Mining Patterns and Rules for Software Specification Discovery

David Lo advised by Siau-Cheng Khoo
Department of Computer Science,National University of Singapore
{dlo,khoosc}@comp.nus.edu.sg

ABSTRACT

Software specifications are often lacking, incomplete and outdated in the industry. Lack and incomplete specifications cause various software engineering problems. Studies have shown that program comprehension takes up to 45% of software development costs. One of the root causes of the high cost is the lack-of documented specification. Also, outdated and incomplete specification might potentially cause bugs and compatibility issues. In this paper, we describe novel data mining techniques to mine or reverse engineer these specifications from the pool of software engineering data.

A large amount of software data is available for analysis. One form of software data is program execution traces. A program trace can be viewed as a sequence of events collected when a program is run. A set of program traces in turn can be viewed as a sequence database. In this paper, we present some novel work in mining software specifications by employing novel pattern mining and rule mining techniques. Performance studies show the scalability of our technique. Case studies on traces of a real industrial application show the utility of our technique in recovering program specifications from execution traces.

1. INTRODUCTION

It's best if all programs and software projects are developed with clear, precise and documented specifications. However, due to hard deadlines and 'short-time-to-market' requirement [5], software products often come with poor, incomplete and even without any documented specifications. This situation is further aggravated by a phenomenon termed as software evolution [4, 17]. As software evolves the documented specification is often not updated. This might render the original documented specification of little use after several cycles of program evolution [10].

The above factors have contributed to high software maintenance costs. It has been investigated that 90% of software cost is due to maintenance [12] and 50% of the maintenance cost is due to comprehending or understanding the code base [28]. Hence, approximately 45% of software cost is due to difficulty in comprehending an existing code base. This is especially true for software projects developed by many developers over a long period of time.

Permission to make digital or hard copies of portions of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyright for components of this work owned by others than VLDB Endowment must be honored.
Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists requires prior specific permission and/or a fee. Request permission to republish from: Publications Dept., ACM, Inc. Fax +1 (212)869-0481 or permissions@acm.org.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-306-8/08/08

To ensure correctness of a software system, model checking [7] 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 [2]. 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 programs. 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 [2, 25, 19, 8], 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.*, [19, 8, 25])
2. Aid program verification (also runtime monitoring) in automating the process of "formulating specifications"(*e.g.*, [2, 33])

In the above line of research we propose data mining algorithms to mine for software specifications. An interesting form of specifications to be mined is *patterns of software temporal behaviors*. These patterns are intuitive and commonly found in software specification documentations. Some of these patterns are:

1. Telecommunication Protocol (*c.f.*, [15]): ⟨*off_hook*, *dial_tone_on*, *dial_tone_off*,*seizure_int*,*ring_tone*,*answer*, *connection_on*⟩

2. Java Authentication and Authorization Service (JAAS)

Authorization Enforcer Strategy Pattern (*c.f.*, [29]): $\langle Subject.getPrincipal$, $PrivilegedAction.create$, $Subject.doAsPrivileged$, $JAAS_Module.invoke$, $Policy.getPermission$, $Subject.getPublicCredential$, $PrivilegedAction.run\rangle$

3. Java Transaction Architecture (JTA) Protocol (*c.f.*, [26]): $\langle TxManager.begin$, $TxManager.commit\rangle$, $\langle Tx Manager.begin$, $TxManager.rollback\rangle$, *etc.*

Another form of useful specifications comes in the form of temporal constraints expressed as rules. One interesting form is rules of the following format:

> "Whenever a series of precedent events occurs, eventually another series of consequent events occurs"

Some sample specifications having the above forms are as follows:

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 money is transferred and a receipt is displayed.

Each of these patterns and rules reflects some interesting program behavior. It can be mined by analyzing a set of program traces – each being a series of method invocations. These program traces can in turn be generated through running a test suite. From data mining point of view, each trace can be considered as a sequence. A pattern or rule (*e.g.*, lock-unlock) can *appear a repeated number of times within a sequence.* Each event can be *separated by an arbitrary number of unrelated events* (*e.g.*, *lock* → resource use → ... → *unlock*). Since a program behavior can be manifested in numerous ways, *analyzing a single trace will not be sufficient.* Usually, a set of test cases satisfying certain code coverage (*i.e.*, every statements are executed) or branch coverage (*i.e.*, every branch decision is taken) criterion (*c.f.*, [3]) is required to test the correctness of a software system. Running this test suite over an instrumented software will generate the desired traces.

