Science of Computer Programming 77 (2012) 743–759

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Mining quantified temporal rules: Formalism, algorithms, and evaluation

David Lo [a], G. Ramalingam [b], Venkatesh-Prasad Ranganath [b,*], Kapil Vaswani [b]

[a] *Singapore Management University, Singapore*

[b] *Microsoft Research, India*

ARTICLE INFO

Article history:
Received 17 February 2010
Received in revised form 28 July 2010
Accepted 12 October 2010
Available online 16 November 2010

Keywords:
Specification mining
Temporal rules
Quantification
Dynamic analysis
Reverse engineering

ABSTRACT

Libraries usually impose constraints on how clients should use them. Often these constraints are not well-documented. In this paper, we address the problem of recovering such constraints automatically, a problem referred to as *specification mining*. Given some client programs that use a given library, we identify constraints on the library usage that are (almost) satisfied by the given set of clients.

The class of rules we target for mining combines simple binary temporal operators with state predicates (composed of equality constraints) and quantification. This is a simple yet expressive subclass of temporal properties (LTL formulae) that allows us to capture many common API usage rules. We focus on recovering rules from execution traces and apply classical data mining concepts to be robust against bugs (API usage rule violations) in clients. We present new algorithms for mining rules from execution traces. We show how a propositional rule mining algorithm can be generalized to treat quantification and state predicates in a unified way. Our approach enables the miner to be *complete* (i.e. , mine all rules within the targeted class that are satisfied by the given traces) while avoiding an exponential blowup.

We have implemented these algorithms and used them to mine API usage rules for several Windows APIs. Our experiments show the efficiency and effectiveness of our approach.

1. Introduction

Libraries and APIs usually impose constraints on how clients should use them but often these constraints are not well-documented. In this paper, we address the problem of recovering such constraints automatically via dynamic analysis of clients of an API (i.e. , from a large number of execution traces that use a given API).

Target class of specifications. A key attribute of any specification miner is the class of properties (or specifications) that can be mined. In this paper, we introduce a class of *quantified binary temporal rules with equality constraints* (QBEC). This is a simple yet expressive class of temporal properties that allows us to capture many common (API usage) rules/constraints such as:

- Temporal rules, such as "every call to $m1()$ must be preceded by a call to $m2()$".
- Rules with equality constraints on parameters such as "every call to $m5(3, \ldots)$ must be followed by a call to $m6(10, ..)$".
- Quantified (temporal) rules such as "for every object x, every call to $m3(x)$ must be followed by a call to $m4(x)$".

* Corresponding author. Tel.: +91 80 6658 6107.
E-mail addresses: davidlo@smu.edu.sg (D. Lo), grama@microsoft.com (G. Ramalingam), rvprasad@microsoft.com (V.-P. Ranganath), kapilv@microsoft.com (K. Vaswani).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.10.003

Importantly, QBEC illustrates three natural dimensions of API usage rules (as shown by the above examples): *temporal operators* which impose constraints on the order in which API calls may be made, *state predicates* which qualify the API calls referred to in a temporal rule, and *quantification* which captures data-flow constraints between API calls.

The binary temporal operators we consider have been addressed by several others previously, but we present a new linear-time algorithm in this respect. Our more important contribution is the formalism we present for handling state predicates and quantification. Our mining algorithm is *complete* (i.e. , it mines all rules within the targeted class that satisfy the desired criteria), while avoiding an exponential blowup that alternative strategies can encounter. An important aspect of our formalism is that it is fairly general and can be easily extended to other temporal operators as well as a large class of state predicates.

Bug-tolerant specification-mining. Another important characteristic of specification miners is whether they can tolerate "input errors". For example, given a set of execution traces, one possibility is to compute the set of rules satisfied by all traces in the given set. Unfortunately, such an approach is not robust to occasional bugs in the programs (under study), which may produce execution traces that violate valid API rules. Instead, we focus on inferring API usage rules that are "almost satisfied" by a given set of clients (in a sense formalized below).

Data-mining based specification mining. Another distinguishing characteristic of our approach is that it is based on data-mining. Specifically, we use well-accepted concepts from the data-mining literature to formalize what it means for a rule to be "almost satisfied" by a given a set of execution traces. This formalization is based on the concepts of *support* and *confidence*. The support for a rule in a given set of traces is a count of the number of "positive instances" of the rule in the set of traces. The confidence is the ratio of the number of instances satisfying the rule to the number of instances where the rule is applicable. The mining problem, then, is to identify all rules whose confidence and support in the given set of traces is above user-provided thresholds.

Efficient mining algorithms. In this paper, we present efficient algorithms for mining QBEC rules. We present a new algorithm for mining must-be-followed-by and must-be-preceded-by rules. We also describe algorithms for mining alternation rules. (These refer to specific temporal operators, which are formally defined later.)

The above algorithms are quite efficient in terms of their dependence on the length of the traces analyzed. Hence, these work well for mining purely temporal rules. However, mining temporal rules that also incorporate *state predicates* (such as constraints on a procedure's arguments) pose a challenge. In general, the number of candidate rules can be very high (or even potentially infinite); consequently, the mining can be very expensive. We tackle this problem by using the *a priori* property from data mining: this technique prunes the search space of candidate rules by inductively constructing terms (which eventually form rules) by composing only subterms that occur frequently enough to justify consideration.

Finally, we consider quantified rules. We show that a logical view of quantification enables the above mentioned propositional rule mining algorithms to be naturally extended to mine quantified rules. This approach also helped us uncover and handle subtle corner cases correctly.

Eliminating redundant rules. In general, the set of mined rules may contain some redundancy. A rule is *redundant* if it is logically implied by another rules. We propose a set of simplifications to eliminate redundancy in a set of mined rules. Such redundancy elimination can serve as a useful filter before the rules are presented to the user.

Implementation and evaluation. We have implemented our mining algorithms (along with the redundancy elimination strategies) and evaluated our implementation by applying it to a set of clients of several Windows API, including the Windows Driver Model (WDM), a framework for device drivers.

Our evaluations show that the mining algorithms are *effective* and *efficient*. We were able to mine several rules shipped with Static Driver Verifier (SDV) [1], a static analysis based tool used to verify that WDM device drivers satisfy a given set of rules. Further, we were able to mine rules that are currently not shipped with SDV but are suggested in the textual documentation of WDM.

We also show that the search space of QBEC rules is very large in practice, and that the a priori technique is effective in pruning this search space. Further, we study the impact of varying confidence and support on the mining. Finally, we show that our redundancy elimination technique is very effective and useful.

Contributions

The contributions of this paper include:

- A formalism for mining *quantified* rules. The formalism is novel in introducing quantification to the framework of data-mining and, in fact, can be applied to standard data-mining problems such as association rule mining.
- A general event (item) quantification technique that enables existing (data-)mining algorithms to mine quantified rules.
- Algorithms to mine binary temporal rules while handling both quantification and state predicates in a unified fashion. The algorithms are complete, i.e. they mine all rules within the targeted class that have the desired support and confidence in the given set of execution traces.
- A new linear-time algorithm for mining must-be-followed-by rules.
- An empirical evaluation of our mining algorithm.

```
1  0x1 = fopen("usr.txt", 'r')
2  0x2 = fopen("pwd.txt", 'r')
3  30 = fread(0x1, . . . )
4  16 = fread(0x2, . . . )
5  0x3 = fopen("db.txt", 'w')
6  void = fwrite(0x3, . . . )
7  void = fclose(0x3)
8  void = fclose(0x2)
9  void = fclose(0x1)
```

Fig. 1. An execution trace capturing file access API invocations.

e_1:[0x1, fopen, "usr.txt", 'r']
e_2:[0x2, fopen, "pwd.txt", 'r']
e_3:[30, fread, 0x1, . . .]
e_4:[16, fread, 0x2, . . .]
e_5:[0x3, fopen, "db.txt", 'w']
e_6:[void, fwrite, 0x3, . . .]
e_7:[void, fclose, 0x3]
e_8:[void, fclose, 0x2]
e_9:[void, fclose, 0x1]

Fig. 2. A trace π_1 of file access API invocations.

2. Running examples

In the rest of this paper, we will use the execution trace in Fig. 1 as a running example to illustrate various concepts and techniques.

Fig. 1 is an execution trace of a program that reads from `"usr.txt"`, reads from `"pwd.txt"`, and writes to `"db.txt"`. Each line in the figure captures an event. For example, line 1 in the trace in Fig. 1 captures the information about an invocation of `fopen` with `"usr.txt"` and `'r'` as the first and second arguments, respectively, and `0x1` as the return value.

For the purpose of simplicity of exposition (via uniformity), invocations that do not return a value are captured as invocations returning a special value `void`.

3. An extensible class of temporal rules

In this section, we define various classes of temporal rules that are easily amenable to mining. These are special kinds of LTL formulae (enhanced to express past-time temporal rules). From an expressiveness perspective, a powerful language like LTL is a convenient specification formalism. However, for mining specifications, it is helpful to consider various restricted classes of specifications, both for efficiency reasons as well effectiveness reasons. The larger the class of targeted specifications, the more the potential for noise in the mined results and the more a user is likely to be overwhelmed by a large number of mined rules. We now introduce a parameterized class of predicates that lets us define various useful classes of temporal rules, including several classes addressed by previous work in specification mining.

3.1. Traces

Formally, we assume that we are given a set of *sequences* of *events*, which we will refer to as *traces*. We use the symbol π, possibly subscripted, to denote a sequence. We use the symbol e, possibly subscripted, to denote an event. An example trace is shown in Fig. 2.

