THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, Singapore 117590

TRA5/08

Mining Past-Time Temporal Rules

David Lo, Siau-Cheng Khoo and Chao Liu

May 2008

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Mining Past-Time Temporal Rules

David Lo[†], Siau-Cheng Khoo[†], and Chao Liu[‡]

[†]Department of Computer Science, National University of Singapore,
[‡]Microsoft Research, Redmond
{dlo,khoosc}@comp.nus.edu.sg,chaoliu@microsoft.com

Abstract. Specification mining is a process of extracting specifications, often from program execution traces. These specifications can in turn be used to aid program understanding, monitoring and verification. There are a number of dynamic-analysis-based specification mining tools in the literature, however none so far extract past time temporal expressions in the form of rules stating: "whenever a series of events occur, previously another series of events happened before". Rules of this format are commonly found in practice and useful for various purposes. Most rule-based specification mining tools only mine future-time temporal expression. Many past-time temporal rules like "whenever a resource is used, it was allocated before" are asymmetric as the other direction does not holds. Hence, there is a need to mine past-time temporal rules.
In this paper, we describe an approach to mine significant rules of the above format occurring above a certain statistical thresholds from program execution traces. The approach start from a set of traces, each being a sequence of events (*i.e.*, method invocations) and resulting in a set of significant rules obeying minimum thresholds of support and confidence. A rule compaction mechanism is employed to reduce the number of reported rules significantly. Experiments on traces of JBoss Application Server shows the utility of our approach in inferring interesting past-time temporal rules.

1 Introduction

Different from many engineering products that rarely change, software changes often throughout its lifespan. This phenomenon has been well studied under the umbrella notion of software evolution. Software maintenance effort deals with the management of such changes, ensuring that the software remains correct while additional features are incorporated [18]. Maintenance cost can contribute up to 90% of software development cost [14]. *Reducing maintenance cost* and ensuring a program *remains correct during evolution* are certainly two worthwhile goals to pursue.

A substantial portion of maintenance cost is due to the difficulty in understanding an existing code base. Studies show that program comprehension can contribute up to 50% of the maintenance cost [17, 39]. A challenge to software comprehension is the maintenance of an accurate and updated specification as program changes. As a study shows, documented specifications often remain unchanged during program evolution [11]. One contributing factor is the short-time-to-market requirement of software products [8]. Multiple cycles of software evolution can render the outdated specification invalid or even misguiding.

To ensure correctness of a software system, model checking [9] has been proposed. It accepts a model and a set of formal properties to check. Unfortunately, difficulty in formulating a set of formal properties has been a barrier to its wide-spread adoption [5]. Adding software evolution to the equation, the verification process is further strained. First, ensuring correctness of software as changes are made is not a trivial task: a change in one part of a code, 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.

Addressing the above problems, there is a need for techniques to automatically reverse engineer or mine formal specifications from program. Recently, there has been a surge in software engineering research to adopt machine learning and statistical approaches to address these problems. One active area is specification discovery [5, 38, 28, 10], where software specification is reverse-engineered from program traces. Employing these techniques ensures specifications remain updated; also it provides a set of properties to verify via formal verification tools like model checking. To re-emphasize, the benefits of specification mining are as follows:

1. Aid program comprehension and maintenance by automatic recovery of program behavioral models (*e.g.*, [28, 10, 38])
2. Aid program verification (also runtime monitoring) in automating the process of "formulating specifications"(*e.g.*, [5, 42])

Most specification miners extract specifications in the form of automata [28, 5, 10, 38] or temporal rules [42, 31]. Usually a mined automata express the whole behavior of a system under analysis. Mined rules express strongly-observed constraints each expressing a property which holds with certain statistical significance.

Rules mined in [42, 30, 31] express future-time temporal expressions. Yang *et al.* mine two event rules of the form: "Whenever a event occur, eventually another event occur in the future" [42]. Lo *et al.* mine temporal rules of arbitrary length of the form: "Whenever a series of event occur, eventually another series of event occur in the future" [30, 31]. In this work, we extend the above work by mining past-time temporal expressions of this format:

> "Whenever a series of events pre occur, previously, another series of events $post$ happened before"

The above rule is denoted as $pre \hookrightarrow_P post$, where pre and $post$ corresponds to the premise (pre-condition) and consequent (post-condition) of the rule. These set of rules can be expressed in past-time temporal logic (Linear Temporal Logic (LTL) + past time operators [24]) and belong to two of the most used families of temporal logic expressions for verification (*i.e.*, precedence and chain-precedence) according to a survey by Dwyer *et al.* [12]. Some example specifications of the above format are as follows:

1. Whenever a file is used, it was opened before.
2. Whenever a socket is written, it was initialized before.
3. Whenever SSL_read is performed, SSL_init was invoked before.

