Mining Quantified Temporal Rules: Formalism, Algorithms, and Evaluation

David Lo *davidlo@smu.edu.sg*

G. Ramalingam *grama@microsoft.com*

Venkatesh Prasad Ranganath *rvprasad@microsoft.com*

Kapil Vaswani *kapilv@microsoft.com*

***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 (involving equality constraints) and quantification. This is a simple yet expressive subclass of temporal properties 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* — 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.

I. 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 $m5(3, ...)$ 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 $m4(x)$".

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 are satisfied by the given traces), while avoiding an exponential blowup that alternative strategies can encounter.

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 that are satisfied by all traces in the given set. Unfortunately, such an approach is not robust to occasional bugs in the programs (used for mining), 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 satifying 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 for these two quantities.

Efficient Mining Algorithms: In this paper, we present efficient algorithms for mining QBEC rules. These algorithms are efficient as their time complexity is linear in the length of the analyzed traces. 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 shows how we can naturally extend the above mentioned propositional rule mining algorithms to mine quantified rules. The formal approach also helped us uncover and handle subtle corner cases correctly.

Eliminating Redundant Rules: In general, the mining rules may contain some redundancy. A rule is *redundant* if it is logically implied by one or more of other rules. We propose a set of simplifications to eliminate redundancy in in a set of 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 efficient. The results also show that the mining is effective: our miner is able to mine device driver rules that are both valid and useful — help identify real bugs in device drivers. Our miner is able to mine rules used by the Static Driver Verifier [1] as well as rules not currently in SDV. The latter set of rules look reasonable based on our preliminary examination of textual documentation. 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. We study the impact of varying confidence and support on the mining. 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 mining algorithm that handles quantification and state predicates in unified fashion. Furthermore, our mining algorithm is complete (i.e., it mines all rules within the targeted class that have the desired support and confidence in the given execution traces).
- A new linear-time algorithm for mining must-be-followed-by rules.
- An empirical evaluation of our mining algorithm.

II. A Class of Temporal Rules

In this section, we define various classes of temporal rules that are interesting from the perspective of mining, including several classes addressed by previous work in specification mining.

Formally, we assume that we are given a set of *sequences* of *events*, which we 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.

Events: In our work, an event is a *tuple of primitive values*. A primitive value is either 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 $[0, \texttt{foo}, 5, 10]$ represents a call to `foo` with parameters 5 and 10 whose return value was 0. We will represent such a tuple mnemonically as "0 $\hookleftarrow$ `foo`(5,10)".

More generally, however, an event could be any concrete or abstract representation of program 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.

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*.

Let $\mathbb{EC}_1$ denote the subset of $\mathbb{EC}$ consisting of predicates of the form "$\$1 = \texttt{procname}$". 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 "$\$1 = \texttt{procname}$". 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 $\langle c_0 \hookleftarrow c_1(c_2, \cdots, c_k)\rangle$, 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, $\langle c_0 \hookleftarrow c_1(c_2, \cdots, c_k)\rangle$ is short for $\bigwedge\{\ \$i = c_i \mid 0 \le i \le k, c_i \ne \_\ \}$. Further, we will abbreviate $\langle c_0 \hookleftarrow c_1(c_2, \cdots, c_k)\rangle$ to $\langle c_1(c_2, \cdots, c_k)\rangle$ when c_0 is "_".

For example, the event "0 $\hookleftarrow$ `foo`(5,10)" satisfies the following predicates in $\mathbb{EC}^*$: $\langle \_ \hookleftarrow \texttt{foo}(\_,\_)\rangle$, $\langle 0 \hookleftarrow \texttt{foo}(\_,\_)\rangle$, and $\langle \_ \hookleftarrow \texttt{foo}(5, 10)\rangle$.

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 formula $\langle \texttt{foo}(1)\rangle \overset{*}{\rightarrowtail} \langle \texttt{bar}(2)\rangle$ represents the rule that any call to `foo` with the first argument as 1 must be followed by a call to `bar` with the first argument as 2.

