7-2008

Mining temporal rules for software maintenance

David LO
Singapore Management University, davidlo@smu.edu.sg

Siau-Cheng Khoo
National University of Singapore

Chao LIU

DOI: https://doi.org/10.1002/smr.375

Citation

LO, David; Khoo, Siau-Cheng; and LIU, Chao. Mining temporal rules for software maintenance. (2008). *Journal of Software Maintenance and Evolution: Research and Practice*. 20, (4), 227-247. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/742

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **20**:227–247
Published online 9 July 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.375

Research

Mining temporal rules for software maintenance

David Lo[1,*,†,‡], Siau-Cheng Khoo[2] and Chao Liu[3]

[1]*School of Information Systems, Singapore Management University, Singapore*
[2]*Department of Computer Science, National University of Singapore, Singapore*
[3]*Microsoft Research, Redmond, WA, U.S.A.*

SUMMARY

Software evolution incurs difficulties in program comprehension and software verification, and hence it increases the cost of software maintenance. In this study, we propose a novel technique to mine from program execution traces a sound and complete set of statistically significant temporal rules of arbitrary lengths. The extracted temporal rules reveal invariants that the program observes, and will consequently guide developers to understand the program behaviors, and facilitate all downstream applications such as verification and debugging. Different from previous studies that were restricted to mining two-event rules (e.g., ⟨*lock*⟩ → ⟨*unlock*⟩), our algorithm discovers rules of arbitrary lengths. In order to facilitate downstream applications, we represent the mined rules as temporal logic expressions, so that existing model checkers or other formal analysis toolkit can readily consume our mining results. Performance studies on benchmark data sets and a case study on an industrial system have been performed to show the scalability and utility of our approach. We performed case studies on JBoss application server and a buggy concurrent versions system application, and the result clearly demonstrates the usefulness of our technique in recovering underlying program designs and detecting bugs. Copyright © 2008 John Wiley & Sons, Ltd.

Received 28 April 2008; Revised 28 April 2008; Accepted 7 June 2008

KEY WORDS: specification mining; software evolution; dynamic analysis; data mining; program comprehension; program verification

1. INTRODUCTION

Software keeps changing throughout its lifespan. Software maintenance deals with the management of such changes, ensuring that the software remains correct while features are added or removed.

*Correspondence to: David Lo, School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902.
†E-mail: davidlo@smu.edu.sg
‡This work was done while the author was with School of Computing, National University of Singapore.

Maintenance cost can contribute up to 60–80% of software cost [1]. A challenge to software maintenance is to keep documented specifications accurate and updated as the program changes. Outdated specifications cause difficulties in program comprehension, which account for up to 50% of program maintenance cost [1].

In order to ensure software correctness, model checking [2] has been proposed and proved useful in many cases. It accepts a model, often automatically constructed from the code, and a set of formal properties to check. However, the difficulty in formulating a set of formal properties has been a barrier to its wide-spread application [3]. Adding software evolution to the equation, the verification process is further strained. First, ensuring the correctness of software as changes are made is not a trivial task: a change in part of a program might induce unwanted effects resulting in bugs in other parts of the code. Furthermore, as a system changes and features are added, there is a constant need to add new properties or modify outdated properties to render automated verification techniques effective in detecting bugs and ensuring the correctness of the system.

While addressing the above problems, there is a need for techniques to automatically mine formal specifications from a program as it changes over time. Employing these techniques ensures that specifications remain updated; it also provides a set of properties to be verified via formal verification tools such as model checking. This family of techniques is commonly referred to as 'specification mining' [3].

There have been a number of studies on specification mining, which relate to either program comprehension (e.g., [4–6]) or verification (e.g., [3,7]). Most specification mining algorithms extract specifications in the form of an automaton (e.g., [3–6]) or two-event rules (e.g., [7]). Although a mined automaton expresses a global picture of a software specification, mined rules break this into smaller parts, each expressing a program property, which is significantly observed. However, existing work on mining rules mines only two-event rules (e.g., $\langle lock\rangle \rightarrow \langle unlock\rangle$), which are limited in their ability to express complex temporal properties.

The focus of this study is to automatically discover rules *of arbitrary lengths* having the following form from program execution traces:

Whenever a series of precedent events occurs, eventually another series of consequent events occurs.

A trace can be viewed as a series of events, with each event corresponding to a software behavior of interest. In the existing literature on specification mining [3,6,7] a trace usually corresponds to a series of signatures of methods that are invoked when a program is executed. A multi-event rule is denoted by $ES_{pre} \rightarrow ES_{post}$, where ES_{pre} and ES_{post} are the premise/pre-condition and the consequent/post-condition, respectively.

The above multi-event rule can be expressed in temporal logic and belongs to two of the most frequently used families of temporal logic expressions for model checking (i.e., response and chain-response) according to a survey in [8]. Examples of such rules include

1. Resource locking: Whenever a lock is acquired, eventually it is released.
2. Initialization–termination: Whenever a series of initialization events is performed, eventually a series of termination events is also performed.
3. Internet banking: Whenever a connection to a bank server is made, an authentication is completed, and money transfer command is issued; eventually the money is transferred and a receipt is displayed.

From traces, many rules can be inferred, but not all are important. Statistics of support and confidence employed in data mining [9] is therefore used to identify important rules. Rules satisfying user-specified thresholds of minimum support and confidence are referred to as *statistically significant* rules.

Effective search space pruning strategies are utilized to efficiently mine multi-event rules from traces. To prevent an explosion in the number of mined rules, we define a *redundancy relation* among rules and propose to generate only a minimal subset of rules containing non-redundant ones (see Section 3). With respect to input traces and given statistical significance thresholds, our algorithm is *statistically sound*, as all mined rules are statistically significant (i.e., they meet the thresholds). It is also statically *complete* as all statistically significant rules of the form $ES_{pre} \rightarrow ES_{post}$ are mined or represented.

Our definitions of statistical soundness and completeness follow data mining definitions. In data mining, soundness and completeness is defined based on the input, i.e., the supplied set of traces and the statistical thresholds. In program analysis, soundness and completeness is defined based on the underlying software. We added the term 'statistically' to the earlier definition to describe this difference. In some sense, for purely dynamic-analysis-based algorithm: 'statistically sound and complete' is the best quality guarantee that an algorithm can achieve. The output can only be as good as the input. Hypothetically speaking, provided that the traces are complete, the result will be complete as well. However, a limitation with dynamic analysis is that it is difficult to generate complete or representative execution traces [10]. In the future, we shall look into the static analysis approach to generate representative traces with some guarantee from the code.

We carried out a performance study on several standard benchmark data sets to demonstrate the effectiveness of our search space pruning strategies. We performed a case study on JBoss application server (JBoss-AS)—a widely used J2EE server—to illustrate the usefulness of our technique in recovering the specifications that a software system obeys. We also performed a case study on a buggy concurrent versions system (CVS) application. It shows the usefulness of our technique in mining bug-revealing properties, thus aiding model checkers in finding bugs.

Our contributions are as follows:

1. We address the limitations of approaches extracting automata-based specification from traces, by discovering statistically sound and complete, and easily understood specifications in temporal logic format frequently used for model checking purposes.
2. We increase the power of existing algorithms' mining two-event linear temporal logic (LTL) rules/properties to mining properties of arbitrary lengths.
3. We propose a mining-maintenance framework composed of instrumentation, trace collection and abstraction, rule mining, post-processing, visualization and model checking.
4. We show the utility of our technique in recovering specifications of a large industrial program.
5. We demonstrate the usefulness of mined LTL rules/properties to reveal bugs from a buggy CVS application (see Section 8.2).

