A Pattern-Based Approach to Parametric Specification Mining

Giles Reger, Howard Barringer, and David Rydeheard

University of Manchester, UK

Abstract. This paper presents a technique for mining parametric temporal specifications from execution traces. The parametric specifications we mine are a form of quantified event automata (QEA), previously introduced as an expressive and efficient formalism for runtime verification. We consider one way in which QEAs may be harnessed for specification mining, namely pattern-based mining that uses a pattern library to generate and check potential properties over given traces, and then combines successful patterns to give a succinct specification capturing the observed behaviour. We have implemented our technique in Scala and evaluated its ability to extract useful specifications. By using predefined models to measure the tool's precision and recall we demonstrate that our approach can effectively and efficiently extract specifications in realistic scenarios.

1 Introduction

In [5], a formalism, and an associated tool, for runtime verification was introduced. Runtime verification [12] is the checking of a trace of events from the run of a computational system against a given specification. In this work specifications are in the form of so-called quantified event automata (QEA) which consist of first-order quantification over variables occurring in a form of finite automaton. This is expressive; allowing us to describe properties not just in terms of what events occur in a trace, but also how the data values associated with events are related within a trace. In addition, as automata, QEAs allow efficient incremental trace checking.

We are concerned with temporal specifications over events parameterised with data values, hence *parametric specifications*. It should be noted that a specification may deal with data parameters in two orthogonal ways - in a *quantified* manner or in a *free* manner. An example of the former, would be to specify that a file should only be read when open by specifying that event $\texttt{open}(f)$ occurs before $\texttt{read}(f)$ for every file f; an example of the latter would be to specify that a counter is strictly increasing by specifying that if $\texttt{count}(x)$ follows $\texttt{count}(y)$ then $x > y$. These are two distinct approaches to dealing with data parameters. QEA addresses both but this paper only considers the first approach, leaving methods for dealing with free variables to future work.

We consider the question: Can QEAs be used in specification mining ([19, 30]), i.e. given a set of traces of events from runs of a computational system, can we build specifications of the system in terms of a QEA? A positive answer would allow us to mine specifications which previous systems cannot, due to the increased expressiveness of QEA. Several parametric specification mining approaches have been proposed, these extend techniques such as *stage-merging* [8] and *active learning* [4] from the propositional to parametric setting - although there is a distinct seperation between approaches that deal with parameters in a quantified or free manner. Here, we choose an approach particularly suited to QEA, namely pattern-based mining [32, 13], in which a given collection of specification "patterns" is checked against the traces. This is a verification process: if the patterns are expressed as QEA, the verification process is that described in [5].

Pattern-based specification mining usually considers only small and simple patterns, so that the checking of traces is sufficiently fast. However, for specification mining to be useful, we wish to mine specifications of some complexity. To bridge this gap, methods by which patterns may be combined to form complex specifications are introduced [13]. We consider methods for combining QEA using what we call "open automata", which we will use as our notion of pattern.

In this paper we present a pattern-based approach to mining parametric specifications which consists of three stages, described further in section 4:

1. *Generating.* A pattern library and candidate alphabet are combined to produce a set of possible patterns.
2. *Checking.* These patterns are checked against a set of given traces.
3. *Combining.* The successful patterns are combined to form a specification.

This general approach is not new (for example [13, 14, 32]), however, we are the first to apply it in a *parametric* setting, although the work of [24] is comparable as discussed in Section 7. The part of this process that deals with quantified parameters is the checking stage - based on a runtime verification algorithm for QEA. This consists of projecting traces of events into subtraces for given assignments to quantified variables and detecting patterns in these. A similar projection-based approach was taken in [17] using an automata-learning instead of a pattern-based approach.

Contributions. We make two main contributions - firstly, adapting the generate-and-check style approach to the quantified parametric setting in a general way, i.e. not specific to any set of patterns; and secondly, introducing the concept of open automata as a method for soundly combining small patterns to build specifications.

Outline. We present *quantified event automata* (QEA) in Section 2. Section 3 introduces our notion of pattern that is used to mine QEA, as described in Section 4 and evaluated in Section 5. Finally, we discuss related work and conclude in Sections 7 and 8.

2 Quantified event automata

In this section we introduce quantified event automata (QEA) as a formalism for describing parametric specifications.

2.1 Notation.

Throughout this paper we write a list as $[l_0, \ldots, l_n]$, with the ith element of a list L given by $L[i-1]$. We use $\overline{s}$ to denote a tuple $\langle s_0, \ldots, s_k \rangle$. A map from X to Y is a partial function $A \in X \rightharpoonup Y$ with a finite domain. We write maps as $\langle x_0 \mapsto v_0, \ldots, x_i \mapsto v_i \rangle$, the empty map as $\langle\,\rangle$ and the domain of map A as $\mathsf{dom}(A)$. Given two maps A and B the map override operator is defined as

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

2.2 Structure

An *event* is a pair of an event name and a list of symbols. More formally, given a set of event names Σ and a set of symbols $Symbol$ an event is a pair $\langle e, \overline{x} \rangle \in \Sigma \times Symbol^*$, which we will write as $\mathsf{e}(\overline{x})$ or $\mathbf{a}$. A symbol can be a value or a variable - for disjoint Val and Var we have $Symbol = Var \cup Val$. An event is *ground* if it only contains values. A trace is a *finite* sequence of ground events - note that we are only concerned with finite traces. Let $Event$, $GEvent$ and $Trace$ be the sets of events, ground events and traces respectively.

A *binding* is a map in $Binding = Var \rightharpoonup Val$. Recall the submap relation $\sqsubseteq$ on bindings such that $\theta_1 \sqsubseteq \theta_2$ iff θ_1 and θ_2 map common variables to the same values and $|\theta_1| \leq |\theta_2|$.

We can apply bindings to symbols and events as substitutions. For symbol s let $\theta(s)$ be the value of s in θ if s is in $\mathsf{dom}(\theta)$ and s otherwise. Given an event $\mathsf{e}(s_1, \ldots, s_n)$ let $\theta(\mathsf{e}(s_1, \ldots, s_n)) = \mathsf{e}(\theta(s_1), \ldots, \theta(s_n))$. Given a set of events $\mathcal{A}$ let $\mathcal{A}(\theta) = \{\theta(\mathbf{a}) \mid \mathbf{a} \in \mathcal{A}\}$. An event $\mathbf{a}$ and a ground event $\mathbf{b}$ match iff there exists a binding θ such that $\theta(\mathbf{a}) = \mathbf{b}$. Let $\mathsf{matches}(\mathbf{a}, \mathbf{b})$ hold iff $\mathbf{a}$ and $\mathbf{b}$ match and let $\mathbf{match}(\mathbf{a}, \mathbf{b})$ be the smallest (wrt $\sqsubseteq$) binding that matches them (and undefined if they do not match).

An event automaton (EA) is a finite state automaton with transitions labelled with events and a quantified event automaton (QEA) is an EA with "quantifications".

Definition 1 (Quantified Event Automata). *An* event automaton E *is a tuple* $\langle \mathcal{A}, Q, \delta, q_0, F, q_\perp \rangle$ *where* $\mathcal{A} \subseteq Event$ *is a set of events,* Q *is a finite set of states,* $\delta : Q \times \mathcal{A} \times Q$ *is a set of transitions,* $q_0 \in Q$ *is the initial state,* $F \subseteq Q$ *is the set of final states and* $q_\perp \in Q \backslash F$ *is the error state.*

A quantified event automaton Q *is a pair* $\langle \Lambda, \mathsf{E} \rangle$ *where* $\Lambda \in (\{\forall, \exists\} \times X)^*$ *is a list of quantifications where* $X \subseteq Var$ *is the set of variables used in* $\mathcal{A}$*. Let* $\mathsf{vars}(\Lambda)$ *be the set of variables in* Λ*.* Q *is well-formed iff all variables in* X *appear in* Λ *exactly once.*

This is a somewhat restricted form of QEA as introduced in [5]. In this work we have chosen to deal with the quantifications and have left the concept of guards and assignments labeling transitions to future work - to see how these are used in QEA see [5]. One reason for this restriction is that our form of combination (Sec. 3) cannot currently deal with *free* variables, which are necessary for guards and assignments.

