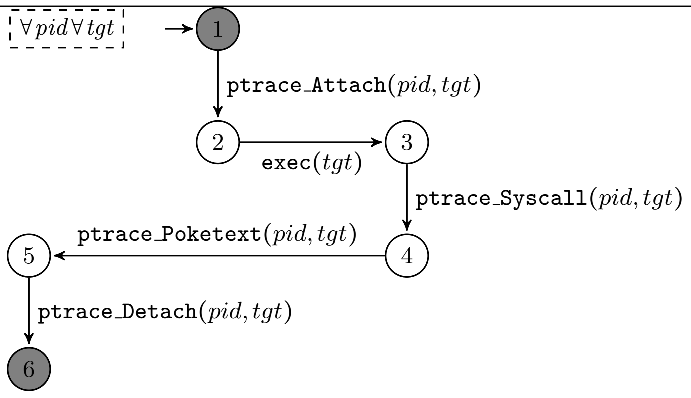
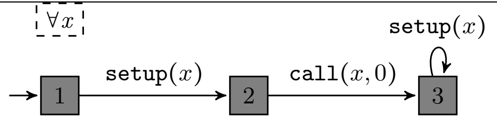
ResLifecycle	[FHR13, BH11b]	 $\forall r$
		request(r)
		rescind(r)
		grant(r)
		deny(r)
		cancel(r)
		1
		2
		3

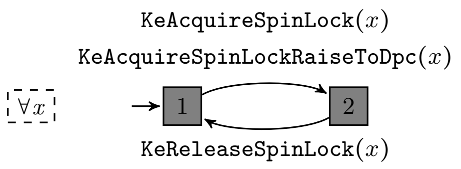
ReleaseRes	[FHR13, BH11b]	 $\forall c \forall t \forall r$
		finish(c)
		grant(t, r)
		schedule(t, c)
		finish(c)
		cancel(t, r)
		1
		2
		3
		4

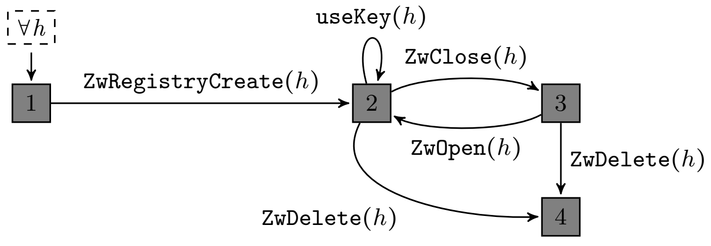
RespectConf	[FHR13, BH11b]	 $\forall r_1 \forall r_2$
		conflict(r_1, r_2)
		grant(r_2)
		conflict(r_2, r_1)
		grant(r_1)
		cancel(r_1)
		1
		2
		3

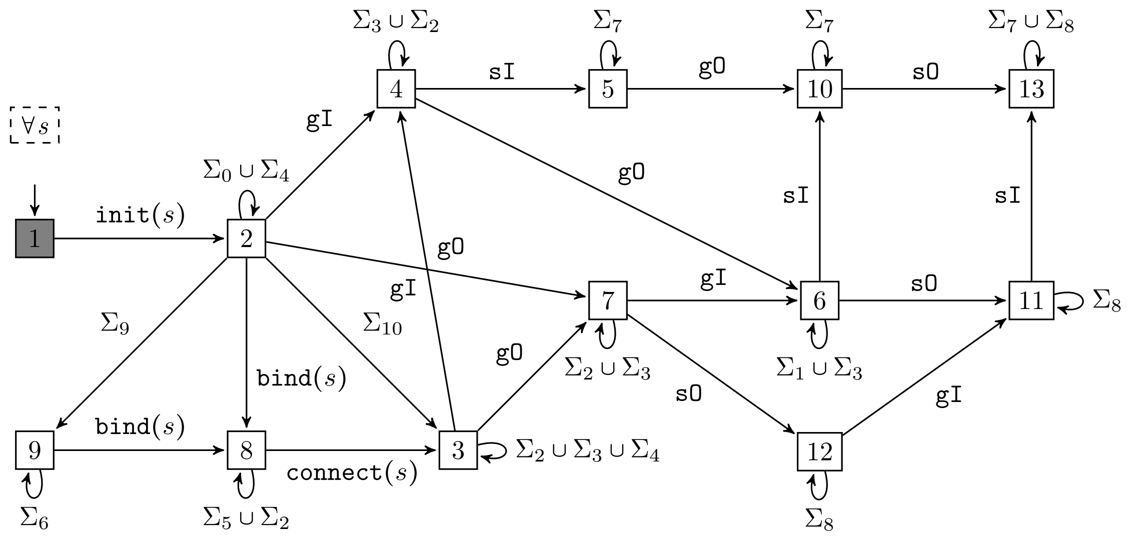
Concurrency		

MutualExcl	[CHP71]	 $\forall t_1 \forall t_2 \forall l$
		lock(t_1, l)
		unlock(t_1, l)
		lock(t_2, l)
		unlock(t_1, l)
		1
		2
		3

LockOrder	[Sto10]	 ∀l_1∀l_2 lock(l_1) unlock(l_1) lock(l_2) unlock(l_2)
ReadWriter	[CHP71]	 ∀w_1∀w_2∀r_1∀r_2∀f enterW(w_1,f) exitW(w_1,f) enterW(w_2,f) exitW(w_2,f) enterR(r_1,f) exitR(r_1,f) enterR(r_2,f) exitR(r_2,f)
Security		
PTrace	[GLO08]	 ∀pid∀tgt ptrace_Attach(pid,tgt) exec(tgt) ptrace_Syscall(pid,tgt) ptrace_Poketext(pid,tgt) ptrace_Detach(pid,tgt)
Drivers		
IOCallDriver	[LRRV12]	

KeAcquire-SpinLock	[LRRV12]	 KeAcquireSpinLock(x) KeAcquireSpinLockRaiseToDpc(x) ∀x 1 2 KeReleaseSpinLock(x)

ZwRegistry-Create	[LRRV12]	 ∀h
		useKey(h)
		ZwClose(h)
		ZwRegistryCreate(h)
		1
		2
		3
		ZwOpen(h)
		ZwDelete(h)
		ZwDelete(h)
		4



