Efficient Mining of Iterative Patterns for Software Specification Discovery

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

Chao Liu
Department of Computer Science
University of Illinois-UC
chaoliu@cs.uiuc.edu

ABSTRACT

Studies have shown that program comprehension takes up to 45% of software development costs. Such high costs are caused by the lack-of documented specification and further aggravated by the phenomenon of software evolution. There is a need for automated tools to extract specifications to aid program comprehension. In this paper, a novel technique to *efficiently* mine common software temporal patterns from traces is proposed. These patterns shed light on program behaviors, and are termed *iterative patterns*. They capture unique characteristic of software traces, typically not found in arbitrary sequences. Specifically, due to loops, interesting iterative patterns can occur *multiple times* within a trace. Furthermore, an occurrence of an iterative pattern in a trace can extend across a sequence of *indefinite length*.Since a program behavior can be manifested in numerous ways, *analyzing a single trace will not be sufficient*. Iterative pattern mining extends sequential pattern and episode minings to discover frequent iterative patterns which occur repetitively both within a program trace and across multiple traces. In this paper, we present CLIPER (CLosed Iterative Pattern minER) to efficiently mine a *closed* set of iterative patterns. A performance study on several simulated and real datasets shows the efficiency of our mining algorithm and effectiveness of our pruning strategy. Our case study on JBoss Application Server confirms the usefulness of mined patterns in discovering interesting software behavioral specification.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Tools*

General Terms

Algorithms, Performance, Experimentation

Keywords

Closed Iterative Patterns, Software Specification Discovery

SIGKDD'07 August 12–15, 2007, San Jose, California, USA.

1. MOTIVATION AND BACKGROUND

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 [6], 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, 21]. 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 has contributed to high software maintenance costs. It has been investigated that up to 90% of software cost is due to maintenance [12] and 50% of the maintenance cost is due to comprehending or understanding the code base [27] (see also [5]). Hence, approximately 45% of software cost is due to difficulty in comprehending an existing code base. This is especially true for software developed by many developers over a long period of time.

The above needs motivate work on building automated tools to extract or mine specifications from programs. An interesting form of specifications to be mined is patterns of software temporal behaviors.

Our motivating application is in the emerging area of dynamic analysis where program traces (each being a series of method invocations) are analyzed in order to infer or mine temporal program properties or patterns of behavior. Some existing work in this domain includes: [2, 22], which mine temporal program behavioral model expressed as an automata. In this paper, we propose mining interesting program temporal properties expressed as patterns rather than an automata. These patterns are intuitive and commonly found in software documentations, such as:

1. Resource Locking Protocol : $\langle lock, unlock \rangle$
2. Telecommunication Protocol (*c.f.*, [16]): $\langle$*off_hook*, *dial_tone_on*, *dial_tone_off*, *seizure_int*, *ring_tone*, *answer*, *connection_on*$\rangle$
3. Java Authentication and Authorization Service (JAAS) Authorization Enforcer Strategy Pattern (*c.f.*, [28]): $\langle Subject.getPrincipal$, $PrivilegedAction.create$, $Subject.doAsPrivileged$, $JAAS\_Module.invoke$, $Policy.getPermission$, $Subject.getPublicCredential$, $PrivilegedAction.run \rangle$
4. Java Transaction Architecture (JTA) Protocol (*c.f.*, [26]): $\langle TxManager.begin$, $TxManager.commit \rangle$, $\langle TxManager.begin$, $TxManager.rollback \rangle$, *etc.*

Each of these patterns reflecting interesting program behavior can be mined by analyzing a set of program traces – each being a series of method invocations. These traces can in turn be generated through running a test suite. From data mining viewpoint, each trace can be considered a sequence. A pattern (*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* $\rightarrow$ resource use $\rightarrow \ldots \rightarrow$ *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 having the above characteristics from traces, iterative pattern mining is proposed. It leverages the techniques found in sequential pattern mining and episode mining to handle 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 pattern is supported by a sequence if it is a sub-sequence of it. It has application 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* [23]. 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*. Similar to sequential pattern mining, we consider a *database of sequences* rather than a single sequence. However, we also mine patterns 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 [32] or generated statically from the source code [30]. Secondly, important patterns for verification, such as, lock acquire and release or stream open and close (*c.f* [32, 7]) 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 of interest. Interestingly, these two notable differences between analysis of events from an alarm management system and program traces are observed by sequential pattern miner first introduced in [1].

To support iterative pattern mining, we need a clear definition and semantics of iterative pattern different from episodes and sequential patterns. Our definition of iterative pattern is inspired by the common languages for specifying software behavioral requirements, namely Message Sequence Chart (MSC) [16] and Live Sequence Chart (LSC) [9].

MSC and LSC are variants of sequence diagram specifying how a system should behave. An example of such chart is a simplified telephone switching protocol (*c.f.*, [16]). Abstracting caller and callee information, it can be represented as a pattern: $\langle$*off_hook, dial_tone_on, dial_tone_off, seizure_int, ring_tone, answer, connection_on*$\rangle$. Such protocol must possess a total-ordering property and satisfy one-to-one correspondence requirements between events in the chart and events in a trace segment satisfying the chart. (Please refer to Section 3.2 for detail.)