The outline of this paper is as follows. Section 2 contains important background information on LTL formalizing our definition of temporal rules. Section 3 presents the principles behind mining temporal rules and the pruning strategies employed. Section 4 presents our algorithm. Section 5 describes our end-to-end mining-maintenance framework. Section 6 describes our performance study on data mining benchmark data sets. Section 7 describes our case studies. Section 8 discusses related work. Section 9 concludes this paper and presents the directions for future work.

2. PRELIMINARIES

This section introduces preliminaries on LTL and its verification, which dictate the semantics of temporal rules. Notations used in this paper are also presented here.

2.1. Linear-time temporal logic

Our mined rules can be expressed in LTL [11]. LTL is a logic that works on possible program paths. A possible program path corresponds to a program trace. A path can be considered as a series of events, where an event is a method invocation. For example, (file_open, file_read, file_write, file_close) is a four-event path.

There are a number of LTL operators, among which we are interested only in the operators 'G','F' and 'X'[‡]. The operator 'G' specifies that *globally* at every point in time a certain property holds. The operator 'F' specifies that a property holds either at that point in time or *finally (eventually)* it holds. The operator 'X' specifies that a property holds at the *next* event. Some examples are listed in Table I.

There are other LTL operators, 'U' (until), 'W' (weak until) and 'R' (release). We leave mining rules expressing other LTL operators for future work. Our purpose is to mine a sub-set of commonly used temporal properties for verification referred to as response and chain response in [8]. As we see in Section 8.2, this limitation can cause one to miss mining complicated bug-revealing property.

Our mined rules state that whenever a series of precedent events occurs eventually another series of consequent events also occurs. A mined rule denoted as $pre \rightarrow post$ can be mapped to its corresponding LTL expression. Examples of such correspondences are given in Table II. Note that although the operator 'X' might seem redundant, it is needed to specify rules such as $\langle a\rangle \rightarrow \langle b, b\rangle$ where the 'b's refer to *different occurrences of* 'b'. The set of LTL expressions minable by our mining framework is represented in the Backus–Naur Form (BNF) as follows:

$$\boxed{\begin{array}{ll} rules := & G(prepost) \\ prepost := & event \rightarrow post | event \rightarrow XG(prepost) \\ post := & XF(event) | XF(event \wedge XF(post)) \end{array}}$$

2.2. Checking/verifying LTL expressions

LTL expressions are mainly used for checking software systems expressed in the form of an automaton [12] (a transition system with start and end nodes). There are existing tools converting code to an automaton. Given an automaton and an LTL property one can check whether the automaton satisfies the LTL property through a well-known technique of model checking [2].

Consider the example in Figure 1, the pseudo-code on the left corresponds to the automaton on the right. Given the property $\langle main, lock\rangle \rightarrow \langle unlock, end\rangle$, a model checking tool (cf. [2]) will

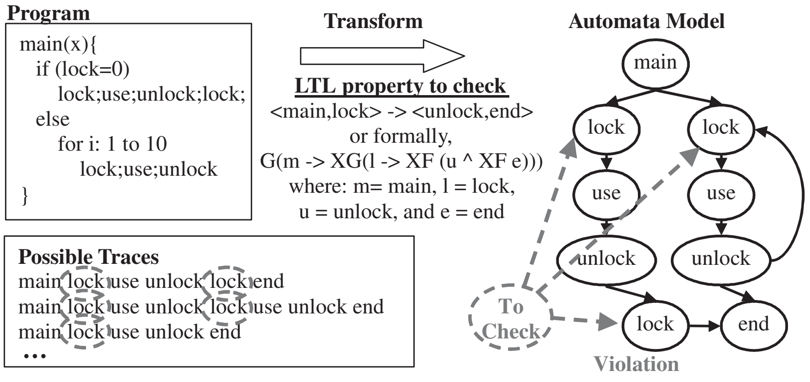
[‡]The operators G, F and X are often also represented by $\Box$, $\Diamond$ and $\circ$, respectively.

Table I. LTL expressions and their meanings.

$F(unlock)$	Eventually *unlock* is called
$XF(unlock)$	From the next event onwards, eventually *unlock* is called
$G(lock \to XF(unlock))$	Globally whenever *lock* is called, then from the next event onwards, eventually *unlock* is called
$G(main \to XG(lock \to (\to XF(unlock \to XF(end)))))$	Globally whenever *main* followed by *lock* are called, then from the next event onwards, eventually *unlock* followed by *end* are called

Table II. Rules and their LTL equivalences.

Notation	LTL notation
$a \to b$	$G(a \to XFb)$
$\langle a,b\rangle \to c$	$G(a \to XG(b \to XFc))$
$a \to \langle b,c\rangle$	$G(a \to XF(b \wedge XFc))$
$\langle a,b\rangle \to \langle c,d\rangle$	$G(a \to XG(b \to XF(c \wedge XFd)))$



Program

```
main(x){
  if (lock=0)
     lock;use;unlock;lock;
  else
     for i: 1 to 10
        lock;use;unlock
}
```

Possible Traces

```
main lock use unlock lock end
main lock use unlock lock use unlock end
main lock use unlock end
...
```

LTL property to check

<main,lock> -> <unlock,end>
or formally,
G(m -> XG(l -> XF (u ^ XF e)))
where: m= main, l = lock,
u = unlock, and e = end

Figure 1. Code –> automaton –> verification.

ensure that for all states in the model where *lock* preceded by a main occurs (marked by the red dashed arrows), eventually (whichever path is taken) *unlock and* then eventually *end* can be reached. For the above example, the property is violated. The *lock* immediately before *end* is not followed by an *unlock*. However, note that the property $\langle main, lock, use\rangle \to \langle unlock, end\rangle$ is satisfied. This is the case since the *lock* immediately before *end* is not followed by a *use*, i.e., the pre-condition of the rule is not satisfied and the rule vacuously holds.

An execution trace can be considered to be a finite path in the automaton and corresponds to a series of events. An event in turn corresponds to a behavior of interest, e.g., method call. A mined rule (or property) $pre \to post$ with a perfect confidence (i.e., confidence $= 1$) states that in the traces from all states where *pre* holds eventually *post* occurs. In the above example, for all points in the traces (i.e., temporal points) where $\langle main, lock\rangle$ occurs (marked with a dashed red circle), one needs to check whether eventually $\langle unlock, end\rangle$ occurs. Based on the definition of LTL properties and how they are verified, our technique analyzes traces and captures *strong* LTL expressions that satisfy given support and confidence thresholds.

Basic notations: Let I be a set of distinct events considered in which an event corresponds to a behavior of interest, e.g., method call. The input to our mining framework is a set of traces. A trace corresponds to a sequence or an ordered list of events from I. For formality, we refer to this set of traces as a sequence database denoted by *SeqDB*. Each trace or sequence is denoted by $\langle e_1, e_2, \ldots, e_{end}\rangle$ where $e_i \in I$.

We define a pattern P to be a series of events. We use $last(P)$ to denote the last event of P. A pattern $P_1 {+\!\!+} P_2$ denotes the concatenation of patterns P_1 and P_2. A pattern P_1 ($\langle e_1, e_2, \ldots, e_n\rangle$) is considered a *subsequence* of another pattern P_2 ($\langle f_1, f_2, \ldots, f_m\rangle$) denoted as $P_1 \sqsubseteq P_2$ if there exist integers $1 \leqslant i_1 < i_2 < \cdots < i_n \leqslant m$ such that $e_1 = f_{i_1}$, $e_2 = f_{i_2}, \ldots, e_n = f_{i_n}$.

3. GENERATION OF TEMPORAL RULES

Each temporal rule of interest has the form $P_1 \rightarrow P_2$, where P_1 and P_2 are two series of events. P_1 is referred to as the *premise* or *pre-condition* of the rule, while P_2 is referred to as the *consequent* or *post-condition* of the rule. The rules correspond to temporal constraints expressible in LTL notations. Some examples are given in Table II.