4. Whenever a client request a resource and the resource is not granted, the resource had been allocated to another client that requested it before.
5. Whenever money is dispensed in an Automated Teller Machine (ATM), previously, card was inserted, pin was entered, user was authenticated and account balance was checked before.

It has been shown that past-time LTL can express temporal properties more succinctly than (pure) future-time LTL [23, 27]. Simple past-time LTL can correspond to more complicated equivalent future-time LTL, many of which are not minable by existing techniques mining future-time LTL rules from traces [42, 31]. The subset of the past-time LTL mined by the approach presented in this paper is not minable by previous approach in [42, 31]. Our work is not meant to replace, rather to complement mining future-time temporal rules.

In static inference of specification, Ramanathan *et al*. [37] mine specifications from program source code of the form: 'Whenever an event occur, previously, another series of events happened before". Different from Ramanathan *et al.* we analyze program traces and we need to address the issue of repeated behaviors within program traces (due to loops and recursions). Ramanathan *et al.* uses an off-the-shelf data mining algorithm [3] which *ignores repetitions within a sequence*. Static analysis has a different set of issues related to difficulty in analyzing pointers & references [7] and the number of infeasible paths [6]. Also, our target specification format is more general capturing pre-conditions of multiple event length, hence enabling user to mine for more complex temporal properties. Also, as described later, we present a method to compact significant rules by 'early' pruning of redundant rules resulting in a potentially combinatorial speed-up and reduction in the set of mined rules. In [37], all significant rules are first generated before redundant ones are removed. The large number of intermediary rules (exponential to the length of the longest rule) might make the algorithm not scalable enough to mine for rules of *long* length. Also, specification pertaining to behavior of the system like the 4th specification above is not easily minable from code.

Our mining algorithm models mining as a search space exploration process. The input is a set of sequences of events, where an event corresponds to an interesting method invocation to be analyzed. The output is a set of significant rules that obeys the minimum thresholds of support and confidence – which are commonly used statistics in data mining [19]. We define the support of the rule to be the number of traces where the rule's premise is observed. We define the confidence of the rule to be the likelihood that the rule's premise is preceded by the consequent. Similar to model checking, the algorithm builds the solution in a bottom up fashion. It first constructs rule of short length, and utilize several properties to throw away sub-search space not yet traversed if some short-length rule is not significant. The search space pruning strategy ensures that the runtime is linearly bounded to the number of significant rules rather than the size of the input.

In addition, we observe that some rules are redundant. To address this, we employ additional pruning strategies to throw away redundant rules. We kept the more comprehensive longer rules that capture more information and hence subsume the shorter ones.

We guarantee that all mined rules are significant and non-redun- dant. Also, all significant and non-redundant rules are mined. In data mining, an algorithm meeting the first and second criteria above is referred to as being correct (sound) and complete respectively (*c.f.*, [19, 25]). In this paper, we refer to the above criteria as *statistical soundness and completeness*.

To demonstrate our ideas, in this paper, we experimented with traces from components of JBoss Application Server [21]. The experiments shows the utility of our approach in mining specifications of an industrial program.

The paper is organized as follows. In Section 2, we discuss the semantics of mined rules. Section 3 describes our mining algorithm. Section 4 describes the experiments performed. Section 5 describes related work. Section 6 discusses some future work and we finally conclude in Section 7.

2 Concepts and Definitions

This section introduces preliminaries on past-time LTL and formalize the scope of rules minable by our approach. Also, notations used in this paper are described.

Past-Time Linear-time Temporal Logic Our mined rules can be expressed in past-time Linear Temporal Logic (LTL) [23, 24, 27]. Past-time temporal logic is an extension of (future-time) LTL [34, 20]. 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 4-event path.

There are a number of LTL operators, among which we are only interested in the operators 'G','F' and 'X' and some of their past-time counterparts 'F^{-1}' and 'X^{-1}' . The operator 'G' specifies that *globally* at every point in time a certain property holds. The operator 'F' specifies that a property holds at that point in time or *at a time in the future*. The operator 'X' specifies that a property holds at the *next* event.

