Automata-based Pattern Mining from Imperfect Traces

Automata-based Pattern Mining from Imperfect Traces

Giles Reger
University of Manchester
Oxford Road, M13 9PL
Manchester, UK
regerg@cs.man.ac.uk

Howard Barringer
University of Manchester
Oxford Road, M13 9PL
Manchester, UK
howard@cs.man.ac.uk

David Rydeheard
University of Manchester
Oxford Road, M13 9PL
Manchester, UK
david@cs.man.ac.uk

ABSTRACT

This paper considers automata-based pattern mining techniques for extracting specifications from runtime traces and suggests a novel extension that allows these techniques to work with so-called imperfect traces i.e. traces that do not exactly satisfy the intended specification of the system that produced them. We show that by taking a so-called edit-distance between an input trace and the language of a pattern we can extract specifications from imperfect traces and identify the parts of an input trace that do not satisfy the mined specification, thus aiding the identification and location of errors in programs.

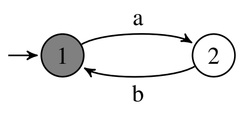
Keywords

Pattern Mining, Specification Mining

1. INTRODUCTION

Formal program specifications are useful for a number of activities but they are often missing or incomplete. The field of specification mining [17] aims to automatically construct formal program specifications from program artifacts. In this work we consider techniques that operate on program traces i.e. finite sequences of events that occur whilst a program is running.

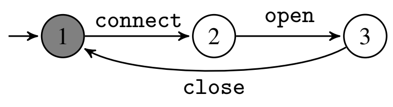
One such approach [19, 5, 6] uses a set of template patterns to detect predefined behaviours and then combine these together to form a specification. Thse regular patterns are described via automata, which allows for efficient checking. For example, consider the following pattern over the metasymbols a and b - shaded states are accepting states.



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 detect three patterns (1) $[a \mapsto$ `connect`$, b \mapsto$ `open`$]$, (2) $[a \mapsto$ `connect`$, b \mapsto$ `close`$]$ and (3) $[a \mapsto$ `open`$, b \mapsto$ `close`$]$. These can then be "combined" to form a larger specification:



This general approach can be used with different patterns and methods of pattern combination. However, it has a major drawback - 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 detect 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 within the given traces. This assumption is not very realistic - 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 [5] to dealing with this issue is to reset a pattern being checked to its initial state when an error occurs - but this technique would not detect the required patterns in our above example . Instead we want to be able to measure how closely a trace matches a pattern. This paper presents an approach that extends the automta-based pattern mining approach to *imperfect traces* by considering so-called *edit distances* between a trace and a pattern's language.

Structure. Section 2 formally introduces the concept of pattern checking and composition. Section. 3 discusses methods for dealing with the imperfect traces problem and Sections 4, 5 and 6 present our proposed solutions. Section. 7 presents two experiments and Section. 8 discusses related work. Finally, we conclude in Section. 9.

2. PATTERN CHECKING

In this section, we introduce a pattern checking framework by first describing how patterns are extracted from traces, then considering how this can be done efficiently and finally discussing how extracted patterns are combined.

2.1 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 1 (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, $\delta \in Q \times \Sigma \rightarrow Q$ 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 2 (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 3 (PROJECTION). *The projection $\pi_\Sigma(\tau)$ of trace τ over alphabet Σ is defined as τ with all elements not in Σ removed.*

Therefore, the detected instantiated patterns are given as follows.

DEFINITION 4 (EXTRACTED PATTERNS). *Given a pattern p and trace τ the extracted patterns* $\mathsf{detect}(p, \tau)$ *are*

$$\{\varphi(p) \mid \mathsf{dom}(\varphi) = p.\Sigma \wedge \forall (a \mapsto s) \in \varphi : s \in \tau \wedge \pi_{\varphi(p).\Sigma}(\tau) \in \mathcal{L}(\varphi(p))\}$$

2.2 Checking patterns efficiently

We discuss two approaches that allow us to check patterns efficiently.

2.2.1 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 in [19]. 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. A more efficient symbolic approach using binary discussion diagrams is explored in [6].

2.2.2 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 labeling states with the patterns that are accepting at that state. This approach was previously presented in [16].