In this paper, as a trace is a series of events, where an event corresponds to a software behavior of interest, e.g., method call, *we formalize a trace as a sequence and a set of input traces as a sequence database*. We use the sample trace or sequence database in Table III as our running example to illustrate the concepts behind generation of temporal rules.

3.1. Concepts and definitions

Mined rules are formalized as LTL expressions with the format: $G(\cdots \rightarrow XF \cdots)$. The semantics of LTL and its verification technique described in Section 2 will dictate the semantics of temporal rules described here. Noting the meaning of the temporal operators illustrated in Table I, to be precise, a temporal rule expresses:

Whenever a series of events has just occurred at a point in time (i.e., a temporal point), *eventually another series of events occurs.*

From the above definition, to generate temporal rules, we need to 'peek' at interesting temporal points (to mine a set of pre-conditions) and 'see' what series of events are likely to occur next (to mine a set of post-conditions). We will first formalize the notion of temporal points and the related notion of occurrences.

Table III. Example database—*DBEX*.

Identifier	Trace/sequence
$S1$	⟨a, b, e, a, b, c⟩
$S2$	⟨a, c, b, e, a, e, b, c⟩
$S3$	⟨a, d⟩

Definition 1 (*Temporal points*). Consider a sequence S of the form $\langle a_1, a_2, \ldots, a_{end}\rangle$. All events in S are indexed by their positions in S, starting at 1 (e.g., a_j is indexed by j). These positions are called *temporal points in* S. For a temporal point j in S, the prefix $\langle a_1, \ldots, a_j\rangle$ is called the j-prefix of S.

Definition 2 (*Occurrences and instances*). Given a pattern P and a sequence S, the *occurrences* of P in S are defined by a set of temporal points $\mathscr{T}$ in S such that for each $j \in \mathscr{T}$, the j-prefix of S is a super-sequence of P and $last(P)$ is indexed by j. The set of *instances* of pattern P in S is defined as the set of j-prefixes of S, for each $j \in \mathscr{T}$.

Example. Consider a pattern P $\langle a, b\rangle$ and the sequence $S1$ in Table III (i.e., $\langle a, b, e, a, b, c\rangle$). The *occurrences* of P in $S1$ form the set of temporal points {2,5}, and the corresponding set of *instances* is $\{\langle a, b\rangle, \langle a, b, e, a, b\rangle\}$.

We define database projection operations to capture events occurring after specified temporal points. The following are two different types of projections and their associated support notions.

Definition 3 (*Projected and sup*). A database *projected* on a pattern p is defined as $SeqDB_P = \{(j, sx)|$ the jth sequence in *SeqDB* is s, where $s = px{+}{+}sx$, and *px is the minimum prefix of s containing p*$\}$.

Given a pattern P_X, we define *sup*$(P_X, SeqDB)$ to be the size of $SeqDB_{P_X}$ (equivalently, the number of sequences in *SeqDB* containing P_X). Reference to the database is omitted, i.e., we write it as *sup*(P_X), if the database is clear from the context, e.g., it refers to input sequence database *SeqDB*.

Definition 4 (*Projected-all and sup-all*). A database *projected-all* on a pattern p is defined as $SeqDB^{all}_P = \{(j, sx)|$ the jth sequence in *SeqDB* is s, where $s = px{+}{+}sx$, and *px is an instance of p* in s and $\text{last}(px) = \text{last}(p)\}$.

Given a pattern P_X, we define $sup^{all}(P_X, SeqDB)$ to be the size of $SeqDB^{all}_{P_X}$. Reference to the database is omitted if it is clear from the context.

Definition 3 defines a standard database projection (*c.f.* [13,14]) capturing events occurring after the *first temporal point*. Definition 4 is a new type of projection to capture events occurring after *each temporal point*.

Example. To illustrate the above concepts, we project and project-all the example database *DBEX* with respect to the pattern $\langle a, b\rangle$. The results are given in Table IV(a) and (b), respectively.

The two projection methods' associated notions of *sup* and sup^{all} are different. Specifically, sup^{all} reflects the number of occurrences of P_X in *SeqDB* rather than the number of sequences in *SeqDB* supporting P_X.

Example. Consider the example database, $sup(\langle a, b\rangle, DBEX) = |DBEX_{\langle a,b\rangle}| = 2$. On the other hand, $sup^{all}(\langle a, b\rangle, DBEX) = |DBEX^{all}_{\langle a,b\rangle}| = 4$.

From the above notions of temporal points, projected databases and pattern supports, we can define the support and confidence of temporal rules.

Table IV. (a) $DBEX_{\langle a,b\rangle}$ and (b) $DBEX^{all}_{\langle a,b\rangle}$.

Identifier	Trace/sequence
(a)	
$S1$	(1,⟨e, a, b, c⟩)
$S2$	(2,⟨e, a, e, b, c⟩)
(b)	
$S1_1$	(1,⟨e, a, b, c⟩)
$S1_2$	(1,⟨c⟩)
$S2_1$	(2,⟨e, a, e, b, c⟩)
$S2_2$	(2,⟨c⟩)

Definition 5 (*Support and confidence*). Consider a temporal rule R_X ($pre_X \rightarrow post_X$). The *support* of R_X is defined as the number of sequences in *SeqDB* where pre_X occurs, which is equivalent to $sup(pre_X, SeqDB)$. The confidence of R_X is defined as the likelihood of $post_X$ occurring after pre_X. This is equivalent to the ratio of $sup(post_X, SeqDB^{all}_{pre_X})$ to the size of $SeqDB^{all}_{pre_X}$.

Example. Consider $DBEX$ and a temporal rule R_X, $\langle a,b\rangle \rightarrow \langle c\rangle$. From the database, the support of R_X is the number of sequences in $DBEX$ supporting (or is a super-sequence of) the rule's pre-condition—$\langle a,b\rangle$. There are two of them; see Table IV(a). Hence, the support of R_X is 2. The confidence of the rule R_X ($\langle a,b\rangle \rightarrow \langle c\rangle$) is the likelihood of $\langle c\rangle$ occurring after each *temporal point* of $\langle a,b\rangle$. Referring to Table IV(b), we see that there is a $\langle c\rangle$ occurring after each temporal point of $\langle a,b\rangle$. Hence, the confidence of R_X is 1.

Significant rules to be mined must have their supports greater than the *min_sup* threshold, *and* their confidences greater than the *min_conf* threshold.

In mining program properties, the confidence of a rule (or property), which is a measure of its certainty, matters the most (cf. [7]). Support values are considered to differentiate high confidence rules from one another according to the frequency of their occurrences in the traces. Rules with confidences less than 100% are also of interest due to the imperfect trace collection and the presence of bugs and anomalies [7]. Similar to the assumption made by the work in statistical debugging (e.g., [15]), simply put, if a program behaves in one way 99% of the time, and the opposite 1% of the time, the latter likely corresponds to a possible bug. Hence, a high confidence and highly supported rule is a good candidate for bug detection using program verifiers.

We added the notions of support and confidence to the temporal rules. The formal notation of temporal rules is defined below.

Definition 6 (*Temporal rules*). A temporal rule R_X is denoted by $pre \rightarrow post$ (sup,$conf$). The series of events *pre* and *post* represent the rule's pre- and post-condition and are denoted by $R_X.Pre$ and $R_X.Post$, respectively. The notions *sup* and *conf* represent the support and confidence of R_X, respectively. They are denoted by $sup(R_X)$ and $conf(R_X)$, respectively.