2.3 Semantics

Here we present the semantics of QEA. Let $\mathsf{Q} = \langle \Lambda, \langle \mathcal{A}, Q, \delta, q_0, F, q_\perp \rangle \rangle$ be a QEA. We first introduce a transition relation for a given binding of quantified variables.

Definition 2 (Transition Relation). *Given binding θ such that* $\mathsf{dom}(\theta) = \mathsf{vars}(\Lambda)$*, let* $\rightsquigarrow_\theta \subseteq Q \times GEvent \times Q$ *be a relation on states s.t. states q and q' are related by the ground event* $\mathbf{a}$ *if and only if*

$$\exists (q, \mathbf{b}, q') \in \delta : \mathtt{matches}(\mathbf{a}, \theta(\mathbf{b}))$$

Let $\rightarrow_\theta$ be Q*'s transition relation for θ, defined as the smallest relation containing $\rightsquigarrow_\theta$ such that for all events* $\mathbf{a}$ *and states q if $\nexists q' : q \overset{\mathbf{a}}{\rightsquigarrow}_\theta q'$ then $q \overset{\mathbf{a}}{\rightarrow}_\theta q_\perp$.*

The relation $\rightarrow_\theta$ is lifted to traces: for states q and q', $q \overset{\epsilon}{\rightarrow}_\theta q$ always holds, and $q \overset{\mathbf{a}\tau}{\rightarrow}_\theta q'$ holds iff there is a q' such that $q \overset{\mathbf{a}}{\rightarrow}_\theta q''$ and $q'' \overset{\tau}{\rightarrow}_\theta q'$.

Therefore, if no transition is defined a transition to the error state is introduced. Next we introduce the concept of *projection* with respect to a binding - this filters out events that are not in the alphabet specialized with the binding.

Definition 3 (Projection). *The projection of $\tau \in Trace$ with respect to alphabet $\mathcal{A}$ for binding θ is defined as:*

$$\begin{aligned} \epsilon \downarrow^\theta_{\mathcal{A}} &= \epsilon \\ \tau\mathbf{a} \downarrow^\theta_{\mathcal{A}} &= \begin{cases} (\tau \downarrow^\theta_{\mathcal{A}})\mathbf{a} & \textit{if } \mathbf{a} \in \mathcal{A}(\theta) \\ (\tau \downarrow^\theta_{\mathcal{A}}) & \textit{otherwise} \end{cases} \end{aligned}$$

We now define when the QEA Q accepts a trace τ by first deriving the domains of the quantified variables from τ.

Definition 4 (Acceptance). *Define the domain of each quantified variable as follows:*

$$\begin{aligned} \mathsf{dom}(\tau)(x) = \{\, \mathtt{match}(\mathbf{a}, \mathbf{b})(x) \mid \mathbf{a} = e(..., x, ...) \in \mathcal{A} \\ \wedge \mathbf{b} \in \tau \wedge \mathtt{matches}(\mathbf{a}, \mathbf{b}) \} \end{aligned}$$

Q *accepts a trace τ iff $\tau \models^{\mathsf{Q}}_{\langle\,\rangle} \Lambda$, where $\models^{\mathsf{Q}}_\theta$ is defined as*

$$\begin{aligned} \tau \models^{\mathsf{Q}}_\theta \forall x \Lambda' &\quad \textit{iff for all } d \textit{ in } \mathsf{dom}(\tau)(x)\ \tau \models^{\mathsf{Q}}_{\theta\dagger\langle x \mapsto d\rangle} \Lambda' \\ \tau \models^{\mathsf{Q}}_\theta \exists x \Lambda' &\quad \textit{iff for any } d \textit{ in } \mathsf{dom}(\tau)(x)\ \tau \models^{\mathsf{Q}}_{\theta\dagger\langle x \mapsto d\rangle} \Lambda' \\ \tau \models^{\mathsf{Q}}_\theta \epsilon &\quad \textit{iff } \exists q \in F : q_0 \overset{\tau\downarrow^\theta_{\mathcal{A}}}{\longrightarrow}_\theta q \end{aligned}$$

This inductive definition of $\models$ builds up all possible bindings of quantified variables and checks whether the trace projected with each required binding reaches a final state.

2.4 Example

Figure 1 gives a QEA for Java iterators that was mined during the development of our tool. The QEA has one quantified variable i and uses the data values true and false in events. Shaded states represent final states and the implicit error state $q_\perp$ is omitted. Consider the trace

$$\tau = \mathtt{hasNext}(\mathrm{A}, \mathrm{true}).\mathtt{next}(\mathrm{A}).\mathtt{hasNext}(\mathrm{B}, \mathrm{false}).\mathtt{hasNext}(\mathrm{A}, \mathrm{true})$$

We have $\mathsf{dom}(\tau)(i) = \{\mathrm{A}, \mathrm{B}\}$ giving two bindings $\theta_1 = \langle i \mapsto \mathrm{A}\rangle$ and $\theta_2 = \langle i \mapsto \mathrm{B}\rangle$. Projection gives us

$$\begin{aligned} \tau \downarrow^{\theta_1}_{\mathcal{A}} &= \mathtt{hasNext}(\mathrm{A}, \mathrm{true}).\mathtt{next}(\mathrm{A}).\mathtt{hasNext}(\mathrm{A}, \mathrm{true}) \\ \tau \downarrow^{\theta_2}_{\mathcal{A}} &= \mathtt{hasNext}(\mathrm{B}, \mathrm{false}) \end{aligned}$$

And therefore, as $0 \overset{\tau\downarrow^{\theta_1}_{\mathcal{A}}}{\longrightarrow}_{\theta_1} 2$ and $0 \overset{\tau\downarrow^{\theta_2}_{\mathcal{A}}}{\longrightarrow}_{\theta_2} 4$, the trace τ is accepted.

As a second example, consider the property that a file must be opened before being read or written to, should not be left open, and if deleted should not be used again. Figure 2 presents a QEA capturing this property. Shaded states represent final states and there is an implicit error state $q_\perp$ that is used if no transitions match. Consider the trace

$$\tau = \mathtt{open}(A).\mathtt{open}(B).\mathtt{read}(A).\mathtt{write}(B).\mathtt{close}(A).\mathtt{close}(B).\mathtt{delete}(A)$$

Again, we consider two bindings for f - $\theta_1 = \langle f \mapsto A\rangle$ and $\theta_2 = \langle f \mapsto B\rangle$and project the trace for each value:

$$\begin{aligned} \tau \downarrow^{\theta_1}_{\mathcal{A}} &= \sigma_1 = \mathtt{open}(A).\mathtt{read}(A).\mathtt{close}(A).\mathtt{delete}(A) \\ \tau \downarrow^{\theta_1}_{\mathcal{A}} &= \sigma_2 = \mathtt{open}(B).\mathtt{write}(B).\mathtt{close}(B) \end{aligned}$$

We have labeled these σ_1 and σ_2 as we will use them later. The trace τ is accepted as both projections reach a final state in the automaton of Figure 2 with f appropriately replaced.

A final example gives a *negative property* i.e. a description of undesired behaviour. This could be extracted from *negative* traces (those that should not occur) and be used to refine a specification - we discuss negative traces later. The property is that there exists a iterator created from a collection that is used after the collection is updated and is given by the QEA in Fig. 3. The following (incorrect) trace would be accepted by this QEA as it reaches the accepting state.

$$\tau = \texttt{iterator}(C, A).\texttt{hasNext}(A).\texttt{next}(A).\texttt{update}(C).\texttt{update}(C).\texttt{hasnext}(A)$$

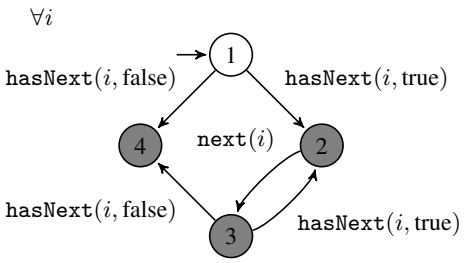


Fig. 1: A mined QEA for Java Iterators

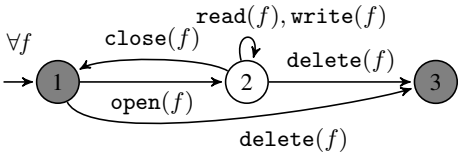


Fig. 2: A QEA for a resource-usage property.