The operator 'F^{-1}' specifies that a property holds at that point in time or *at a time in the past*. The operator 'X^{-1}' specifies that a property holds at the *previous* event.

Let us consider three examples listed in Table 1.

$X^{-1}F^{-1}(file\_open)$
Meaning: *At a time in the past* file is opened
$G(file\_read \rightarrow X^{-1}F^{-1}(file\_open))$
Meaning: *Globally* whenever file is read, *at a time in the past* unlock is called
$G((account\_deducted \wedge XF(money\_dispensed)) \rightarrow$ $(X^{-1}F^{-1}(balance\_suffice \wedge (X^{-1}F^{-1}(cash\_requested \wedge (X^{-1}F^{-1}(correct\_pin \wedge$ $(X^{-1}F^{-1}(insert\_debit\_card)))))))))$
Meaning: *Globally* whenever one's bank account is deducted and money is dispensed (from an ATM), previously user inserted debit card, entered correct pin, requested for cash to be dispensed and account balance was checked and it sufficed.

Table 1. Past-time LTL Expressions and their Meanings

Notation	LTL Notation
$a \hookrightarrow_P b$	$G(a \rightarrow X^{-1}F^{-1}b)$
$\langle a, b\rangle \hookrightarrow_P c$	$G((a \wedge XFb) \rightarrow (X^{-1}F^{-1}c))$
$a \hookrightarrow_P \langle b, c\rangle$	$G(a \rightarrow X^{-1}F^{-1}(c \wedge X^{-1}F^{-1}b))$
$\langle a, b\rangle \hookrightarrow_P \langle c, d\rangle$	$G((a \wedge XFb) \rightarrow (X^{-1}F^{-1}(d \wedge X^{-1}F^{-1}c)))$

Table 2. Rules and their Past-time LTL Equivalences

Our mined rules state whenever a series of premise events occurs it was be preceded by another series of consequent events. A mined rule denoted as $pre \hookrightarrow_P post$, can be mapped to its corresponding LTL expression. Examples of such correspondences are shown in Table 2.

Mapping to common English language expressions and for uniformity purpose, in both the premise and consequent of the rule the time goes forward to the future (e.g., a is followed by b, is preceded by, c is followed by d). In the corresponding past-time LTL expression we need to reverse the order of c and d. Also note that although the operator 'X' might seem redundant, it is needed to specify rules such as $\langle a\rangle \hookrightarrow_P \langle b, b\rangle$ 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[1]:

$$\begin{array}{|l} rules := G(pre \rightarrow post) \\ \quad pre := (event) | (event \wedge XF(pre)) \\ \quad post := (event) | (event \wedge X^{-1}F^{-1}(post)) \end{array}$$

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. 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 $first(P)$ to denote the first 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 $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $e_1 = f_{i_1}, e_2 = f_{i_2}, \cdots, e_n = f_{i_n}$.

3 Mining Past Time Temporal Rules

Each temporal rule of interest has the form $P_1 \hookrightarrow_P P_2$, 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 past-time LTL notations. Some examples are shown in Table 2.

In this paper, since 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*

[1] post is in reversed order

and a set of input traces as a sequence database. We use the sample trace or sequence database in Table 3 as our running example to illustrate the concepts behind generation of temporal rules.

Identifier	Trace/Sequence
S1	$\langle$c, b, a, e, b, a$\rangle$
S2	$\langle$c, b, e, a, e, b, c, a$\rangle$
S3	$\langle$d, a$\rangle$

Table 3. Example Database – DBX

3.1 Concepts & Definitions

Mined rules are formalized as past-time Linear Temporal Logic expressions with the format: G($\ldots \rightarrow X^{-1}F^{-1}\ldots$). The semantics of past-time LTL described in Section 2 will dictate the semantics of temporal rules described here. Noting the meaning of the temporal operators illustrated in Table 1, to be precise, a mined past-time temporal rule expresses:

> "Whenever a series of events *occurred starting at a point in time (i.e. a temporal point)*, previously, another series of events happened before."

From the above definition, to generate temporal rules, we need to "peek" at interesting temporal points and "see" what series of events are likely to occur *before*. We first formalize the notion of temporal points and the related notion of occurrences.