3.2. Events

In the particular instantiation we study, an event is a *tuple of primitive values*. We consider a primitive value to be a string or an integer (which can also represent an address or pointer value). Let V denote the set of all primitive values. In our context, an event represents a particular call to an API method (during an execution) by a client. The tuple captures the name of the method called, as well as the values of the parameters of this particular call and the return value. Thus, a tuple [0, `foo`, 5, 10] represents a call to `foo` with parameters 5 and 10 whose return value was 0. We will represent such tuples mnemonically as "0 $\hookleftarrow$ `foo`(5,10)".

In this representation, the execution trace in Fig. 1 could be represented as an event trace shown in Fig. 2.

More generally, however, an event could be any concrete or abstract representation of a program's state, and a sequence could be a corresponding representation of an execution trace. Note that a tuple, in our usage as described above, actually represents an abstraction of a pair of states, one representing a point where the procedure call begins and one representing a point where the procedure call ends.

3.3. Event predicates

An *event-predicate* is a predicate ξ over events. We will use the notation $e \models \xi$ to denote that an event e satisfies an event-predicate ξ.

In our setting, an event predicate typically combines a specific procedure name with potential constraints on the parameter values or return value. As an example, we could have an event predicate ξ that matches calls to a procedure `bar` where the second parameter value is 13. If t is a tuple, we will use the representation $t \downarrow i$ to denote the $i+1$-th element of the tuple t. Thus, $t \downarrow 0$ denotes the first element of the tuple. We define a set of predicates $\mathbb{EC}$ of the form $\$i = c$, where i is a non-negative integer and c is a primitive value, whose meaning is defined as follows: a tuple t satisfies the predicate $\$i = c$ iff $t \downarrow i$ equals c. We refer to these predicates as *equality constraint predicates*.

For example, event e_1 in trace π_1 (in Fig. 1) satisfies the equality constraint predicate $\$2 =$ "usr.txt", i.e. $e_1 \models (\$2 =$ "usr.txt").

Let $\mathbb{EC}_1$ denote the subset of $\mathbb{EC}$ consisting of predicates of the form "$\$1 =$ `procname`". These are the simplest event predicates of interest to us, which check if the given event is a call to a specific procedure.

More generally, we would like to consider any event predicate that can be expressed as the conjunction of one or more equality constraint predicates. In our context, an event predicate "$\$2 = 5$" is satisfied by an event iff it represents a call whose first parameter has the value 5. Such an event predicate is typically meaningless without any information about the procedure that is called. Hence, we restrict ourselves only to conjunctions of equality constraint predicates that include a constraint of the form "$\$1 =$ `procname`". Let $\mathbb{EC}^*$ denote the set of all such event predicates. Thus, $\mathbb{EC}^*$ is isomorphic to $\mathbb{EC}_1 \times 2^{\mathbb{EC} \setminus \mathbb{EC}_1}$.

We will use the notation $\langle c_0 \hookleftarrow c_1(c_2, \ldots, c_k)\rangle$, where each c_i, except c_1, is either a primitive value or an underscore, and c_1 is a primitive value (representing a procedure name) to represent an element of $\mathbb{EC}^*$. This represents an event predicate that is satisfied by a call to c_1 with parameter values c_2 through c_k and a return value of c_0. Furthermore, we will use an underscore in place of a constant c_i if we do not want any constraint on the corresponding tuple element. In other words, $\langle c_0 \hookleftarrow c_1(c_2, \ldots, c_k)\rangle$ is short for $\bigwedge\{\$i = c_i \mid 0 \leq i \leq k, c_i \neq \_\}$. Further, we will abbreviate $\langle c_0 \hookleftarrow c_1(c_2, \ldots, c_k)\rangle$ to $\langle c_1(c_2, \ldots, c_k)\rangle$ when c_0 is "_".

Example 1. Consider event $e_1 =$ `[0x1, fopen, "usr.txt", 'r']` in trace π_1. $\langle$`_ ↩ fopen(_, _)`$\rangle$, $\langle$`0x1 ↩ fopen(_, _)`$\rangle$, and $\langle$`0x1 ↩ fopen(_, 'r')`$\rangle$ are predicates in $\mathbb{EC}^*$ that are satisfied by e_1.

3.4. Temporal operators

We construct *temporal formulae* (or temporal rules) by combining event predicates using temporal operators. We currently consider two types of temporal operators, the "*eventual*" operator and the "*alternation*" operator, in two flavors each (forward and backward). This gives us the following temporal operators: $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ (forward eventual operator), $\xi_2 \overset{*}{\leftarrowtail} \xi_1$ (backward eventual operator), $\xi_1 \overset{a}{\rightarrowtail} \xi_2$ (forward alternation operator), and $\xi_2 \overset{a}{\leftarrowtail} \xi_1$ (backward alternation operator) where ξ_1 and ξ_2 represent event predicates. The meaning of these operators is defined below.

The temporal formula $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ represents the rule that any occurrence of ξ_1 must eventually be followed by an occurrence of ξ_2. More formally, we say that a sequence $\pi = e_1 e_2 \cdots e_n$ satisfies the temporal formula $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ (denoted $\pi \models \xi_1 \overset{*}{\rightarrowtail} \xi_2$) iff for any $e_i \models \xi_1$ there exists a $j > i$ such that $e_j \models \xi_2$.

For example, the temporal formula $\langle$`fopen(_,'r')`$\rangle \overset{*}{\rightarrowtail} \langle$`fclose(_)`$\rangle$ represents the rule that any call to `fopen` with 'r' as the second argument should be followed by a call to `fclose`. In trace π_1, $e_1 \models \langle$`fopen(_,'r')`$\rangle$, $e_2 \models \langle$`fopen(_,'r')`$\rangle$, $e_4 \models \langle$`fopen(_,'r')`$\rangle$, and $e_7 \models \langle$`fclose(_)`$\rangle$; hence, $\pi_1 \models \langle$`fopen(_,'r')`$\rangle \overset{*}{\rightarrowtail} \langle$`fclose(_)`$\rangle$.

Similarly, the temporal formula $\xi_2 \overset{*}{\leftarrowtail} \xi_1$ represents the rule that any occurrence of ξ_1 must be preceded by an occurrence of ξ_2.

The temporal formula $\xi_1 \overset{a}{\rightarrowtail} \xi_2$ represents the rule that (i) any occurrence of ξ_1 must eventually be followed by an occurrence of ξ_2; (ii) furthermore, an occurrence of ξ_1 cannot be followed by another occurrence of ξ_1 before an occurrence of ξ_2.

For example, the temporal formula $\langle$`lock(0x4)`$\rangle \overset{a}{\rightarrowtail} \langle$`unlock(0x4)`$\rangle$ represents the rule that any call to `lock` with the first argument of 0x4 should be followed by a call to `unlock` with the first argument of 0x4 and no intervening calls to `lock` with the first argument of 0x4. In trace π_2, $e_1 \models \langle$`lock(0x4)`$\rangle$, $e_3 \models \langle$`unlock(0x4)`$\rangle$, $e_5 \models \langle$`lock(0x4)`$\rangle$, and $e_6 \models \langle$`unlock(0x4)`$\rangle$; hence, $\pi_2 \models \langle$`lock(0x4)`$\rangle \overset{a}{\rightarrowtail} \langle$`unlock(0x4)`$\rangle$ due to event pairs (e_1, e_3) and (e_5, e_6) (see Fig. 3).

The formula $\xi_2 \overset{a}{\leftarrowtail} \xi_1$ is similar, but in the backward direction.

In all of the rule forms described above, we refer to ξ_1 as the *antecedent* and ξ_2 as the *consequent* of the rule. (Notice that an occurrence of an event satisfying the antecedent implies the occurrence of an event satisfying the consequent. However, the temporal order of the antecedent and the consequent events are different for forward and backward rules.)

We will use the symbol $\overleftrightarrow{\mathbb{EA}}$ to denote the set of four temporal operators defined above.

```
e1  [void, lock, 0x4]
e2  [void, lock, 0x5]
e3  [void, unlock, 0x4]
e4  [void, unlock, 0x5]
e5  [void, lock, 0x4]
e6  [void, unlock, 0x4]
```

Fig. 3. An event trace π_2 capturing synchronization API invocations.

Example 2. Consider the typestate property $(\texttt{lock};\texttt{unlock})^*$ that says that `lock` and `unlock` operations have to occur in strict alternation. This property can now be expressed using the above operators as $(\texttt{lock} \overset{a}{\rightarrowtail} \texttt{unlock}) \wedge (\texttt{lock} \overset{a}{\leftarrowtail} \texttt{unlock})$.

3.5. Quantification

We will refer to the type of temporal rules considered so far as *propositional temporal rules*. We use *quantification* to introduce constraints involving parameter (and return) values of different events in a temporal rule, as in "Every call to $\texttt{foo}(x)$ must be preceded by a call to `bar` that returned x". This rule can be expressed formally as: $\forall x.\langle x \hookleftarrow \texttt{bar}() \rangle \overset{*}{\leftarrowtail} \langle \texttt{foo}(x) \rangle$.

Formally, we need to first generalize event predicates to allow event predicates that contain free variables. In our instantiation, we allow equality constraints of the form "$\$i = x$", where x is a free variable. For clarity, we shall refer to event predicates with free variables as *quantifiable event predicates* and to event predicates with no free variables as *propositional event predicates*. A binding θ is a map from free variables to (primitive) values. Given a quantifiable event predicate ξ, and a binding θ for all the free variables occurring in ξ, we will use $\xi[\theta]$ to denote the event predicate obtained by replacing every variable x in ξ by its value $\theta(x)$.