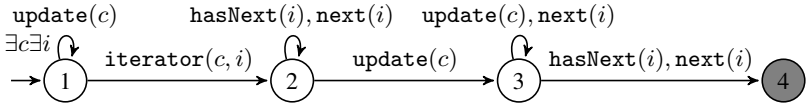


Fig. 3: A QEA for the incorrect usage of collections and iterators.

3 A notion of pattern

In this section we introduce a notion of pattern and pattern libraries. As outlined earlier, patterns are combinable predicates on traces. An initial suggestion might be to use standard finite state automata, but it has previously been shown [14] that these have an inadequate form of combination which fails to capture relationships between non-shared symbols. We address this with a new form of automata.

3.1 Open Automata

We introduce *open automata* as a formalism for patterns.

Definition 5 (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.*

Our notion of 'pattern' is to be open automata. 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:

$$\begin{aligned} \Delta_p(S, \epsilon) &= S \\ \Delta_p(\{\}, \sigma) &= \{q_\perp\} \\ \Delta_p(S, a\sigma) &= \Delta_p(S', \sigma) \\ \text{for } S' = \left\{ q' \mid \exists q \in S : \begin{array}{l} (q, a, q') \in \delta \text{ if } a \in \Sigma \\ (q, \bullet, q') \in \delta) \text{ if } a \notin \Sigma \end{array} \right\} & \end{aligned}$$

This adds transitions to an error state if no other transition can be made. The language of p is then

$$\mathcal{L}(p) = \{\sigma \mid \Delta_p(\{q_0\}, \sigma) \cap F \neq \emptyset\}.$$

Two open automata can be combined by synchronizing on hole symbols:

Definition 6 (Combination). *Given open automata* $p_1 = \langle \Sigma_1, Q_1, q_1^0, \delta_1, F_1, q_1^\perp \rangle$ *and* $p_2 = \langle \Sigma_2, Q_2, q_2^0, \delta_2, F_2, q_2^\perp \rangle$*, their combination 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*

1. $(q_1, a, q_1') \in \delta_1$*, or* $a \in \Sigma_2 \backslash \Sigma_1$ *and* $(q_1, \bullet, q_1') \in \delta_1$
2. $(q_2, a, q_2') \in \delta_2$*, or* $a \in \Sigma_1 \backslash \Sigma_2$ *and* $(q_2, \bullet, q_2') \in \delta_2$

This combination is sound:

Proposition 1. *For any two open automata* p_1 *and* p_2 *we have* $\mathcal{L}(p_1) \cap \mathcal{L}(p_2) = \mathcal{L}(p_1 \cap p_2)$.

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

$$S_1 = \Delta_{p_1}(\{q_1^0\}, \tau) \quad S_2 = \Delta_{p_1}(\{q_1^0\}, \tau) \quad S = \Delta_{p_1 \cap p_2}(\{(q_1^0, q_2^0), \tau)$$

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:

$$S_1 = \{q_1^0\} \quad S_2 = \{q_2^0\} \quad S = \{(q_1^0, q_2^0)\}$$

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,

$$S_1 = \{q' \mid q \in S_1^\sigma \wedge (q, \alpha, q') \in \delta_1\} \quad S_2 = \{q' \mid q \in S_2^\sigma \wedge (q, \alpha, q') \in \delta_2\}.$$

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:

$$\begin{aligned} S &= \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge ((q_1, q_2), \alpha, (q_1', q_2')) \in \delta\} \\ &= \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1 \wedge (q_2, \alpha, q_2') \in \delta_2\} \\ &= \{(q_1', q_2') \mid q_1 \in S_1^\sigma \wedge q_2 \in S_2^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1 \wedge (q_2, \alpha, q_2') \in \delta_2\} \\ &= \{q_1' \mid q_1 \in S_1^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1\} \times \{q_2' \mid q_2 \in S_2^\sigma \wedge (q_2, \alpha, q_2') \in \delta_2\} \end{aligned}$$

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

$$S_1 = \{q' \mid q \in S_1^\sigma \wedge (q, a, q') \in \delta_1\} \quad S_2 = \{q' \mid q \in S_2^\sigma \wedge (q, \bullet, q') \in \delta_2\}$$

and again by rewriting S by expanding δ for $p_1 \cap p_2$ and using our induction hypothesis we get

$$\begin{aligned} S &= \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge ((q_1, q_2), a, (q_1', q_2')) \in \delta\} \\ &= \{(q_1', q_2') \mid (q_1, q_2) \in S^\sigma \wedge (q_1, a, q_i') \in \delta_i \wedge a \in \Sigma_i \backslash \Sigma_j \wedge (q_j, \bullet, q_j') \in \delta_j\} \\ &= \{(q_1', q_2') \mid q_1 \in S_1^\sigma \wedge q_2 \in S_2^\sigma \wedge (q_i, a, q_i') \in \delta_1 \wedge (q_j, \bullet, q_j') \in \delta_j\} \\ &= \{q_1' \mid q_1 \in S_1^\sigma \wedge (q_1, \alpha, q_1') \in \delta_1\} \times \{q_2' \mid q_2 \in S_2^\sigma \wedge (q_2, \alpha, q_2') \in \delta_2\} \end{aligned}$$

Finally, open automata can be instantiated with a mapping from symbols to ground events.

Definition 7 (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* $p(\varphi) = \langle GEvent, Q, q_0, \delta(\varphi), F, q_\perp \rangle$ *where* $(q, \varphi(a), q') \in \delta(\varphi)$ *iff* $(q, a, q') \in \delta$.

3.2 Pattern libraries

Our approach takes a predefined pattern library as an input. A k-pattern library is a list of patterns over a fixed set of k metasymbols, which we will take as the first k letters of the alphabet i.e. $\{a, b, c, \ldots\}$. A pattern library is a set of k-pattern libraries. We can have any value k, but generally small patterns seem to be sufficient for capturing meaningful specifications.

The pattern library will determine the specifications that may be mined. Whilst we do not present a methodology for developing an appropriate pattern library (this remains further work) we have built a library of (around 150) useful patterns for $k \leq 3$ that fit into these broad categories:

- *Restriction* - restricting symbols to certain parts of the trace e.g. the end or before another symbol.
- *Alternation* - the order in which symbols may alternate.
- *Choice* - using symbols to select next behaviour.
- *Combination* - allowing other patterns to be combined e.g. sequentially or in alternation.

The size of this library is large as we need to capture many variations of a simple pattern - with holes in different positions. Later we will see that the size of the library does not significantly effect the running time of the mining process. In our framework, patterns can be generated automatically based on some criteria or entered manually. Much of the library has been systematically generated, but some patterns have been added as the result of exploratory study. This is part of the nature of the framework - patterns can be easily added if some prior knowledge of possible behaviour is available.

3.3 Pattern Checkers

We now build a generation function G that uses a pattern library and an alphabet of events to realise a function from traces of events to sets of instantiated patterns. We begin by introducing a *pattern checker* that maps a trace of metasymbols to a set of patterns that will accept the trace.

Definition 8 (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*

$$\mathsf{C}(\tau) = \{p \in \Gamma(q) \mid q_0 \stackrel{\tau}{\Rightarrow} q\}$$

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 $0 < i < |\mathsf{L}|$
- $\langle \ldots, S_i, \ldots \rangle \stackrel{a}{\Rightarrow} \langle \ldots, S'_i, \ldots \rangle$ iff for $0 < i < |\mathsf{L}|$

$$S'_i = \begin{cases} \mathsf{L}[i].\delta(q^i, a) & \text{if } a \in \Sigma \\ \mathsf{L}[i].\delta(q^i, \bullet) & \text{otherwise} \end{cases}$$

- $\mathsf{L}[i] \in \Gamma(\langle \ldots, q^i, \ldots \rangle)$ 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 $\mathsf{L}[i]$.

An *event pattern checker* consists of a pattern checker and an instantiation and maps a trace of events to a set of instantiated pattern that accept the trace. Let $\varphi^{-1} \in GEvent \rightharpoonup \Sigma$ be the inverse of instantiation φ.