Also states $2 - 13 \xrightarrow{\texttt{close}(s)} 14$ with $14 \in F$

gI	getInputStream(s)
gO	getOutputStream(s)
sI	shutdownInput(s)
sO	shutdownOutput(s)
Σ_0	{ isConnected(s), isBound(s) }
Σ_1	$\Sigma_0 \cup$ { getLocalPort, getLocalAddress(s) }
Σ_2	$\Sigma_1 \cup$ { getLocalSocketAddress(s) }
Σ_3	{ getPort(s), getInetAddress(s), sendUrgentData(s), getRemoteSocketAddress(s)}
Σ_4	{ getSendBufferSize(s), setSoLinger(s), getTcpNoDelay(s), getSoTimeout(s), setKeepAlive(s), setOOBInline(s), getOOBInline(s), getSoLinger(s), setSoTimeout(s), setSendBufferSize(s), getKeepAlive(s), setTcpNoDelay(s)}
Σ_5	$\Sigma_4 \cup$ { getReceiveBufferSize(s), setPerformancePreferences(s), setTrafficClass(s), setReceiveBufferSize(s), getTrafficClass(s)}
Σ_6	$\Sigma_0 \cup \Sigma_5 \cup$ { setReuseAddress(s), getReuseAddress(s) }
Σ_7	$\Sigma_2 \cup \Sigma_3 \cup$ { isInputShutdown(s) }
Σ_8	$\Sigma_2 \cup \Sigma_3 \cup$ { isOutputShutdown(s) }
Σ_9	{ setTrafficClass(s), getReuseAddress(s), setReuseAddress(s), getReceiveBufferSize(s), setReceiveBufferSize(s), getTrafficClass(s), setPerformancePreferences(s) }
$\Sigma_1 0$	{ sendUrgentData(s), getLocalSocketAddress(s), getRemoteSocketAddress(s), getInetAddress(s), getLocalAddress(s), getPort(s), getLocalPort(s)}

Figure D.1: The QEA model for Socket where the table summarises events and sets of events used.

Table D.2: Statistics about the generated traces. N=None, S=State, P=Path. Model names have been shortened - they are in the same order as presented previously.

	Training												Test	
	Shortest			Short			Medium			Long				
	N	S	P	N	S	P	N	S	P	N	S	P	+	-
HN	-	5	11	-	-	43	-	-	1757	-	-	23519	2774	93
SO	4	5	7	-	12	15	28	27	27	-	9	60	52	6
CI	8	10	-	67	68	43	616	709	94	62	91	-	45	8
So	10	-	-	49	-	-	1282	1583	-	-	29995	-	2596	60
UR	9	13	-	36	44	-	-	749	1462	-	-	28457	2855	48
Fo	4	8	16	-	19	33	-	-	273	-	-	461	511	35
JF	5	-	-	-	-	-	106	113	-	144	163	227	152	11
IS	-	3	6	-	-	15	-	-	108	-	-	218	241	52
OS	3	5	11	4	6	29	-	-	344	-	-	665	565	60
Ja	6	12	22	14	31	23	-	410	140	-	107	322	238	31
SA	-	-	3	-	-	16	-	-	79	-	-	177	136	135
SL	-	-	3	-	-	15	-	-	98	-	-	189	135	116
Co	3	-	12	-	-	44	-	-	1620	-	-	43914	5501	11
NC	6	-	4	4	-	9	10	-	6	8	-	10	9	4
RL	4	7	13	-	-	26	-	-	1404	-	-	19884	2557	51
RR	7	7	-	9	10	15	15	16	24	14	17	20	15	9
RC	6	5	6	5	9	14	-	20	14	-	14	16	19	6
ME	3	1	4	18	-	23	87	-	105	-	-	-	10	4
LO	7	9	16	14	15	25	-	-	72	-	-	28	7	6
RW	-	-	-	-	8	14	-	17	20	-	-	29	31	7
PT	-	11	-	-	-	-	-	-	1435	-	591	2097	2764	12
IO	1	2	8	-	-	42	-	-	2036	-	-	-	2455	48
KA	-	5	12	-	-	56	-	-	1518	-	-	21971	1247	52
ZR	6	15	16	10	21	26	-	-	120	-	-	276	220	29

D.3 Traces

Table. D.2 reports on the generated traces - dashes indicate unused categories as discussed in Sec. 10.2.3.

D.4 Precision and Recall

Here we report the full precision and recall results. The tables in this section are as follows:

- Tables D.3 and D.4 give precision and recall, respectively, for the AdHoc pattern library
- Tables D.5 and D.6 give precision and recall, respectively, for the Designed pattern library
- Tables D.7 and D.8 give precision and recall, respectively, for the DesignedAug pattern library
- Table D.9 gives the precision and recall for the AdHoc pattern library in connected mode
- Table D.10 gives the precision and recall for the Designed pattern library in connected mode
- Table D.11 gives the precision and recall for the DesignedAug pattern library in connected mode

Table D.3: Precision for the Adhoc pattern library.

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.0/0.62	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/0.83
SO	**1.0**/0.63	**1.0**/0.63	0.93/**1.0**		0.93/**1.0**	0.93/**1.0**	0.55/0.64	0.93/**1.0**	0.93/**1.0**		**1.0**/0.63	0.93/**1.0**
CI	0.0/0.5	0.0/0.5		**1.0**/0.51	0.95/0.92	to	0.91/0.55	0.96/0.97	to	0.0/0.5	0.91/0.55	
So	to			to			to	to			to	
UR	to	to		to	to			to	**1.0**/**1.0**			to
Fo	0.0/0.62	0.0/0.62	0.0/0.62		**1.0**/0.62	**1.0**/0.63			**1.0**/**1.0**			**1.0**/**1.0**
JF	0.0/0.62						0.94/0.93	0.94/0.93		0.5/0.62	0.93/**1.0**	0.0/0.61
IS		**1.0**/0.67	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
OS	0.0/0.62	0.0/0.62	**1.0**/**1.0**	0.0/0.62	0.0/0.62	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
Ja	**1.0**/0.34	**1.0**/0.39	**1.0**/0.38	to	**1.0**/0.35	**1.0**/0.52		0.0/0.33	**1.0**/**1.0**		**1.0**/**1.0**	**1.0**/**1.0**
SA			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
SL			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
Co	0.0/0.37		0.0/0.37			**1.0**/0.65			**1.0**/0.65			**1.0**/0.65
NC	0.0/0.82		0.0/0.82	0.0/0.82		**1.0**/0.87	0.0/0.82		**1.0**/0.9	0.0/0.82		**1.0**/0.93
RL	0.0/0.66	to	to			to			to			to
RR	0.0/0.63	0.0/0.58		0.0/0.63	0.6/0.64	0.53/0.70	0.0/0.63	**1.0**/0.63	0.6/0.66	0.0/0.63	0.0/0.63	0.88/0.69
RC	0.0/0.71	0.0/0.71	0.28/0.71	0.0/0.71	0.6/0.72	0.41/0.74		0.79/0.84	0.54/0.76		to	0.52/0.89
ME	-	-	**1.0**/0.73	-		0.0/0.55	-		-			
LO	**1.0**/0.81	**1.0**/0.81	**1.0**/0.84	**1.0**/0.80	**1.0**/0.79	**1.0**/0.82			**1.0**/0.80			**1.0**/0.84
RW					-	to		to	**1.0**/0.80			**1.0**/0.79
PT		-							-		-	-
IO	0.0/0.39	0.0/0.39	**1.0**/**1.0**			**1.0**/0.52			**1.0**/0.52			
KA		0.0/0.36	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
ZR	0.0/0.66	**1.0**/**1.0**	**1.0**/**1.0**	**1.0**/0.67	**1.0**/0.67	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**

Table D.4: Recall for the Adhoc pattern library.