For example, event $e_1 =$ `[0x1, fopen, "usr.txt", 'r']` in trace π_1 can be denoted by the quantifiable event predicate $\$1 = \texttt{fopen} \wedge \$3 = x$ along with the binding $[x \mapsto \text{'r'}]$.

In such a setting, a quantified forward-eventual rule is of the form $\forall \vec{X}.\xi_1 \overset{*}{\rightarrowtail} \xi_2$ where ξ_1 and ξ_2 have the *same* set of free variables $\vec{X}$. We say that a sequence $\pi = e_1 e_2 \cdots e_n$ satisfies the quantified formula $\forall \vec{X}.\xi_1 \overset{*}{\rightarrowtail} \xi_2$ iff for some binding θ of values to the free variables in $\vec{X}$, if $e_i \models \xi_1[\theta]$ then there exists a $j > i$ such that $e_j \models \xi_2[\theta]$.

In trace π_1, $e_1 \models \langle \texttt{lock}(x) \rangle[\theta]$ relates to $e_3 \models \langle \texttt{unlock}(x) \rangle[\theta]$ and $e_5 \models \langle \texttt{lock}(x) \rangle[\theta]$ relates to $e_6 \models \langle \texttt{unlock}(x) \rangle[\theta]$ via $\theta = [x \mapsto \texttt{0x4}]$. Similarly, $e_2 \models \langle \texttt{lock}(x) \rangle[\theta]$ relates to $e_4 \models \langle \texttt{unlock}(x) \rangle[\theta]$ via $\theta = [x \mapsto \texttt{0x5}]$. Hence, $\pi_1 \models \forall x.\langle \texttt{lock}(x) \rangle \overset{*}{\rightarrowtail} \langle \texttt{unlock}(x) \rangle$.

The quantified form of other rules and their meanings are defined in a similar fashion.

Note. The above definition constrains the antecedent and the consequent to have the same set of free variables, without any loss of generality. A rule such as $\forall x.\langle \texttt{foo}(x) \rangle \overset{*}{\rightarrowtail} \langle \texttt{bar}() \rangle$ is equivalent to the unquantified rule $\langle \texttt{foo}(\_) \rangle \overset{*}{\rightarrowtail} \langle \texttt{bar}() \rangle$. The rule $\forall x.\langle \texttt{foo}() \rangle \overset{*}{\rightarrowtail} \langle \texttt{bar}(x) \rangle$ cannot be satisfied if the quantification is over an infinite domain, and is equivalent to a conjunction of unquantified rules $\wedge_{i=1}^{k} \langle \texttt{foo}() \rangle \overset{*}{\rightarrowtail} \langle \texttt{bar}(v_i) \rangle$ otherwise.

3.6. Summary

Note that we can obtain different classes of rules by considering the following dimensions: (a) the set of temporal operators allowed, (b) the set of event predicates allowed, and (c) the degree of quantification allowed. Let EP denote a set of event predicates, T denote a set of temporal operators, and n denote a non-negative number. We define $\mathfrak{F}(\text{EP}, \text{T}, n)$ to be the set of temporal rules built out of event predicates in EP and the temporal operators in T and consisting of at most n quantified variables.

In the rest of the paper, we present a formalization of the mining problem and present our mining algorithms by considering the following, increasingly richer, classes of rules.

1. $\mathfrak{F}(\mathbb{EC}_1, \overleftrightarrow{\mathbb{EA}}, 0)$: We first consider *propositional temporal* rules such as $\langle foo \rangle \overset{*}{\rightarrowtail} \langle bar \rangle$.
2. $\mathfrak{F}(\mathbb{EC}^*, \overleftrightarrow{\mathbb{EA}}, 0)$: We then consider *propositional temporal rules with equality constraints*, such as $\langle foo(3) \rangle \overset{*}{\rightarrowtail} \langle bar(7) \rangle$. While we use equality constraints to illustrate our algorithm and our experimental evaluation is restricted to this class, the algorithm we present is more generally applicable. It can handle event predicates that can be expressed as the conjunction of predicates belonging to a *finitely instantiable* set of predicates, defined as follows. A set S of event predicates is said to be *finitely instantiable* if any event can satisfy at most a finite number of predicates belonging to S. Any finite set S is trivially finitely instantiable. Note that the set $\mathbb{EC}$ is infinite but finitely instantiable.
3. $\mathfrak{F}(\mathbb{EC}^*, \overleftrightarrow{\mathbb{EA}}, k \geq 1)$: We then consider *quantified rules with equality constraints* such as $\forall x.\langle foo(x) \rangle \overset{*}{\rightarrowtail} \langle bar(x) \rangle$. In our experimental evaluation, we restrict attention to rules with one level of quantification ($k = 1$), but our algorithms apply to any value of k.

Discussion. We briefly discuss other interesting classes of rules. One important class is that of non-temporal rules [2]: e.g., a rule that the first parameter of `foo` must be non-null. This can also be expressed as a trivial (temporal) operator $\xi_1 \overset{s}{\rightarrowtail} \xi_2$, whose meaning is that any event satisfying ξ_1 must also satisfy ξ_2.

Some of the previous work on specification mining has focused on mining temporal rules captured as a finite-state automaton. We note that many commonly observed finite-state automaton rules can be expressed as the conjunction of simple temporal rules of the above form. The temporal operators we study correspond to (templates) for a small finite-state automaton. However, unlike techniques for mining a *single* finite-state automaton, we focus on mining the set of all finite-state automaton of a given template satisfied by a given set of traces. This approach has certain advantages. For instance, assume that we have a library with k different temporal rules $f_i \overset{*}{\rightarrowtail} g_i$, $1 \leq i \leq k$. Expressing these rules as a *single* finite automaton would require $O(2^k)$ states. Since finite-state automaton mining algorithms tend to limit their attention to automatons with a limited number of states, it would be easy to miss mining these temporal rules in a *single* automaton-mining approach. In contrast, approaches that mine a *set* of finite automaton, such as ours, can handle this scenario easily enough. In particular, mining the set of k temporal rules in the above example is straightforward with our approach.

Though we restrict our attention to a specific class of temporal operators, namely $\overleftrightarrow{\mathbb{EA}}$, our approach for handling event predicates and quantification can be used for other temporal operators as well.

4. The problem

In this section we formally define the problem considered in this paper. Informally, our goal is: given a set T of traces, identify the set of all rules r from the class of temporal rules (defined in the previous section) such that we can say with "*high confidence*" that T satisfies r.

We use a well-accepted definition from the data-mining literature [3] to formalize the problem. The first step in this formalization is to define the notion of *support* and *confidence* for a rule r in a set T of traces.

Propositional temporal rules. Let $\pi_1, \ldots, \pi_n$ be the set of given sequences. Let $\pi[j]$ denote the j-th element of a sequence π and the ordered pair (i, j) denote the *position* of the j-th element of the i-th sequence π_i. We say that position (i, j) is a *witness* for the event predicate ξ if $\pi_i[j] \models \xi$. Similarly, we say that position (i, j) is a *witness* for the temporal rule $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ if (i, j) is a witness for ξ_1 and there exists a witness (i, k) for ξ_2 with $k > j$.

Given a temporal rule $\xi_1 \overset{*}{\rightarrowtail} \xi_2$, we define its *support* as the number of witnesses for the rule and its *confidence* as the ratio of its support to the number of witnesses for ξ_1.

For example, in trace π_1, positions (1, 1) and (1, 2) (events e_1 and e_2, respectively) are witnesses to $\langle$`fopen(_,'r')`$\rangle$. These positions are also witnesses for the rule $\langle$`fopen(_, 'r')`$\rangle \overset{*}{\rightarrowtail} \langle$`fclose(_)`$\rangle$. Hence, the number of witnesses for the antecedent of this rule is 2, the the support for the rule is 2, and the confidence for the rule is 1 (=2/2).

Support and confidence of rules with other temporal operators are similarly defined. The support of any event predicate ξ is also defined to be the number of witnesses for ξ.

Quantified temporal rules. While dealing with quantification, we say that position (i, j) is a *witness* for a quantifiable event predicate $\xi(\vec{X})$ if there exists a binding θ for $\vec{X}$ such that $\pi_i[j] \models \xi[\theta]$. As in the case of propositional temporal rules, we say that position (i, j) is a *witness* for the quantified temporal rule $\forall \vec{X}.\xi_1 \overset{*}{\rightarrowtail} \xi_2$ if there exists a binding θ for $\vec{X}$ such that (i, j) is a witness for $\xi_1[\theta]$ and there exists a witness (i, k) for $\xi_2[\theta]$ with $k > j$. With these definitions of witnesses, the notion of *support* and *confidence* for propositional temporal rules carries over to quantified temporal rules.

For example, in trace π_2, there are 3 witnesses (2, 1), (2, 2), and (2, 5) for $\langle$`lock(x)`$\rangle$. These are also witnesses for the rule $\langle$`lock(x)`$\rangle \overset{*}{\rightarrowtail} \langle$`unlock(`$x$`)`$\rangle$. Hence, the number of witnesses for the antecedent of this rule is 3, the support for the rule is 3, and the confidence for the rule is 1 (=3/3).

Problem definition. Given a set of traces T, a positive integer S_{min}, and a value C_{min} in the range [0,1], identify all rules belonging to class *QBEC* whose support in T is at least S_{min} and whose confidence in T is at least C_{min}.