To mine software temporal patterns and rules having the above characteristics from traces, we propose novel techniques referred to as *iterative pattern mining* and *recurrent rule mining*. They leverages and extends sequential pattern mining and episode mining to address software specification mining.

Sequential pattern mining first addressed by Agrawal and Srikant in [1] discovers temporal patterns that are supported by a *significant number of sequences.* A sequential pattern is supported by a sequence if it is a sub-sequence of it. It has potential applications in many areas, from analysis of market data to gene sequences. On the other hand, Mannila *et al.* perform episode mining to discover *frequent episodes within a sequence of events* [22]. An episode is defined as a series of events occurring *relatively close* to one another (*i.e.* they occur at the same window). An episode is supported by a window if it is a sub-sequence of the series of events appearing in the window. Episode mining focuses on mining from a single sequence of events, and has its application in analyzing events from telecommunication alarm management system.

Iterative pattern is a series of events supported by a *significant number of instances repeated within and across sequences.* Recurrent rule is a temporal constraint that holds a *significant number of times within and across sequences.* Similar to sequential pattern mining, we consider a *database of sequences* rather than a single sequence. We also mine patterns and rules occurring repeatedly within a sequence. This is similar in spirit to episode mining, but we remove the restriction that related events must happen in the same window.

Due to looping, a trace can contain repeated occurrences of interesting patterns. In fact, a series of events in an alarm management system used by Manilla *et al.* is similar to a series of system calls in a software system. However, there are 2 notable differences.

First, program properties are often inferred from a set of traces instead of a single trace. These are either produced by executing a test suite [33] or generated statically from the source code [31]. Secondly, important software patterns and rules, such as lock acquire and release or stream open and close (*c.f* [33, 6]), often have their events occur at some arbitrary distance away from each other in a program trace. Hence, there is a need to 'break' the 'window barrier' in order to capture these patterns or rules of interest. Interestingly, these two notable differences between analysis of events from an alarm management system and program traces are observed by sequential pattern mining first introduced in [1].

To support iterative pattern and recurrent rule mining, we need clear definitions and semantics of iterative patterns and recurrent rules different from those of episodes and sequential patterns. Our definition of iterative patterns is inspired by several commonly used languages for specifying software behavioral requirements, namely Message Sequence Chart (MSC) [15] and Live Sequence Chart (LSC) [9]. For recurrent rules, we based and express our rules in Linear Temporal Logic [14] which is commonly used in program verification and monitoring.

In this paper, we mine a *closed* set of iterative patterns and a *non-redundant* set of recurrent rules. Search space pruning strategies employed for *early identification and pruning* of non-closed patterns and redundant rules are respectively used to mine a closed set of iterative patterns and non-redundant set of rules efficiently. Performance studies performed on synthetic and real-world datasets shows the major success of our non-closed pattern and redundant rules pruning strategies: the algorithms run with up to over two orders of magnitude speedup especially on low support thresholds or when the frequent patterns/significant rules are long.

As a case study we experimented with traces collected from components of JBoss Application Server. Our mined patterns highlight important design patterns and temporal constraints shedding light on program behaviors.

The outline of this paper is as follows: Section 4 provides a discussion on mining closed iterative pattern. Section 5 presents the principles behind the generation of *non-redundant recurrent rules* Section 6 presents the results of our performance studies. Section 7 discusses case studies on

mining program behavioral design from traces of JBoss Application Server. We present future work in Section 8 and conclude in Section 9.

2. RELATED WORK

Iterative pattern mining is an extension of sequential pattern mining, which was originated by Agrawal and Srikant [1]. To remove redundant patterns, closed sequential pattern mining was proposed by Yan *et al.* [32] and later improved by Wang and Han [30]. Different from sequential pattern mining, iterative pattern mining captures multiple occurrences of a pattern not only those repeated *across multiple sequences* but also those repeated *within each sequence*. In this aspect, iterative pattern mining resembles episode mining initiated by Mannila *et al.* [22] which was later extended by Casas-Garriga to replace a fixed-window size with a gap constraint between one event to the next in an episode [13]. Both versions of episode mining mine events occurring close to one another, expressed by "window size" and gap constraint respectively. This is *different* from iterative pattern mining, which does not have the notion of "episode". This deviation is significant, since important program behavioral patterns, for example: lock acquire and release, or file open and close (*c.f* [33, 6]), often have their events occur at some arbitrary distance away from one another in a trace. To address the potentially large number of patterns formed due to the removal of window size constraint, we only mine for closed patterns. As far as we know, there is no work on mining closed episodes. In addition, both versions of episode mining handle only one single sequence, whereas iterative pattern mining operates over a set of sequences.