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.0/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			0.66/**1.0**
SO	0.01/**1.0**	0.01/**1.0**	**1.0**/0.96		**1.0**/0.96	**1.0**/0.96	0.08/0.96	**1.0**/0.96	**1.0**/0.96		0.01/**1.0**	**1.0**/0.96
CI	0.0/**1.0**	0.0/**1.0**		0.06/**1.0**	0.92/0.96	to	0.22/0.98	0.98/0.96	to	0.0/**1.0**	0.21/0.98	
So	to			to			to	to			to	
UR	to	to		to	to			to	**1.0/1.0**			to
Fo	0.0/**1.0**	0.0/**1.0**	0.0/**1.0**		0.01/**1.0**	0.03/**1.0**			**1.0/1.0**			**1.0/1.0**
JF	0.0/**1.0**						0.88/0.97	0.88/0.97		0.05/0.97	**1.0**/0.96	0.0/0.97
IS		0.02/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
OS	0.0/**1.0**	0.0/**1.0**	**1.0/1.0**	0.0/**1.0**	0.0/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
Ja	0.02/**1.0**	0.22/**1.0**	0.18/**1.0**	to	0.06/**1.0**	0.54/**1.0**		0.0/**1.0**	**1.0/1.0**		**1.0/1.0**	**1.0/1.0**
SA			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
SL			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
Co	0.0/**1.0**		0.0/**1.0**			0.68/**1.0**			0.68/**1.0**			0.68/**1.0**
NC	0.0/**1.0**		0.0/**1.0**	0.0/**1.0**		0.29/**1.0**	0.0/**1.0**		0.47/**1.0**	0.0/**1.0**		0.64/**1.0**
RL	0.0/**1.0**	to	to			to			to			to
RR	0.0/**1.0**	0.0/0.83		0.0/**1.0**	0.05/0.98	0.44/0.77	0.0/**1.0**	0.01/**1.0**	0.20/0.92	0.0/**1.0**	0.0/**1.0**	0.25/0.98
RC	0.0/**1.0**	0.0/**1.0**	0.05/0.95	0.0/**1.0**	0.07/0.98	0.3/0.83		0.57/0.94	0.32/0.89		to	0.8/0.71
ME	-	-	0.54/**1.0**	-		0.0/**1.0**	-		-			
LO	0.46/**1.0**	0.46/**1.0**	0.56/**1.0**	0.43/**1.0**	0.36/**1.0**	0.48/**1.0**			0.43/**1.0**			0.56/**1.0**
RW					-	to		to	0.11/**1.0**			0.03/**1.0**
PT		-							-		-	-
IO	0.0/**1.0**	0.0/**1.0**	**1.0/1.0**			0.41/**1.0**			0.41/**1.0**			
KA		0.0/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
ZR	0.0/**1.0**	**1.0/1.0**	**1.0/1.0**	0.02/**1.0**	0.02/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**

Table D.5: Precision for the Designed pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.0/0.62	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
SO	-	-	-		-	-	-	-	-		-	-
CI	-	0.0/0.47		-	-	-	-	-	-	-	-	
So	0.0/0.29			0.24/**1.0**			to	to			to	
UR	0.0/0.62	0.0/0.62		0.62/**1.0**	0.79/**1.0**			**1.0**/0.63	0.81/**1.0**			to
Fo	0.0/0.62	**1.0/1.0**	**1.0**/0.62		0.81/**1.0**	0.84/**1.0**			0.79/**1.0**			0.84/**1.0**
JF	0.0/0.62						-	-		-	-	-
IS		**1.0**/0.67	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
OS	0.0/0.62	**1.0/1.0**	**1.0/1.0**	0.0/0.62	0.0/0.62	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
Ja	**1.0/1.0**	**1.0/1.0**	**1.0/1.0**	**1.0**/0.35	**1.0**/0.35	**1.0/1.0**		0.0/0.33	**1.0/1.0**		**1.0/1.0**	**1.0/1.0**
SA			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
SL			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
Co	0.0/0.37		0.99/**1.0**			**1.0**/0.65			**1.0**/0.65			**1.0**/0.65
NC	0.0/0.82		0.0/0.82	0.0/0.82		-	-		-	-		-
RL	0.0/0.66	0.74/**1.0**	0.74/**1.0**			0.74/**1.0**			0.74/**1.0**			0.74/**1.0**
RR	0.0/0.63	0.0/0.52		-	-	-	-	-	-	-	-	-
RC	-	-	-	0.0/0.71	-	-		-	-		-	-
ME	-	-	0.0/0.55	-		0.0/0.55	-		-			
LO	0.0/0.70	0.0/0.70	0.0/0.70	-	-	-			-			-
RW					-	-		-	-			-
PT		-							-		-	-
IO	0.0/0.39	0.0/0.39	0.81/**1.0**			**1.0**/0.52			**1.0**/0.52			
KA		0.0/0.36	**1.0/1.0**			**1.0**/0.66			**1.0**/0.55			**1.0/1.0**
ZR	0.85/**1.0**	0.85/**1.0**	0.85/**1.0**	0.85/**1.0**	0.85/**1.0**	0.85/**1.0**			0.85/**1.0**			0.85/**1.0**

Table D.6: Recall for the Designed pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.0/**1.0**	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
SO	-	-	-		-	-	-	-	-		-	-
CI	-	0.0/0.9		-	-	-	-	-	-	-	-	
So	0.0/**1.0**			**1.0**/0.41			to	to			to	
UR	0.0/**1.0**	0.0/**1.0**		**1.0**/0.65	**1.0**/0.80			0.05/**1.0**	**1.0**/0.82			to
Fo	0.0/**1.0**	**1.0**/**1.0**	0.01/**1.0**		**1.0**/0.82	**1.0**/0.85			**1.0**/0.80			**1.0**/0.83
JF	0.0/**1.0**						-	-		-	-	-
IS		0.02/**1.0**	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
OS	0.0/**1.0**	**1.0**/**1.0**	**1.0**/**1.0**	0.0/**1.0**	0.0/**1.0**	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
Ja	**1.0**/**1.0**	**1.0**/**1.0**	**1.0**/**1.0**	0.06/**1.0**	0.06/**1.0**	**1.0**/**1.0**		0.0/**1.0**	**1.0**/**1.0**		**1.0**/**1.0**	**1.0**/**1.0**
SA			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
SL			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
Co	0.0/**1.0**		**1.0**/0.98			0.68/**1.0**			0.68/**1.0**			0.68/**1.0**
NC	0.0/**1.0**		0.0/**1.0**	0.0/**1.0**		-	-		-	-		-
RL	0.0/**1.0**	**1.0**/0.83	**1.0**/0.83			**1.0**/0.83			**1.0**/0.83			**1.0**/0.83
RR	0.0/**1.0**	0.0/0.63		-	-	-	-	-	-	-	-	-
RC	-	-	-	0.0/**1.0**	-	-		-	-		-	-
ME	-	-	0.0/**1.0**	-		0.0/**1.0**	-		-			
LO	0.0/**1.0**	0.0/**1.0**	0.0/**1.0**	-	-	-			-			-
RW					-	-		-	-			-
PT		-							-		-	-
IO	0.0/**1.0**	0.0/**1.0**	**1.0**/0.82			0.41/**1.0**			0.41/**1.0**			
KA		0.0/**1.0**	**1.0**/**1.0**			0.71/**1.0**			0.53/**1.0**			**1.0**/**1.0**
ZR	**1.0**/0.85	**1.0**/0.85	**1.0**/0.85	**1.0**/0.85	**1.0**/0.85	**1.0**/0.85			**1.0**/0.85			**1.0**/0.85