5. Mining algorithms

In this section, we describe our mining algorithms. For the sake of brevity, we only describe the algorithms to mine the forward forms of rules as these algorithms can be trivially adapted to mine the backward forms of rules.

5.1. Forward-eventually rules

In this section we present our algorithm for mining $\overset{*}{\rightarrowtail}$ rules. We will start with the simplest form of these rules, and then consider increasingly richer forms of these rules.

5.1.1. Propositional rules with no equality constraints

We first consider mining rules such as $\langle foo \rangle \overset{*}{\rightarrowtail} \langle bar \rangle$, which involve only the temporal ordering of procedure calls and not the parameters or return-values of these calls. The design of our algorithm is influenced by two key insights.

```
PropositionalMustFollowEventually(T, S_min, C_min)
 1  # First Phase: Initialize
 2  Preds ← ⋃_{t∈T} ⋃_{e∈t} PredsOf(e)
 3  FreqPreds ← {ξ ∈ Preds | Supp(ξ, T) ≥ S_min}
 4  for each (ξ1, ξ2) ∈ FreqPreds × Preds
 5  do N_R[ξ1, ξ2] ← 0
 6  # Second Phase: Mine rule instances
 7  for each t in T
 8  do for each i ← 1 to |t|
 9     do for each ξ ∈ PredsOf(t[i])
10        do last[ξ] ← i
11     for each ξ ∈ Preds
12     do N_P[ξ] ← 0
13     for i ← 1 to |t|
14     do e ← t[i]
15        for each ξ_e in PredsOf(e)
16        do if last[ξ_e] = i
17              then for each ξ_f in FreqPreds
18                   do N_R[ξ_f, ξ_e] ← N_R[ξ_f, ξ_e] + N_P[ξ_f]
19           N_P[ξ_e] ← N_P[ξ_e] + 1
20 # Third Phase: Identify significant rules
21 Rules ← ∅
22 for each N_R[ξ1, ξ2] = s
23 do if s ≥ S_min ∧ (s/Supp(ξ1, T)) ≥ C_min
24       then Rules ← Rules ∪ {ξ1 →* ξ2}
25 return Rules
```

Fig. 4. Algorithm to mine $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ rules composed of propositional event predicates. *PredsOf* (e) is the set of all propositional event predicates satisfied by event e and $Supp(\phi, T)$ is the total number of ϕ satisfying events in T.

The first insight is based on the following observation. To compute the support and confidence for the rule $\langle foo \rangle \overset{*}{\rightarrowtail} \langle bar \rangle$ in a trace, it is sufficient to consider the last occurrence of *bar* in the trace (ignoring the earlier occurrences of *bar*). Given the last occurrence of *bar* in the trace, we then just need the number of occurrences of *foo* that precede it: this gives us the support for rule $\langle foo \rangle \overset{*}{\rightarrowtail} \langle bar \rangle$ in the given trace. The support for the rule in a set of traces can be computed by just adding its support from each trace. Given the support, we just need to know the total number of occurrences of *foo* in all the traces to compute the confidence.

We can identify the last occurrence of every procedure in a trace in a single pass through the trace. Similarly, we can compute the occurrence count of every procedure in every prefix of the given trace in a single pass through the trace. This leads to an algorithm whose complexity is linear in N_l, the sum of the lengths of the input traces. The worst-case time complexity of the algorithm is $N_l.N_p$ where N_p is the number of distinct procedures.

The second insight draws from the use of an *a priori* property [3] in the data mining community. The following (straightforward) theorem serves as the basis for applying the a priori property:

$$support(\xi_1 \overset{*}{\rightarrowtail} \xi_2) \leq support(\xi_1).$$

This theorem says that the support for a rule $\langle foo \rangle \overset{*}{\rightarrowtail} \langle bar \rangle$ can be no more than the support for *foo* (i.e. , the total number of occurrences of *foo* in the traces). We say that a procedure p is *frequent* if its support is at least S_{min}. It follows that in mining rules of the form $\langle f \rangle \overset{*}{\rightarrowtail} \langle g \rangle$ it suffices to consider only procedures f that are frequent.

Using this optimization reduces the time complexity of the algorithm to $N_l.F_p$ where F_p is the number of *frequent* procedures (which is often smaller than the number of procedures recorded in a trace). While this improvement may not appear exciting, the a priori property will be significant as we expand our scope to more general forms of rules.

5.1.2. Propositional rules with equality constraints

We now consider mining rules of the form $\xi_1 \overset{*}{\rightarrowtail} \xi_2$, where either ξ_i may include equality constraints. The intuitions of the earlier approach carry over. A key distinction, however, is that in the earlier approach every event satisfies only one predicate of interest. In the current setting, an event can satisfy many different event predicates. Note that it is straightforward to enumerate the set of predicates from $\mathbb{EC}^*$ that an event satisfies. The algorithm is presented in Fig. 4 and it operates in three phases.

In the first phase, it constructs the set *Preds* of all event predicates (line 2) satisfied by some event in the input traces and the set *FreqPreds* of frequent event predicates (line 3). (An event predicate ξ is said to be *frequent* if its support is at

least S_{min}.) It also initializes a map N_R that will be used to compute the support for rules (lines 4–5) that are represented as a pair of event predicates. Driven by the second insight, only pairs of event predicates from the set *FreqPreds* × *Preds* are considered.

In the second phase, each trace is processed to compute the support for rules (lines 7–19). Driven by the first insight, the position of the last occurrence of every predicate from *Preds* in a trace is identified and recorded in *last* (lines 8–10). Every event e in a trace π, say at the i-th position, is then processed, in order, to calculate the total support of frequent event predicates in $\pi[1..i]$. This support is recorded in the map N_P (line 19). During this processing, if the last occurrence of an event predicate ξ_e is encountered (line 16), the cumulative support for rules $\xi_f \overset{*}{\rightarrowtail} \xi_e$ involving frequent event predicate $\xi_f \in$ *FreqPreds* is incremented by $N_P[\xi_f]$, the support for ξ_f (again, driven by the first insight) (line 18).

In the final phase the algorithm selects rules that have/exceed the minimum desired support S_{min} and confidence C_{min} (lines 21–24) and returns them.

The worst-case time complexity of the algorithm is $N_l.F_e.M_e$ where N_l is the total length of the input traces, F_e is the number of frequent event predicates (which, as stated previously, will often be smaller than the number of all event predicates possible in the traces), and M_e is the maximum number of event predicates satisfied by any event. Thus, the complexity of the algorithm is linear in the total length of all the traces.

Note. The computation of the set of frequent event predicates (line [3]) is done using a *frequent item set* mining algorithm [3]. The key idea exploited here is again the a priori property: a predicate $\xi_1 \wedge \xi_2$ is frequent only if ξ_1 and ξ_2 are both frequent. Hence, this conjunction needs to be considered by the algorithm only if both conjuncts are frequent. Since most event predicates (conjunctions of equality constraints involving different parameters) will be infrequent and uninteresting, this technique allows the exploration of a big space of candidate predicates effectively.

5.1.3. Quantified rules with equality constraints

We now extend our algorithm to mine quantified rules. As motivation, consider the common rule that lock and unlock operations on a given lock must strictly alternate. Consider the trace $\pi = 0 \hookleftarrow lock(3), 0 \hookleftarrow lock(7), 0 \hookleftarrow unlock(3), 0 \hookleftarrow unlock(7)$. Events 1 and 3 together are a positive witness to this rule, and so are events 2 and 4. The key to noting that these two pairs are witnesses to the *same* rule is to *abstract* away the parameter that couples the antecedent and consequent together: both event pairs are positive witnesses to the parameterized rule $\langle lock(x)\rangle \overset{*}{\rightarrowtail} \langle unlock(x)\rangle$ with different bindings for x. From this, we would like to infer the quantified rule $\forall x.\langle lock(x)\rangle \overset{*}{\rightarrowtail} \langle unlock(x)\rangle$.

A key first step in our previous algorithm was to enumerate the set of event predicates that a given event e satisfied. We generalize this step as follows: we will now enumerate for every event e the set of ordered pairs (ξ, θ), consisting of a quantifiable event predicate ξ and a binding θ for the free variables of ξ such that e satisfies $\xi[\theta]$. We refer to a pair (ξ, θ), as described above, as a *generalized event predicate* (denoted as ζ).

We assume a fixed variable naming scheme for quantified variables in the mined rules. If we want to mine rules with k quantifiers, then we will use the set of variables $\{v_1, \ldots, v_k\}$ for this purpose. Let *GenPredsOf*(e, k) denote the set of generalized event predicates (ξ, θ) satisfied by e, where the set of free variables in ξ and the domain of binding θ are both $\{v_1, \ldots, v_k\}$. Note that *GenPredsOf*$(e, 0)$ is just the set of event predicates (with no variables) satisfied by e, and that elements ζ_{n+1} of *GenPredsOf*$(e, k+1)$ can be obtained from elements ζ_n of *GenPredsOf*(e, k) in a straightforward fashion by binding v_{k+1} to every possible value occurring in ζ_n, as illustrated below.

As an example, consider the event $e = 1 \hookleftarrow baz(1, 2)$. The set of generalized event predicates with at most 2 free variables satisfied by this event are given below, where $1^?$ can be replaced by either 1 or _ and $2^?$ can be replaced by either 2 or _.

No free variables *GenPredsOf*$(e, 0) =$
$(\langle 1^? \hookleftarrow baz(1^?, 2^?)\rangle, \{\})$.