Definition 31 (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 = $\langle a_1, \ldots, a_n\rangle$, the suffix $\langle a_{n-(j-1)}, \ldots, a_n\rangle$ is called the j-suffix of S.*

Definition 32 (Occurrences & Instances) *Given a pattern P and a sequence S, the* occurrences *of P in S are defined by a set of temporal points $\mathcal{T}$ in S such that for each $j \in \mathcal{T}$, the j-suffix of S is a super-sequence of P and $first(P)$ is indexed by j. The set of* instances *of pattern P in S is defined as the set of j-suffixes of S, for each $j \in \mathcal{T}$.*

Example. Consider a pattern P $\langle b, a\rangle$ and the sequence $S1$ in Table 3 (*i.e.*, $\langle c, b, a, e, b, a\rangle$). The *occurrences* of P in $S1$ form the set of temporal points {2,5}, and the corresponding set of *instances* are $\{\langle b, a\rangle, \langle b, a, e, b, a\rangle\}$.

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

Definition 33 (Projected-past & Sup-past) *A database* projected-past *on a pattern p is defined as:*

$SeqDB_P^{past}$ = $\{(j, px) \mid$ *the* j^{th} *sequence in SeqDB is s, where* $s = px{+}{+}sx$*, and* sx is the minimum suffix of s containing $p\}$

Given a pattern P_X*, we define* $\sup^{past}(P_X, SeqDB)$ *to be the size of* $SeqDB_{P_X}^{past}$ *(i.e., 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 34 (Projected-past-all & Sup-past-all) *A database* projected-past-all *on a pattern* p *is defined as:* $SeqDB_P^{past-all}$ = $\{(j, px) \mid$ *the* j^{th} *sequence in* $SeqDB$ *is* s*, where* $s = px{+}{+}sx$*, and* sx is an instance of p in s *and first(*sx*) = first(*p*)*$\}$

Consider for example, a pattern P_X*, we define* $sup^{past-all}(P_X, SeqDB)$ *to be the size of* $SeqDB_{P_X}^{past-all}$*. Reference to the database is omitted if it is clear from the context.*

Definition 33 captures events occurring after the *first temporal point*. Definition 34 capture events occurring after *each temporal point*.

Example. To illustrate the above concepts, we project and project-all the example database DBX with respect to the pattern $\langle b, a\rangle$. The results are shown in Table 4 (a) & (b) respectively.

(a)

Identifier.	**Trace/Sequence**
S1	(1,⟨c, b, a, e⟩)
S2	(2,⟨c, b, e, a, e⟩)

(b)

Identifier	**Trace/Sequence**
$S1_1$	(1,⟨c, b, a, e⟩)
$S1_2$	(1,⟨c⟩)
$S2_1$	(2,⟨c, b, e, a, e⟩)
$S2_2$	(2,⟨c⟩)

Table 4. $(a); DBX_{\langle b,a\rangle}^{past}$ & $(b); DBX_{\langle b,a\rangle}^{past-all}$

The two projection methods' associated notions of $\sup^{past}$ and $sup^{past-all}$ are different. Specifically, $sup^{past-all}$ 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, $\sup^{past}(\langle b, a\rangle, DBX) = |DBX_{\langle b,a\rangle}^{past}| = 2$. On the other hand, $sup^{past-all}(\langle b, a\rangle, DBX) = |DBX_{\langle b,a\rangle}^{past-all}| = 4$.

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

Definition 35 (Support & 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* $\sup^{past}(pre_X, SeqDB)$*. The confidence of* R_X *is defined as the likelihood of* $post_X$ *happening before* pre_X*. This is equivalent to the ratio of* $\sup^{past}(post_X, SeqDB_{pre_X}^{past-all})$ *to the size of* $SeqDB_{pre_X}^{past-all}$*.*

Example. Consider DBX and a temporal rule R_X, $\langle b, a\rangle \hookrightarrow_P \langle c\rangle$. From the database, the support of R_X is the number of sequences in DBX supporting (or is a super-sequence of) the rule's pre-condition – $\langle b, a\rangle$. There are 2 of them – see Table 4 (a). Hence support of R_X is 2. The confidence of the rule R_X ($\langle b, a\rangle \hookrightarrow_P \langle c\rangle$) is the likelihood of $\langle c\rangle$ occurring before each *temporal point* of $\langle b, a\rangle$. Referring to Table 4(b), we see that there is a $\langle c\rangle$ occurring before each temporal point of $\langle b, a\rangle$. 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 (*c.f.*, [42]). 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 [42]. Similar to the assumption made by work in statistical debugging (*e.g.*, [13]), 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 or runtime monitors.