Table D.7: Precision for the DesignedAug pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.0/0.62	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
SO	**1.0**/0.63	**1.0**/0.63	**1.0**/**1.0**		**1.0**/**1.0**	**1.0**/**1.0**	0.0/0.63	**1.0**/**1.0**	**1.0**/**1.0**		0.0/0.63	**1.0**/**1.0**
CI	0.0/0.5	0.0/0.5		**1.0**/0.50	**1.0**/0.98	0.97/0.63	**1.0**/0.56	**1.0**/0.97	0.0/0.49	0.0/0.5	**1.0**/0.59	
So	to			to			to	to			to	
UR	to	to		to	to			to	to			to
Fo	0.0/0.62	0.0/0.62	0.0/0.62		0.0/0.62	**1.0**/0.71			**1.0**/**1.0**			**1.0**/**1.0**
JF	0.0/0.62						**1.0**/0.63	0.98/**1.0**		**1.0**/0.62	0.98/**1.0**	0.0/0.62
IS		**1.0**/0.67	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
OS	0.0/0.62	0.0/0.62	**1.0**/0.63	0.0/0.62	0.0/0.62	**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
Ja	**1.0**/0.34	**1.0**/0.35	**1.0**/0.38	0.0/0.33	**1.0**/0.34	**1.0**/0.39		0.0/0.33	**1.0**/**1.0**		**1.0**/0.34	**1.0**/**1.0**
SA			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
SL			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
Co	0.0/0.37		**1.0**/**1.0**			**1.0**/0.65			**1.0**/0.65			**1.0**/0.65
NC	0.0/0.82		0.0/0.82	0.0/0.82		0.88/0.89	0.0/0.82		0.0/0.82	0.0/0.82		0.90/0.91
RL	0.0/0.66	0.0/0.66	0.0/0.66			**1.0**/**1.0**			**1.0**/**1.0**			**1.0**/**1.0**
RR	0.0/0.63	0.0/0.62		0.0/0.63	0.0/0.63	0.71/0.70	**1.0**/0.63	**1.0**/0.64	0.61/0.65	0.0/0.63	0.0/0.63	0.66/0.64
RC	**1.0**/0.71	0.0/0.71	0.0/0.71	0.0/0.71	**1.0**/0.72	**1.0**/0.72		**1.0**/0.84	to		0.0/0.71	0.93/0.89
ME	-	-	0.0/0.55	-		0.0/0.55	-		-			
LO	0.0/0.70	0.0/0.70	0.0/0.70	**1.0**/0.81	**1.0**/0.84	**1.0**/0.81			**1.0**/0.85			**1.0**/0.75
RW					to	to		to	to			to
PT		-							-		-	-
IO	0.0/0.39	0.0/0.39	**1.0**/**1.0**			**1.0**/0.52			**1.0**/0.52			
KA		0.0/0.36	0.0/0.36			**1.0**/0.66			**1.0**/0.55			**1.0**/**1.0**
ZR	0.0/0.66	**1.0**/0.68	**1.0**/0.68	0.0/0.66	**1.0**/0.67	**1.0**/0.82			**1.0**/**1.0**			**1.0**/**1.0**

Table D.8: Recall for the DesignedAug pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.0/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
SO	0.01/**1.0**	0.01/**1.0**	**1.0/1.0**		**1.0/1.0**	**1.0/1.0**	0.0/**1.0**	**1.0/1.0**	**1.0/1.0**		0.0/**1.0**	**1.0/1.0**
CI	0.0/**1.0**	0.0/**1.0**		0.03/**1.0**	0.98/**1.0**	0.42/0.99	0.22/**1.0**	0.97/**1.0**	0.0/0.99	0.0/**1.0**	0.31/**1.0**	
So	to			to			to	to			to	
UR	to	to		to	to			to	to			to
Fo	0.0/**1.0**	0.0/**1.0**	0.0/**1.0**		0.0/**1.0**	0.33/**1.0**			**1.0/1.0**			**1.0/1.0**
JF	0.0/**1.0**						0.03/**1.0**	**1.0**/0.99		0.01/**1.0**	**1.0**/0.99	0.0/**1.0**
IS		0.02/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
OS	0.0/**1.0**	0.0/**1.0**	0.03/**1.0**	0.0/**1.0**	0.0/**1.0**	**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
Ja	0.02/**1.0**	0.09/**1.0**	0.2/**1.0**	0.0/**1.0**	0.05/**1.0**	0.23/**1.0**		0.0/**1.0**	**1.0/1.0**		0.03/**1.0**	**1.0/1.0**
SA			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
SL			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
Co	0.0/**1.0**		**1.0/1.0**			0.68/**1.0**			0.68/**1.0**			0.68/**1.0**
NC	0.0/**1.0**		0.0/**1.0**	0.0/**1.0**		0.47/0.98	0.0/**1.0**		0.0/**1.0**	0.0/**1.0**		0.58/0.98
RL	0.0/**1.0**	0.0/**1.0**	0.0/**1.0**			**1.0/1.0**			**1.0/1.0**			**1.0/1.0**
RR	0.0/**1.0**	0.0/0.95		0.0/**1.0**	0.0/0.99	0.34/0.92	0.01/**1.0**	0.03/**1.0**	0.13/0.95	0.0/**1.0**	0.0/0.99	0.06/0.98
RC	0.02/**1.0**	0.0/**1.0**	0.0/**1.0**	0.0/**1.0**	0.05/**1.0**	0.07/**1.0**		0.52/**1.0**	to		0.0/**1.0**	0.7/0.98
ME	-	-	0.0/**1.0**	-		0.0/**1.0**	-		-			
LO	0.0/**1.0**	0.0/**1.0**	0.0/**1.0**	0.46/**1.0**	0.56/**1.0**	0.46/**1.0**			0.58/**1.0**			0.21/**1.0**
RW					to	to		to	to			to
PT		-							-		-	-
IO	0.0/**1.0**	0.0/**1.0**	**1.0/1.0**			0.41/**1.0**			0.41/**1.0**			
KA		0.0/**1.0**	0.0/**1.0**			0.71/**1.0**			0.53/**1.0**			**1.0/1.0**
ZR	0.0/**1.0**	0.1/**1.0**	0.1/**1.0**	0.0/**1.0**	0.02/**1.0**	0.58/**1.0**			**1.0/1.0**			**1.0/1.0**

Table D.9: Precision and Recall for the AdHoc pattern library in connectedness mode