Definition 9 (Event Pattern Checker). *An event pattern checker $\mathsf{E} = \langle \mathsf{C}, \varphi \rangle$ with pattern checker C and instantiation φ produces the following instantiated pattern from the trace $\tau \in GEvent^*$:*

$$\begin{aligned} \mathsf{E}(\tau) = \{p(\varphi) \mid \exists \sigma \in \mathsf{C}.\Sigma^* : |\sigma| = |\tau| \wedge p \in \mathsf{C}(\sigma) \wedge \\ \forall i.0 \leq i < |\sigma|.\exists \mathbf{b} \in \varphi^{-1}(\sigma[i]).\mathsf{matches}(\tau[i], \mathbf{b})\} \end{aligned}$$

i.e. 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 lift this concept to a set of event pattern checkers F such that F's successful patterns for trace τ are

$$\mathsf{F}(\tau) = \bigcup_{\mathsf{E} \in \mathsf{F}} \mathsf{E}(\tau)$$

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:

$$\mathsf{G}(\mathsf{L}, \mathcal{A}) = \{\langle \mathsf{C}, \varphi \rangle \mid \forall a \in \mathsf{C}.\Sigma : a \in \mathsf{dom}(\varphi) \wedge \varphi(a) \in \mathcal{A}\}$$

Finally, the generation function for pattern library P and alphabet of events $\mathcal{A}$ is

$$\mathsf{G}(\mathsf{P}, \mathcal{A}) = \bigcup_{\mathsf{L} \in \mathsf{P}} \mathsf{G}(\mathsf{L}, \mathcal{A})$$

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.

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. The compiled version of our pattern library consists of 3 pattern checkers for 1, 2 and 3 symbols that have 38, 1159 and 9 states respectively.

3.4 Example

Here we demonstrate how pattern checkers are generated - later we will show how they are used. Consider the six patterns in Fig. 4. The first three are 2-patterns as they haven alphabets of size 2, 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.

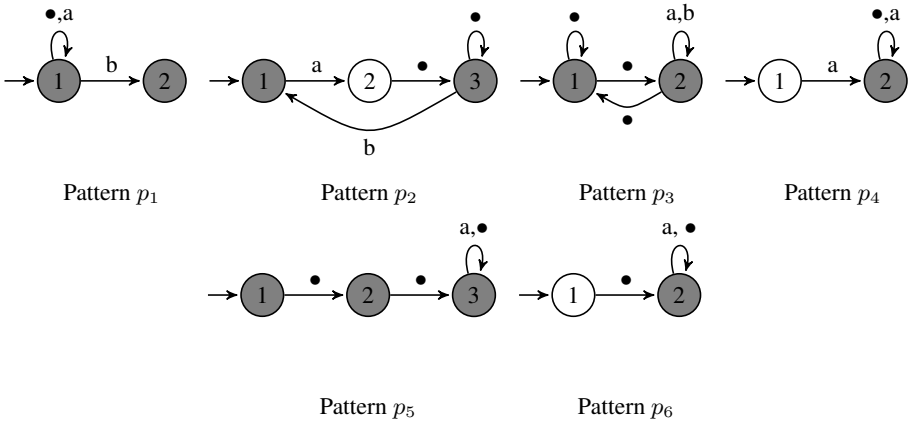


Fig. 4: An example (small) pattern library

The pattern checker for the first three patterns is given in Fig. 5 and the pattern checker for the second three patterns is given in Fig. 6. The patterns written on each state are those given by the output function for that state.

4 Mining Patterns

In this section we describe how we mine QEA using patterns (i.e. open automata). Our approach is split into three main stages, as outlined in Figure 7. Here we give an overview of these three stages.

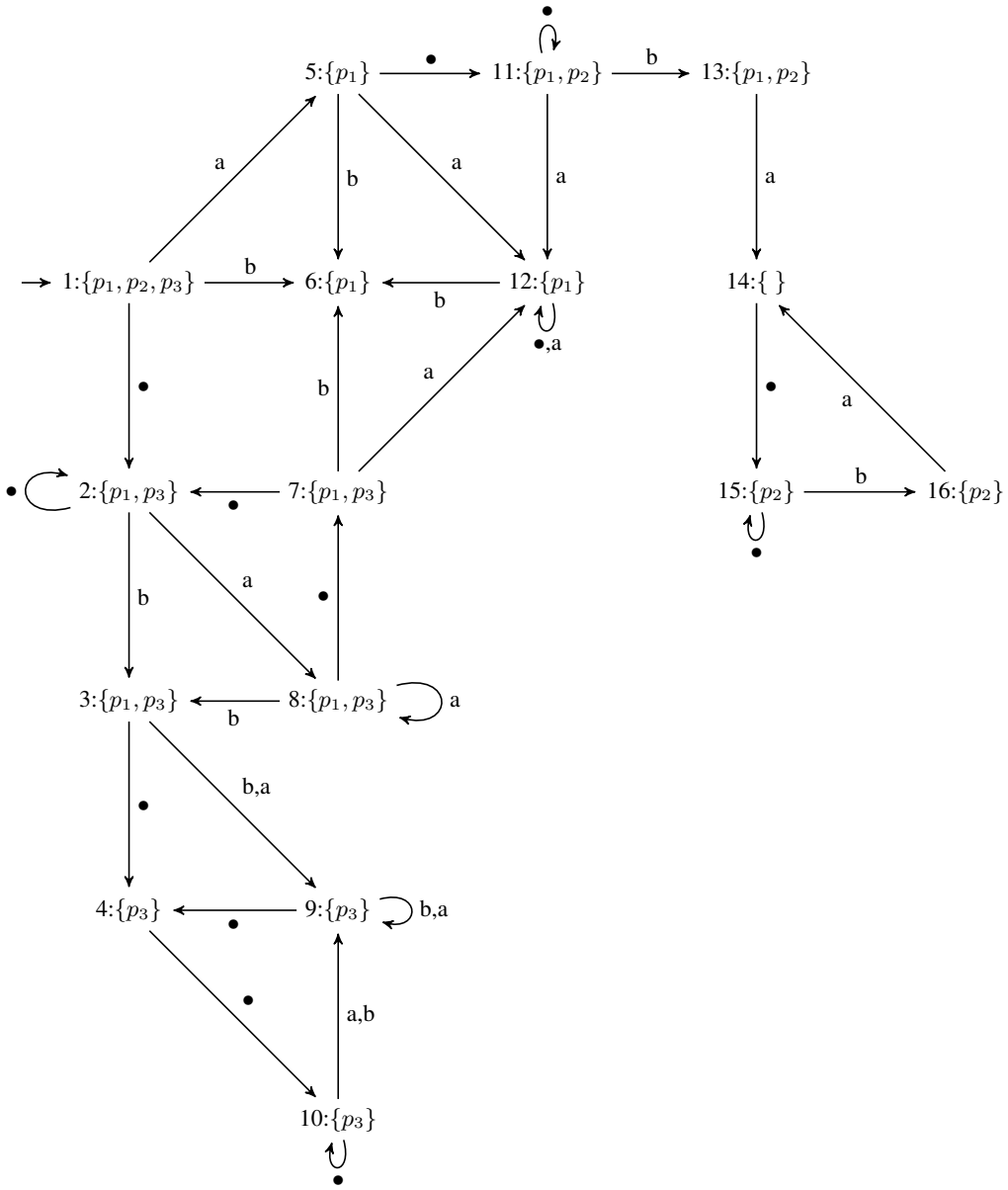


Fig. 5: An example pattern checker for a 2-pattern library.

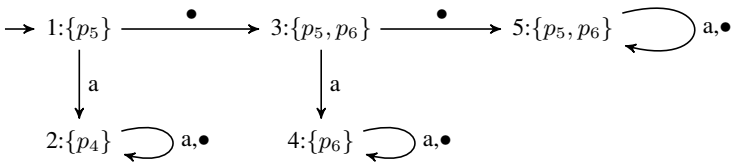


Fig. 6: An example pattern checker for a 1-pattern library.