Recurrent rule can be thought as an extension to sequential rule (*e.g.*, [27]) and episode rule (*e.g.*, [22, 13]). Different from sequential rule, recurrent rule considers satisfaction of the rule not only *across multiple sequences* but also repeatedly *within each sequence*. For episode rule, the rule's constituent event occur close to one another – this is expressed by "window size" or gap constraint.

In mining DNA sequences, Zhang *et al.* introduced the idea of "gap requirement" in mining periodic patterns from sequences [34]. Similar to ours, they detect repeated occurrences of patterns within a sequence and across multiple sequences. However, the gap requirement used there does not always hold for other purposes. Consider analyzing software traces, the useful patterns of lock acquire followed-by lock release can be separated by any number of events, and will violate the gap requirement. In addition, the pattern definition proposed in [34] does not follow apriori property and hence potentially reduces the efficiency of the mining process. Lastly, the method only guarantees the mining of a complete set of patterns with length less than n, where n is a user defined parameter. Often, the appropriate value of this parameter n is not obvious to the user.

El-Ramly *et al.* mined user-usage scenarios of GUI based program composed of screens – these scenarios are termed as interaction patterns [11]. Given a set of series of screen ids, frequent patterns of user interactions are obtained. Similar to iterative pattern mining, interaction pattern mining takes as an input a set of sequences and discover patterns occurring repeatedly within sequences.

However, due to differences in the nature of data mined, there are significant differences between interaction and iterative pattern mining. First, the semantics of the patterns mined are different. Iterative pattern adheres to the semantics of MSC/LSC specification language in describing software behavioral requirements, whereas interaction pattern does not. Second, apriori property is not observed by interaction patterns. In contrast, iterative patterns observe apriori property. Third, for each pattern instance, interaction pattern imposes a limit on the number of 'insertions' between one event to the next by a fixed constant. For many useful software temporal patterns (*e.g.* $\langle lock, unlock\rangle$) the number of 'insertions' is irrelevant – events can be separated by an arbitrary number of events; iterative patterns capture such "behavior" well.

In the area of specification mining, a number of studies on mining software temporal properties have been performed [33, 2, 19, 18]. Most of them mine an automata (*e.g.*, [2, 19]) and hence are very different from our work. Of the most relevance is the work on mining rule-based specification [33], where the rules have a similar semantics as our recurrent rules but are limited to two-event rules (*e.g.*, $\langle lock\rangle \rightarrow \langle unlock\rangle$). Their algorithms do not scale for mining multi-event rules since they first list all possible two-event rules and then check the significance of each rules. For rules of arbitrary lengths, the number of possible rules is arbitrarily large. Our work generalizes their work by mining a complete set of rules of arbitrary lengths that satisfy given support and confidence thresholds. To enable efficient mining, we devise a number of search space pruning strategies.

3. PRELIMINARIES

This section describes some definitions, preliminaries on Message & Live Sequence Charts and preliminaries on temporal logics.

3.1 Basic Definitions

Let I be a set of distinct events. Let a *sequence* S be an ordered list of events. We denote S as $\langle e_1, e_2, \ldots, e_{end}\rangle$ where each e_i is an event from I. We refer to the ith event in the sequence S as $S[i]$. The sequence database under consideration is denoted by $SeqDB$. Also, we denote a single arbitrary event as ev and a series of arbitrary events as evs.