DEFINITION 5 (PATTERN CHECKER). *Given an alphabet of symbols Σ and a set of patterns $p_1, \ldots, p_n$ over Σ let the 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 detected by pattern checker C in trace τ are therefore

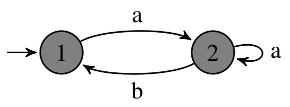
$$\mathsf{C}(\tau) = \{p \mid q_0 \overset{\tau}{\Rightarrow} q \wedge p \in \Gamma(q)\}$$

We can extend the notion of instantiation to pattern checkers and define extracted patterns for a pattern checker as follows.

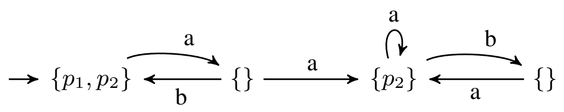
DEFINITION 6 (PATTERN CHECKER EXTRACTED PATTERNS). *Given a pattern checker* C *and trace τ the extracted patterns* $\mathsf{detect}(\mathsf{C}, \tau)$ *are*

$$\{p \mid \exists \varphi : \mathsf{dom}(\varphi) = p.\Sigma \wedge \forall (a \mapsto s) \in \varphi : s \in \tau \wedge p \in \varphi(\mathsf{C})(\pi_{\varphi(p).\Sigma}(\tau))\}$$

For example, if we call the pattern in the introductory example p_1 and call the following pattern p_2



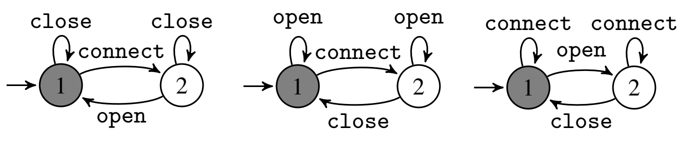
then the pattern checker for p_1 and p_2 would be



where states are labeled using the output function Γ.

2.3 Combining patterns.

The following is based on the technique introduced by Gabel and Su in [5]. 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 detected 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 7 (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.However, this might be costly and not all extracted patterns necessarily contain useful information. Therefore, an alternative approach (which we do not consider further here) is to develop specific combination rules for given patterns, as is done in [5]. For example, they introduce a sequencing rule for the pattern in our introductory example, which we will represent by the regular expression $(ab)^*$.

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

We applied this by taking $\mathcal{L}_1 = \mathcal{L}_2 = \emptyset$. The use of $\mathcal{L}_1$ and $\mathcal{L}_2$ allows for repeated application of the rule. The *closure* of the set of extracted patterns with respect to the set of combination rules can then be computed.

3. DEALING WITH IMPERFECT TRACES

The previous framework will only detect 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.

3.1 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 efforts.

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.

3.2 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.

$$\begin{tikzcd} \arrow[r] & 1 \arrow[r, bend left, "a"] & 2 \arrow[l, bend left, "c"] \arrow[loop right, "b"] \end{tikzcd}$$

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.

3.3 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; 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 [10]. Formally, this distance is given as follows.

DEFINITION 8 (LEVENSHTEIN DISTANCE). *The Levenshtein distance between traces* τ_1 *and* τ_2 *is* $\mathsf{distance}(\tau_1, \tau_2)$, *defined as*

$$\begin{array}{ll} \mathsf{distance}(\tau_1, \epsilon) & = |\tau_1| \\ \mathsf{distance}(\epsilon, \tau_2) & = |\tau_2| \\ \mathsf{distance}(a\tau_1, b\tau_2) & = \min \begin{cases} \mathsf{distance}(\tau_1, b\tau_2) + 1 & \\ \mathsf{distance}(a\tau_1, \tau_2) + 1 & \\ \mathsf{distance}(\tau_1, \tau_2) + 1 & \textit{if } a \neq b \\ \mathsf{distance}(\tau_1, \tau_2) & \textit{if } a = b \end{cases} \end{array}$$

We define an updated notion of extracted patterns using this metric.

DEFINITION 9 (IMPERFECT EXTRACTED PATTERNS). *Given a pattern p, trace* τ *and integer* $\gamma > 0$, *which we call the tolerance, the imperfect extracted patterns* $\mathsf{imperfect\_detect}(p, \tau, \gamma)$ *are*

$$\left\{ \begin{array}{l|l} \varphi(p) & \mathsf{dom}(\varphi) = p.\Sigma \wedge \forall (a \mapsto s) \in \varphi : s \in \tau \wedge \\ & \exists \tau' \in \mathcal{L}(\varphi(p)) : \mathsf{distance}(\tau', \pi_{\varphi(p).\Sigma}(\tau)) < \gamma \end{array} \right\}$$

We extend this definition for pattern checkers as we did before (Sec. 2.2.2).