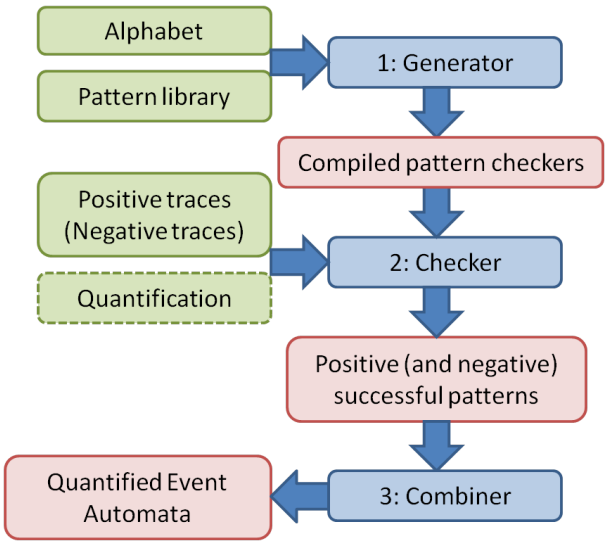


Fig. 7: An overview of the tool.

1. **Generator.** Patterns in the pattern library are defined in terms of abstract symbols and the job of the generator stage is to use the given alphabet to instantiate those abstract symbols to produce patterns which can be checked against the trace. 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 takes a pattern library and a set of events and produces a set of event pattern checkers.
2. **Checker.** Projection (Definition 3) and event pattern checkers are used to find the patterns which hold in each trace for each binding of quantified variables. A list of quantifications is used to select the positive and negative patterns - if no list of quantifications is supplied all possible quantifications are explored.
3. **Combiner.** The notion of combination for open automata is used to produce an instantiated open automaton that, with the quantification list, forms a QEA.

We have implemented this framework in the `Scala` programming language.

4.1 Restricting the search space

The *monitoring process* takes as input the following:

- A pattern library P
- An alphabet of events $\mathcal{A}$
- A list of quantifications Λ over X, the variables in $\mathcal{A}$
- A set of positive T_+ and a (possibly empty) set of negative T_- traces

The list of quantifications Λ need not be given as all possible values for Λ can be explored efficiently after the trace has been traversed, however this will result in a separate specification per quantification list. In general, the assumption would be that if a specification can be found for all universal quantification then this is the most general specification, however this may be an over-generalisation.

It is generally difficult to generate negative traces and the monitoring process does not require them. We include negative traces as they serve as a 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.

4.2 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 10 (Scorecard). *A scorecard is a map from bindings to sets of patterns:*

$$\mathsf{Scorecard} = \mathit{Binding} \rightharpoonup 2^{\mathit{Pattern}}$$

We use the notion of projection and the generator function G to generate a scorecard $\mathsf{S}(\tau)$ for a trace τ:

Definition 11 (Scoring). *Given a pattern library* P*, alphabet* $\mathcal{A}$*, set of quantified variables* X *and trace* τ*, let the set of relevant bindings be*

$$\mathsf{relevant} = \{\theta \mid \mathsf{dom}(\theta) = X \wedge \forall x \in X : \theta(x) \in \mathsf{dom}(\tau)(x)\}$$

and the scorecard $\mathsf{S}(\tau)$ *given as*

$$\mathsf{S}(\tau) = \langle \theta \mapsto \mathsf{G}(\mathsf{P}, \mathcal{A}(\theta))(\tau \downarrow^{\theta}_{\mathcal{A}(\theta)}) \mid \theta \in \mathsf{relevant} \rangle$$

This definition can be implemented directly to produces scorecards. However, it requires passing over each trace multiple times. In *Runtime Verification*, techniques have been introduced which may be adapted here to produce more efficient implementations. For example, a simple optimisation would be to pass over the trace once to construct relevant and once to compute the patterns for each binding θ by keeping track of the states reached in the event pattern checkers. Further details are beyond the scope of this paper and we refer the reader to work discussed in [5].

Once scorecards have been constructed the list of quantifications is used to extract successful patterns. This is achieved by following the approach given in Definition 4 for QEA acceptance.

Definition 12 (Successful patterns). *Given a scorecard* $\mathsf{S}(\tau)$*, the successful patterns for the list of quantifications* Λ *are given by* $\mathsf{pat}(\mathsf{S}(\tau), \langle\,\rangle, \Lambda)$*, defined as*

$$\begin{aligned}
\mathsf{pat}(\mathsf{S}, \theta, \forall x \Lambda') &= \textstyle\bigcap_{d \; in \; \mathsf{dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger \langle x \mapsto d \rangle, \Lambda') \\
\mathsf{pat}(\mathsf{S}, \theta, \exists x \Lambda') &= \textstyle\bigcup_{d \; in \; \mathsf{dom}(\tau)(x)} \mathsf{pat}(\mathsf{S}, \theta \dagger \langle x \mapsto d \rangle, \Lambda') \\
\mathsf{pat}(\mathsf{S}, \theta, \epsilon) &= \mathsf{S}(\theta)
\end{aligned}$$

The positive and negative successful patterns are therefore

$$\mathsf{suc}_\alpha = \textstyle\bigcap_{\tau \in T_\alpha} \mathsf{pat}(\mathsf{S}(\tau), \langle\,\rangle, \Lambda) \qquad \text{for } \alpha \in \{-, +\}$$

4.3 Combining patterns

This stage straightforwardly combines the successful positive and negative patterns produced by the checking stage.

Definition 13 (Combination). *Given a set of positive successful patterns* suc_+ *and a set of negative successful patterns* suc_-*. Their combination is given as*

$$\left(\bigcap_{p \in \mathsf{suc}_+} p \right) \bigcap \overline{\left(\bigcap_{p \in \mathsf{suc}_-} p \right)}$$

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 QEA via removing holes is straightforward. The size of the pattern library can affect combination time as it will limit the number of successful patterns.

4.4 Complexity

To calculate the 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 $O(|p|^{|\mathsf{L}|})$.

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 represent 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 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.

4.5 Limitations of the mining technique

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. Some techniques [24] use the notions of *support* and *confidence* from the field of pattern-mining to deal with 'almost correct' traces. An alternative approach is taken in [32], which partitions the trace and checks if their alternating pattern holds in each partition. In either case it is not immediately clear how to apply this to our general notion of patterns as automata.

There are two other limitations stemming from the reliance on prior knowledge. Firstly, 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. Secondly, 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 [17], 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.

4.6 Example

We demonstrate the mining process by showing how we might extract the QEA in Fig. 2. Consider a system that deals with files which we instrument to output events from the following alphabet:

$$\mathcal{A} = \{\mathtt{open}(f), \mathtt{read}(f), \mathtt{write}(f), \mathtt{close}(f), \mathtt{delete}(f)\}$$

We then take the following trace:

$$\begin{aligned}\tau_1 = \ & \mathtt{open}(1).\mathtt{write}(1).\mathtt{open}(2).\mathtt{read}(2).\mathtt{read}(1).\mathtt{close}(1).\mathtt{write}(2).\\ & \mathtt{open}(1).\mathtt{read}(1).\mathtt{write}(1).\mathtt{close}(2).\mathtt{open}(2).\mathtt{read}(2).\mathtt{write}(1).\\ & \mathtt{read}(1).\mathtt{delete}(1).\mathtt{write}(2).\mathtt{delete}(2)\end{aligned}$$

Lastly, let us take the pattern library given in Sec. **??** and therefore, the pattern checkers given in Fig. 5 and Fig. 6 - let us call these 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

$$\begin{aligned}\tau_1 \downarrow_{\mathcal{A}([f\mapsto 1])} &= \sigma_1 = \mathtt{open}(1).\mathtt{write}(1).\mathtt{read}(1).\mathtt{close}(1).\mathtt{open}(1).\mathtt{read}(1).\\ &\qquad \mathtt{write}(1).\mathtt{write}(1).\mathtt{read}(1).\mathtt{delete}(1)\\ \tau_1 \downarrow_{\mathcal{A}([f\mapsto 2])} &= \sigma_2 = \mathtt{open}(2).\mathtt{read}(2).\mathtt{write}(2).\mathtt{close}(2).\mathtt{open}(2).\mathtt{read}(2).\\ &\qquad \mathtt{write}(2).\mathtt{delete}(2)\end{aligned}$$

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. Finally, we must compute the set of pattern instantiates given by the trace for each event pattern checker.