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

Definition 36 (Past-Time Temporal Rules) *A temporal rule R_X is denoted by $pre \hookrightarrow_P post$ $(sup, conf)$. 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 DBX and the rule R_X, $\langle b, a\rangle \hookrightarrow_P \langle c\rangle$ shown in the previous example. It has support of 2 and confidence of 1. It is denoted by $\langle b, a\rangle \hookrightarrow_P \langle c\rangle(2, 1)$.

3.2 Monotonicity and Non-Redundancy

Our algorithm is a member of the family of pattern mining algorithms, e.g. [3, 40]. Monotonicity (a.k.a. apriori) properties have been widely used to ensure efficiency of many pattern mining techniques (*e.g.*, [3, 40]). Different mining algorithm often require new or additional apriori property. Fortunately, past-time temporal rules obey the following apriori properties:

Theorem 1 (Monotonicity Property – Support) *If a rule $evs_P \hookrightarrow_P evs_C$ does not satisfy the* min_sup *threshold, neither will all rules $evs_Q \hookrightarrow_P evs_C$ where evs_Q is a super-sequence of evs_P.*

Theorem 2 (Monotonicity Property – Confidence) *If a rule $evs_P \hookrightarrow_P evs_C$ does not satisfy the min_conf threshold, neither will all rules $evs_P \hookrightarrow_P evs_D$ 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 *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 [40].

Definition 37 (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., $post_X ++ pre_X \sqsubseteq post_Y ++ pre_Y$)

(2) Both rules' support and confidence are the same

Also, in the case that the concatenations are the same (i.e., $post_X ++ pre_X = post_Y ++ pre_Y$), 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 (ATM):

R1	*money_dispensed* $\hookrightarrow_P$ *card_inserted, enter_pin, pin_correct cash_request*
R2	*money_dispensed* $\hookrightarrow_P$ *card_inserted*
R3	*money_dispensed* $\hookrightarrow_P$ *enter_pin*
R4	*money_dispensed* $\hookrightarrow_P$ *card_inserted, enter_pin*
R5	*money_dispensed* $\hookrightarrow_P$ *enter_pin, cash_request*

If all of the above rules have the same support and confidence values, rules R2-R5 are redundant since 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 ones. 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.

Theorem 3 (Pruning Redundant Pre-Conds) *Given two pre-conditions P_X and P_Y where $P_X \sqsubset P_Y$, if $SeqDB^{past}_{P_X} = SeqDB^{past}_{P_Y}$ then for all sequences of events post, rules $P_X \hookrightarrow_P post$ is rendered* redundant *by $P_Y \hookrightarrow_P post$ and can be pruned.*

Proof. Since $P_X \sqsubset P_Y$, from Definition 37 of rule redundancy, we only need to prove that the rules R_X ($P_X \rightarrow post$) and R_Y ($P_Y \rightarrow post$) have the same values of support and confidence.

Since $SeqDB^{past}_{P_X} = SeqDB^{past}_{P_Y}$, the followings are guaranteed: (1) P_X and P_Y must share the same prefix (at least $first(P_X) = first(P_Y)$) and (2) $\forall s \in SeqDB$. the first instance of P_X corresponds to the first instance of P_Y. From points (1) and (2) above, not only the first instance, but every instance of P_X in $SeqDB$ must also correspond to an instance of P_Y (and vice versa). In other words, $SeqDB^{past}_{P_X} = SeqDB^{past}_{P_Y}$ iff $SeqDB^{past-all}_{P_X} = SeqDB^{past-all}_{P_Y}$.

Since $SeqDB^{past-all}_{P_X} = SeqDB^{past-all}_{P_Y}$, and R_X and R_Y share the same post-condition, R_X and R_Y must have the same support and confidence values. Hence, R_X is rendered redundant by R_Y and can be pruned.

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

Proof. Since $P_X \sqsubset P_Y$, from Definition 37 of rule redundancy, we only need to prove that the rule R_X ($pre \rightarrow P_X$) and R_Y ($pre \rightarrow P_Y$) have the same values of support and confidence. The equality of support values is guaranteed since the two rules have the same pre-condition.