One free variable *GenPredsOf*$(e, 1)$
$(\langle v_1 \hookleftarrow baz(1^?, 2^?)\rangle, \{v_1 \mapsto 1\})$,
$(\langle 1^? \hookleftarrow baz(v_1, 2^?)\rangle, \{v_1 \mapsto 1\})$,
$(\langle v_1 \hookleftarrow baz(v_1, 2^?)\rangle, \{v_1 \mapsto 1\})$,
$(\langle 1^? \hookleftarrow baz(1^?, v_1)\rangle, \{v_1 \mapsto 2\})$.

Two free variables *GenPredsOf*$(e, 2)$
$(\langle v_1 \hookleftarrow baz(v_2, 2^?)\rangle, \{v_1 \mapsto 1, v_2 \mapsto 1\})$,
$(\langle v_1 \hookleftarrow baz(1^?, v_2)\rangle, \{v_1 \mapsto 1, v_2 \mapsto 2\})$,
$(\langle v_1 \hookleftarrow baz(v_1, v_2)\rangle, \{v_1 \mapsto 1, v_2 \mapsto 2\})$,
$(\langle v_2 \hookleftarrow baz(v_1, 2^?)\rangle, \{v_1 \mapsto 1, v_2 \mapsto 1\})$,
$(\langle 1^? \hookleftarrow baz(v_1, v_2)\rangle, \{v_1 \mapsto 1, v_2 \mapsto 2\})$,
$(\langle v_2 \hookleftarrow baz(1^?, v_1)\rangle, \{v_1 \mapsto 2, v_2 \mapsto 1\})$,
$(\langle 1^? \hookleftarrow baz(v_2, v_1)\rangle, \{v_1 \mapsto 2, v_2 \mapsto 1\})$, and
$(\langle v_2 \hookleftarrow baz(v_2, v_1)\rangle, \{v_1 \mapsto 2, v_2 \mapsto 1\})$.

A couple of points are worth noting here. First, the example illustrates what happens when a concrete value is repeated multiple times in an event. In the above example, the value 1 occurs twice in the tuple e. When we produce generalized event predicates corresponding to the binding $\{v_1 \mapsto 1\}$, we have a choice of replacing various occurrence of 1 by v_1, and we consider all these possibilities (except that we require at least one occurrence of 1 to be replaced by v_1).

Second, note that in the case of generalized event predicates with two variables, we have some redundancy. For example, in *GenPredsOf* $(e, 2)$, $(\langle v_1 \hookleftarrow baz(1^?, v_2)\rangle, \{v_1 \mapsto 1, v_2 \mapsto 2\})$ and $(\langle v_2 \hookleftarrow baz(1^?, v_1)\rangle, \{v_2 \mapsto 1, v_1 \mapsto 2\})$ are the equivalent modulo variable renamings. We can eliminate this redundancy and reduce the size of the set of generalized event predicates of an event; however, this requires that checking equivalence of variable bindings at a later step in the algorithm be adapted to check for equality modulo variable renaming. (This is straightforward, but not shown in our algorithm presentation.)

Description. The algorithm to mine $\overset{*}{\rightarrowtail}$ rules involving quantifiable event predicates is given in Fig. 5. This algorithm is identical to the one in Fig. 4 except for processing generalized event predicates instead of propositional event predicates and the two minor yet key differences described below.

We represent a rule $\forall \vec{x}.\xi_1 \overset{*}{\rightarrowtail} \xi_2$ by the pair (ξ_1, ξ_2) where each ξ_i is a quantifiable event predicate. Hence, the map N_R used to compute the support of rules maps pairs of quantifiable event predicates to non-negative numbers (lines 4–5).

Within the algorithm, when we consider a pair of generalized event predicates (ξ_1, θ_1) and (ξ_2, θ_2) to determine if they are a witness to a rule, we check to see if the bindings θ_1 and θ_2 are equal (line 18): if they are, then these constitute a witness for the rule $\forall \vec{v_i}.\xi_1 \overset{*}{\rightarrowtail} \xi_2$.

As this algorithm is similar in structure to the algorithm for propositional rule mining, its complexity will be $N_l.F_e.M_e$ where F_e and M_e are defined over generalized event predicates (instead of event predicates). Note that when F_e and M_e are independent of N_l, the complexity is linear in N_l. However, if F_e and M_e increase when N_l increases, the complexity may not be linear in N_l.

When $k = 0$, the set of generalized event predicate corresponds to the propositional event predicates (with empty bindings). Hence, this algorithm generalizes our previous algorithm for mining propositional rules.

5.2. A comparison with alternative techniques

We now contrast our technique with an alternative approach based on the idea of *trace slicing* [4,5]. Let $\pi = f(1); f(2); g(1); g(2)$ be a trace (where the return values have been omitted for simplicity). The *slice* of π with respect to the value 1 is $f(x); g(x)$ and the slice of π with respect to value 2 is $f(x); g(x)$. The rule $f(x) \overset{*}{\rightarrowtail} g(x)$ can be mined from each of these trace slices, and this serves as the basis for mining quantified rules.

However, this approach raises several subtle questions. How do we define the slice of the trace $f(1, 1); g(1, 1)$ with respect to the value 1? Should the first event be abstracted into $f(x, x)$ or $f(x, 1)$ or $f(1, x)$? All are reasonable possibilities. If we choose just one of these possibilities, the mining algorithm becomes *incomplete* and may fail to mine some valid rules. If we consider all of these possibilities, then we need to consider 9 different slices (since we have 3 such choices for the second event as well). In general, the number of slices we need to consider could be exponential in the *length of the trace*.

Furthermore, computing support and confidence from the slices is tricky. If a single trace produces multiple slices, many of these may not satisfy a given rule, but cannot be treated as negative witnesses.

The same problem arises in the mining of rules with equality constraints. Yang et al. [4] use a *context-sensitive* mining approach for such rules. Essentially, this amounts to transforming a trace $f(1); g(2)$ into a trace $f\_1; g\_2$ and applying the basic mining algorithm to this trace (where f_1 is treated as a procedure name). Consider an event $f(1, 2)$. Should this be transformed into f or f_1 or f_2 or f_1_2? Again we face a choice between *incompleteness* or an *exponential blowup*.

One of our key contributions is a mining algorithm that is complete, yet avoids the above-mentioned exponential blowup. Our approach works by generalizing a trace into a sequence of sets of predicates (while the trace slicing approach relies on generalizing a trace into a set of sequence of predicates).

5.3. Alternation rule mining algorithm

We now describe our algorithm for mining alternation rules of the form $\xi_1 \overset{a}{\rightarrowtail} \xi_2$ (Fig. 6). The treatment of equality constraints and quantification in this case is the same as before and, hence, is not described here.

In comparison with the algorithms for mining eventually rules, this algorithm differs primarily in the overall iterative structure of the algorithm, which is guided by the nature of the alternation operator.

First, consider the following theorem:

$$support(\xi_1 \overset{a}{\rightarrowtail} \xi_2) \leq \min(support(\xi_1), support(\xi_2)).$$

As before, this theorem allows us to use the a priori property while mining. We have a stronger theorem here, which says that while mining rules of the form $f_1 \overset{a}{\rightarrowtail} f_2$ it is sufficient to consider only frequent f_1 and frequent f_2. Driven by this insight, we only capture information for rules composed of frequent event predicates (lines 4–5).

```
QuantifiedMustFollowEventually(T, S_min, C_min, k)
 1  # First Phase: Initialize
 2  Preds ← ⋃_{t∈T} ⋃_{e∈t} GenPredsOf(e, k)
 3  FreqPreds ← {ξ | (ξ, θ) ∈ Preds ∧ Supp(ξ, T) ≥ S_min}
 4  for each (ξ1, (ξ2, θ)) ∈ FreqPreds × Preds
 5  do N_R[ξ1, ξ2] ← 0
 6  # Second Phase: Mine rule instances
 7  for each t in T
 8  do for each i ← 1 to |t|
 9     do for each ζ ∈ PredsOf(t[i])
10        do last[ζ] ← i
11     for each ζ ∈ Preds
12     do N_P[ζ] ← 0
13     for i ← 1 to |t|
14     do e ← t[i]
15        for each ζ_e in GenPredsOf(e, k)
16        do if last[ζ_e] = i
17              then for each ζ_f in Preds
18                   do if ξ_f ∈ FreqPreds ∧ θ_e = θ_f
19                         then N_R[ξ_f, ξ_e] ←
20                              N_R[ξ_f, ξ_e] + N_P[ζ_f]
21           N_P[ζ_e] ← N_P[ζ_e] + 1
22 # Third Phase: Identify significant rules
23 Rules ← ∅
24 for each N_R[ξ1, ξ2] = s
25 do if s ≥ S_min ∧ (s/Supp(ξ1, T)) ≥ C_min
26       then Rules ← Rules ∪ {ξ1 ↣* ξ2}
27 return Rules
```

Fig. 5. Algorithm to mine $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ rules composed of quantifiable event predicates. *GenPredsOf* (e, k) is the set of generalized event predicates, with k quantifiers, satisfied by event e.

We also leverage the form of the rule to optimize the search of event predicates used to construct witnesses. Specifically, an event e cannot be paired with an event f to construct a witness for the rule $\xi_1 \overset{a}{\rightarrowtail} \xi_2$ if there exists an intermediate event e_i that satisfies either ξ_1 or ξ_2. Given an event e that satisfies ξ_1, the algorithm searches for events following e that can be paired with e to construct witnesses for alternation rules (lines 10–17). Due to the optimization, the algorithm aborts the search if it encounters an event e_i that satisfies ξ_1 (lines 12–13). Likewise, upon finding a rule f that satisfies ξ_2, the support for the rule $\xi_1 \overset{a}{\rightarrowtail} \xi_2$ (recorded in $N_R[\xi_e, \xi_f]$) is incremented by 1 (line 16) and the search is aborted.