Example. Consider $DBEX$ and the rule R_X, $\langle a,b\rangle \rightarrow \langle c\rangle$ shown in the previous example. It has a support of 2 and a confidence of 1. It is denoted by $\langle a,b\rangle \rightarrow \langle c\rangle(2,1)$.

3.2. Monotonicity properties and non-redundancy

Our algorithm is a new addition to the family of pattern mining algorithms, e.g., [13,14,16,17]. Monotonicity (a.k.a. *a priori*) properties have been widely used to ensure efficiency of many pattern mining techniques (e.g., [16,17]). One of the novelty in our new mining algorithm is the identification of new and suitable *a priori* properties that apply. Fortunately, temporal rules obey the following *a priori* properties:

Theorem 1 (Monotonicityproperty—support). *If a rule* $evs_P \rightarrow evs_C$ *does not satisfy the min_sup threshold, neither will all rules* $evs_Q \rightarrow evs_C$, *where* evs_Q *is a super-sequence of* evs_P.

Theorem 2 (Monotonicityproperty—confidence). *If a rule* $evs_P \rightarrow evs_C$ *does not satisfy the min_conf threshold, neither will all rules* $evs_P \rightarrow evs_D$, *where* evs_D *is a super-sequence of* evs_C.

To reduce the number of rules and improve efficiency, we define a notion of rule redundancy defined based on the *super-sequence relationship* among rules having the same support and confidence values. This is similar to the notion of *closed* patterns applied to sequential patterns [13,14].

Definition 7 (*Rule redundancy*). A rule R_X ($pre_X \rightarrow post_X$) is redundant if there is another rule R_Y ($pre_Y \rightarrow post_Y$) where

(1) R_X is a sub-sequence of R_Y (*i.e.*, $pre_X{+\!+}post_X \sqsubseteq pre_Y{+\!+}post_Y$).
(2) Both rules have the same support and confidence values

In addition, in the case where the concatenations are the same (i.e., $pre_X{+\!+}post_X = pre_Y{+\!+}post_Y$), to break the tie, we call the one with the longer premise as being redundant (i.e., we wish to retain the rule with a shorter premise and longer consequent).

To illustrate redundant rules, consider the following set of rules describing an automated teller machine:

R1	*accept_card* → *enter_pin,display_goodbye,eject_card*
R2	*accept_card* → *enter_pin*
R3	*accept_card* → *display_goodbye*
R4	*accept_card* → *enter_pin*, *eject_card*
R5	*accept_card* → *display_goodbye*, *eject_card*

If all of the above rules have the same support and confidence values, rules R2–R5 are redundant as they are represented by rule R1. To keep the number of mined rules manageable, we remove redundant rules. Noting the combinatorial nature of redundant rules, removing redundant rules can drastically reduces the number of reported rules.

A simple approach to reduce the number of rules is to first mine a full set of rules and then remove redundant rules. However, this 'late' removal of redundant rules is inefficient due to the exponential explosion of the number of intermediary rules that need to be checked for redundancy. To improve efficiency, it is therefore necessary to identify and prune a search space containing redundant rules 'early' during the mining process. The following two theorems are used for 'early' pruning of redundant rules. The proofs are available in our technical report [18].

Theorem 3 (Pruning redundant pre-conds). *Given two pre-conditions P_X and P_Y where $P_X \sqsubset P_Y$, if $SeqDB_{P_X} = SeqDB_{P_Y}$, then for all sequences of events post, the rule $P_X \rightarrow post$ is rendered* redundant *by $P_Y \rightarrow post$ and can be pruned.*

Theorem 4 (Pruning redundant post-conds). *Given two rules R_X ($pre \rightarrow P_X$) and R_Y ($pre \rightarrow P_Y$) if $P_X \sqsubset P_Y$ and $(SeqDB^{all}_{pre})_{P_X} = (SeqDB^{all}_{pre})_{P_Y}$, then R_X is rendered* redundant *by R_Y and can be pruned.*

Using Theorems 3 and 4, many redundant rules can be pruned ‘early’. However, the theorems only provide sufficient conditions for the identification of certain redundant rules—there are other redundant rules that are not identified by them. To remove the remaining redundant rules, we perform a post-mining filtering step based on Definition 7.

Our approach to mining a set of non-redundant rules satisfying the support and confidence thresholds is as follows:

Step 1: Leveraging Theorems 1 and 3, we generate a *pruned* set of pre-conditions satisfying *min_sup*.

Step 2: For each pre-condition *pre*, we create a *projected-all* database $SeqDB^{all}_{pre}$.

Step 3: Leveraging Theorems 2 and 4, for each $SeqDB^{all}_{pre}$, we generate a *pruned* set containing such a post-condition *post*, such that the rule $pre \rightarrow post$ satisfies *min_conf*.

Step 4: Using Definition 7, we filter any remaining redundant rules.

In the following section, we describe our algorithm in detail.

4. MINING ALGORITHM

In the previous section, the process of mining non-redundant rules has been divided into four steps. Steps 1 and 3 sketch how a pruned set of pre- and post-conditions is mined. The following paragraphs will elaborate them in more detail.

Before proceeding, we first describe a set of patterns called *projected database closed* (or LS-Set) first mentioned in [13]. A pattern is in the set if *there does not exist any super-sequence pattern having the same projected database*. Patterns having the same projected database must have the same support, but patterns with the same support can have different projected databases. *Projected database closed* patterns are of special interest to us, as explained in the following paragraphs.

At step 1, a pruned set of pre-conditions is generated from the input database *SeqDB*. From Theorem 3, a pattern is in the pruned pre-condition set if *there does not exist any super-sequence pattern having the same projected database*. Comparing with the definition of *projected database closed* patterns in the previous paragraph, we note that this pruned set of pre-conditions corresponds to the *projected database closed set* (or LS-Set) mined from *SeqDB*.

At step 3, starting with a projected-all database $SeqDB^{all}_{pre}$, we generate a pruned set of post-conditions. From Theorem 4, a pattern is in the pruned post-condition set if *there does not exist any super-sequence pattern having the same projected database*. Again, this set of pruned post-conditions corresponds to the *projected database closed set* (or LS-Set) mined from $SeqDB^{all}_{pre}$.

Our mining algorithm is shown in Figure 2. First, a pruned set of pre-conditions satisfying the minimum support threshold (i.e., *min_sup*) is mined using an LS-Set miner modified from

```
Procedure Mine Non-Redundant Temporal Rules
Inputs: $SeqDB$ : Set of input traces represented as a sequence database;
$min\_sup$ : Min. Support Thresh.; $min\_conf$ : Min. Conf. Thresh.
Output:
$Rules$: Non Redundant Set of Temporal Rules
Method:
1: Let $PreCond$ = Generate an LS-Set from $SeqDB$ with the threshold
                   set at $min\_sup$
2: For every $pre$ in $PreCond$
3:    Let $SeqDB^{all}_{pre}$ = $SeqDB$ projected-all by pattern $pre$
4:    Let $bthd = min\_conf \times |SeqDB^{all}_{pre}|$
5:    Let $PostCond$ = Generate an LS-Set from $SeqDB^{all}_{pre}$ with the
                       threshold set at $bthd$
6:    For every $post$ in $PostCond$
7:       Add ($pre \rightarrow post$) to $Rules$
8: For every $rx$ in $Rules$
9:    If ($rx$ is redundant according to Def. 7)
10:      Remove $rx$ from $Rules$
11: Output $Rules$
```

Figure 2. Mining algorithm.

BIDE [14], the state-of-the-art closed sequential pattern miner[§]. Next, for each pre-condition mined, a database projected-all on it is formed. Consequently, another LS-Set generator is run on each projected-all database to mine the set of post-conditions of the corresponding candidate rules having enough confidence values. Finally, a filtering step to remove any remaining redundant rules based on Definition 7 is performed. To perform the final filtering step scalably, each remaining rule is first hashed based on its support and confidence values. Only rules falling into the same hash bucket need to be checked for super-sequence relationship.