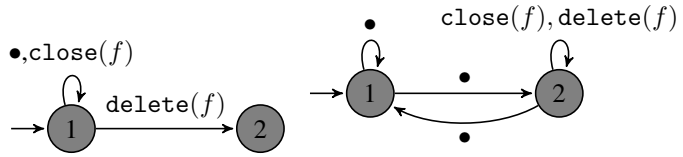
Before we give the results, let us briefly consider the patterns extracted from σ_2 by the event pattern checker $\langle \mathsf{C}_3, [a \mapsto \mathtt{close}(f), b \mapsto \mathtt{delete}(f)], [f \mapsto 2]\rangle$ in detail. The following table shows the states that the trace σ_2 passes through.

σ_2	State
open(2)	2
read(2)	2
write(2)	2
close(2)	8
open(2)	7
read(2)	2
write(2)	2
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 connected to 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. 1 and 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.

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}

Table 1: The reached states for traces σ_1 and σ_2 for the patterns and their event checker for the 2-pattern library.

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. 3 describes the successful 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 QEA. Figure 8 gives the combined open automata that is the result of combining all successful patterns in Table. 3 together. Figure. 2 gives the resulting QEA - holes can be removed as we can take the universal set of symbols to be the alphabet.

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 2: The reached states for traces σ_1 and σ_2 for the patterns and their event checker for the 1-pattern library.

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)	$\{p_5,p_6\}$

Table 3: The successful patterns.

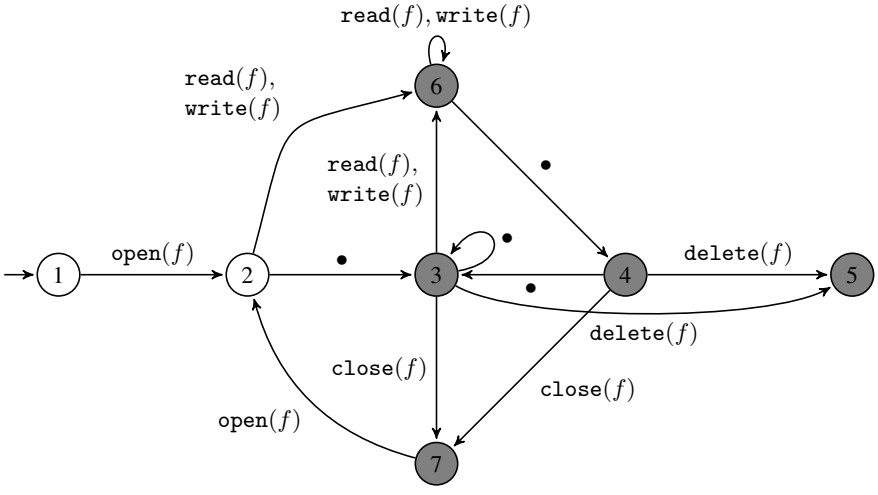


Fig. 8: The combined open automata with holes.

5 Evaluation

In this section we establish the viability of our approach by measuring its ability to extract specifications from traces generated from a range of models.

Experiments were carried out on an Apple Mac Pro with two 2.26GHz quad-core Intel Xeon processors and 16GB of memory. The pattern library used was built using a mixture of automatic generation and intuition and was developed independently from the chosen models and prior to evaluation.

5.1 Selecting an evaluation method

There are two approaches found in the literature that are used to evaluate specification mining methods. The first (e.g.[20, 24, 31]), derived from machine learning, uses a set of representative specifications to generate training and test sets of traces, builds a model from the training set and measures its accuracy using the test set. The second (e.g.[3, 13, 15, 17, 32]) is an explorative approach that selects one or more real-world application and applies 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.

We have chosen to focus on the first approach as it gives a strong quantitative measure of the effectiveness of the technique and allows us to explore the effects of targeting different kinds of specifications with varying levels of information. 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 from the literature.

5.2 Evaluation setup

As illustrated in Fig. 9 our evaluation will follow four steps:

1. Select a set of realistic reference models capturing a range of different kinds of specifications.
2. Randomly generate training and test traces, with different categories of training traces and both positive and negative test traces.
3. For each category of training trace, use the tool to extract a specification.
4. Use the test traces to compute the precision and recall of the extracted specification.

We discuss these stages further below.

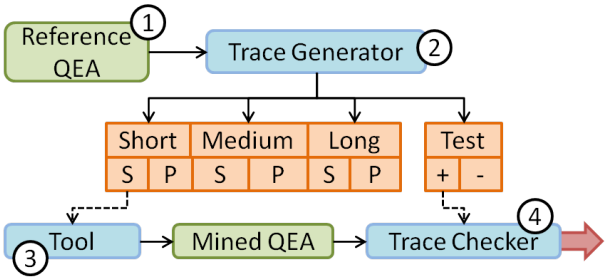


Fig. 9: An overview of the evaluation setup.

5.3 The models

For our evaluation we use ten predefined models to represent realistic specifications we might expect to extract. Table 4 gives an overview of the models used, including the number of states, events and quantified variables. These specifications can be grouped into the following domains:

- **Communication.** *James* is taken from [17] as a specification extracted from the Apache James mail server program. *Satellites* and *Commands* are (simple) specifications of correct communication between planetary rovers, inspired by the work of [6].
- **Java API.** *SocketOutput* and *ColIter* describe the correct usage of data structures in the Java API.

Name	Description	$\|Q\|$	$\|\mathcal{A}\|$	$\|X\|$
James	The SMTP protocol (without authentication) as used in the Apache JAMES mail server	7	5	1
Satellites	There exists a satellite that has established a communication link with all known field units.	4	2	2
Commands	Issued commands succeed and are only reissued after failure.	4	4	1
SocketOutput	A stream is used after being connected to a socket and not after that socket is closed.	6	3	2
ColIter	An iterator created from a collection is not used after the collection is updated.	10	5	2
MutualExcl	No lock should be held by two threads at the same time.	4	4	3
LockOrder	Locks are always taken in a consistent order.	12	4	2
IOCallDriver	The I/O stack must be setup before calling the IOCallDriver	4	2	1
KeAcquireSpinLock	Locks and releases of a spin lock alternate, with two alternate locking calls.	3	3	1
ZwRegistryCreate	Registry keys are created before being used, open when used and not used after being deleted.	5	5	1

Table 4: Target specifications

– **Concurrency.** *MutualExcl* and *LockOrder* describe common desired concurrent behaviour.
– **Drivers.** The last three models are based on rules described in the SDV tool [1] and were taken from [24] as interesting specifications to address.

As well as covering a variety of domains, these ten models also capture different levels of complexity in number of states, size of alphabet and number of quantified variables. Satellites is the only model that uses existential quantification.

5.4 Generating traces

We used the predefined models to generate traces for both 'training' and testing.

Coverage To explore how the mining process is effected by the quality of the data provided we consider two coverage criteria for traces. This is similiar to the approach taken by [22], however is updated for our scenario with quantifications and non prefix-closed automata.

– 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

Random generation To randomly generate traces from QEA we require a set of possible values for each quantified variable to give us a set of possible ground events. The general trace-generation method randomly generates traces over these ground events and checks whether they are accepted by the given QEA. We refine this approach in two ways. Firstly, we avoid selecting events that would automatically lead to failure by verifying the trace incrementally. Secondly, we note that if the alphabets for different bindings are disjoint then we can generate a trace per binding and merge these together.

Generated traces We generate short, medium and long traces for each coverage level. We generate a range of sizes as we expect that traces containing more information will lead to more accurate specifications. The sizes are chosen so that the short traces should give a minimum demonstration of behaviour and the long traces should represent realistic usage. We generate 10 traces for each training set and 100 traces for each testing set. Table 5 gives the average length of the generated traces - the dashes indicate that no traces were produced as the coverage criteria were not met.

5.5 Measurements

We evaluate our technique for accuracy and efficiency.

Model	Training						Test	
	Short		Medium		Long			
	State	Path	State	Path	State	Path	Positive	Negative
James	12	22	31	23	410	140	137	829
Satellites	-	5	-	11	-	93	57	31
Commands	-	12	-	44	-	1620	2526	2160
SocketOutput	5	7	12	15	27	27	39	82
ColIter	10	-	68	43	709	94	39	122
MutualExcl	1	3	-	26	-	100	10	40
LockOrder	9	16	13	23	-	88	10	18
IOCallDriver	2	8	-	42	-	2036	1318	1462
KeAcquireSpinLock	5	12	-	56	-	1518	951	907
ZwRegistryCreate	25	39	44	66	554	666	356	798

Table 5: The average lengths of traces for each size and coverage level.