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
							Precision					
SO	0.81/0.66	0.89/0.70	0.96/0.93		0.96/0.93	0.89/0.70	0.0/0.63	0.96/0.93	0.96/0.93		**1.0**/0.65	0.96/0.93
CI	0.85/0.57	0.95/0.55		0.5/0.5	0.91/0.68	0.91/0.68	0.89/0.56	0.91/0.68	0.89/0.56	0.80/0.55	0.77/0.52	
JF	**1.0**/0.72						**1.0**/0.72	**1.0**/0.75		**1.0**/0.72	**1.0**/0.76	**1.0**/0.74
RR	**1.0**/0.65	0.41/0.64		**1.0**/0.65	0.88/0.66	0.61/0.72	**1.0**/0.65	**1.0**/0.66	0.65/0.68	**1.0**/0.65	**1.0**/0.65	0.90/0.72
RC	**1.0**/0.71	**1.0**/0.71	0.37/0.71	0.0/0.71	0.66/0.72	0.46/0.76		0.79/0.84	0.54/0.76		0.5/0.71	0.53/0.91
							Recall					
SO	0.15/0.98	0.29/0.98	0.87/0.98		0.87/0.98	0.29/0.98	0.0/0.99	0.87/0.98	0.87/0.98		0.10/**1.0**	0.87/0.98
CI	0.29/0.95	0.2/0.99		0.02/0.98	0.56/0.95	0.57/0.95	0.25/0.97	0.56/0.95	0.25/0.97	0.25/0.94	0.14/0.96	
JF	0.38/**1.0**						0.38/**1.0**	0.45/**1.0**		0.38/**1.0**	0.48/**1.0**	0.43/**1.0**
RR	0.08/**1.0**	0.20/0.83		0.08/**1.0**	0.13/0.99	0.44/0.84	0.10/**1.0**	0.13/**1.0**	0.25/0.92	0.08/**1.0**	0.08/**1.0**	0.34/0.98
RC	0.02/**1.0**	0.02/**1.0**	0.07/0.95	0.0/**1.0**	0.05/0.99	0.37/0.83		0.57/0.94	0.32/0.89		0.02/0.99	0.82/0.71

Table D.10: Precision and Recall for the Designed pattern library in connectedness mode

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
					Precision							
SO	0.82/0.92	0.89/0.8	0.82/0.92		0.82/0.92	-	0.0/0.63	0.89/0.8	-		**1.0**/0.65	-
CI	-	0.86/0.62		0.61/0.50	-	-	-	-	-	-	-	
JF	**1.0**/0.72						**1.0**/0.75	-		-	-	-
RR	**1.0**/0.65	0.13/0.54		-	-	-	-	-	-	-	-	-
RC	-	-	-	-	-	-		-	-		-	-
					Recall							
SO	0.87/0.89	0.58/0.96	0.87/0.89		0.87/0.89	-	0.0/0.99	0.58/0.96	-		0.10/**1.0**	-
CI	-	0.44/0.93		0.08/0.95	-	-	-	-	-	-	-	
JF	0.38/**1.0**						0.45/**1.0**	-		-	-	-
RR	0.08/**1.0**	0.10/0.63		-	-	-	-	-	-	-	-	-
RC	-	-	-	-	-	-		-	-		-	-

Table D.11: Precision and Recall for the DesignedAug pattern library in connectedness mode

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
							Precision					
SO	0.81/0.66	0.87/0.66	0.96/0.93		0.85/0.68	0.96/0.93	0.0/0.63	0.97/0.80	0.96/0.93		**1.0**/0.65	0.96/0.93
CI	0.78/0.53	0.95/0.55		0.33/0.49	0.89/0.62	0.91/0.68	0.86/0.54	0.89/0.62	0.88/0.56	0.76/0.54	0.77/0.52	
JF	**1.0**/0.72						**1.0**/0.72	**1.0**/0.75		**1.0**/0.72	**1.0**/0.76	**1.0**/0.74
RR	**1.0**/0.65	0.54/0.64		**1.0**/0.65	**1.0**/0.66	0.78/0.76	**1.0**/0.66	**1.0**/0.66	0.8/0.71	**1.0**/0.65	0.83/0.65	**1.0**/0.68
RC	**1.0**/0.71	0.0/0.71	0.0/0.71	0.0/0.71	**1.0**/0.71	**1.0**/0.72		**1.0**/0.84	0.88/0.79		0.0/0.71	0.93/0.89
							Recall					
SO	0.15/0.98	0.12/0.99	0.87/0.98		0.20/0.98	0.87/0.98	0.0/0.99	0.58/0.99	0.87/0.98		0.10/**1.0**	0.87/0.98
CI	0.15/0.96	0.2/0.99		0.01/0.98	0.44/0.95	0.57/0.95	0.19/0.97	0.44/0.95	0.24/0.97	0.23/0.93	0.14/0.96	
JF	0.38/**1.0**						0.38/**1.0**	0.45/**1.0**		0.38/**1.0**	0.48/**1.0**	0.43/**1.0**
RR	0.08/**1.0**	0.10/0.95		0.08/**1.0**	0.13/**1.0**	0.5/0.92	0.12/**1.0**	0.13/**1.0**	0.34/0.95	0.08/**1.0**	0.08/0.99	0.22/**1.0**
RC	0.02/**1.0**	0.0/**1.0**	0.0/**1.0**	0.0/**1.0**	0.02/**1.0**	0.07/**1.0**		0.52/**1.0**	0.37/0.98		0.0/**1.0**	0.7/0.98

D.5 Efficiency

Here we report the full efficiency results. The tables in this section are as follows:

- Tables D.12 to D.14 give the checking and combining times and patterns passed, respectively, for the AdHoc pattern library
- Tables D.15 to D.17 give the checking and combining times and patterns passed, respectively, for the Designed pattern library
- Tables D.18 to D.20 give the checking and combining times and patterns passed, respectively, for the DesignedAug pattern library

Table D.12: Checking times for the AdHoc pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		2.223	1.556			0.743			3.35			23.09
SO	1.72	2.17	1.546		1.567	0.626	0.873	0.981	0.502		1.493	1.075
CI	1.399	2.584		2.397	2.867	to	7.125	11.50	to	3.162	2.061	
So	to			to			to	to			to	
UR	to	to		to	to			to	525.7			to
Fo	1.762	1.887	1.717		2.073	1.369			5.781			7.977
JF	0.735						3.263	2.949		3.096	3.187	0.891
IS		0.68	1.863			0.873			1.033			1.461
OS	0.855	0.766	1.933	0.674	0.71	1.052			2.012			3.039
Ja	0.976	1.179	2.354	to	1.301	1.334		3.891	2.673		0.823	4.807
SA			2.087			1.084			1.085			1.395
SL			2.106			1.085			1.07			1.326
Co	1.068		2.209			1.293			5.406			86.85
NC	1.611		2.316	0.778		2.092	0.869		1.23	1.404		1.679
RL	1.494	to	to			to			to			to
RR	1.629	0.656		1.007	1.167	0.739	1.185	1.02	0.871	1.693	0.893	1.05
RC	2.378	1.679	1.453	1.088	2.501	2.091		2.766	2.457		to	2.664
ME	-	-	1.49	-		4.516	-		-			
LO	2.578	1.523	2.469	2.286	2.856	2.147			1.338			1.454
RW					-	to		to	30.43			17.66
PT		-							-		-	-
IO	1.896	0.907	0.989			0.679			1.147			
KA		1.296	0.96			0.693			1.995			18.79
ZR	2.071	1.634	1.327	1.402	1.288	1.061			2.148			4.136