[§]The details are available in our technical report [18].

The algorithm can be adapted easily to generate a full set of temporal rules satisfying *min_sup* and *min_conf* thresholds. This is performed to serve as a point of reference for investigating the benefit of early identification and pruning of redundant rules. To generate the full set we can simply (1) generate a full set of pre- and post-conditions of rules satisfying the support and confidence thresholds at lines 1 and 5 of the algorithm, respectively (we use PrefixSpan [19] for this purpose) and (2) skip the final redundancy filtering step (i.e., lines 8–10 of the algorithm in Figure 2).

5. MINING-MAINTENANCE FRAMEWORK

Section 4 discusses how to convert a set of traces to a statistically complete and sound set of non-redundant rules. However, how do we generate traces? Also, how to integrate the mining algorithm in an end-to-end framework starting from a program and ending in visualization of mined properties and/or program verification via model checking? To address the above questions we propose a mining framework that comprises four parts:

Part 1: Program instrumentation.
Part 2: Trace generation and abstraction to sequences of symbols.
Part 3: Execution of the mining algorithm.
Part 4: Presentation of mined rules, post-processing and model checking.

Our proposed mining framework is outlined diagrammatically in Figure 3. At the start of a mining task, four inputs are provided: a program (in source code, byte code or binary) to analyze, a test suite, a set of thresholds and an abstract model of the corresponding system to verify. The abstract model can be generated automatically from the code (cf. [20]) or provided by the user. These inputs

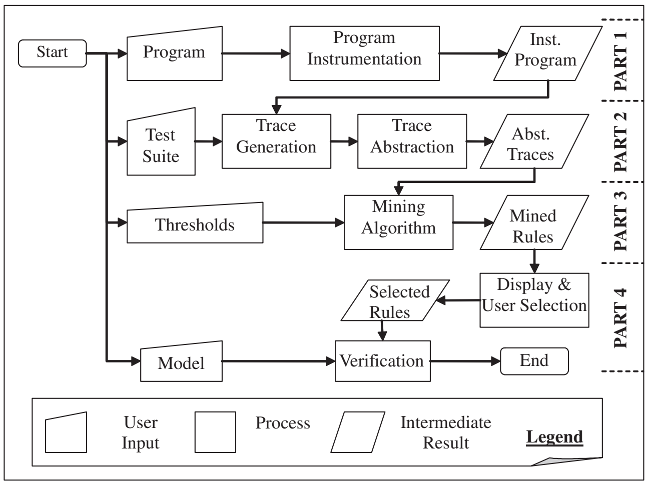


Figure 3. Mining framework.

will be fed to various parts of the mining framework resulting in a set of mined rules ready for presentation to the user and for inputs to a model checker.

The input program needs to be instrumented. Instrumentation inserts ‘print’ statements at the entry points of various function definitions. When the instrumented program is run, a trace will be generated. A trace is a series of signatures of the methods being invoked. Various instrumentation strategies can be employed including aspects, byte code manipulation [21] and binary editing [22].

Much open-source and industrial software comes with a test suite. Running the instrumented program over a test suite will produce a set of traces. Each trace corresponds to a series of method invocations. The set of raw traces can then be abstracted to a set of sequences of symbols by mapping each method signature to a unique symbol. This set of sequences of symbols is the input to our mining algorithm.

The limitation with generation of traces by running a set of test cases is the dependence on the quality of test cases. One needs to balance the running of both unit and system tests. With unit tests alone, running these tests may produce smaller traces and may not be representative of true execution scenarios.

At the end of the mining process, a set of rules will be obtained. These rules can be sorted according to their support and confidence. The rules are then presented to the user to help the user understand the temporal properties and implicit rules of the software system under analysis. Selected rules can be input to a model checker for bug finding and verification of a system.

The mining process can be applied periodically along with the various maintenance tasks performed during the software lifespan. Mined rules can help developers understand program behaviors, update obsolete specifications and verify a program via model checking.

6. PERFORMANCE EVALUATION

Experiments have been performed on both synthetic and real data sets to evaluate the *scalability of our mining framework* on standard data mining benchmark data sets. Low support threshold similar to the range considered in [13,14,23] is utilized to test for scalability. Our algorithms are the *first* algorithms mining multi-event temporal rules; hence, we compare and contrast the runtime required when full and non-redundant sets of temporal rules are mined to evaluate the *effectiveness of our non-redundant rule pruning strategies* (i.e., Theorems 3 and 4).

We use two data sets in our experiments: one synthetic and another real. Synthetic data generator provided by IBM (cf. [17]) was used with modification to generate synthetic traces. We produce a synthetic data set by running the IBM synthetic data generator with the following parameter setting: D (no. of sequences—in 1000 s) $=5$, C (average sequence length) $=20$, N (no. of unique events—in 1000 s) $=10$ and S (average no. of events in maximal sequences) $=20$. We also experimented on a click stream data set (i.e., Gazelle data set) from KDD Cup 2000 [24]. It contains 23 639 sequences with an average length of 3 and a maximum length of 651. Both the synthetic data generator and Gazelle data set have been standard benchmarks used in pattern mining research [13,14,17,23].

The Gazelle data set, based on KDD Cup 2000, is not evenly distributed—the maximum length is 651 and the average length is 3. Although the average length is 3, it contains almost 30 000 sequences. At high support threshold the data set is relatively easy to mine. However, at very low support thresholds, as considered in this study (see [13,14]), the data set is hard to mine as the rules are long and the number of significant rules is many.

All experiments were performed on a Pentium M 1.6 GHz tablet PC with 1.5 GB memory, running Windows XP Tablet PC Edition 2005. Algorithms were written using C#.Net.

Experiments were performed by varying *min_sup* and *min_conf* thresholds. The *min_sup* value is represented as a percentage ratio to the number of sequences in the database. The results are plotted as line graphs. ‘Full’ and ‘NR’ correspond to the full set and non-redundant set of rules, respectively. The x-axis of each graph corresponds to one of the thresholds considered, whereas the y-axis represents either the algorithm runtime or the number of mined rules.

The experiment results for mining rules from the synthetic data set are shown in Figure 4(A)–(D). The experiment results for the Gazelle data set are shown in Figure 4(E) and (F).

From the two experiments we note that the runtime is significantly increased when the *min_sup* threshold is lowered. On the other hand, lowering the *min_conf* threshold has lesser effect on the runtime. The results also show that we can efficiently mine temporal rules from standard data mining benchmark data sets even at a low *min_sup* threshold. The lower the threshold, the more difficult it is to mine the rules. We did not experiment with low *min_conf* thresholds as we believe that the usefulness of low confidence rules (if any) is minimal.