A pattern P_1 ($\langle e_1, e_2, \ldots, e_n\rangle$) is considered a *subsequence* of another pattern P_2 ($\langle f_1, f_2, \ldots, f_m\rangle$) if there exist integers $1 \le i_1 < i_2 < i_3 < i_4 \ldots < i_n \le m$ where $e_1 = f_{i_1}$, $e_2 = f_{i_2}, \cdots, e_n = f_{i_n}$. Notation-wise, we write this relation as $P_1 \sqsubseteq P_2$. We also say that P_2 is a *super-sequence* of P_1. We use the notations *first*(P) and *last*(P) to denote the first event and the last event of P respectively. Reference to the database is omitted if it refers to the input sequence database $SeqDB$. The concatenation of two patterns P_1 and P_2 is denoted by $P_1{+\!+}P_2$.

3.2 MSC, LSC & Iterative Patterns

Our definition of iterative pattern is inspired by several commonly used languages for specifying software behavioral requirement: Message Sequence Chart (MSC) (a standard of International Telecommunication Union (ITU) [15]) and its extension, Live Sequence Chart (LSC) [9].

MSC and LSC are variants of the well known UML sequence diagram describing behavioral requirement of software. Not only does they specify system interaction through ordering of method invocation, but they also specify caller and callee information. An example of such charts is a sim-

plified telephone switching protocol (*c.f.*, [15]): abstracting caller and callee information and simplifying the protocol, it can be represented as a pattern: ⟨*off_hook*, *dial_tone_on*, *dial_tone_off*, *seizure_int*, *ring_tone*, *answer*, *connection_on*⟩.

In verifying traces for conformance to an event sequence specified in MSC/LSC, the sub-trace manifesting the event sequence must satify the total-ordering property: Given an event ev_i in an MSC/LSC, the occurrence of ev_i in the sub-trace occurs before the occurrence of every ev_j where $j > i$ and after ev_k where $k < i$ [15]. Kugler *et al.* strengthened the above requirement to include a one-to-one correspondence between events in a pattern and events in any sub-trace satisfying it [16]. Basically, this requirement ensures that, if an event appears in the pattern, then it appears as many times in the pattern as it appears in the sub-trace.

For the telephone switching example, the following traces are not in conformance to the protocol:

off_hook, *seizure_int*, *ring_tone*, *answer*,*ring_tone*, *connection_on*
off_hook, *seizure_int*, *ring_tone*, *answer*, *answer*, *answer*, *connection_on*

The first trace above doesn't satisfy the total-ordering requirement due to the out-of-order second occurrence of the *ring-tone* event. The second doesn't satisfy the one-to-one correspondence requirement due to the multiple occurrences of the *answer* event.

The full language of MSC/LSC is complicated and it is not our intention to mine MSC/LSC. Among other things, iterative pattern abstracts away the caller and callee information but retains the one-to-one correspondence and total ordering requirements between a pattern and its instances.

3.3 Temporal Logic & Recurrent Rules

Our mined rules can be expressed in Linear Temporal Logic (LTL) [14]. 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'. 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. Let us consider four examples listed in Table 1.

Our mined rules state 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 shown in Table 2. 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:

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

4. MINING ITERATIVE PATTERNS

The following describes the methodology to mine closed iterative patterns.

Our pattern instance definition following definitions of MSC and LSC and capturing instances of iterative patterns can be expressed unambiguously in the form of Quantified Regular Expression (QRE) [23]. Quantified regular expression is very similar to standard regular expression with ';' as concatenation operator, '[-]' as exclusion operator (*i.e.* [-P,S] means any event except P and S) and * as the standard kleene-star.

Definition 4.1 (**Pattern Instance - QRE**). *Given a pattern P ($p_1 p_2 \ldots p_n$), a substring SB ($sb_1 sb_2 \ldots sb_m$) of a sequence S in $SeqDB$ is an instance of P iff it is of the following QRE expression*

$$p_1; [-p_1, \ldots, p_n]*; p2; \ldots; [-p_1, \ldots, p_n]*; p_n.$$

To mine iterative patterns scalably we use the following apriori properties.

Theorem 1 (**Apriori Property**). *If P is not frequent then its extensions (P++evs or evs++P) (where evs is a series of events) are also not frequent.*

Iterative pattern instances can be mined using depth first pattern growth and prune strategy. However, rather than using the usual projection that extracts sequential patterns (see PrefixSpan [24]), we perform a different type of projection capturing instances of iterative patterns.

Also, to address scalability we mine for only closed patterns as defined below.