Table D.13: Combining times for the AdHoc pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.231	0.232			0.24			0.562			0.323
SO	1.176	0.468	0.242		0.233	0.244	0.47	0.258	0.308		0.508	0.289
CI	163.2	8.778		0.972	7.476	to	1.668	1.647	to	12.8	18.65	
So	to			to			to	to			to	
UR	to	to		to	to			to	17.93			to
Fo	13.69	6.347	16.03		1.41	1.49			1.088			1.11
JF	4.465						0.421	0.44		0.147	0.441	0.149
IS		0.647	0.331			0.333			0.34			0.362
OS	2.076	0.851	0.66	2.949	1.117	0.697			0.657			0.712
Ja	6.346	14.22	31.68	to	12.69	18.12		4.165	8.179		8.515	9.74
SA			0.219			0.229			0.232			0.239
SL			0.219			0.242			0.239			0.254
Co	1.2		16.16			24.04			23.19			30.44
NC	0.998		1.778	1.336		7.178	1.041		1.934	19.34		8.965
RL	3.621	to	to			to			to			to
RR	0.103	8.601		2.311	5.811	6.68	6.056	8.389	37.30	4.791	0.563	4.723
RC	0.417	9.702	122.8	9.905	3.236	452.3		5.765	27.39		to	11.11
ME	-	-	1.32	-		1.363	-		-			
LO	5.891	1.112	0.035	0.037	0.045	0.054			0.048			6.07
RW					-	to		to	49.81			108.2
PT		-							-		-	-
IO	0.157	0.204	0.155			0.443			0.352			
KA		0.134	0.024			0.039			0.041			0.136
ZR	5.543	1.39	1.49	2.126	1.86	1.509			1.614			1.739

Table D.14: Passed patterns for the AdHoc pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		176	90			90			90			91
SO	30	27	24		24	24	26	24	24		30	24
CI	80	103		39	37	to	38	35	to	75	38	
So	to			to			to	to			to	
UR	to	to		to	to			to	464			to
Fo	969	210	221		168	168			164			164
JF	728						26	26		27	24	28
IS		395	95			95			95			95
OS	861	152	127	861	226	127			127			127
Ja	322	75	77	to	97	73		108	71		71	71
SA			177			177			177			177
SL			177			177			177			177
Co	485		26			27			27			27
NC	92		142	180		18	64		60	32		18
RL	960	to	to			to			to			to
RR	35	29		38	31	18	34	29	18	34	38	17
RC	69	68	53	161	40	37		35	37		to	34
ME	-	-	76	-		94	-		-			
LO	24	12	8	8	8	8			8			16
RW					-	to		to	324			188
PT		-							-		-	-
IO	187	177	60			76			76			
KA		20	14			14			14			14
ZR	146	122	122	350	170	122			122			122

Table D.15: Checking times for the Designed pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		2.041	1.544			0.681			2.187			22.97
SO	-	-	-		-	-	-	-	-		-	-
CI	-	2.571		-	-	-	-	-	-	-	-	
So	125.1			480.7			to	to			to	
UR	7.677	5.243		16.32	18.96			24.73	492.8			to
Fo	0.825	0.796	1.699		0.925	1.089			4.466			6.223
JF	0.731						-	-		-	-	-
IS		0.645	1.845			0.772			0.87			1.168
OS	0.915	0.74	1.926	0.392	0.68	0.923			1.835			2.773
Ja	1.076	0.986	2.239	0.759	1.046	1.158		3.638	2.046		0.753	3.274
SA			2.135			1.047			0.989			1.208
SL			2.089			1.056			1.026			1.219
Co	1.139		2.238			1.282			5.376			89.05
NC	1.415		2.244	0.707		-	-		-	-		-
RL	1.38	1.388	1.647			1.703			8.994			91.96
RR	1.556	0.445		-	-	-	-	-	-	-	-	-
RC	-	-	-	0.994	-	-		-	-		-	-
ME	-	-	1.151	-		3.01	-		-			
LO	2.563	1.449	1.799	-	-	-			-			-
RW					-	-		-	-			-
PT		-							-		-	-
IO	1.729	1.232	0.91			0.432			1.188			
KA		1.302	0.934			0.545			1.906			20.15
ZR	2.09	1.545	1.225	1.292	0.635	0.818			1.701			2.993

Table D.16: Combining times for the Designed pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.055	0.047			0.05			0.082			0.065
SO	-	-	-		-	-	-	-	-		-	-
CI	-	0.003		-	-	-	-	-	-	-	-	
So	2.054			1.334			to	to			to	
UR	0.414	0.595		0.374	0.442			0.525	0.123			to
Fo	0.219	0.14	0.151		0.146	0.131			0.154			0.158
JF	0.561						-	-		-	-	-
IS		0.09	0.051			0.063			0.064			0.068
OS	0.126	0.064	0.064	0.157	0.076	0.083			0.073			0.081
Ja	0.043	0.038	0.027	0.114	0.081	0.026		0.179	0.027		0.037	0.031
SA			0.03			0.034			0.037			0.038
SL			0.031			0.036			0.035			0.039
Co	0.125		0.038			0.062			0.063			0.078
NC	0.02		0.117	0.131		-	-		-	-		-
RL	0.04	0.069	0.12			0.071			0.143			0.094
RR	0.019	0.053		-	-	-	-	-	-	-	-	-
RC	-	-	-	0.003	-	-		-	-		-	-
ME	-	-	0.007	-		0.013	-		-			
LO	0.01	0.01	0.011	-	-	-			-			-
RW					-	-		-	-			-
PT		-							-		-	-
IO	0.023	0.031	0.003			0.009			0.013			
KA		0.01	0.007			0.014			0.02			0.032
ZR	0.019	0.023	0.026	0.024	0.035	0.032			0.032			0.037

Table D.17: Passed patterns for the Designed pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		43	31			31			31			31
SO	-	-	-		-	-	-	-	-		-	-
CI	-	1		-	-	-	-	-	-	-	-	
So	293			200			to	to			to	
UR	66	76		64	64			71	64			to
Fo	88	58	66		58	58			58			58
JF	122						-	-		-	-	-
IS		58	37			37			37			37
OS	86	40	40	86	42	40			40			40
Ja	24	18	18	51	39	18		53	18		18	18
SA			30			30			30			30
SL			30			30			30			30
Co	67		26			29			29			29
NC	10		28	24		-	-		-	-		-
RL	34	14	14			14			14			14
RR	11	7		-	-	-	-	-	-	-	-	-
RC	-	-	-	4	-	-		-	-		-	-
ME	-	-	6	-		8	-		-			
LO	8	8	8	-	-	-			-			-
RW					-	-		-	-			-
PT		-							-		-	-
IO	26	30	7			12			12			
KA		12	10			11			12			10
ZR	20	20	20	25	20	20			20			20