Since $(SeqDB^{past-all}_{pre})^{past}_{P_X} = (SeqDB^{past-all}_{pre})^{past}_{P_Y}$, it implies $\sup^{past}(P_X, SeqDB^{past-all}_{pre}) = \sup^{past}(P_Y, SeqDB^{past-all}_{pre})$. Hence, the two rules will have the same confidence values. Hence, we have shown that R_X is rendered redundant by R_Y and can be pruned.

Utilizing Theorems 3 & 4, many redundant rules can be pruned 'early'. However, the theorems only provide sufficient conditions for the identification of redundant rules – there are redundant rules which are not identified by them. To remove remaining redundant rules, we perform a post-mining filtering step based on Definition 37.

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

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

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

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

Step 4 Using Definition 37, we filter any remaining redundant rules.

4 Experiments

In this section we discuss our experiments on mining past-time temporal rules from traces of JBoss Application Server. It shows the utility of our method in recovering specifications of an industrial system.

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

on the traces with the minimum support and confidence thresholds set at 15 and 90% respectively, 4 non-redundant rules were mined. The algorithm completed within 2.5 seconds.

A sample of the mined rules is shown in Figure 1 (left). It describes authentication using Java Authentication and Authorization Service (JAAS) for EJB within JBoss-AS. Roughly it describes a rule that states: "Whenever principal and credential information is required (the premise of the rule), previously configuration information is checked to determine authentication service availability (event 1-5 in the consequent), actual authentication events are invoked (event 6-8) and principal information is bound to the subject being authenticated (event 9-12)".

Premise	→ P	Consequent
SimplePrincipal.toString() SecAssoc.getPrincipal() SecAssoc.getCredential() SecAssoc.getPrincipal() SecAssoc.getCredential()		XLoginConfImpl.getConfEntry() PolicyConfig.get() XLoginConfImpl$1.run() AuthenticationInfo.copyAppConfEntry() AuthenticationInfo.getName() ClientLoginModule.initialize() ClientLoginModule.login() ClientLoginModule.commit() SecAssocActs.setPrincipalInfo() SetPrincipalInfoAction.run() SecAssocActs.pushSubjectContext() SubjectThreadLocalStack.push()

Premise	→ P	Consequent
TransactionImpl.isDone()		TransManLocator.getInstance() TransManLocator.locate() TransManLocator.tryJNDI() TransManLocator.usePrivateAPI() TxManager.getInstance() TxManager.begin() XidFactory.newXid() XidFactory.getNextId() XidImpl.getTrulyGlobalId() TransImpl.assocCurrentThread() TransImpl.lock() TransImpl.unlock() TransImpl.getLocalId() XidImpl.getLocalId() LocalId.hashCode() TxManager.getTransaction()

Fig. 1. A sample rule from JBoss-Security (top) and another from JBoss-Transaction (bottom). Each of the rules are read from top to bottom, left to right.

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 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 traces with the minimum support and confidence thresholds set at 25 traces and 90%

respectively, 36 non-redundant rules were mined. The algorithm completed within 30 seconds.

A sample of the mined rules is shown in Figure 1 (right). The rule describes that: "Whenever a check is performed on whether transaction is completed (the premise of the rule), previously connection to a server instance (event 1-4 in the consequent), initialization and utilization of transaction manager and implementation (event 5-6,10-12), acquiring of ids (event 7-9,13-15) and obtaining of transaction from the manager (event 16) are performed before."

5 Related Work

One of the most well-known specification mining tool is Daikon [16]. It returns value-based invariants (*e.g.*, x $>$ 5, etc.) by monitoring a fixed set of templates as a program is run. Different from Daikon, in this work, we consider temporal invariants capturing ordering constraint among events.

Most specification mining tools mine *temporal* specifications in the form of automata [33, 2, 28, 38, 10]. An automata specify a global behavior of a system. Different from work mining automata, mined rules describe *strongly observed* sub-behaviors of a system or properties that occur with statistically significance (*i.e.*, appear with enough support and confidence).

In [42], Yang et al. 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 [20]), which are statistically significant with respect to a user-defined 'satisfaction rate'. These rules express: "whenever an event occurs, eventually in the future another event occurs". 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 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 [30, 31], Lo et al. extended the work by Yang et al. to mine future-time temporal rules of *arbitrary lengths*. The algorithm is statistically sound and complete. Rules of arbitrary lengths is able to capture more complex temporal properties. Often, simple properties are already known by the programmers while complex properties might be missed or might be an emergent behavior.