5.4. Completeness and soundness

We now prove the completeness and soundness of PropositionalMustFollowEventually algorithm. As the algorithms are similar in structure and flow, the proof of soundness and completeness can be carried over to other algorithms.

Theorem 1 (*Soundness*)**.** *Every rule mined by* PropositionalMustFollowEventually *satisfies the support and confidence thresholds.*

Proof. In the third phase, PropositionalMustFollowEventually identifies only rules that satisfy support and confidence thresholds and emits them as the mined rules. Hence, the algorithm is sound. □

Theorem 2 (*Completeness*)**.** PropositionalMustFollowEventually *mines every rule that satisfies the support and confidence threshold.*

Proof. In the second phase, in a trace, PropositionalMustFollowEventually combines every event predicate with the last occurrence of every event predicate to construct the witness for the rule involving these two event predicates. Hence, all possible forward eventually rules are constructed by the algorithm. Further, the algorithm also calculates the correct support for every rule. Hence, in combination with the soundness theorem, the algorithm is complete. □

6. Eliminating redundancy in mined rules

Note that a number of logical implication relations hold between various different temporal rules. These implications are relevant in two ways. First, these implications can be used to make the mining process more efficient. Second, and more

```
MustFollowWithStrictAlternation(T, S_min, C_min, k)
# First Phase: Initialize
Preds ← ⋃_{t∈T} ⋃_{e∈t} GenPredsOf(e, k)
FreqPreds ← {ξ | (ξ, θ) ∈ Preds ∧ Supp(ξ, T) ≥ S_min}
for each (ξ1, ξ2) ∈ FreqPreds × FreqPreds
do N_R[ξ1, ξ2] ← 0
# Second Phase: Mine rule instances
for each t in T
do for i ← 1 to |t|
   do e ← t[i]
      for each ζ_e in FreqPredsOf(e, k)
      do for j ← i + 1 to |t|
         do if t[j] ⊨ ζ_e
               then break
               else for each ζ_f ∈ FreqPredsOf(f, k)
                    do if θ_e = θ_f
                          then N_R[ξ_e, ξ_f] ← N_R[ξ_e, ξ_f] + 1
                               break
# Third Phase: Identify significant rules
Rules ← ∅
for each N_R[ξ1, ξ2] = s
do if s ≥ S_min ∧ (s/Supp(ξ1, T)) < C_min
      then Rules ← Rules ∪ {ξ1 ↣^a ξ2}
return Rules
```

Fig. 6. Algorithm to mine $\xi_1 \overset{a}{\rightarrowtail} \xi_2$ rules where $FreqPredsOf(e, k) = \{(\xi, \theta) \in GenPredsOf(e, k) | \xi \in FreqPreds\}$.

importantly, they can be used to simplify the output set of mined rules by eliminating redundant rules, which can make it easier for end users to study the set of mined rules.

Theorem 3. *Let $\xi_1, \xi_2, \xi_3, \xi_1'$, and ξ_2' be event predicates. Then,*

1. $\xi_1' \Rightarrow \xi_1, \xi_1 \overset{*}{\rightarrowtail} \xi_2, \xi_2 \Rightarrow \xi_2'$ *imply* $\xi_1' \overset{*}{\rightarrowtail} \xi_2'$. *(Similarly for all other temporal operators.)*
2. $\xi_1 \overset{a}{\rightarrowtail} \xi_2$ *implies* $\xi_1 \overset{*}{\rightarrowtail} \xi_2$. *(Similarly for* $\overset{a}{\leftarrowtail}$ *and* $\overset{*}{\leftarrowtail}$*.)*
3. $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ *and* $\xi_2 \overset{*}{\rightarrowtail} \xi_3$ *imply* $\xi_1 \overset{*}{\rightarrowtail} \xi_3$. *(Similarly for* $\overset{*}{\leftarrowtail}$*.)*

Our algorithm for eliminating redundant rules iteratively identifies (using the above implications) and removes redundant rules. The complexity of our algorithm is quadratic in the number of rules. Our algorithm takes support and confidence of the rules into account to decide whether to eliminate a logically redundant rule: if a logically redundant rule has greater support/confidence than the rules that imply it, we retain the redundant rule.

7. Experimental evaluation

In this section, we empirically evaluate many aspects of our formalism and algorithms.

7.1. Expressiveness of QBEC

We chose to target QBEC rules in our work because we believe that it offers a good tradeoff between expressiveness and the complexity of mining. We performed a simple study to validate our belief that QBEC is quite expressive in practice. We manually analyzed a set of 78 widely used rules from the WDM API (incorporated in the Static Driver Verifier [1]).

Our study shows that 23 of the 78 rules can be directly expressed in QBEC and that 45 (including the 23 that are directly expressible) of the 78 rules can be expressed in a simple extension of QBEC where we permit inequality constraints (involving $<$ and $>$) over finite enumeration types and allow predicates involving global variables. (The use of global variables requires only a generalization of the concept of an event, rather than a generalization of QBEC.) Our mining algorithms can directly handle these extensions. (See our description of finitely instantiable predicates in Section 3.)

The other 33 rules cannot be expressed in QBEC for one or more of the following reasons.[1]

(a) 24 rules require ternary temporal operators, such as "between every occurrence of events e_1 and e_2, there must be an occurrence of event e_3".

[1] The sum of these three classes is more than 33 because some of these rules fall into multiple classes.

Table 1
The APIs used in evaluation along with the mined rules. An entry X/Y/Z in the Rules column denotes X propositional binary rules, Y binary rules w/ quantification, and Z binary w/ quantification and equality constraints were mined.

API	# Procs	# Traces	# Calls	# Rules	Supp/Conf
WDM	68	7038	99736	15/15/7	1000/0.9
IO	72	54	56721	8/9/53	1000/0.95
Registry	44	41	35435	13/12/54	2000/0.95
Memory	50	63	581187	10/11/8	20000/0.9
Printing	36	34	4172	21/13/110	200/0.95

(b) 18 rules require disjunction in the rule: e.g., "every occurrence of e_1 must be followed by an occurrence of e_2 or e_3".
(c) 17 rules require negation: e.g., a rule that an occurrence of e_1 must not be followed by an occurrence of e_2.

This preliminary analysis suggests that despite its simplicity, QBEC is quite expressive. Furthermore, QBEC is a very good basis for various natural extensions (of the form described above) that will make its expressiveness even better.

7.2. Implementation

We now describe a couple of aspects in which our implementation differs from the algorithms presented earlier.

Restricted $\overset{*}{\rightarrowtail}$ rules. While mining rules of the form $\xi_1 \overset{*}{\rightarrowtail} \xi_2$ and $\xi_2 \overset{*}{\leftarrowtail} \xi_1$, our implementation only considers consequents with an equality constraint involving at most one parameter. We made this pragmatic choice as the *a priori* property is not applicable to the consequent of these rules. We plan to relax this restriction in our ongoing work.

Value equality. One of the limitations of mining rules using dynamic execution traces is that it relies on the equality of values to relate events. However, it is possible that two unrelated events may have the same parameter or return values. Consider the following trace.

```
0x40000 = HeapAlloc(0x56000, 0);
...
HeapFree(0x56000, 0x40000);
...
0x40000 = HeapAlloc(0x56000, 0);
```

The memory object `0x40000` is reused by the memory allocator and hence returned by two calls to `HeapAlloc`. However, note that the call to `HeapFree` and the second call to `HeapAlloc` are logically unrelated, although they share a common value. Such accidental value equalities can lead to imprecision in the mined rules. We use a simple heuristic to work around this problem. While mining quantified rules, we restrict attention to those pairs of compatible generalized event predicates that do not have an intervening event whose return value equals the bound variable's value. We found that this heuristic helps improve the quality of mined rules.

7.3. Experimental methodology

We applied our algorithms to several commonly used Windows APIs. Table 1 lists the APIs, the number of procedures in the API, the number of traces we generated for each API, and the number of API calls in the traces.

The traces for the Windows APIs (Registry, IO, Memory Management and Printer) were generated using `logger` [6]. As clients of these APIs, we selected a number of desktop applications such as Adobe Reader, XEmacs, Windows media player, Outlook etc. We ran each application several times and during each run, we performed a series of actions simulating realistic usage of the applications.

The Windows Driver Model (WDM) is a framework for device drivers. We generated WDM traces from a set of 20 device drivers using a software model checker because we could not find a logging utility that generates device driver traces. The model checker executed the drivers to verify their correctness. We instrumented the model checker to generate a function call trace during every execution. Compared to the Windows API traces, the traces generated by the model checker tend to be smaller (an average of 14 API calls per trace). To compensate for the size of the traces, we generated a significantly larger number of traces.

We ran our experiments in a system with a 1.6 GHz Intel Pentium Core2 Duo processor with 3 GB RAM running Windows Vista. We measure the running times of the mining algorithms using the .NET `TimeSpan` class. The execution times we report are averages across 3 runs of the algorithm.

7.4. Size of the search space

We now present some metrics that characterize the size of the search space (of QBEC rules) considered by our miner.