Accuracy To measure the accuracy of the mining process we use the common precision-recall evaluation measures taken from the field of information retrieval [26]. *Recall* measures the mined specification's ability to identify correct behaviours. *Precision* measures the extent to which incorrect traces are rejected by the mined specification. To compute these measures we test the randomly generated traces on the extracted specifications and add them to REL (relevant) and RET (retrieved) sets as follows:

Reference model	Extracted model	RET	REL
accept	accept	add	add
accept	reject		add
reject	accept	add	
reject	reject		

Precision and recall are then defined as

$$\text{precision} = \frac{|\text{REL} \cap \text{RET}|}{|\text{RET}|} \qquad \text{recall} = \frac{\text{REL} \cap \text{RET}|}{|\text{RET}|}$$

Efficiency It is necessary to demonstrate that the technique can be applied to realistic traces, therefore we measure the time it takes to extract specifications, focussing separately on checking and combining times.

5.6 Results

Model	Recall						Precision					
	Short		Medium		Long		Short		Medium		Long	
	State	Path	State	Path	State	Path	State	Path	State	Path	State	Path
James	0.55	0.56	0.23	0.75	0.05	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Satellites	-	1.0	-	1.0	-	1.0	-	1.0	-	1.0	-	1.0
Commands	-	0.28	-	0.78	-	0.78	-	1.0	-	1.0	-	1.0
SocketOutput	0.08	1.0	1.0	1.0	1.0	1.0	1.0	0.98	0.98	0.98	0.98	0.98
ColIter	0.0		0.75		0.87	0.4	0.0		0.94		0.94	0.97
MutualExcl	-	1.0	-	0.64	-	-	-	1.0	-	1.0	-	-
LockOrder	0.0	0.0	1.0	1.0	-	1.0	0.0	0.0	1.0	1.0	-	1.0
IOCallDriver	0.0	1.0	-	0.67	-	0.67	0.0	1.0	-	1.0	-	1.0
KeAcquireSpinLock	0.09	1.0	-	1.0	-	1.0	1.0	1.0	-	1.0	-	1.0
ZwRegistryCreate	0.44	0.4	1.0	0.56	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 6: Precision and Recall results

The results of our evaluation are detailed in Table 6, giving precision and recall results, Table 7, giving efficiency results, and Table 8, describing the size of mined specifications. Overall the results are positive, with even the more complex specifications being reconstructed well.

Model	Times		Patterns passed
	Check	Combine	
James	1.98	11.68	79
Satellites	0.365	0.209	144
Commands	4.474	19.77	39
SocketOutput	0.515	0.299	22
ColIter	3.818	175.0	33
MutualExcl	24.38	2.761	1824
LockOrder	2.627	0.044	8
IOCallDriver	1.145	0.142	78
KeAcquireSpinLock	1.833	0.043	20
ZwRegistryCreate	2.435	1.568	341

Table 7: Efficiency results - times in seconds

Model	Short		Medium		Long	
	State	Path	State	Path	State	Path
James	13	17	9	13	13	10
Satellites	-	4	-	4	-	4
Commands	-	21	-	15	-	15
SocketOutput	7	4	8	4	8	8
ColIter	13		101		34	136
MutualExcl	-	13	-	17	-	-
LockOrder	29	13	5	5	-	5
IOCallDriver	4	4	-	5	-	5
KeAcquireSpinLock	6	4	-	4	-	4
ZwRegistryCreate	9	8	6	10	6	6

Table 8: The size of mined specifications.

There were only two cases where a specification was not identified - both for MutualExcl. It is likely that this is due to the necessary patterns, i.e. the ones present in the successful cases, failing to hold for all bindings (the 3 quantified variables increase the chances of this) - recall that, with universal quantification, a pattern is included only if it is true for all projections of the trace. An interesting extension would be to consider cases where patterns hold for 'almost' all bindings.

Looking at Table 8 we can see that generally mined specifications are larger than their reference models. In the extreme, the mined specification for ColIter with long traces and path coverage has 136 states, compared to the reference model's 10. This explosion in size is due to capturing non-existent orderings between events.

Interestingly, the mined specification for LockOrder with long traces and path coverage achieves full recall and precision even though it has 5 states and the reference model has 12. The reference model describes the order in which locks can be taken and contains symmetric behaviour relating to when each of the two locks are taken first. However, this has some non-determinism introduced via the quantified variables ($\texttt{lock}(l_1)$ and $\texttt{lock}(l_2)$ when $l_1 = l_2$) and the extracted model is a minimised version of the model that makes use of this non-determinism to omit one half of the behaviours. This demonstrates that our technique an extract *concise* specifications.

We now discuss results for recall, precision and efficiency individually.

Recall We expect mined specifications to be reasonable at accepting correct traces as we are generalising from a set of accepting traces, effectively taking this set and adding similar traces that we expect also to be accepted.

Generally, path-cover leads to better recall than state-cover, and longer traces lead to better recall. However, there are a few cases where this does not hold. In all of these cases the mined specification incorrectly conclude that loops must be unfolded a number of times as there are no short traces in their training sets. This demonstrates how over-precise patterns in the pattern-library can prevent useful generalisation, and the need for traces of varying lengths. In the case of James we have very poor recall for state-cover . A likely reason for this is that James has a single accepting state (when the protocol completes with a quit message) with many different paths to it, which are not fully captured by shorter traces. Recall that path-coverage only means that all paths are covered in the trace as a whole, not each individual per-binding subtrace.

There are four cases where we have zero recall (and precision) as the mined specifications reject all trace tests. In all cases this was due to loops not being exercised in any trace in the training set and then being used in every trace in the test set. In practice these training traces were unrealisticly short.

Precision Mined specifications are generally precise. As expected, path coverage leads to better precision than state coverage. In some cases shorter traces lead to better precision. This is due to longer traces including 'noise' causing patterns with irrelevant parts to be matched and included in the mined specification.

Efficiency Table 7 records the time taken in the check and combine stages (the generate stage is a pre-compilation stage that took under a minute) and the number of patterns being combined for the long traces with path coverage. We focus on this category as it contains the most complex traces and should demonstrate realistic traces.

Let us first consider checking time. There is no clear correlation between these times and the lengths of traces being used. Instead the correlation can be found with the size of alphabets being checked, as this determines the number of event pattern checkers that must be updated for each event. Note that the number of quantified variables also has an effect - demonstrated in the case of MutualExcl.

Let us now consider combining time. The first thing we note is that there is no strong correlation between number of patterns passed and the time it takes to combine them. In this case of MutualExcl it takes under 3 seconds to combine 1824 patterns, yet in the case of ColIter it takes 175 seconds to combine 33 patterns (as the resultant specification has 136 states these 33 patterns are likely to be complex). Again this is due to the relative alphabet sizes - combining two open automata with the same symbols is linear and the likelihood of successful patterns sharing symbols increases greatly with shorter alphabets.

As we might expect, and as we concluded when discussing the complexity of our approach, the size of the alphabet plays a large part in deciding the time it takes to extract a specification. However, the pattern-library also plays a key part, it is likely that in the cases where combination was very expensive many of the successful patterns were redundant.

5.7 Summary

We have demonstrated that our technique can effectively mine specifications for target systems of varying complexity and traces of varying levels of length and coverage. As expected, more information leads to richer specifications, although in many cases accurate specifications were mined from little information. In addition, having only long traces was detrimental as it caused the mined specification to exclude shorter traces. In terms of performance, the limiting factors are size of alphabet and number of quantified variables.

5.8 Validity of Results

There are a two ways in which the validity of our results could be questioned. Firstly, that the models used are not representative of realistic specifications. To combat this threat we selected models based on real-world scenarios and focused on a range of domains to ensure that our approach was not biased towards a certain common structure such as resource usage. Secondly, that the randomly generated traces are not representative of realistic traces. However, the coverage criteria we used were relatively weak - we did not require that every accepting state accept a trace, only that they were visited, and in the path-coverage section we only require each path be visited at least once.

6 Examining the Java standard library

We apply our technique to the `Java` standard library as used by five benchmarks from the DaCapo benchmark suite [9]. We identify likely specification alphabets by manually examining the library documentation and instrument the benchmarks using `AspectJ` to record instances of each event. Table 9 records the results, demonstrating that our technique scales to traces taken from real-world systems. We discuss these further below.