In [32], Lo et al. mine Live Sequence Chart (LSC) from program execution traces. LSC can be viewed as a formal form of a sequence diagram. In [32], the LSCs mined are of the format: "whenever a chart pre is satisfied, eventually another chart main is satisfied". Different from standard temporal rules, LSCs impose specific constraints on the satisfaction of a chart (pre or main). When translated to LTL, LSC corresponds to a rather complex temporal expressions [22]. Also, different from this work, the work in [32] only mine LSC that express future time temporal expressions. To the best of our knowledge, LSCs focus on expressing future-time temporal expressions as it is usually employed to express liveness properties for reactive systems.

In [29], we proposed iterative patterns to mine frequent patterns of program behavior. Different from rules, patterns do not express any constraints. A rule on the other

hand expresses a constraint that state when its premise is satisfied, its consequent is satisfied as well. For monitoring and verification purposes, constraints are needed.

There are several studies on extracting specifications from code (*e.g.* [4, 26, 13, 41, 37]). The above techniques belong to the *static analysis* family. In contrast, we adopt a *dynamic analysis* approach in extracting specifications from execution traces. Static and dynamic analyses complement each other (*c.f.*, [15]). Their pros and cons have been discussed in the literature [15, 7]. With dynamic analysis, even with the best algorithms, the quality of specification mined is only as good as the quality of traces. With static analysis, one is faced with the problem of pointers and infeasible paths. Some specifications pertaining to the dynamic behavior of a system can only be mined (or are much easier to mine) via dynamic analysis.

Studies in [26, 13, 41, 37] mine specifications from code, and present them in the form of rules. The study in [26] ignores ordering of events and hence mined rules do not describe temporal properties. Past studies on extracting rules expressing temporal properties from code [13, 41] are limited to extract two-event rules. In [37], Ramanathan et al. mine past-time temporal rules from program code. Different with our approach, we mine rules from program execution traces, address the issue of repetitions due to loop and recursion in the traces and mine rules with pre-conditions of arbitrary lengths.

6 Discussion

In this section, we discuss some related issues and potential future work.

Currently we only consider method calls without consideration of their return and parameter values. Each method call is mapped to a symbol. Often, it is necessary to differentiate a method based on its input parameters or return values. For example, a socket read can return success or failure. Sometimes, the return values can be inferred from a sequence of method call. For example in OpenSSL [1], if SSL_read() is followed by SSL_get_error(), it is likely to mean that something wrong has happened to the read. In the future, we are looking into incorporation of the return and parameter values to the mining process.

There are alternative definition and other sub-set of past-time temporal rules to be mined. In this paper, the occurrence of the rule post-condition occurs before the first event of the rule's premise. Sometimes this condition need to be modified. For example, consider the specification: "Whenever a function perform a blocking read on the channel and the function exit successfully, a channel write happened before." In the case above, the channel write can occur before the blocking read or between the blocking read and the function exit. A rule capturing the above specification corresponds to a different past-time LTL expression. We leave the modification of the mining algorithm to mine the above variant of our past-time temporal rules as a future work.

In this paper, only ordering constraint is considered. Often, both ordering and value-based constraints are needed. Consider an embedded software running on a vending machine. A possible specification is: "Whenever a coin is entered but later ejected, it must be the case that the number of drinks to be dispensed == 0". Also, consider the specification: "Whenever a non-blocking resource request is made, but it is not granted, it must be the case that the number of resource left == 0". We plan to follow

the approach by Mariani *et al.* [35] and Lorenzoli *et al.* [33], by merging Daikon [16], that mines value-based invariants, with our tool.

Also similar to the argument made in [23], often there is a need to 'forget' past information. Adapting the example in [23], consider an automotive system with an alarm notifying problem and a reset button to restore the system to last known good state. A good property is to ensure that the car alarm doesn't sound unless there is a problem before. This corresponds to the past-time property: "whenever an alarm sound, there must be a problem before". However, this problem must appear after the last reset. One need to "forget" past information whenever a reset action is taken. In mining software specifications, if these "reset"-like events can be input to the miner, this will improve the mining results and speed up the mining process as well.

In our experiments, we note that some of our mined rules are minor variations of one another. This is the case since we only mine for total order. If there are two events whose order is irrelevant, the miner might return this partial order as two separate rules. In the future we plan to adopt the approach by Acharya *et al.* [2], used for mining automata using static analysis by composing mined partial orders, and adapt it to mine for rules expressing partial order. The partial order miner [36] used in [2] ignores repetitions in the input sequence set. A new data mining algorithm might need to be developed to merge our approach with the partial order miner.