3.4 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 10 (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 11 (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.

4. 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.

4.1 Failing edit-distance

We introduce an alternative formulation of the edit-distance that only applies edits when a trace 'fails'. We say a pattern fails for a trace if no extensions of the trace can satisfy the pattern. We define this metric as follows.

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 there exists a trace τ' such that $\mathsf{good}(\tau).\tau' \in \mathcal{L}(p)$ but for all traces τ' we have $\mathsf{good}(\tau).a.\tau' \notin \mathcal{L}(p)$.

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. The failing-edit-distance is still an edit-distance, but not necessarily minimal.

4.2 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

Algorithm 1 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 failing($q'$) then
            if $|\rho| < \gamma$ then
                $C' \leftarrow C' \cup \{\langle (i, -).\rho, q\rangle\}$
                $C' \leftarrow C' \cup \{\langle (i, +b).\rho, \delta(a, \delta(b, q))\rangle \mid b \in \Sigma\}$
                $C' \leftarrow C' \cup \{\langle (i, \%b).\rho, \delta(b, q)\rangle \mid b \in \Sigma\}$
        else
            $C' \leftarrow C' \cup \{\langle \rho, q'\rangle\}$
    $C \leftarrow \{\langle \rho, q\rangle \in C' \mid \neg$failing$(q)\}$
return min$(\{|\rho| \mid \langle \rho, q\rangle \in C \wedge q \in F\})$
```

the instantiated pattern. A configuration is a pair consisting of a rewrite (Def. 11) 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 1 gives an algorithm for computing the 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 failing that returns true if a final state is not reachable from the given state.

The use of γ helps restrict the exponential blowup introduced by the non-determinism of edit functions. Other optimisations that can reduce this blowup include restricting the number of edits allowed in a row and combining similar rewrites together.

4.3 Example of computing failing edit-distance

Let us take the resource usage pattern introduced in Sec. 3.2 and consider the trace

`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.

5. USING THE TRUE EDIT DISTANCE

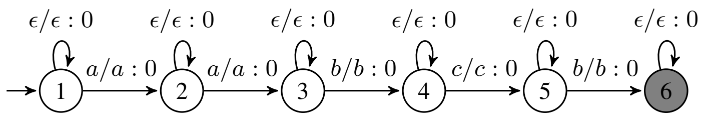
We now consider an approach that uses the true edit distance between the trace and language. We consider a technique that uses weighted transducers to compute the edit-distance between a trace and a finite automaton [2]. The general idea is that we model the trace and pattern as weighted transducers 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.

5.1 Weighted transducers

A weighted transducer has transitions labeled 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.

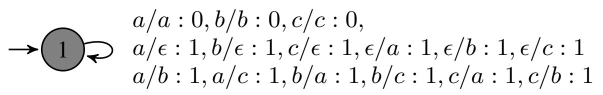
DEFINITION 12 (WEIGHTED TRANSDUCER). *A weighted transducer is a 5-tuple $T = \langle Q, \Sigma, \Delta, \delta, F\rangle$ where Q is a finite set of states, Σ is a finite input alphabet of symbols, Δ is a finite output alphabet of symbols, $\delta \subset Q \times (\Sigma \cup \{\epsilon\}) \times (\Delta \cup \{\epsilon\}) \times \{0, 1\} \times Q$ 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 labeling 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
$a/a : 0, b/b : 0, c/c : 0,$
$a/\epsilon : 1, b/\epsilon : 1, c/\epsilon : 1, \epsilon/a : 1, \epsilon/b : 1, \epsilon/c : 1$
$a/b : 1, a/c : 1, b/a : 1, b/c : 1, c/a : 1, c/b : 1$

5.2 Composition

The composition $T \circ X$ of two transducers T and X considers all possible sequencing 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 in [1] and Algorithm. 2 gives an algorithm for three-way composition.

5.3 An example of computing edit-distance

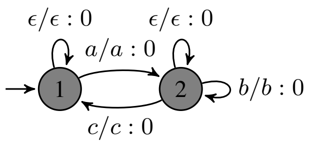
Let us take the same example we used for the failing edit-distance i.e. the trace

`open.open.use.close.use`

and the resource usage pattern introduced in Sec. 3.2. 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. 5.1 above. We therefore already have our weighted transducer T. We then compute the weighted transducer P for the resource usage pattern as follows.

Algorithm 2 Computing the three-way composition of transducers T, X and P with the same input and output alphabets Σ and Δ.