Mining non-redundant rules rather than a full set of rules reduced the runtimes and the number of rules by up to more than *28 times less* and up to *8500 times less*, respectively. Admittedly, the pruning strategies themselves require some computation cost; hence, for cases where the benefit

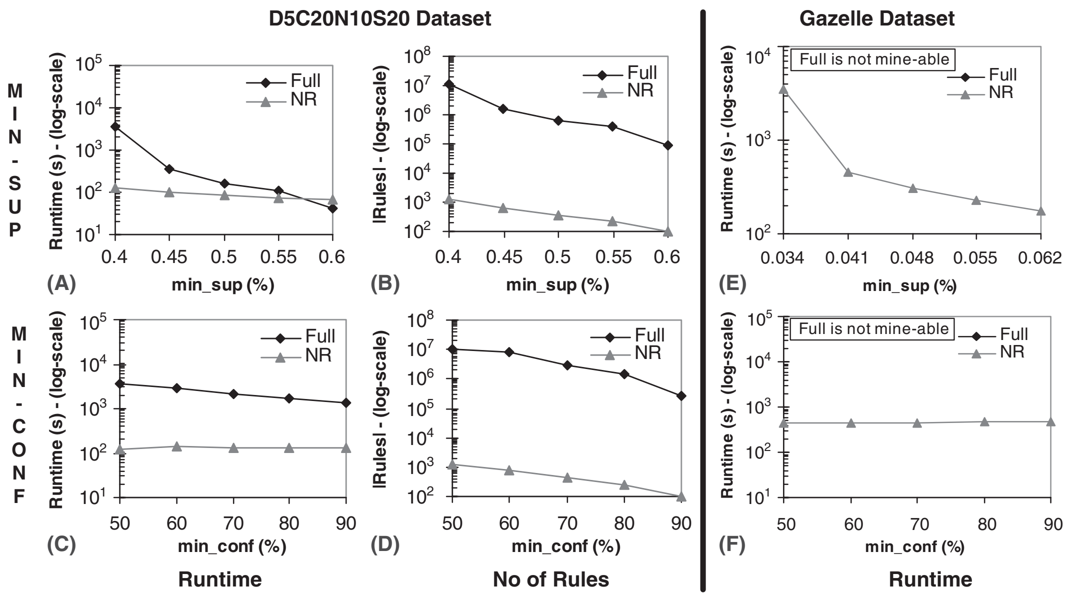


Figure 4. Varying *min_sup* at *min_conf* = 50% (A) and (B) and *min_conf* at *min_sup* = 0.4% (C) and (D) for D5C20N10S20 data set. Varying *min_sup* at *min_conf* = 50% (E) and *min_conf* at *min_sup* = 0.041% (F) for Gazelle data set.

of the pruning strategies is less, mining a full set of rules might be slightly faster than the non-redundant set. However, the desired result is the non-redundant set of rules. The full set of rules contains too many redundant rules—up to more than 10 million rules were produced! In addition, for experiments with the real-life benchmark data set Gazelle, the full set of rules is not mine-able even at the highest *min_sup* threshold shown in Figure 4(E)—our attempt produced a gigantic 51 GB file before we had to stop the process after it runs for many hours.

In the case study section (i.e., Section 7) we analyze traces from real software. In this section, we analyze traces obtained from related works on sequences in the field of data mining. These traces are generated by the IBM data generator and Gazelle data set. These traces have customarily been used to test the scalability of data-mining algorithms, such as ours. Furthermore, since the data generators produce random sequences, it helps to ensure that the algorithm is not biased or tuned to work on only particular software or group of software under analysis.

7. CASE STUDIES

In this section we discuss our case studies on two different systems: JBoss-AS and a buggy CVS application. The first study shows the utility of our method in recovering specifications of a large industrial system. The second study demonstrates the usefulness of mined LTL rules/properties to reveal bugs from a buggy application. A discussion on additional strategies to improve the scalability of our approach is also presented in the end of this section.

7.1. JBoss-AS

JBoss-AS is the most widely used J2EE application server. It contains over 100 000 lines of code and comments. The purpose of this study is to show the usefulness of the mined rules to describe the behavior of a real software system.

Case 1: *JBoss-AS security component*. We instrumented the security component of JBoss-AS using JBoss-AOP and generated traces by running the test suite that comes with the JBoss-AS distribution. In particular, we ran the regression tests on enterprise Java bean (EJB) security implementation of JBoss-AS. Twenty-three traces of a total size of 4115 events, with 60 unique events, were generated. Running the algorithm with the minimum support and confidence thresholds set at 15 and 90%, respectively, six non-redundant rules were mined. The algorithm completed within 3 s.

A sample of the mined rules is shown in Figure 5 (left). It describes authentication using Java authentication and authorization service for EJB within JBoss-AS. When authentication scenario starts, first configuration information is checked to determine authentication service availability—this is described by the premise of the rule. This is followed by invocations of actual authentication events, binding of principal information to the subject being authenticated and utilizations of subject's principal and credential information in performing further actions—these are described by the consequent of the rule. The meaning of the rule has been checked against JBoss documentation.

Case 2: *JBoss-AS transaction component*. We instrumented the transaction component of JBoss-AS using JBoss-AOP and generated traces by running the test suite that comes with the JBoss-AS distribution. In particular, we ran a set of transaction manager regression tests of JBoss-AS. Each

Premise	Consequent
XmlLoginCI.getConfEntry()	ClientLoginMod.initialize()
AuthenInfo.getName()	ClientLoginMod.login()
	ClientLoginMod.commit()
	SecAssocActs.setPrincipalInfo()
	SetPrincipalInfoAction.run()
	SecAssocActs.pushSubjectCtxt()
	SubjectThreadLocalStack.push()
	SimplePrincipal.toString()
	SecAssoc.getPrincipal()
	SecAssoc.getCredential()
	SecAssoc.getPrincipal()
	SecAssoc.getCredential()

Premise	Consequent
TxManLoc.getInstance()	TransImpl.instanceDone()
TxManLoc.locate()	TxManager.getInstance()
TxManLoc.tryJNDI()	TxManager.releaseTransImpl()
TxManLoc.usePrivateAPI()	TransImpl.getLocalId()
TxManager.getInstance()	XidImpl.getLocalId()
TxManager.begin()	LocalId.hashCode()
XidFactory.newXid()	LocalId.equals()
XidFactory.getNextId()	TransImpl.unlock()
XidImpl.getTrulyGlobalId()	XidImpl.hashCode()
LocalId.assocCurThread()	
TransactionImpl.lock()	

Figure 5. A sample rule from JBoss security (left) and another from JBoss transaction (right). Each of the rules is read from top to bottom, left to right.

trace is abstracted as a sequence of events, where an event corresponds to a method invocation. Twenty-eight traces with a total size of 2551 events containing 64 unique events were generated. Running the algorithm on the abstracted traces with the minimum support and confidence thresholds set at 25 traces and 90%, respectively, 182 non-redundant rules were mined. The algorithm completed within 30 s.

In the presentation of mined rules, we display first rules in which their constituent events rarely repeat and sort them according to their support and confidence values. These help to distinguish more interesting rules from the others.

A sample of the mined rules is shown in Figure 5 (right). The 20-event rule in Figure 5 (right) describes that the series of events ⟨connection to a server instance events, transaction manager and implementation set up event⟩ (events 1–11) at the start of a transaction is always followed by the series of events ⟨transaction completion events and resource release events⟩ (events 12–20) at the end of the transaction. The above rule describing the temporal relationship and constraint between the 20 events is difficult to identify manually. The rule sheds light on the *implementation details* of JBoss-AS, which are implemented at various locations in (i.e., crosscuts) the JBoss-AS transaction code base. It corresponds to the code in TransactionManagerLocator, TransactionManager, TransactionImpl, etc.

Discussion: For many applications, data mining is an iterative process; there are no hard-and-fast rules as to the best thresholds to be used. Similarly, our rule mining tool is used in an iterative manner. One can start at a reasonably high support and confidence thresholds and iteratively reduce the thresholds.

Some mined rules are shorter (e.g., 8) whereas others are longer (e.g., 56). In our result, we saw a number of similar rules due to small minor variations. Note that, by definition, employing rule redundancy removal removes only those rules with the same support and confidence. In the future, we shall look into ways to group these rules with minor variations together, or introduce new weaker notions of rule redundancy. We leave a detailed study on the effect of degree of incompleteness of trace and result of mined specification for future work. There are various studies that work on ensuring coverage of tests or automated test generation. In this paper, we give the guarantee of statistical soundness and completeness given input trace and thresholds.