Table D.18: Checking times for the DesignedAug pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		2.149	1.522			0.84			2.462			23.51
SO	0.758	1.838	1.408		1.303	0.77	0.894	1.168	0.741		1.106	1.532
CI	1.446	3.215		2.667	3.227	3.245	8.018	12.50	6.905	4.957	2.099	
So	to			to			to	to			to	
UR	to	to		to	to			to	to			to
Fo	1.488	1.472	2.082		1.682	1.83			6.164			8.702
JF	0.948						3.115	2.699		3.696	2.899	1.049
IS		0.751	1.896			0.966			1.217			1.74
OS	0.991	0.929	2.108	0.607	0.792	1.223			2.355			3.576
Ja	1.618	1.523	2.794	1.299	1.509	1.853		4.558	3.88		1.021	6.89
SA			2.139			1.142			1.167			1.529
SL			2.156			1.165			1.202			1.513
Co	0.993		2.488			1.506			5.692			88.25
NC	2.24		2.504	0.917		2.436	0.82		1.039	1.189		1.638
RL	1.851	1.288	1.229			1.288			8.987			98.11
RR	1.91	1.132		1.087	1.2	1.075	1.471	1.082	1.344	1.763	0.865	1.626
RC	2.741	2.089	2.189	1.448	3.189	2.899		3.849	to		1.513	3.927
ME	-	-	2.382	-		11.60	-		-			
LO	5.065	3.341	5.311	4.875	5.425	3.589			3.353			1.775
RW					to	to		to	to			to
PT		-							-		-	-
IO	1.882	1.176	0.957			0.548			1.272			
KA		1.395	1.113			0.709			2.101			20.41
ZR	2.446	2.042	1.731	1.572	0.996	1.462			3.231			6.166

Table D.19: Combining times for the DesignedAug pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		0.728	0.515			0.539			0.681			0.638
SO	0.423	0.302	0.258		0.288	0.355	0.385	0.331	0.423		0.473	0.448
CI	8.654	4.6		2.717	1.848	3.477	2.914	2.634	2.74	2.664	5.659	
So	to			to			to	to			to	
UR	to	to		to	to			to	to			to
Fo	19.98	4.639	6.943		2.607	2.397			2.024			2.116
JF	9.776						0.788	0.397		0.535	0.462	0.412
IS		1.637	0.537			0.585			0.606			0.66
OS	4.153	1.754	0.936	4.334	2.862	0.942			0.957			1.012
Ja	6.818	2.189	1.819	4.325	1.801	1.786		3.605	1.38		1.564	1.478
SA			0.504			0.554			0.535			0.576
SL			0.518			0.562			0.545			0.577
Co	2.856		0.66			0.785			0.778			0.826
NC	1.098		2.445	2.434		2.534	1.74		2.186	2.626		2.836
RL	6.149	6.754	7.299			1.183			1.189			1.212
RR	0.497	1.411		1.943	1.67	11.37	1.102	3.318	6.832	2.081	0.756	5.918
RC	1.078	3.771	22.38	15.46	1.184	2.85		3.126	to		7.329	2.972
ME	-	-	1.915	-		1.321	-		-			
LO	0.819	0.399	0.449	0.27	0.271	0.303			0.287			0.787
RW					to	to		to	to			to
PT		-							-		-	-
IO	0.443	0.519	0.24			0.294			0.28			
KA		0.257	0.14			0.134			0.137			0.179
ZR	2.717	1.675	1.441	4.002	2.446	2.024			1.455			1.586

Table D.20: Passed patterns for the DesignedAug pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		538	309			309			309			309
SO	88	77	79		74	74	83	74	74		90	74
CI	161	210		167	141	107	166	141	118	265	154	
So	to			to			to	to			to	
UR	to	to		to	to			to	to			to
Fo	2066	566	678		523	521			520			520
JF	1378						104	86		106	86	97
IS		1066	290			290			290			290
OS	2097	445	343	2097	653	342			342			342
Ja	627	293	287	856	354	285		400	284		287	284
SA			437			437			437			437
SL			437			437			437			437
Co	1159		211			220			220			220
NC	172		346	278		88	146		162	208		88
RL	1993	237	206			195			195			195
RR	121	59		145	110	63	109	96	53	100	99	56
RC	301	273	127	469	153	130		154	to		173	124
ME	-	-	146	-		162	-		-			
LO	88	82	82	74	74	74			74			82
RW					to	to		to	to			to
PT		-							-		-	-
IO	515	437	141			146			146			
KA		93	53			53			54			52
ZR	453	364	342	806	427	340			339			339

D.6 Conciseness

Here we report the full conciseness results. The tables in this section are as follows:

- Table D.21 gives the size and complexity of extracted models for the AdHoc pattern library
- Table D.22 gives the size and complexity of extracted models for the Designed pattern library
- Table D.23 gives give the size and complexity of extracted models for the DesignedAug pattern library
- Table D.24 gives give the size and complexity of extracted models for all pattern libraries in connectedness mode

Table D.21: Size (number of states)/Complexity for the AdHoc pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		4/ 2	4/ 3			4/ 3			4/ 3			6/ 4
SO	6/ 12	7/ 16	8/ 19		8/ 19	8/ 19	5/ 12	8/ 19	8/ 19		6/ 12	8/ 19
CI	16/ 47	12/ 31		27/ 119	101/ 460	to	22/ 107	34/ 161	to	26/ 86	65/ 279	
So	to			to			to	to			to	
UR	to	to		to	to			to	7/ 74			to
Fo	10/ 18	28/ 70	43/ 83		14/ 38	17/ 50			5/ 8			5/ 8
JF	7/ 1						13/ 57	13/ 57		13/ 56	9/ 37	13/ 55
IS		4/ 1	4/ 2			4/ 2			4/ 2			4/ 2
OS	4/ 1	8/ 3	5/ 3	4/ 1	6/ 2	5/ 3			5/ 3			5/ 3
Ja	9/ 12	13/ 16	17/ 24	to	9/ 11	13/ 17		13/ 13	10/ 13		10/ 13	10/ 13
SA			4/ 1			4/ 1			4/ 1			4/ 1
SL			4/ 1			4/ 1			4/ 1			4/ 1
Co	6/ 3		21/ 26			15/ 12			15/ 12			15/ 12
NC	10/ 7		9/ 2	9/ 2		51/ 132	17/ 34		11/ 22	13/ 24		51/ 132
RL	4/ 4	to	to			to			to			to
RR	6/ 11	85/ 237		8/ 21	17/ 55	93/ 338	14/ 43	35/ 107	84/ 301	18/ 54	18/ 52	60/ 214
RC	22/ 109	30/ 113	82/ 338	10/ 16	32/ 145	92/ 370		21/ 107	72/ 281		to	70/ 273
ME	-	-	13/ 24	-		17/ 8	-		-			
LO	45/ 124	29/ 78	5/ 12	5/ 12	5/ 12	5/ 12			5/ 12			45/ 126
RW					-	to		to	37/ 124			51/ 156
PT		-							-		-	-
IO	3/ 1	4/ 1	4/ 4			5/ 2			5/ 2			
KA		6/ 5	4/ 4			4/ 4			4/ 4			4/ 4
ZR	12/ 26	8/ 19	8/ 19	10/ 17	12/ 26	8/ 19			8/ 19			8/ 19