```
$\mathsf{Enqueue}(S, (T.q_0, X.q_0, P.q_0))$
$Q \leftarrow \{(T.q_0, X.q_0, P.q_0)\}$
$\delta, F \leftarrow \emptyset$
while $\neg\mathsf{isEmpty}(S)$ do
    $(q_1, q_2, q_3) \leftarrow \mathsf{Dequeue}(S)$
    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 $i_2 = o_1 \wedge o_2 = i_3$ do
            if $(q_1, q_2, q_3) \notin Q$ then
                $Q \leftarrow Q \cup \{(q_1, q_2, q_3)\}$
                $\mathsf{Enqueue}(S, (q_1, q_2, q_3))$
            $\delta \leftarrow \delta \cup ((q_1, q_2, q_3), i_1, o_3, w_1 + w_2 + w_3, (q_1', q_2', q_3'))$
return $\langle Q, \Sigma, \Delta, \delta, F\rangle$
```



We now compute $T \circ X \circ P$, using the edit transducer X presented in Sec. 5.1 above. This gives us the weighted transducer in Figure 1. We then use Djkistra'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 labeled with that pattern.

6. 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.

6.1 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

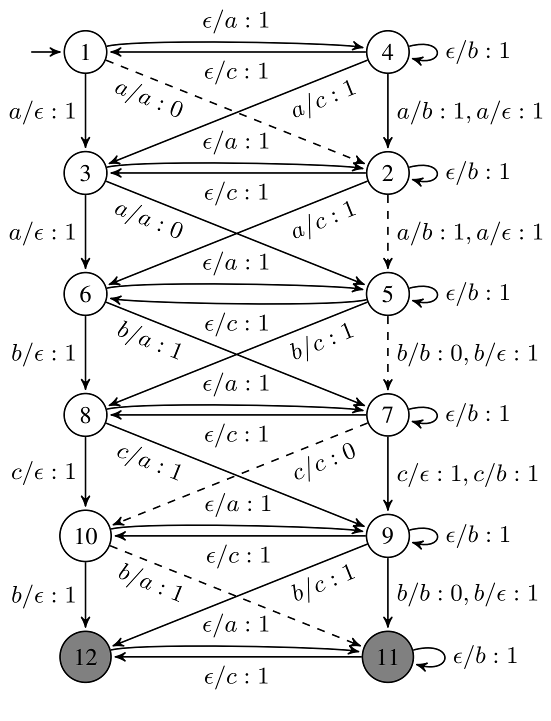


Figure 1: An example of the composition $T \circ X \circ P$

given by the language of the composition - therefore it is in theory 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 $\{\ldots\rho_i\ldots \mid \rho_i \in R_i\}$ such that every pair of rewrites is compatible. Two rewrites are compatible if they do not attempt to make different rewrites at the same point in a trace. The edit-distance of $\langle p_n, R_n\rangle \cap \ldots \cap \langle p_n, R_n\rangle$ is $|\rho_1 \cup \ldots \cup \rho_n|$ i.e. the number of 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.

However, rewrites only contain information about the parts of the trace they update. If one rewrite updates an element in the trace and the other rewrite does not then this should also appear as an incompatibility. We therefore incorporate this information when computing compatibility.

6.2 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 is given in Algorithm 3. 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.

6.3 Saturating the set of patterns

Given a set of imperfect patterns P_0 extracted from a trace we compute the i-the 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, it is highly likely that many combinations in P_{i_1} are empty (i.e. accept no traces) and therefore

Algorithm 3 Computing the minimum compatibility between sets of rewrites R_1 and R_2 from imperfect patterns extracted from trace τ where R_i is related 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
        $D \leftarrow \{\rho \mid \rho(i)$ 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 $\mathsf{min}(\{|\bigcup g| \mid g \in G_{okay}\})$
```

```java
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 2: A hypothetical piece of Java code.

can be removed. Even though many patterns can be removed, the saturation can grow exponentially. Let P_∞ be the fixed-point of P_i i.e. the set P_i such that $P_{i+1} = P_i$. We can either compute P_∞ or place an upper bound on the number of saturations we want to perform.

Once we have generated a set of patterns we can rank them by two properties - firstly, the size of alphabet, and secondly the edit-distance. We are interested in patterns with large alphabets and small edit-distance.

7. 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.

7.1 Application to example code

Consider the Java code in Figure 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.
```