Also in the list of our future work is incorporation of negation, disjunction and the remaining LTL/past-time LTL operators not considered so far namely: until and since operators. It is also of interest to consider merging of static and dynamic analysis for temporal rule mining.

7 Conclusion

In this paper, we propose a technique to mine past-time temporal rules from program execution traces. The rules state: "Whenever a series of events occur, previously another series of events happened before". These rules capture important properties useful for verification, monitoring and program understanding.

Existing work on mining temporal rules focuses on future-time temporal expressions. Past-time temporal logic is more intuitive and compact to express some class of important properties. We consider our work to complement existing techniques mining future-time temporal expressions. To the best of our knowledge, this is the first work on mining past-time temporal rules from program execution traces where repetitions due to loop and recursion need to be considered. Our rule format is also more general than the precedence rule mined by static-analysis-based approach in [37].

Also, 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. Experiments on JBoss Application Server show the utility of our technique in recovering specifications of an industrial program.

References

1. OpenSSL: Documents, sll(3). http://www.openssl.org/docs/ssl/SSL_read.html.

2. M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In *SIGSOFT FSE*, 2007.
3. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of IEEE Int. Conf. on Data Engineering*, 1995.
4. R. Alur, P. Cerny, G. Gupta, and P. Madhusudan. Synthesis of interface specifications for java classes. In *POPL*, 2005.
5. G. Ammons, R. Bodik, and J. R. Larus. Mining Specification. In *POPL*, 2002.
6. N. Baskiotis, M. Sebag, M.-C. Gaudel, and S. Gouraud. A machine learning approach for statistical software testing. In *Proc. of Int. Joint Conf. on Artificial Intelligence*, 2007.
7. L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE TSE*, 32(9):642–663, 2006.
8. R. Capilla and J. Dueñas. Light-weight product-lines for evolution and maintenance of web sites. In *CSMR*, 2003.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *TOSEM*, 7(3):215–249, July 1998.
11. S. Deelstra, M. Sinnema, and J. Bosch. Experiences in software product families: Problems and issues during product derivation. In *SPLC*, 2004.
12. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.
13. D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of Symp. on Operating Systems Principles*, 2001.
14. L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, pages 17–23, 2000.
15. M. Ernst. Static and dynamic analysis: Synergy and duality. In *Work. on Dynamic Ana.*, 2003.
16. M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *TSE*, 27(2):99–123, 2001.
17. R. Fjeldstad and W. Hamlen. Application program maintenance-report to our respondents. *Tutorial on Software Maintenance*, pages 13–27, 1983.
18. P. Grubb and A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific, 2003.
19. J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann, 2006.
20. M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge, 2004.
21. JBoss. http://www.jboss.org.
22. H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal Logic for Scenario-Based Specifications. In *TACAS*, 2005.
23. F. Laroussinie, N. Markey, and P. Schnoebelen. Temporal logic with forgettable past. In *LICS*, 2002.
24. F. Laroussinie and P. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 2(148):303–324, 1995.
25. J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *AAAI*, 2006.
26. Z. Li and Y. Zhou. PR–miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *SIGSOFT FSE*, 2005.
27. O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Proc. Logics of Programs Workshop*, pages 196–218, 1985.
28. D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.
29. D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. *Proc. of SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2007.

30. D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules from program execution traces. In *Proc. of Int. Work. on Program Compre. through Dynamic Ana. (Informal Proceedings)*, 2007.
31. D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of recurrent rules from a sequence database. In *Proc. of Int. Conf. on Database Systems for Advanced Apps.*, 2008.
32. D. Lo, S. Maoz, and S.-C. Khoo. Mining Modal Scenario-Based Specification from Execution Traces of Reactive Systems. In *ASE*, 2007.
33. D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *ICSE*, 2008.
34. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.
35. L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis for component integration. In *ICECCS*, 2005.
36. J. Pei, H. Wang, J. Liu, K. Wang, J. Wang, and P. S. Yu. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering*, 18:1467–1481, 2006.
37. M. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE*, 2007.
38. S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, 2001.
39. T. Standish. An essay on software reuse. *IEEE TSE*, 5(10):494–497, 1984.
40. J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. of IEEE Int. Conf. on Data Engineering*, 2004.
41. W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.
42. J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.