Definition 4.2 (**Closed Pattern**). *A frequent pattern P is* closed *if there exists no super-sequence Q s.t.:*

1. *P and Q has the same support*
2. *Every instance of P corresponds to a unique instance of Q.*

An instance of P ($seq_P, start_P, end_P$) corresponds to an instance of $Q(seq_Q, start_Q, end_Q)$ iff $seq_P = seq_Q$ and $start_P \geq start_Q$ and $end_P \leq end_Q$.

Several additional pruning properties are used to prune the search space containing non-closed patterns. The full details are available in [20].

5. MINING RECURRENT RULES

To be precise, a recurrent rule $pre \rightarrow post$ expresses:

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

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

Definition 5.1 (**Temporal Points & Occurrences**). *Consider a sequence S of the form $\langle a_1, a_2, \ldots, a_{end}\rangle$. All events in S are indexed by their position in S, starting at* 1 *(e.g., a_j is indexed by j). These positions are called* temporal points *in S. The* occurrences *of a pattern P in S is defined by a set of temporal points $\mathcal{T}$ such that for each $j \in \mathcal{T}$, the prefix of S ending in j is a super-sequence of P and $last(P) = S[j]$.*

$F(unlock)$
Meaning: *Eventually* unlock is called

$XF(unlock)$
Meaning: From the *next* event onwards, *eventually* unlock is called

$G(lock \rightarrow XF(unlock))$
Meaning: *Globally* whenever lock is called, then from the *next* event onwards, *eventually* unlock is called

$G(main \rightarrow XG(lock \rightarrow (\rightarrow XF(unlock \rightarrow XF(end)))))$
Meaning: *Globally* whenever main followed by lock are called, then from the *next* event onwards, *eventually* unlock followed by end are called

Table 1: LTL Expressions and their Meanings

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

Table 2: Rules and their LTL Equivalences

To mine for recurrent rules, we propose another new projected database operation to capture events occurring after *each temporal point*. The sequence support (s-support) of a rule $pre \rightarrow post$ is the number of sequences or traces the premise *pre* of the rule occurs (or is satisfied). The instance support (i-support) of a rule $pre \rightarrow post$ is the number of occurrences of $pre{+}{+}post$. The confidence of a rule is the likelihood that each temporal point corresponding to the occurrences of *pre* is followed by the consequent *post*. Rules obeying the minimum thresholds of sequence support (*min_s-sup*), instance support (*min_i-sup*) and confidence (*min_conf*) are referred to as being significant.

The following apriori properties hold and are used to prune the search space containing non-significant rules.

THEOREM 2 (**Apriori Property – S-Support**). *If a rule $evs_P \rightarrow evs_C$ does not satisfy the min_s-sup threshold, neither will all rules $evs_Q \rightarrow evs_C$ where evs_Q is a super-sequence of evs_P.*

THEOREM 3 (**Apriori Property – 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 *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 [32, 30].

DEFINITION 5.2 (**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., $pre_X{+}{+}post_X \sqsubset pre_Y{+}{+}post_Y$)
(2); Both rules have the same supports and confidence values
Also, in the case that 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).

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, we employ several properties to prune the search space containing redundant rules "early" during the mining process – see details in [21].

Our approach to mining a set of non-redundant rules satisfying the supports and confidence thresholds is described, at high-level, as follows:

- **Step 1** First, we generate a *pruned* set (with many redundant pre-conditions removed) of pre-conditions satisfying *min_s-sup*.
- **Step 2** For each pre-condition *pre*, we find all temporal points corresponding to *pre*.
- **Step 3** We then generate a *pruned* set (with many redundant post-conditions removed) containing such post-condition *post*, such that the rule $pre \rightarrow post$ satisfies *min_conf*.
- **Step 4** Checking the rules' instance-supports, we remove rules from step 3 that do not satisfy *min_i-sup*.
- **Step 5** Using Definition 5.2, we filter any remaining redundant rules.

6. PERFORMANCE STUDIES

For iterative pattern and recurrent rule mining, we implement two versions of the mining algorithms. The first mine a closed set of patterns or non-redundant set of rules. The second mine all frequent patterns or all significant rules. The purpose of these studies is to test the scalability of our mining algorithm and the effectiveness of our non-closed pattern/redundant rule pruning strategy

Synthetic data generator provided by IBM was used with modification to ensure generation of sequences of events. The generators accept a set of parameters. The parameters D, C, N and S correspond respectively to the number of sequences (in 1000s), the average number of events per sequence, the number of different events (in 1000s) and the average number of events in the maximal sequences. We experimented with the dataset D5C20N10S20.