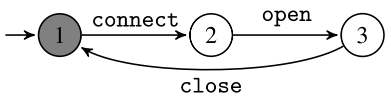
We now consider mining this trace with two patterns - the alternating pattern given in the introduction and the resource usage pattern given in Section 3.2. We take the alternating pattern first.

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. The instantiation $[a \mapsto \texttt{open}, b \mapsto \texttt{connect}]$ 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 $[a \mapsto \texttt{close}, b \mapsto \texttt{connect}]$ and $[a \mapsto \texttt{close}, b \mapsto \texttt{open}]$ there is a shorter true edit-distance as this approach is allowed to make edits without failure - here this involves 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.

		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	-

For one case, $[a \mapsto \texttt{connect}, b \mapsto \texttt{open}]$ there is a distance of 0 - this is 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`).

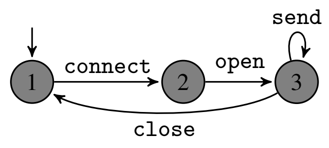
Combining the three instantiated patterns with an edit distance of 0 or 1 we get the following pattern.

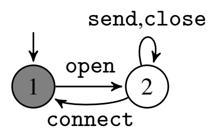


Now let us consider the resource usage pattern. The following table gives the failing and true edit distances as before - with each entry in the table representing the c dimension using a 4-tuple. Here, again, computed distances are the same but the true edit-distance approach generates some distances 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 only the ninth event (as before) and one set removes the only first `connect` event, therefore they are incompatible. When combined these give the two following patterns respectively.

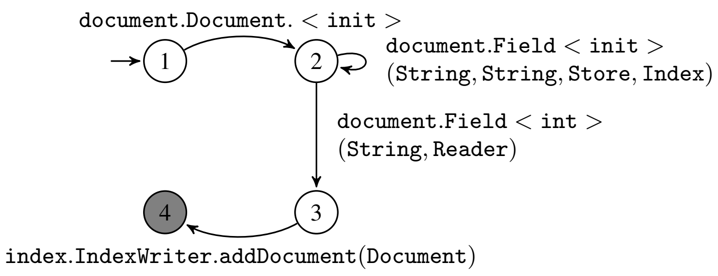




The rewrite for first pattern is compatible with the rewrite for pattern extracted using the alternation pattern and we can combine these patterns to form a final specification, which is the same as the one on the left above, but with only the initial state accepting.

7.2 An accuracy experiment

In the following we attempt to evaluate the *accuracy* of our approach by generating traces from the following specification for the Lucene tool described in [5].


document.Document. < init >
document.Field < init >
(String, String, Store, Index)
document.Field < int >
(String, Reader)
index.IndexWriter.addDocument(Document)
1
2
3
4

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 1 gives the results - for each approach it reports the average accuracy of all produced patterns, the minimum edit required to produce a pattern with maximum accuracy, the time (in milliseconds) taken for checking and then saturation, the extracted patterns and the size of the 3-saturated set. Experiments were carried out with a range of trace lengths and noise levels and we record the min, mean and max edits made at each noise level.

Every experiment produces at least one pattern with perfect accuracy. Although the average accuracy is low, this is over a very large saturated set, containing many (over 50%) patterns with zero accuracy. As expected, with zero noise we detect a pattern with total accuracy that requires no edits. As expected, as the level of noise (and therefore level of errors) increases accuracy decreases.

Interestingly, for trace length of 100 and noise level of 0.05, giving an average of 6 errors, we achieve 0.57% average accuracy. This is due to a small saturated set being produced. Methods for pruning this set should be explored.

We can see that the saturated sets are very large and the majority of time is spent computing this set. Future work should look at methods for reducing this by using a more guided approach.

8. RELATED WORK

We consider alternative techniques that mine specifications from runtime traces. A recent survey paper [17] gives a good overview of the field. Here we focus on how techniques deal with imperfect traces, in particular we are interested in automata-based pattern mining approaches.