Table D.22: Size (number of states)/Complexity for the Designed library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		4/ 2	4/ 3			4/ 3			4/ 3			4/ 3
SO	-	-	-		-	-	-	-	-		-	-
CI	-	5/ 16		-	-	-	-	-	-	-	-	
So	13/ 242			4/ 38			to	to			to	
UR	4/ 29	9/ 87		3/ 16	3/ 16			6/ 56	3/ 16			to
Fo	5/ 8	4/ 5	5/ 7		4/ 5	4/ 5			4/ 5			4/ 5
JF	7/ 1						-	-		-	-	-
IS		4/ 1	4/ 2			4/ 2			4/ 2			4/ 2
OS	4/ 1	4/ 3	4/ 3	4/ 1	5/ 4	4/ 3			4/ 3			4/ 3
Ja	4/ 6	4/ 6	4/ 6	5/ 8	5/ 8	4/ 6		6/ 6	4/ 6		4/ 6	4/ 6
SA			4/ 1			4/ 1			4/ 1			4/ 1
SL			4/ 1			4/ 1			4/ 1			4/ 1
Co	4/ 2		4/ 4			7/ 7			7/ 7			7/ 7
NC	11/ 26		8/ 3	10/ 5		-	-		-	-		-
RL	6/ 18	3/ 6	3/ 6			3/ 6			3/ 6			3/ 6
RR	11/ 20	18/ 44		-	-	-	-	-	-	-	-	-
RC	-	-	-	3/ 7	-	-		-	-		-	-
ME	-	-	5/ 10	-		10/ 23	-		-			
LO	10/ 23	10/ 23	10/ 23	-	-	-			-			-
RW					-	-		-	-			-
PT		-							-		-	-
IO	3/ 1	4/ 1	3/ 3			5/ 3			5/ 3			
KA		6/ 5	4/ 4			5/ 4			8/ 6			4/ 4
ZR	3/ 5	3/ 5	3/ 5	3/ 5	3/ 5	3/ 5			3/ 5			3/ 5

Table D.23: Size (number of states)/Complexity for the DesignedAug pattern library

	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
HN		4/ 2	4/ 3			4/ 3			4/ 3			4/ 3
SO	7/ 14	8/ 17	7/ 15		6/ 13	6/ 13	6/ 13	6/ 13	6/ 13		8/ 16	6/ 13
CI	18/ 51	13/ 20		7/ 23	14/ 49	24/ 86	11/ 37	8/ 26	10/ 28	12/ 36	25/ 80	
So	to			to			to	to			to	
UR	to	to		to	to			to	to			to
Fo	7/ 11	35/ 85	35/ 56		12/ 30	10/ 25			5/ 8			5/ 8
JF	7/ 1						10/ 29	6/ 22		10/ 28	6/ 22	7/ 22
IS		4/ 1	4/ 2			4/ 2			4/ 2			4/ 2
OS	4/ 1	8/ 3	10/ 7	4/ 1	6/ 2	5/ 3			5/ 3			5/ 3
Ja	10/ 10	11/ 11	13/ 15	10/ 9	8/ 8	9/ 11		7/ 5	7/ 8		10/ 11	7/ 8
SA			4/ 1			4/ 1			4/ 1			4/ 1
SL			4/ 1			4/ 1			4/ 1			4/ 1
Co	6/ 3		5/ 3			6/ 4			6/ 4			6/ 4
NC	9/ 4		8/ 3	9/ 2		48/ 113	22/ 21		10/ 21	17/ 16		48/ 113
RL	4/ 4	19/ 30	37/ 67			5/ 6			5/ 6			5/ 6
RR	16/ 26	29/ 40		12/ 31	16/ 43	53/ 165	14/ 43	20/ 55	70/ 231	26/ 80	8/ 21	57/ 160
RC	7/ 28	20/ 67	127/ 501	11/ 19	8/ 27	51/ 212		69/ 340	to		23/ 85	20/ 79
ME	-	-	13/ 8	-		41/ 26	-		-			
LO	16/ 17	17/ 24	17/ 24	5/ 12	5/ 12	5/ 12			5/ 12			18/ 43
RW					to	to		to	to			to
PT		-							-		-	-
IO	3/ 1	4/ 1	4/ 5			5/ 3			5/ 3			
KA		10/ 8	10/ 12			5/ 4			8/ 6			4/ 4
ZR	17/ 36	10/ 24	14/ 37	11/ 18	12/ 27	11/ 28			5/ 8			5/ 8

Table D.24: Size (number of states)/Complexity for the connectedness results

AdHoc												
	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
SO	5/ 7	7/ 12	5/ 6		4/ 4	7/ 14	4/ 2	4/ 4	4/ 6		5/ 1	4/ 6
CI	11/ 29	8/ 9		11/ 12	17/ 52	61/ 188	13/ 45	9/ 26	26/ 78	21/ 67	12/ 31	
JF	7/ 1						8/ 6	12/ 27		7/ 13	49/ 135	10/ 19
RR	6/ 11	85/ 237		9/ 21	18/ 55	74/ 264	14/ 43	35/ 107	84/ 301	14/ 37	18/ 52	41/ 133
RC	22/ 109	17/ 61	82/ 340	10/ 23	41/ 175	92/ 370		21/ 107	72/ 281		24/ 87	70/ 273

Designed												
	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
SO	3/ 3	4/ 4	3/ 3		3/ 3	-	4/ 2	4/ 4	-		5/ 1	-
CI	-	4/ 8		9/ 14	-	-	-	-	-	-	-	
JF	7/ 1						5/ 9	-		-	-	-
RR	11/ 20	18/ 44		-	-	-	-	-	-	-	-	-
RC	-	-	-	-	-	-		-	-		-	-

DesignedAug												
	Shortest			Short			Medium			Long		
	N	S	P	N	S	P	N	S	P	N	S	P
SO	5/ 7	6/ 5	5/ 6		7/ 12	4/ 6	4/ 2	5/ 5	4/ 6		5/ 1	4/ 6
CI	14/ 34	12/ 16		7/ 5	12/ 33	16/ 52	4/ 8	11/ 31	11/ 27	8/ 19	10/ 26	
JF	7/ 1						7/ 4	9/ 19		7/ 11	17/ 43	12/ 20
RR	16/ 26	29/ 40		12/ 30	13/ 32	53/ 165	14/ 43	16/ 44	70/ 231	8/ 20	8/ 21	32/ 103
RC	7/ 28	13/ 40	127/ 501	11/ 25	17/ 74	51/ 212		69/ 340	56/ 218		23/ 85	20/ 79