Iterative pattern. The results of experiments performed

on the D5C20N10S20 dataset using closed and full-set of frequent iterative pattern miners are shown in Figure 1. The Y-axis (in log-scale) corresponds to the runtime taken or the number of generated patterns. The X-axis corresponds to the minimum support thresholds. The thresholds are reported relative to the number of sequences in the database.

Comparing the results of mining a closed set of patterns that of mining a full set of frequent patterns, we note that for the non-redundant set both the runtime and the number of mined rules were reduced by a large amount: up to *92 times less* for the runtime, and *1250 times less* for the number of mined rules.

Recurrent Rules. Experiments were performed by varying *min_s-sup* & *min_conf* thresholds. Varying the i-support threshold does not affect the runtime because we do not have any pruning property involving the instance support of mined rules. The experiment results for the synthetic dataset are shown in Figure 2 & 3. 'Full' and 'NR' correspond to the full set and non-redundant set of rules respectively. The x-axis of the graph corresponds to the thresholds used while the y-axis represents the runtime required, or the number of mined rules.

Comparing the results of mining a non-redundant set with that of mining a full set of rules, we note that for the non-redundant set both the runtime and the number of mined rules were reduced by a large amount: up to *147 times less* for the runtime, and *8500 times less* for the number of mined rules.

The results above show the effectiveness of our non-closed pattern/redundant rule pruning strategy. For both cases, the algorithms run well even on very low support thresholds. For more details on performance studies conducted including studies on other datasets, please refer to [20, 21].

7. CASE STUDIES

Case studies were performed on the components of JBoss Application Server (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. 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. The details on trace length and parameters used for the experiments are available in [20, 21].

A sample of the mined iterative patterns from transaction component of JBoss AS is shown in Figure 4. The rule read from top to bottom, left to right, specifies a common behavior where: a connection is first set up to the server, the transaction manager is set up, the transaction is set up, the transaction is committed and the transaction is finally disposed.

A sample of the mined recurrent rules (with abbreviated class and method names) from security component of JBoss AS is shown in Figure 5. The rule, read from top to bottom, left to right, describes authentication using Java Authentication and Authorization Service (JAAS) 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.

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

Figure 5: A Rule from JBoss-Security

8. FUTURE WORK

As a future work, we are looking into improving the scalability of the mining algorithms further. Yet additional pruning strategies can be employed to cut down more search spaces and make the mining more scalable to handle large industrial traces.

We are also looking into mining generators of iterative patterns. The set of frequent patterns can be grouped into equivalence classes. Simply put, each class contains patterns having the same support. Generators are minimal members of equivalence classes of frequent patterns. Merging generators with closed patterns potentially form interesting rules with minimal pre-conditions and maximal post-conditions.

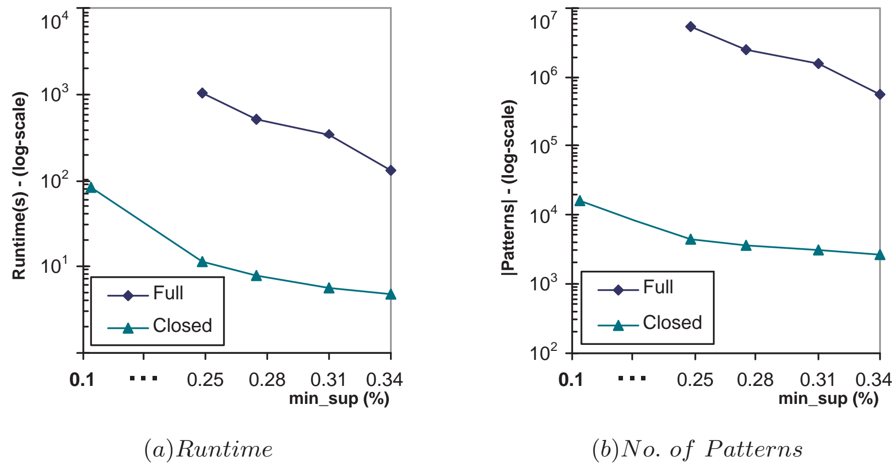
So far we have been looking into rule redundancy via syntactic relationship among rules (i.e., sub-sequence relationship). In the future, we would like to investigate detecting rule redundancy via logical relationship: one rule is redundant if it can be *logically* inferred by another reported rule.