Ammons et al. [3] developed an early approach that used a probabilistic finite automata learner from the field of grammar inference. This techniques requires us to know the alphabet of the inferred specification beforehand. Imperfect traces require human experts to check violations of the inferred specification in a coring phase. Lo et al. [13] extend this approach - one extension that is relevant to mining with imperfect traces 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. [12, 18]) and *closed frequent sequential pattern mining* (i.e. [14]) rely on computing *support* and *confidence* values where support reflects the level of imperfection. Mined specifications can then be checked against the original program to detect bugs.

The automata-based pattern-mining technique was first used by Engler et al. [4]. They focus on the alternating pattern $(ab)^*$ and to deal with imperfect traces they count the number of times that a and b appear together in order and the number of times a appears without b and compute the likelihood that a and b form a specification. Goues and Weimer [8] extend this approach by considering techniques for pruning false positives by examining the source code.

Yang et al. [19] 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. [6, 5] extend this approach by introducing a symbolic method for specification mining using binary decision diagrams (BDDs) and the Javert tool that uses two patterns $(ab)^*$ and $(ab^*)^*$ and composition rules based on the notion of automata combination to extract large patterns. They deal with imperfect traces by restarting a pattern at the initial state on failure. In [7] 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. [11] extend this approach to mine specifications with timing bounds and more complex pattern composition 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 [16, 9, 15] have considered so-called *parametric* where traces contain data i.e. $\texttt{open}(123).\texttt{open}(456).\texttt{close}(123).$ $\texttt{close}(456)$. Whilst some approaches use ad-hoc approaches to deal with context, these approaches focus on *slicing* the trace based on this data and extracting traces from the resulting data-free traces. The work in [9] extends the approach taken by [3] and therefore use the same coring technique to deal with imperfect traces and [15] uses the notions of *support* and *confidence* from data mining.

9. CONCLUSION

In this paper 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.

Trace	Noise	(min,mean,max)			Failing					Perfect		
length	level	edits	Accuracy	Edit	Time	Extracted	P_3	Accuracy	Edits	Time	Extracted	P_3
10	0.0	(0,0,0)	0.10	0	(318,9181)	36	2779	0.14	0	(171,364)	36	706
10	0.01	(1,1,2)	0.18	1	(94,21908)	34	2349	0.16	3	(69,29)	36	22
10	0.05	(1,1,1)	0.13	2	(77,20113)	36	2890	0.00	1	(55,18)	36	12
10	0.1	(1,2,4)	0.24	3	(78,30180)	36	2673	0.00	1	(53,24)	36	16
100	0.0	(0,0,0)	0.05	0	(138,4606)	18	333	0.07	0	(491,549)	36	705
100	0.01	(1,1,2)	0.25	0	(130,26967)	18	333	0.20	0	(419,339)	36	116
100	0.05	(4,6,8)	0.57	6	(273,31782)	14	53	0.0	-	(428,132)	36	0
100	0.1	(6,10,16)	0.0	-	(325,1584)	11	0	0.0	51	(427,75)	36	2

Table 1: Results from accuracy experiment

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 should involve the combination of this approach with an existing technique, for example the symbolic mining technique of [6], and the composition rules of [5] and [11]. We also plan on combining this approach with the author's pattern-mining approach taken in [16], which targets a specific alphabet of events to extract a parametric specification. This approach uses so-called open automata that means that all extracted patterns can be sound combined to form a specification. Therefore, we would be able to use pattern combination directly, rather than introducing pattern composition rules.

10. REFERENCES

[1] 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.

[2] C. Allauzen and M. Mohri. Linear-space computation of the edit-distance between a string and a finite automaton. *CoRR*, abs/0904.4686, 2009.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, Jan. 2002.

[4] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, Oct. 2001.

[5] 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.

[6] 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.

[7] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[8] C. Goues and W. Weimer. Specification mining with few false positives. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 292–306, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] 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.

[10] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[11] 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.

[12] 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, Sept. 2005.

[13] D. Lo and S.-C. Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 265–275, New York, NY, USA, 2006. ACM.

[14] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *J. Softw. Maint. Evol.*, 20(4):227–247, July 2008.

[15] 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.

[16] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, November 2013. To appear.

[17] 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.

[18] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. *Automated Software Engg.*, 18(3-4):263–292, Dec. 2011.

[19] 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.