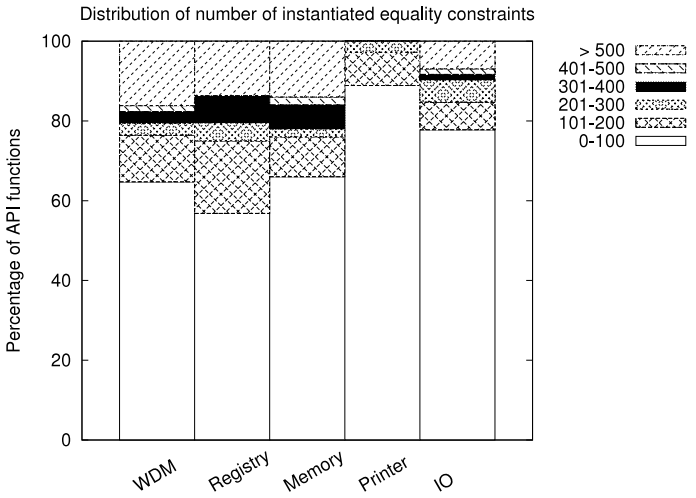

Fig. 7. The distribution of the number of equality constraints associated with functions in various APIs.

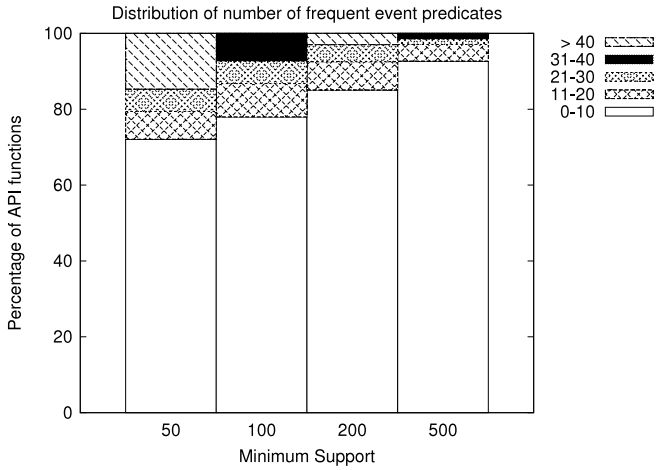

Fig. 8. The distribution of the number of frequent event predicates associated with each function in the WDM API for different support thresholds.

First, we measured the number of *instantiated* equality constraints associated with functions in the APIs. We say that an equality constraint $\$i = c_i$ where $i \neq 1$ is an instantiated equality constraint associated with a function f if there exists some event e in the traces such that $e \models (\$1 == f \wedge \$i == c_i)$. Fig. 7 shows the distribution of the number of instantiated equality constraints for various APIs. The bottom bar in each column of the figure shows that a majority of the functions are associated with less than 100 instantiated equality constraints. However, between 2% and 20% of functions have more than 500 constraints (represented by the top bar in each column). This observation shows that the search space of QBEC rules is extremely large. The observation also shows the need for mining algorithms that search this space efficiently.

As described earlier, our algorithms exploit the *a priori* property to search this space efficiently by considering only *frequent predicates* where possible. Fig. 8 shows the distribution of the number of frequent event predicates for functions from the WDM API for various minimum support values. Not surprisingly, most functions in the API have a small number (between 0 and 10) of frequent event predicates. The percentage of API functions with $\leq$10 frequent predicates increases from 70% to 90% as the support threshold is increased from 50 to 500. Note that the number of frequent predicates is significantly smaller than the number of instantiated equality constraints (Fig. 7). This shows the effectiveness of using the a priori property. The data from other APIs is qualitatively similar.

Fig. 9 shows the distribution of the size of frequent event predicates in the WDM API. The size of an event predicate ξ is the number of equality constraints in ξ. We find that most frequent event predicates are conjunctions of 1 or 2 equality constraints.

7.5. Impact of confidence/support threshold

We performed a sensitivity analysis to study the impact of the support and confidence thresholds on the number of QBEC rules mined by our algorithm. Tables 2 and 3 show the number of $\overset{*}{\rightarrowtail}$ rules mined, before and after redundant rule

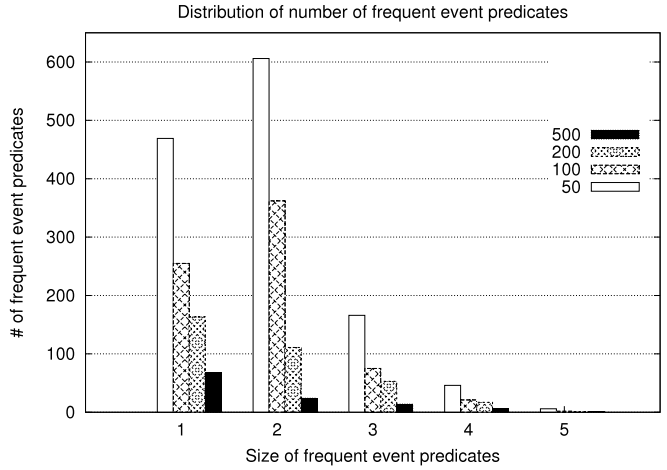

Fig. 9. The distribution of the size of frequent event predicates associated with functions in the WDM API for different support thresholds.

Table 2
Number of QBEC rules with equality constraints generated for the WDM API.

Support↓	Before elimination				After elimination			
Conf. →	0.8	0.9	0.95	0.98	0.8	0.9	0.95	0.98
50	2732	2212	2092	1079	315	268	230	261
100	1295	1117	1053	750	114	108	96	112
200	563	526	510	456	80	76	70	72
500	222	199	186	141	57	57	47	37
1000	176	156	149	111	39	37	31	23

Table 3
Number of QBEC rules generated for the memory management API.

Support↓	Before elimination			After elimination		
Conf. →	0.9	0.95	0.98	0.9	0.95	0.98
10 000	4301	3150	2967	3728	2777	2759
20 000	229	155	95	29	23	16
50 000	55	25	7	6	3	2

elimination, for the WDM and memory management APIs, respectively, when the support threshold or the confidence threshold is increased, showing that these parameters can be effective filters for choosing rules for subsequent manual examination. Note that in a few cases, the number of non-redundant rules mined increases with an increase in the confidence threshold. This is because of an interaction between the confidence threshold and redundant rule elimination. Consider two rules r_1 and r_2 and a rule r_s such that $r_s \implies r_1$ and $r_s \implies r_2$, where $\implies$ denotes the logical implication. Also assume that the rules r_1 and r_2 have a higher confidence than r_s. If the minimum confidence threshold is greater than the confidence of r_s, but less than the confidence of r_1 and r_2, the rule r_s will be dropped while the other rules will be retained. However, if the confidence threshold is decreased and falls below the confidence of r_s, this rule will be included as part of the rule set. Consequently, the redundancy elimination step, which is currently insensitive to the confidence threshold, will eliminate the rules r_1 and r_2 and retain r_s instead. Hence, decreasing the confidence threshold may result in the smaller non-redundant number of rules.

7.6. Effectiveness of redundancy elimination

Tables 2 and 3 also illustrate that redundancy elimination is very effective in reducing the number of mined rules that one must consider. On the average, this phase eliminates 77% of the rules in the WDM API and 82% of the rules in the Windows APIs. Thus, this phase is critical in ensuring that the mined rules do not overwhelm users.

7.7. Efficiency of the mining algorithms

We now evaluate the efficiency of our mining algorithms. Fig. 10 shows the total running time of the two algorithms for mining rules from WDM traces for various support thresholds. The running times are averages across four different

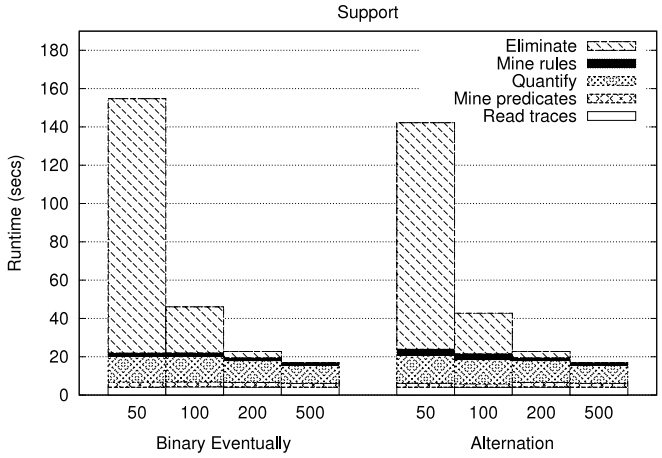

Fig. 10. Time taken to mine rules for the WDM API. The support threshold corresponding to each bar is mentioned beneath each bar.

Table 4
A selection of the QBEC rules mined using our tool. V represents a quantified variable.

Rule			Support	Confidence
KeAcquireSpinLock(_,V)	$\overset{*}{\rightarrowtail}$	KeReleaseSpinLock(_,V)	5124	1.00
ExAcquireFastMutex(V)	$\overset{*}{\rightarrowtail}$	ExReleaseFastMutex(V)	1242	1.00
InterlockedIncrement(V)	$\overset{*}{\rightarrowtail}$	InterlockedDecrement(V)	3571	1.00
IoGetNextIrpStackLocation(V)	$\overset{*}{\rightarrowtail}$	IoCallDriver(V,_)	2599	0.99
IoCopyCurrentIrpStackLocationToNext(V)	$\overset{*}{\leftarrowtail}$	IoCallDriver(V,0)	3346	0.98
V ExAllocatePoolWithTag(_,_,_)	$\overset{*}{\rightarrowtail}$	ExFreePool(V)	2060	0.97
IoCopyCurrentIrpStackLocationToNext(V)	$\overset{*}{\leftarrowtail}$	PoCallDriver(V,_)	519	0.94
IoCopyCurrentIrpStackLocationToNext(V)	$\overset{*}{\rightarrowtail}$	IoCallDriver(V,_)	4473	0.90

confidence thresholds; we did not find any significant variation in the running times as the confidence thresholds are changed. We can attribute the total running time to the time consumed in reading and parsing raw traces, mining frequent event predicates, mining rules, and eliminating redundant rules.