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 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 and (ii) an occurrence of ξ_1 cannot be followed by another occurrence of ξ_1 before an occurrence of ξ_2. 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.

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

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 `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 $\theta(x)$.

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 any 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 $e_j \models \xi_2[\theta]$. The quantified form of other rules and their meanings are defined in a similar fashion.

The above definition constrains the antecedent and the consequent to have the same set of free variables, without 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.

Summary: We obtain different classes of rules by varying (a) the temporal operators allowed, (b) the 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 whole number. We define $\mathfrak{F}(\mathsf{EP}, \mathsf{T}, n)$ to be the set of temporal rules built out of event predicates in EP and temporal operators in T and consisting of at most n quantified variables. In this paper, we present our formalization and mining algorithms by considering the following, increasingly richer, classes of rules in that order.

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 shall illustrate and evaluate our algorithm in this class, our algorithm 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 can handle any value of k.

III. 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 [2] 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 $\pi_1, \cdots, \pi_n$ 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. 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.

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}.

IV. 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.

A. 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.

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.

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 *Apriori* property [2] in the data mining community. The following (straight-forward) theorem serves as the basis for applying the Apriori property:

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

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 atleast S_{min}. It follows that in mining rules of the form $\langle f\rangle \overset{*}{\rightarrowtail} \langle g\rangle$ it suffices to consider only functions 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. While this improvement may not appear exciting, the Apriori property will be significant as we expand our scope to more general forms of rules.

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 Figure 1 and it operates in three phases.

In the initial 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 atleast 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 pair of event predicates from the set *FreqPreds* $\times$ *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 and F_e is the number of frequent event predicates, 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 [2]. The key idea exploited here is again the Apriori 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 lets us explore a big space of candidate predicates effectively.

3) Quantified Rules with Equality Constraints: We now extend our algorithm to mine quantified rules. As motivation, consider the common rule involving lock and unlock operations, on a given lock, must strictly alternate. Consider the trace π = $0\hookleftarrow$*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

```
PROPOSITIONALMUSTFOLLOWEVENTUALLY($T, S_{min}, C_{min}$)
 1  # (1) Initialize
 2  Preds ← $\bigcup_{t \in T} \bigcup_{e \in t}$ PredsOf($e$)
 3  FreqPreds ← $\{\xi \in$ Preds $\mid$ Supp$(\xi, T) \geq S_{min}\}$
 4  for each $(\xi_1, \xi_2) \in$ FreqPreds × Preds
 5  do $N_R[\xi_1, \xi_2] \leftarrow 0$
 6  # (2) Mine rule instances
 7  for each $t$ in $T$
 8  do for each $i \leftarrow 1$ to $|t|$
 9        do for each $\xi \in$ PredsOf($e$)
10              do last[$\xi$] ← $i$
11     for each $\xi \in$ Preds
12     do $N_P[\xi] \leftarrow 0$
13     for $i \leftarrow 1$ to $|t|$
14     do $e \leftarrow t[i]$
15           for each $\xi_e$ in PredsOf($e$)
16           do if last[$\xi_e$] $= i$
17                  then for each $\xi_f$ in FreqPreds
18                         do $N_R[\xi_f, \xi_e] \leftarrow N_R[\xi_f, \xi_e] + N_P[\xi_f]$
19              $N_P[\xi_e] \leftarrow N_P[\xi_e] + 1$
20  # (3) Identify significant rules
21  Rules ← $\emptyset$
22  for each $N_R[\xi_1, \xi_2] = s$
23  do if $s \geq S_{min} \wedge (s/$Supp$(\xi_1, T)) \geq C_{min}$
24        then Rules ← Rules $\cup \{\xi_1 \overset{*}{\rightarrowtail} \xi_2\}$
25  return Rules
```

Figure 1. 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*(ϕ, T) is the total number of ϕ satisfying events in T.

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 $\{\ v_1, \cdots, v_k\ \}$ 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 $\{\ v_1, \cdots, v_k\ \}$. 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.