At the moment we only mine for rules that express forward temporal constraints. We are planning to mine rules that express backward and in-between temporal constraints, e.g., whenever a series of events occurs, another series of events must have happened before, etc. We are also planning to mine rules involving negation, disjunction and partial order. These rules can capture more complex constraints in a software system.

In the near future, we plan to do more case studies on more varieties of software systems. We are also planning to build a visualization tool to help user in navigating and visualizing the mined specifications. Integration of the mined specification with automated testing and verification tools are also some things in the pipeline. It will also be interesting to develop a method to rank mined patterns and rules. We also plan to support more user feedback in terms of domain knowledge to the mining process aside from simply minimum support and confidence thresholds.

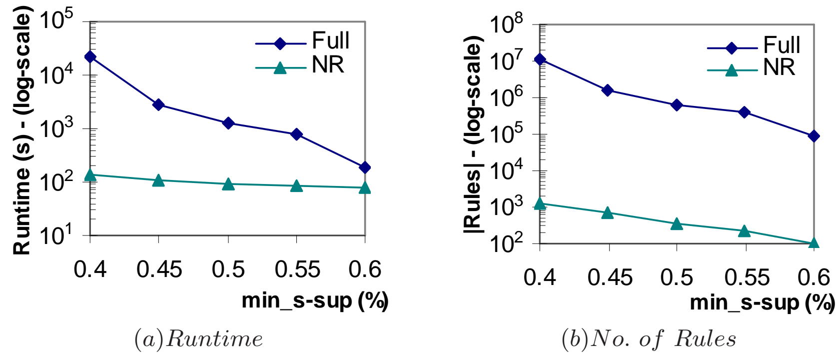
9. CONCLUSION

In this paper, we present novel algorithms to mine for closed frequent *iterative patterns* and non-redundant significant *recurrent rules* from a sequence database corresponding



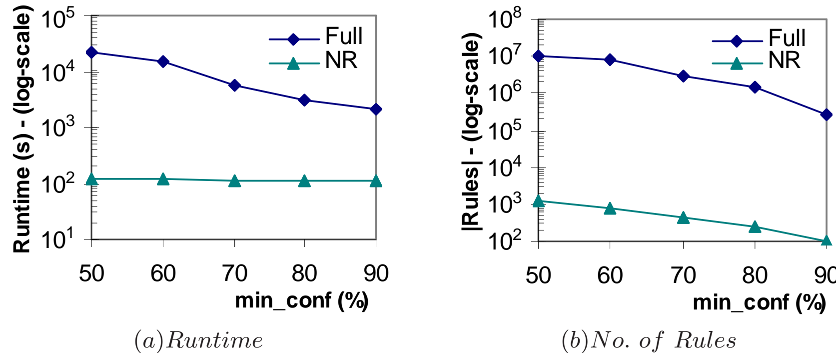
$(a)Runtime$ $(b)No.\ of\ Patterns$

Figure 1: Performance results of iterative pattern mining algorithms



$(a)Runtime$ $(b)No.\ of\ Rules$

Figure 2: Performance results of recurrent rule mining algorithms at *min_conf*=50% and *min_i-sup*=1



$(a)Runtime$ $(b)No.\ of\ Rules$

Figure 3: Performance results of recurrent rule mining algorithms at *min_s-sup*=0.4% and *min_i-sup*=1

to a set of program execution traces. Frequent iterative patterns are patterns that frequently repeat across multiple sequences and within each sequence. Recurrent rules have the form "whenever a series of precedent events occurs, eventually a series of consequent events occurs". Statistics of support and confidence are attached to iterative patterns and recurrent rules to distinguish significant ones. Several apriori properties, non-closed pattern and redundant rule pruning strategies are employed to cut the search space and improve the efficiency of our mining algorithm. Our performance study shows the effectiveness of our pruning strategies in reducing runtime (up to 147 times less) and in removing redundant patterns/rules (up to 8500 times less). Closed iterative patterns and non-redundant recurrent rules can be efficiently mined even at low support thresholds by our proposed mining framework. Case studies on two components of JBoss Application Server show the applicability of our mined patterns and rules in shedding light to program designs & behaviors.