Iterators. Figure 1 gives a QEA mined for the `java.util.Iterator` class. This specifies that `hasNext` must be called on each iterator and return true before calling `next` on that iterator, and that after `hasNext` returns false the iterator can no longer be used. A local-guard was introduced to capture the value returned by `hasNext`.

Table 9: Results for the Java standard library.

Class name	total trace lengths	Successful patterns	Time (sec)	
			Check	Combine
`Iterator`	14,832,710	4	1180	6
`InputStream`	51,357	20	18	5
`OutputStream`	8,427,358	57	725	3

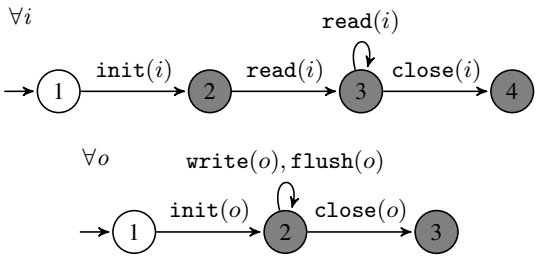


Fig. 10: Mined QEA for `InputStream` (top) and `OutputStream` (bottom) from `java.io`.

Input and Output Streams. The mined QEA for `InputStream` and `OutputStream` from `java.io` are given in Figure 10. The QEA for `InputStream` is more restrictive than expected, requiring that an input stream be read before being closed, as this occurred in all usages of input streams in the mined traces. This demonstrates that the specifications found will depend heavily on the given traces.

7 Related Work

The field of specification mining is growing, and specifically the idea of extending these approaches to deal with data has received much attention recently.

7.1 Mining with quantified parameters.

There are two other approaches that focus on mining parametric specifications with quantified variables. TARK [23, 24] uses techniques from data-mining to identify predetermined quantified binary temporal rules with equality constraints (QBEC). Their focus is on extracting sets of rules rather than a single specification and therefore do not include a notion of combination. However, the use of support and confidence allows them to address imperfections in traces. JMINER [17] mines parametric specifications with universally quantified variables with the restriction that an event name may only occur in the alphabet once. The sk-strings algorithm [29] is used to infer a probabilistic finite state automata from *trace slices* using an approach similar to our trace projection. JMINER provides functionality for automatically discovering likely specification alphabets, whereas TARK considers all events in the trace. Some other techniques can abstract over a single value by slicing the trace [32] or collecting seperate traces per value [10], but do not deal with quantification in a general manner.

Neither approach is able to mine the MutualExcl model, depicted in Figure 11, as JMINER's underlying language cannot represent it and it cannot be decomposed into the binary rules of TARK. Additionally, neither could capture Satellites as it uses existential quantification. Our evaluation included examples of models that were mined by both of these tools.

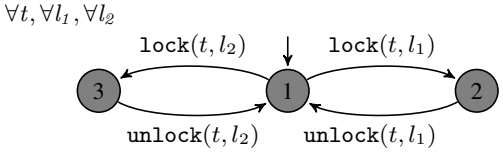


Fig. 11: The MutualExl property as a QEA.

7.2 Mining with free parameters.

There have also been approaches that extract unquantified parametric specifications with free variables and guard transitions in both an *static* and *active* setting.

The *static* setting, which our work is in, assumes a static set of traces is given. gkTail [25] attempts to combine the Daikon invariant-detection tool [11] with a state-merging technique based on the k-tails approach [8] to mine a form of Extended Finite State Machine. Traces are annotated with Daikon and used to construct a canonical automata 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. KLFA [27] extracts data recurrence patterns by augmenting the names of events with symbols representing discovered recurrence patterns and then applying the propositional kBehaviour [28] technique. Lo et al. [21] mine live sequence charts enriched with invariants by first identifying frequent charts and then using these to identify subtraces for invariant mining.

There also exist approaches in the *active* setting that infer parametric specifications with free variables. The active setting assumes the existence of a *teacher* who can answer queries about the target system. Approaches [7, 16, 2] extend Angluin's original L^* algorithm [4] from the propositional to parametric setting. The general approach builds up a consistent *abstraction* function that maps parametric traces to propositional traces. These techniques can infer different forms of unquantified parametric specifications with equality guards.

7.3 The need for quantified parametric specifications.

In [22] Lo et al. explore whether techniques that augment mined models with free variables and guards necessarily produce higher quality models than their propositional counterparts. They compare gkTail and KLFA with their underlying propositional algorithms kTails and kBehaviour and conclude that neither significantly improves precision or recall.

There is, however, 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 $\texttt{open}(1).\texttt{open}(2).\texttt{open}(3).\texttt{close}(1).\ \texttt{close}(3).\texttt{close}(2)$ would not produce the specification that correctly abstracts the data, i.e., $\forall f.\texttt{open}(f).\texttt{close}(f)$.

7.4 Non-parametric pattern-mining

The pattern-mining approach taken in this paper has been previously explored in the context of propositional specification mining. The approach was first taken by Yang et al. [32] in the Peracotta tool and later extended by Gabel and Su [13, 14] in the Javert tool. These approaches use a small (< 10) set of patterns in the form of DFA, which are composed using predetermined rules based on combination. Li et al. [18] introduce patterns based on Linear Temporal Logic and related composition rules. Our approach extends these approaches by introducing the concept of open automata.

8 Conclusion

In this paper we positively answer the question 'can quantified event automata be used in specification mining?' by describing and evaluating a pattern-based technique for mining QEA from sets of parametric traces using a given alphabet. We have established that our technique is good at extracting specifications that simulate complex specifications well.

This work is significant as the form of specifications that can be mined are richer than previous approaches. Importantly, this work also lays the foundations for further work allowing us to extend our specification mining technique to the full expressiveness of QEA, incorporating both free and quantified variables, along with transition guards and assignments.

Extending our technique to the form of QEA described in [5] will involve extracting likely guards by checking 'guard-patterns' against sequences of bindings of free variables, and updating our notion of open automata combination to reason about how variables are updated in holes. Two further extensions that should be considered are the automatic identification of likely alphabets by applying heuristics to traces or the source code that produces them, and introducing techqniques for dealing with imperfect traces.

References

1. Windows Driver Development. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff551714\%28v=vs.85\%29.aspx`.
2. 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.
3. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, Jan. 2002.
4. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
5. 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.
6. 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.
7. T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines with parameters. In *FASE*, pages 107–121, 2006.
8. 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.
9. 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 (extended version). Technical Report TR-CS-06-01, ANU, 2006. http://www.dacapobench.org.
10. V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 17–24, New York, NY, USA, 2006. ACM.
11. M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 449–458, New York, NY, USA, 2000. ACM.
12. 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*. IOS Press, 2013.
13. M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.
14. M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 51–60, New York, NY, USA, 2008. ACM.
15. 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.
16. 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.
17. 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.
18. 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.
19. D. Lo, K. Cheng, and J. Han. *Mining Software Specifications: Methodologies and Applications*. Chapman and Hall/CRC Data Mining and Knowledge Discovery Series. Taylor & Francis Group, 2011.
20. D. Lo and S. cheng Khoo. Quark: Empirical assessment of automaton-based specification miners. In *In WCRE*, pages 51–60, 2006.
21. D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. *Autom. Softw. Eng.*, 19(4):423–458, 2012.
22. D. Lo, L. Mariani, and M. Santoro. Learning extended fsa from software: An empirical assessment. *J. Syst. Softw.*, 85(9):2063–2076, Sept. 2012.
23. D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Mining quantified temporal rules: Formalism, algorithms, and evaluation. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 62–71, Washington, DC, USA, 2009. IEEE Computer Society.
24. 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.
25. 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, New York, NY, USA, 2008. ACM.
26. C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
27. 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.
28. L. Mariani, F. Pastore, M. Pezzè, and M. Santoro. Mining finite-state automata with annotations. In D. Lo, S.-C. Khoo, J. Han, and C. Liu, editors, *Mining Software Specifications: Methodologies and Applications*, Data Mining and Knowledge Discovery. CRC Press, 2011.

29. A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In *International Conference on Machine Learning*, 1997.
30. 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.
31. 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.
32. 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.