For example, let event $e = 1\hookleftarrow$*baz*(2). Then, *GenPredsOf*$(e, 0)$ consists of $(\langle 1^? \hookleftarrow baz(2^?)\rangle, \{\})$, where $1^?$ can be replaced by either 1 or _ and $2^?$ can be replaced by either 2 or _. $\mathit{GenPredsOf}(e,1)$ contains $(\langle v_1 \hookleftarrow baz(2^?)\rangle,\ \{v_1 \mapsto 1\})$ and $(\langle 1^? \hookleftarrow baz(v_1)\rangle,\ \{v_1 \mapsto 2\})$. $\mathit{GenPredsOf}(e,2)$ consists of $(\langle v_1 \hookleftarrow baz(v_2)\rangle,\ \{v_1 \mapsto 1, v_2 \mapsto 2\})$ and $(\langle v_2 \hookleftarrow baz(v_1)\rangle,\ \{v_1 \mapsto 2, v_2 \mapsto 1\})$. (The two predicates in $\mathit{GenPredsOf}(e,2)$ are equivalent modulo variable renaming, but we retain such redundant predicates to simplify presentation.)

Description: The algorithm to mine $\overset{*}{\rightarrowtail}$ rules involving quantifiable event predicates is identical to the one in Figure 1 with the exception of processing generalized event predicates (controlled by a parameter k instead of propositional event predicates (at lines 2, 9, and 15) and except for 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 (at lines 4-5).

While selecting witnesses for a rule at line 17, we consider only pairs of generalized event predicates (ξ_1, θ_1) and (ξ_2, θ_2) where ξ_1 is frequent (i.e. $\xi_1 \in \mathit{FreqPreds}$) and the bindings are equal, i.e. $\theta_1 = \theta_2$.

For details of the algorithm, please refer to the technical report [3].

B. A Comparison with Alternative Techniques

We now contrast our technique with an alternative approach based on the idea of *trace slicing* [4], [5]. Let π = $f(1); f(2); g(1); g(2)$ 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 (i.e. mine all rules within the targeted class that are satisfied by the given traces), 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).

C. Alternation Rules

Similar to algorithms for mining eventually rules, we have designed algorithms for mining both propositional and quantified forms of alternation rules. While the treatment of equality constraints and quantification in these algorithms is the same as in the algorthims for mining eventually rules, the key differences are 1) the overall iterative structure of the algorithms, which are guided by the nature of the alternation operator, and 2) the application of Apriori property: we rely on a relatively stronger theorem: $\mathit{support}(\xi_1 \overset{a}{\rightarrowtail} \xi_2) \leq \mathit{min}(\mathit{support}(\xi_1), \mathit{support}(\xi_2))$ to only consider frequent events f_1 and f_2 for mining $f_1 \overset{a}{\rightarrowtail} f_2$.

Due to lack of space, we present these algorithms in the techincal report [3].

V. Eliminating Redundancy

Note that a number of logical implication relations hold between various different temporal rules. These implications 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 1: 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.

VI. Experimental Evaluation

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

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

Table I
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.

A. *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 II).

The other 33 rules cannot be expressed in QBEC for one or more of the following reasons: (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". (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., "an occurrence of e_1 must not be followed by an occurrence of e_2".