Note that the time taken for mining is *much improved* with search space pruning strategies. Without employing any search space pruning strategy (i.e., if an approach similar to that in [7,25] to mine two-event rules is employed), the mining process will require E^L check operations, where E is the number of unique events and L is the maximum length of the trace. For traces from JBoss-AS considered, the mining process will require more than 50^{100} operations. Considering 1 ps per operation, it will only complete *in about* 2.501×10^{148} *centuries*! This highlights the power and importance of search space pruning strategies in improving the scalability of the mining process.

7.2. Buggy CVS application

This section describes a case study conducted on a buggy CVS application adapted from the one studied in [6,26]. This case study shows the usefulness of mined rules for model checking and bug detection.

7.2.1. CVS scenarios

The CVS application is built on top of the FTP library provided by Jakarta Commons Net [27]. Jakarta Commons Net is a set of reusable open-source java code implementing the client side of many commonly used network protocols. The CVS application can be considered as a client of Jakarta Commons Net, and it follows a certain protocol described by a set of scenarios.

There are eight common FTP interaction scenarios in our CVS implementation: Initialization and re-initialization of a repository, multiple-file upload, download, rename and deletion, and multiple-directory creation and deletion. All scenarios begin by connecting and logging-in to the FTP server. They end by logging-off and disconnecting from the FTP server. The CVS interaction protocol can be represented as a 33-state automata partially drawn in Figure 6 (see [6,26] for a more complete diagram). We focus on the following two scenarios:

Multiple-file upload scenario: One can store one or more files. For each file, the following is performed. First, the type of the file to be transferred is set. If the file is new, store the file directly and append the new file information to the *CVS system file* in the server. Otherwise, rename the old file by adding to the filename the time it is replaced and proceed to store the new file—the *CVS system file* need not be updated.

Multiple-file deletion scenario: One can delete one or more files. For each file, first go to its directory. List all files in the current working directory. Delete the file and all its previous versions. If the files are not located in the same directory, change the working directory accordingly. Finally, record file deletion information by appending it to the *CVS system file*.

7.2.2. Bug description and trace generation

Each invocation of a method of `FTPClient` may raise exceptions, especially `FTPConnectionClosedException` and `IOException`. Hence, the code accessing `FTPClient` methods need to be enclosed in a `try..catch.. finally` block. Every time such an exception occurs the program simply logs out and disconnects from the FTP server. These are represented by adding *error transitions* (shown as dashed lines) to the automata as shown in Figure 7.

The above exceptions might cause a number of bugs, but we focus only on 4 bugs causing *the system log file to be in an inconsistent state*. The system file describes the state of the CVS repository and should be kept consistent with the stored files. Bugs of this type are illustrated by

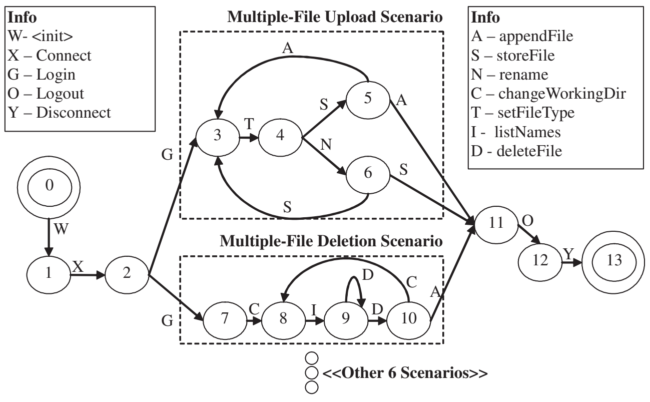


Figure 6. CVS protocol.

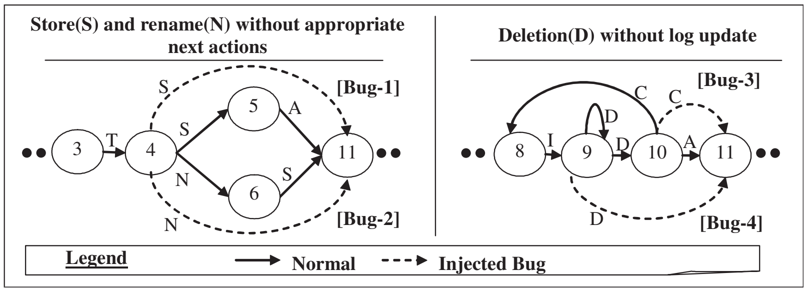

Store(S) and rename(N) without appropriate next actions
[Bug-1]
[Bug-2]
Deletion(D) without log update
[Bug-3]
[Bug-4]
Legend
Normal
Injected Bug

Figure 7. Injected bug.

the error transitions shown in Figure 7. Because of the bugs, a file can be added or deleted without a proper log entry being made. An old version of a file can be renamed by appending a time-stamp without the new version being stored in the CVS. These bugs occur because the CVS scenarios are not executed atomically.

To generate traces, we followed the process discussed in [6]. First, we instrumented the CVS application byte code using an adapted version of Java runtime analysis toolkit [21]. Next, we ran the instrumented CVS application over a set of test cases to generate traces. Via a trace post-processing step, we then filtered events in the traces not corresponding to the interactions between the CVS application and the Jakarta Commons Net FTP library. Thirty-six traces of a total size of 416 events were generated.

7.2.3. Mined rules and model checking results

We ran our mining algorithm on the generated traces. It completed within 1 s and mined five rules with minimum support and confidence thresholds set at 15 traces and 90%, respectively. Among the mined rules, the following two rules are the bug-revealing program properties:

1. Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, the file type is set (T) and an old file is renamed (N), *then eventually* a new file is stored (S), followed by a logout (O) and a disconnection from the server (Y). This is *denoted as* $\langle W, X, G, T, N\rangle \rightarrow \langle S, O, Y\rangle$.
2. Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, a working directory is set (C), *then eventually* the files in the directory is listed (I), a file is deleted (D), a log entry is made (A), followed by a logout (O) and a disconnection from the server (Y). This is *denoted as* $\langle W, X, G, C\rangle \rightarrow \langle I, D, A, O, Y\rangle$.

We used a model checker outlined in [28]. We drafted an abstract model of the buggy CVS system from the code manually in the format accepted by the model checker and checked against the above two properties. Alternatively, one can try to use some tools that automate model generation from the code. As the focus of this case study is on mining bug-revealing properties, we leave this for future work. The model checker reported violations of the above properties. These violations correspond to 3 out of the 4 bugs in the model. Bug-2 violates the first property. Bug-3 and Bug-4 violate the second property.

It is interesting to note that no two-event rules can be used in detecting these bugs, as invocations of FTP file delete (D), rename (R) and store (S) follow a different protocol in different scenarios (e.g., 'D' must be followed by 'A' in file deletion but not in the repository re-initialization scenario). Multi-event rules provide more precise contexts for identifying the scenarios where specified events occur, thus enabling the detection of the bugs.

One of the bugs (Bug-1) cannot be revealed because the required bug-revealing property is outside the bound of the LTL expressions mineable by our algorithm. This bug-revealing property is whenever the application is initialized (W), the connection (X) and login (G) to the server are made, a file type is set (T), *and there is no rename* (N) *until a new file is stored* (S), then eventually a log entry is made (A), followed by a logout (O) and a disconnection from the server (Y). Note that the phrase in italics is not expressible by any of our mined rules whose format (in BNF) is described in Section 2. To mine such properties, we need to mine rules containing the LTL operators not($\neg$) and until (U)—this is a possible future work.

8. RELATED WORK

Ernst *et al.* propose an interesting work called Daikon that discovers value-based program invariants occurring at a certain program point by analyzing program execution traces [29]. These value-based invariants are usually in the form of algebraic equations or boolean expressions (e.g., $X>Y$, $Z<Y$, etc.). Different from Daikon, in this work we focus on mining temporal properties, capturing ordering among events.