The figure shows that the time for mining the binary event rules and the alternation rules are almost the same. At low support thresholds, a large fraction of the running time is due to redundancy elimination. This is because redundancy elimination takes time quadratic in the number of rules and at low support thresholds, we mine a large number of rules. The running time decreases sharply as the minimum support threshold is increased. However, even with high support thresholds, the time consumed in mining QBEC rules itself is small ($\tilde{1}$0% on average), with a large fraction of the runtime is attributed to trace quantification. E.g., rules with support and confidence greater than 200 and 0.9 respectively are mined in approximately 20 s, out of which 11 s are spent in trace quantification and 3.4 s in mining rules.

7.8. Quality of mined rules

An important measure of a mining algorithm is its precision: how many of the candidate rules mined are indeed valid rules? However, evaluating this metric is challenging (in our context) as it can only be done manually. Table 4 shows some of the mined rules with their support and confidence. Following are a few interesting observations from this data.

- Many of the rules we mine are known rules documented as part of the Static Driver Verifier (SDV) tool. For instance, the quantified rule involving `KeAcquireSpinLock` is part of the SDV toolkit.

  Some of the rules we mine are not known rules but represent common idioms that most clients follow. For instance, consider the forward binary-eventually rule involving the function `IoCopyCurrentIrpStackLocation` and `IoCallDriver`. This rule is informally documented as follows [7].

  > "After calling `IoCopyIrpStackLocationToNext`, a driver *typically* sets an I/O completion routine with `IoSetCompletionRoutine` before passing the IRP to the next-lower driver with `IoCallDriver`."

  The documentation suggests that this rule is perhaps a common idiom. This may explain the number of violations of this rule. The documentation for the rule involving `PoCallDriver` is along similar lines. On the other hand, consider the

backward-eventually rule between the same functions (with the additional equality constraint on the second parameter). This rule is documented as follows, which points to a stronger rule that must be followed.

> "Before a calling `IoCallDriver`, the calling driver *must* set up the I/O stack location in the IRP for the target driver."

We did not find a violation of this rule in any of the drivers in the validation suite.

- In general, one may expect the confidence to correlate with the number of violations of the rule. However, we do not find a strong correlation between these quantities. On the contrary, we find that several rules with high confidence are also violated by many clients. There may be several reasons for the lack of correlation, including the presence of rare paths such as error handling code that do not show up in the traces and that some of the rules we mine represent common idioms that all clients may not follow.
- We found that the violations of some of the rules (such as a few forward-eventually rules involving `KeAcquireSpinLock` and `ExAllocatePoolWithTag` in antecedent positions) *a*re definite bugs in the drivers.

Similarly, we validated some of the rules mined for the other APIs (the rules mined at a reasonably high support and confidence specific to each API) against the informal documentation of the APIs and found that a large fraction of the rules were stated in the documentation.

8. Related work

The topic of specification mining has attracted wide attention in the recent years. The early work of [2] addresses the problem of mining program invariants (as opposed to API usage rules), but restricts attention to non-temporal invariants. Some researchers (e.g., [8]) explore the problem of mining API usage rules by analyzing the library, while others (including us) use clients to mine API usage rules. Within the space of client-based mining, several researchers (e.g., [9]) have pursued a static-analysis based approach to mining, while we (and several others) address the problem of mining specifications from traces. We now compare our work with related work in the space of trace-based specification-mining.

We first state some distinguishing features of our work. (a) One of our significant contributions relates to *quantification*. We provide a simple yet rigorous formalization of event quantification along with a general algorithm to mine quantified rules. Further, our algorithm is complete with respect to possible quantifications that are considered during rule mining. (b) We provide a unified formalism and mining algorithm that combine *state predicates* with temporal constraints. (c) The algorithm we present for mining binary-eventually rules is novel, and is linear in the total size of the input traces. (d) Our approach exploits the classical *a priori* property from data-mining to make the mining more efficient.

The work most closely related to ours is that of Yang et al. [4], Yang and Evans [10]. Yang et al. also focus on mining binary temporal rules, but differ from us in several respects. They rely on *trace slicing* for quantification and *context-sensitive* mining for *equality constraints*. Section 5.2 compares these approaches and outlines the advantages of our approach.

Quantified rules are very common, but only a few past efforts support the mining of quantified rules. Ammons et al. [11] support only quantification over the first argument to a procedure call. Chen and Rosu [5] is the only prior work that provides a complete formalism for quantification. Our formalism, done independently, is similar in some respects to their formalism, but there are very significant differences as well. The Chen and Rosu algorithm is based on *trace slicing* and we contrast our work with trace slicing in Section 5.2.

Some of the previous work [5,11,12] has focused on mining API usage rules in the form of a *single* finite-state automaton. Our approach may be viewed as mining a *number of small* automata of a special form (corresponding to the temporal operators), which has some advantages. E.g., consider an API with k rules $f_i \overset{*}{\rightarrowtail} g_i$, $1 \leq i \leq k$. Expressing these as a *single* automaton would require 2^k states. Since mining algorithms tend to limit their attention to automata with a limited number of states, a single-automaton-miner is likely to miss some of these temporal rules. In contrast, mining this set of k temporal rules is straightforward with our approach. Furthermore, our mining algorithms are linear in the total size of the trace, while the automaton-based approaches are cubic. It would be interesting to generalize our approach to mine *a set of arbitrary automatons within some size*. Recently, Gabel and Su [13] showed how simpler rules (like ours) can be combined to form more complex rules (or automaton).

Recently, Lorenzoli et al. [12] have presented a technique for mining an *Extended FSM*, which combines state predicates with finite-state automaton, but this neither supports quantification nor tolerates erroneous inputs. In contrast, our techniques can mine binary temporal rules involving state predicates and quantification (more efficiently) while tolerating erroneous inputs.

The work in [14,15] also use data mining, but they mine frequent patterns rather than rules. While rules capture constraints, patterns only capture series of events that appear frequently. De Sousa et al. [16] address the mining of implied scenarios, in the form of message sequence charts, which describe sequences of events that could occur.

9. Conclusion and future work

In this paper, we have introduced QBEC, a class of quantified binary temporal rules with equality constraints, that is equivalent to a subset of LTL formulae. As common API usage patterns can be easily expressed as QBEC rules, we explored

the possibility of mining QBEC rules in the context of mining API usage patterns (as QBEC rules). Consequently, as part of devising algorithms to mine QBEC rules, we described a novel formalism for mining quantified rules along with a general event quantification technique that can be seamlessly leveraged by existing standard data-mining frameworks. Further, we empirically evaluated and demonstrated the expressiveness of QBEC and the feasibility of efficiently and effectively mining QBEC rules in the context of rules shipped with an industrial strength program verification tool.

Going forward, there are few possibilities to extend our work. The first possibility is to explore extensions to QBEC to express higher arity rules i.e. , rules involving more than two events. The second possibility is to admit disjunctive predicates, i.e. , predicates formed by the disjunction of equality constraints, in QBEC rules. Along the same lines, it would be interesting to admit non-equality constraints in QBEC rules. It would be useful to extend QBEC with the support to capture various forms of timing constraints, e.g. event e will be followed by event f within 3 s.

Distinct from extensions to QBEC, it would be interesting to explore various applications of QBEC rules in the context of developer assistance, fault analysis, compatibility checking, program verification, and system optimization.

References

[1] Static driver verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx.
[2] M. Ernst, J. Cockrell, W. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, TSE 27 (2).
[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proc. of VLDB, 1994.
[4] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, M. Das, Perracotta: mining temporal API rules from imperfect traces, in: Proc. of ICSE, 2006.
[5] F. Chen, G. Rosu, Mining parametric state-based specifications from executions, in: Technical Report, 2008, Unpublished.
[6] Debugging tools for windows. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.
[7] Windows Driver Development. http://www.osronline.com/.
[8] R. Alur, P. Cerny, G. Gupta, P. Madhusudan, Synthesis of interface specifications for java classes, in: Proc. of POPL, 2005.
[9] M. K. Ramanathan, A. Grama, S. Jagannathan, Static specification inference using predicate mining, in: Proc. of PLDI, 2007.
[10] J. Yang, D. Evans, Dynamically inferring temporal properties, in: Proc. of PASTE, 2004.
[11] G. Ammons, R. Bodik, J.R. Larus, Mining specification, in: Proc. of POPL, 2002.
[12] D. Lorenzoli, L. Mariani, M. Pezzè, Automatic generation of software behavioral models, in: Proc. of ICSE, 2008.
[13] M. Gabel, Z. Su, Javert: fully automatic mining of general temporal properties from dynamic traces, in: Proc. of FSE, 2008.
[14] H. Safyallah, K. Sartipi, Dynamic analysis of software systems using execution pattern mining, in: Proc. of ICPC, 2006.
[15] M. El-Ramly, E. Stroulia, P. Sorenson, Interaction-pattern mining: extracting usage scenarios from run-time behavior traces, in: Proc. of KDD, 2002.
[16] F. de Sousa, N. Mendonca, S. Uchitel, J. Kramer, Detecting implied scenarios from execution traces, in: Proc. of Work. Conf. on Reverse Engineering, 2007.