B. *Implementation*

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

Restricted to $\overset{*}{\rightarrowtail}$ 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);
```

Here, 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 rules we generate, and we plan to formally study this phenomenon in a future work.

C. *Experimental Methodology*

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

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 in the process of verifying 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 1.6GHz Intel Pentium Core2 Duo processor with 3GB RAM running Windows Vista. We measured 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.

D. *Size of the Search Space of QBEC rules*

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

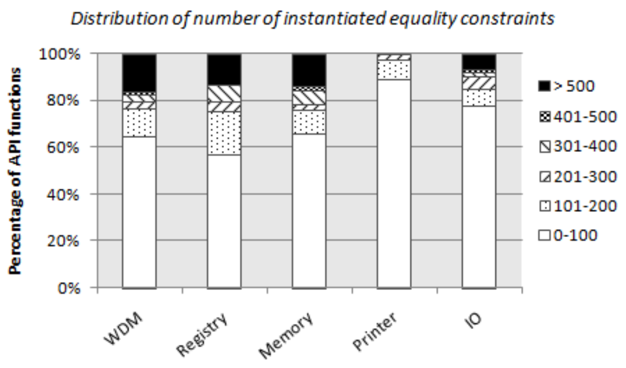

Figure 2. The distribution of the number of base event predicates associated with functions in various APIs.

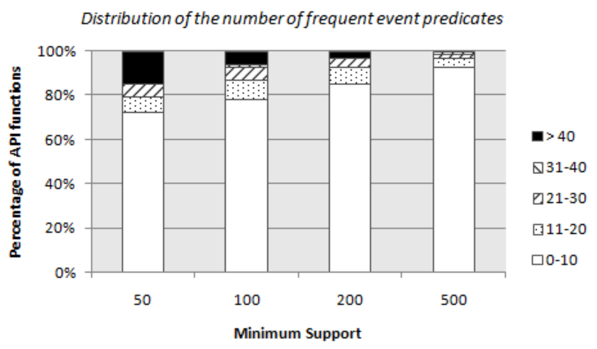

Figure 3. The distribution of the number of frequent event predicates associated with each function in the WDM API for different support thresholds.

$e \models (\$1 == f \wedge \$i == c_i)$. Figure 2 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. Figure 3 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 (Figure 2). This shows the effectiveness of using the a priori property. The data from other APIs is qualitatively similar; we omit the results due to space constraints.

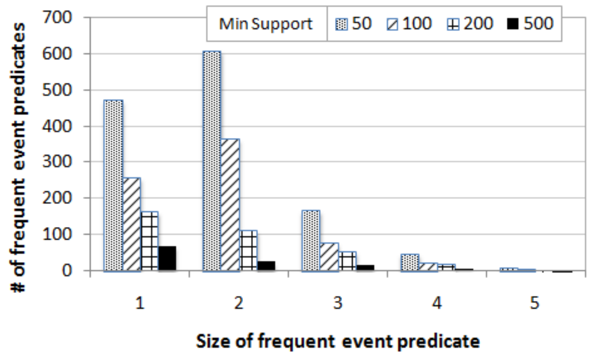

Figure 4. The distribution of the size of frequent event predicates associated with functions in the WDM API for different support thresholds.

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

Table II
NUMBER OF QBEC RULES WITH EQUALITY CONSTRAINTS GENERATED FOR THE WDM API.

Figure 4 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.

E. 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 II and III show the number of $\overset{*}{\rightarrowtail}$ rules mined, before and after redundant rule elimination, for the WDM and memory management APIs. As these tables illustrate, the number of mined rules decrease significantly 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 can happen when the increase in the confidence threshold causes a stronger rule r in the output to be replaced with two or more weaker rules (which were originally redundant because of r).

F. Effectiveness Of Redundancy Elimination

Tables II and III 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.

Support↓	**Before elimination**			**After elimination**		
Conf. →	**0.9**	**0.95**	**0.98**	**0.9**	**0.95**	**0.98**
10000	4301	3150	2967	3728	2777	2759
20000	229	155	95	29	23	16
50000	55	25	7	6	3	2

Table III
NUMBER OF QBEC RULES GENERATED FOR THE MEMORY MANAGEMENT API.

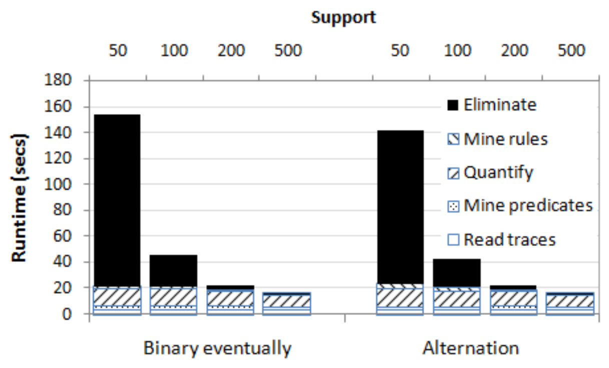

Figure 5. Time taken to mine rules for the WDM API.

G. *Efficiency of the Mining Algorithms*

Figure 5 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 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, performing trace quantification, mining rules themselves and finally, eliminating redundant rules.

The figure shows that the time for mining the binary eventually 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 (10% 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 seconds, out of which 11 seconds are spent in trace quantification and 3.4 seconds in mining rules.

H. *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. A manual examination of a subset of the rules mined from the WDM traces produced promising results. We found that several of the examined rules were valid rules, documented as part of the Static Driver Verifier (SDV) tool. Several of the remaining rules appeared to be valid, based on an examination of the textual documentation [7]. Several of the mined rules appear to correspond to idiomatic programming practice recommendations [7], though it is unclear if these are mandatory rules.