Yang *et al.* [7] present an interesting work on mining two-event temporal logic rules (i.e., of the form $G(a \rightarrow XF(b))$, where G, X and F are LTL operators [11]), which are statistically significant with respect to a user-defined 'satisfaction rate'. The algorithm presented, however, does not scale to mine multi-event rules of arbitrary length. To handle longer rules, Yang *et al.* suggest a partial solution based on the concatenation of mined two-event rules. Yet, the method proposed might miss some multi-event rules or introduce superfluous rules that are not statistically significant—it is neither statistically sound nor complete. In contrast, we mine LTL rules of arbitrary sizes; scalability is accomplished by utilizing search space pruning strategies inspired from research in the data mining domain. Our method is also statistically sound and complete.

Many specification mining algorithms extract temporal specifications in the form of automata [3–6]. Different from the above, we mine a set of multi-event temporal logic rules. While an automaton expresses a global picture of software specification, mined LTL rules break this into smaller parts, each expressing program properties observed in the traces.

Technique wise, this work is a new addition to the family of pattern mining algorithms [13,14,16,17]. Different from other pattern mining algorithms, in this work we mine LTL rules that have a different semantics and require different mining strategies than previous approaches.

9. CONCLUSION AND FUTURE WORK

In this paper, a novel method to mine a *non-redundant* set of *statistically significant* rules *of arbitrary lengths* of the form: 'Whenever a series of events ES_{Pre} occurs, eventually another series of events ES_{Post} also occurs' is proposed. According to a survey in [8], these rules belong to two of the most frequently used families of temporal logic expressions for model checking. Our approach is *statistically sound and complete*, meaning that all mined rules are statistically significant and all statistically significant rules are mined or represented. The problems of a potentially exponential runtime cost and a huge number of reported rules have been effectively mitigated by employing search space pruning strategies and elimination of redundant rules. A case study on JBoss-AS shows the utility of our technique in recovering specifications of a large industrial program. Another case study on a buggy CVS application demonstrates the usefulness of our approach in mining bug-revealing properties for bug detection using a model checker.

As the future work, we shall investigate advanced techniques to reduce the size of input traces while retaining the quality of the specification mined. One can do so by throwing away non-important or less important events. Zaidman *et al.* [30] identify important key classes using a webmining algorithm. Hamou-Lhadj and Lethbridge [31] propose a technique to summarize the content of large traces. An approach similar to that in [30,31] can potentially be employed to identify and remove less important events from the traces. Kuhn and Greevy [32] partition a trace into different phases. We are investigating on the possibility of separately mining specifications for each phase of a program. We also plan to investigate additional pruning strategies to further improve the performance of our algorithm. Additionally, we plan to mine for more properties involving disjunction, negation and additional sub-sets of LTL.

REFERENCES

1. Canfora G, Cimitile A. Software maintenance. *Handbook of Software Engineering and Knowledge Engineering*. World Scientific: River Edge NJ, 2001; **1**:91–120.
2. Clarke E, Grumberg O, Peled D. *Model Checking*. MIT Press: Cambridge MA, 1999.
3. Ammons G, Bodik R, Larus JR. Mining specification. *Proceedings of Symposium on Principles of Programming Languages (POPL)*, 2002; 4–16.

4. Reiss SP, Renieris M. Encoding program executions. *Proceedings of International Conference on Software Engineering (ICSE)*, 2001; 221–230.
5. Cook JE, Wolf AL. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology* 1998; **7**(3):215–249.
6. Lo D, Khoo SC. SMArTIC: Toward building an accurate, robust and scalable specification miner. *Proceedings of International Symposium on the Foundations of Software Engineering (FSE)*, 2006; 265–275.
7. Yang J, Evans D, Bhardwaj D, Bhat T, Das M. Perracotta: Mining temporal API rules from imperfect traces. *Proceedings of International Conference on Software Engineering (ICSE)*, 2006; 282–291.
8. Dwyer M, Avrunin G, Corbett J. Patterns in property specifications for finite-state verification. *Proceedings of International Conference on Software Engineering (ICSE)*, 1999; 411–420.
9. Han J, Kamber M. *Data Mining Concepts and Techniques*. Morgan Kaufmann: Los Altos CA, 2006.
10. Ball T. The concept of dynamic analysis. *Proceedings of European Software Engineering Conference/International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 1999; 216–234.
11. Huth M, Ryan M. *Logic in Computer Science*. Cambridge University Press: Cambridge, 2004.
12. Hopcroft J, Motwani R, Ullman J. *Introduction to Automata Theory*, *Languages*, *and Computability*. Addison-Wesley: Reading MA, 2001.
13. Yan X, Han J, Afhar R. CloSpan: Mining closed sequential patterns in large data sets. *Proceedings of SIAM International Conference on Data Mining (SDM)*, 2003.
14. Wang J, Han J. BIDE: Efficient mining of frequent closed sequences. *Proceedings of International Conference on Data Engineering (ICDE)*, 2004; 79–90.
15. Engler D, Chen DY, Hallem S, Chou A, Chelf B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *Proceedings of Symposium on Operating System Principles (SOSP)*, 2001; 57–72.
16. Agrawal R, Srikant R. Fast algorithms for mining association rules. *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 1994; 487–499.
17. Agrawal R, Srikant R. Mining sequential patterns. *Proceedings of International Conference on Data Engineering (ICDE)*, 1995; 3–14.
18. Lo D, Khoo SC, Liu C. Mining recurrent rules from sequence database. *NUS SoC Technical Report TR 12/07*, 2007 [27 June 2008].
19. Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu MC. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. *Proceedings of International Conference on Data Engineering (ICDE)*, 2001; 215–224.
20. Corbett J, Dwyer M, Hatcliff J, Laubach S, Pasareanu CS, Robby, Zheng H. Bandera: Extracting finite-state models from Java source code. *Proceedings of International Conference on Software Engineering (ICSE)*, 2000; 439–448.
21. JRat. http://jrat.sourceforge.net/ [27 June 2008].
22. Larus J, Schnarr E. EEL: Machine-independent executable editing. *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1995; 291–300.
23. Lo D, Khoo SC, Liu C. Efficient mining of iterative patterns for software specification discovery. *Proceedings of SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2007; 460–469.
24. Kohavi R, Brodley C, Frasca B, Mason L, Zheng Z. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations* 2000; **2**:86–98.
25. Weimer W, Necula G. Mining temporal specifications for error detection. *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005; 461–476.
26. Lo D, Khoo SC. QUARK: Empirical assessment of automaton-based specification miners. *Proceedings of Working Conference on Reverse Engineering (WCRE)*, 2006; 51–60.
27. Apache. Jakarta commons/net. Available from: http://jakarta.apache.org/commons/net/ [27 June 2008].
28. Hinton A, Kwiatkowska M, Norman G, Parker D. PRISM: A tool for automatic verification of probabilistic systems. *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006; 441–444.
29. Ernst M, Cockrell J, Griswold W, Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 2001; **27**(2):99–123.
30. Zaidman A, Calders T, Demeyer S, Paredaens J. Applying web mining techniques to execution traces to support the program comprehension process. *Proceedings of European Conference on Software Maintenance and Reengineering (CSMR)*, 2005; 134–142.
31. Hamou-Lhadj A, Lethbridge T. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. *Proceedings of International Conference on Program Comprehension (ICPC)*, 2006; 181–190.
32. Kuhn A, Greevy O. Exploiting analogy between traces and signal processing. *Proceedings of International Conference on Software Maintenance (ICSM)*, 2006; 320–329.