Acknowledgement. Special thanks to Chao Liu who collaborated on the above work.

Connection Set Up
TransactionManagerLocator.getInstance
TransactionManagerLocator.locate
TransactionManagerLocator.tryJNDI
TransactionManagerLocator.usePrivateAPI

Tx Manager Set Up
TxManager.begin
XidFactory.newXid
XidFactory.getNextId
XidImpl.getTrulyGlobalId

Transaction Set Up
TransactionImpl.associateCurrentThread
TransactionImpl.getLocalId
XidImpl.getLocalId

Transaction Set Up (Con't)
LocalId.hashCode
TransactionImpl.equals
TransactionImpl.getLocalIdValue
XidImpl.getLocalIdValue
TransactionImpl.getLocalIdValue
XidImpl.getLocalIdValue

Transaction Commit
TxManager.commit
TransactionImpl.commit
TransactionImpl.beforePrepare
TransactionImpl.checkIntegrity
TransactionImpl.checkBeforeStatus

Transaction Commit (Con't)
TransactionImpl.endResources
TransactionImpl.completeTransaction
TransactionImpl.cancelTimeout
TransactionImpl.doAfterCompletion
TransactionImpl.instanceDone

Transaction Dispose
TxManager.releaseTransactionImpl
TransactionImpl.getLocalId
XidImpl.getLocalId
LocalId.hashCode
LocalId.equals

Figure 4: The Longest Iterative Pattern Mined from JBoss Transaction Component

10. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, 1995.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *SIGPLAN POPL*, 2002.

[3] R.V. Binder. *Testing Object-Oriented Systems Models, Patterns, and Tools*. Addison-Wesley, 2000.

[4] B. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[5] R. Capilla and J.C. Duenas. Light-weight product-lines for evolution and maintenance of web sites. In *CSMR*, 2003.

[6] W-N. Chin, S-C. Khoo, S. Qin, C. Popeea, and H.H. Nguyen. Verifying safety policies with size properties and alias controls. In *ICSE*, 2005.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[8] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3):215–249, July 1998.

[9] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.

[10] S. Deelstra, M. Sinnema, and J. Bosch. Experiences in software product families: Problems and issues during product derivation. In *SPLC*, 2004.

[11] M. El-Ramly, E. Stroulia, and P. Sorenson. From run-time behavior to usage scenarios: an interaction-pattern mining approach. In *KDD*, 2002.

[12] E. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, pages 17–23, 2000.

[13] G.C. Garriga. Discovering unbounded episodes in sequential data. In *PKDD*, 2003.

[14] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge, 2004.

[15] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC). 1999.

[16] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal logic for scenario-based specifications. In *TACAS*, 2005.

[17] M.M. Lehman and L.A. Belady. *Program Evolution - Processes of Software Change.* Academic Press, 1985.

[18] D. Lo and S-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.

[19] D. Lo and S-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.

[20] D. Lo, S-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. *SIGKDD*, 2007.

[21] D. Lo, S-C. Khoo, and C. Liu. Efficient mining of recurrent rules from a sequence database. In *DASFAA*, 2008.

[22] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of frequent episodes in event sequences. *DMKD*, 1:259–289, 1997.

[23] K. Olender and L. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE TSE*, 16:268–280, 1990.

[24] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICDE*, 2001.

[25] S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, 2001.

[26] Java Transaction API Specification. java.sun.com/products/jta/.

[27] M. Spiliopoulou. Managing interesting rules in sequence mining. In *PKDD*, 1999.

[28] T. Standish. An essay on software reuse. *IEEE TSE*, pages 494–497, 1984.

[29] C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns*. Sun Microsystem, 2006.

[30] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, 2004.

[31] W.Weimer and G.Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.

[32] X. Yan, J. Han, and R. Afhar. CloSpan: Mining closed sequential patterns in large datasets. In *SDM*, 2003.

[33] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.

[34] M. Zhang, B. Kao, D.W. Cheung, and K.Y. Yip. Mining periodic patterns with gap requirement from sequences. In *SIGMOD*, 2005.