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.

VII. RELATED WORK

The topic of specification mining has attracted wide attention in the recent years. The early work of [8] addresses the problem of mining program invariants (as opposed to API usage rules), but restricts attention to non-temporal invariants. Some researchers (e.g., [9]) 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., [10]) 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) 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.* [11], [4]. 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 IV-B 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.* [12] 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

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

Table IV
A SELECTION OF THE QBEC RULES MINED USING OUR TOOL. V REPRESENTS A QUANTIFIED VARIABLE.

well. The Chen and Rosu algorithm is based on *trace slicing* and we contrast our work with trace slicing in Section IV-B.

Some of the previous work [12], [13], [5] 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 $f_i \overset{*}{\rightarrowtail} g_i$, $1 \le i \le k$. 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 [14] showed how simpler rules (like ours) can be combined to form more complex rules (or automaton).

Recently, Lorenzoli *et al.* [13] 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 [15], [16] 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.* [17] address mining of implied scenarios, in the form of message sequence charts, which describe sequences of events that could occur.

REFERENCES

[1] "Static driver verifier," http://www.microsoft.com/whdc/devtools/tools/sdv.mspx.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of VLDB*, 1994.

[3] D. Lo, G. Ramalingam, V. P. Ranganath, and K. Vaswani, "Mining quantified temporal rules: Formalism, algorithms, and evaluation," Microsoft Research, Tech. Rep., 2008.

[4] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining temporal API rules from imperfect traces." in *Proc. of ICSE*, 2006.

[5] F. Chen and G. Rosu, "Mining Parametric State-Based Specifications from Executions," in *Technical Report (Unpublished)*, 2008.

[6] "Debugging tools for windows," http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

[7] "Windows Driver Development," http://www.osronline.com/.

[8] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *TSE*, vol. 27, no. 2, 2001.

[9] R. Alur, P. Cerny, G. Gupta, and P. Madhusudan, "Synthesis of interface specifications for java classes." in *Proc. of POPL*, 2005.

[10] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Static specification inference using predicate mining," in *Proc. of PLDI*, 2007.

[11] J. Yang and D. Evans, "Dynamically inferring temporal properties." in *Proc. of PASTE*, 2004.

[12] G. Ammons, R. Bodik, and J. R. Larus, "Mining specification," in *Proc. of POPL*, 2002.

[13] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic Generation of Software Behavioral Models," in *Proc. of ICSE*, 2008.

[14] M. Gabel and Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in *Proc. of FSE*, 2008.

[15] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining." in *Proc. of ICPC*, 2006.

[16] M. El-Ramly, E. Stroulia, and P. Sorenson, "Interaction-pattern mining: Extracting usage scenarios from run-time behavior traces." in *Proc. of KDD*, 2002.

[17] F. de Sousa, N. Mendonca, S. Uchitel, and J. Kramer, "Detecting implied scenarios from execution traces.." in *Proc. of Work. Conf. on Reverse Engineering*, 2007.