The full language of MSC/LSC is complicated and it is not our intention to mine MSC/LSC. In this paper, iterative pattern mined abstracts away the caller and callee information but ensures total-ordering property and one-to-one correspondence between a pattern and its instance (*i.e.*, a segment of a trace).

Pattern mining in general is an NP-hard problem. For it to be practical, efficient search space pruning strategies need to be employed. One of the most important property to help in ensuring scalability is the apriori property. There are several variants of it. Iterative pattern obeys the following apriori property utilized by depth-first search sequential pattern miners (*e.g.*, FreeSpan [14] and PrefixSpan [25]) which states:

If P is not frequent then P++evs (where evs is a series of events) is also not frequent.

Apriori property holds for both sequential patterns and episodes. To ensure efficiency, it is desirable to maintain this property for iterative patterns. Fortunately, the formulation of iterative pattern guarantees this property as described in Section 3.

Due to possibly combinatorial number of frequent subsequences of a long pattern, it's best to mine a closed set of patterns (*c.f.*, [31] & [29]). Closed pattern mining discovers patterns without any super-sequence having corresponding set of instances. Resultant pattern set is likely to be *more compact and yet still complete* (*i.e.* every frequent pattern is represented by a closed pattern). Closed pattern mining can also lead to more efficient pattern mining strategy. *Early identification and pruning* of non-closed patterns can reduce the runtime significantly.

In this paper, we mine a *closed* set of iterative patterns. A search space pruning strategy employed by *early identification and pruning* of non-closed patterns is used to mine a closed set of iterative patterns efficiently. Our performance study on synthetic and real-world datasets shows the major success of our pruning strategy: it runs with over an order of magnitude speedup especially on low support thresholds or when the frequent patterns are long.

As a case study we experimented with traces collected from transaction sub-component of JBoss Application Server. Our mined patterns highlight important design patterns shedding light on program behavior.

The contributions of this work are as follows:

1. We propose an *efficient* algorithm to mine a *closed* set of software iterative patterns from program execution traces.
2. We present a novel formulation of iterative pattern inspired by standards adopted for specifying software behavioral requirements (*i.e.*, MSC and LSC).

3. We extend episode mining by: (1) analyzing multiple sequences, (2) removing the 'window' barrier and (3) extracting a closed set of patterns for software specification mining purpose.
4. We extend closed pattern mining by considering repeated pattern occurrences within a sequence and across multiple sequences for software specification mining purpose.

The outline of the paper is as follows: We present related work in Section 2. Section 3 provides an in-depth discussion on semantics of iterative pattern. Section 4 presents the principles behind the generation of *closed iterative patterns* and its associated pruning strategy. Section 5 describes our closed pattern mining algorithm. Section 6 presents the results of our performance study. Section 7 discusses a case study on mining program behavioral design from traces of JBoss Application Server. We conclude in Section 8.

2. RELATED WORK

Our work is a variant 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.* [31] and later improved by Wang and Han [29]. Different from sequential pattern, our pattern capture multiple occurrences of pattern not only *across multiple sequences* but also those repeated *within each sequence*. In this aspect, iterative pattern mining resembles episode mining initiated by Mannila *et al.* [23] 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* with 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* [32, 7]), often have their events occur at some arbitrary distance away from one another in a trace. In addition, both versions of episode mining handle only one single sequence, whereas iterative pattern mining operates over a set of sequences.

In mining DNA sequences, Zhang *et al.* introduced the idea of "gap requirement" in mining periodic patterns from sequences [33]. 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 [33] 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, all with length less than n, where n is a user defined parameter. The appropriate value of this parameter n might not be obvious to the user.

In the software engineering domain, Yang *et al.* mined a restricted form of two-event temporal rules, instead of patterns, from program traces [32]. To handle more than two events, they proposed concatenation of 2-event rules to form longer ones. Unfortunately, this method is not sound as only an approximation to significance values of reported rules is made. It is also not complete since potentially many of the more-than-two-event rules cannot be generated by simple concatenation of 2-event rules.

In a similar domain, 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 ours, 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.

Firstly, 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. Consequently, the apriori property is not observed by interaction patterns – a pattern can have a larger support than its subsequences. In contrast, iterative patterns observe the apriori property.

Secondly, 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.

3. ITERATIVE PATTERNS

In this section, we define formally iterative pattern, and provide the reasoning behind its semantics.

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 $SeqDB$.

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 $1 \leq i_1 < i_2 < i_3 < i_4 \ldots < i_n \leq m$ 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 $SeqDB$.

DEFINITION 3.1 (**Concatenation and Truncation**). *Concatenation of two patterns P_1 ($\langle a_1, \ldots, a_n \rangle$) and P_2 ($\langle b_1, \ldots, b_m \rangle$) will result in a longer pattern P_3 ($\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$). Truncation operation is only applicable between a pattern and its suffix. Truncation of a pattern P_3 ($\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle$) and a suffix P_2 ($\langle b_1, \ldots, b_m \rangle$) will result in the pattern P_1 ($\langle a_1, \ldots, a_n \rangle$). Patterns concatenation is denoted by $++$, while pattern truncation is denoted by $--$.*

Another important operation used in this work is the erasure operation, as defined below.

DEFINITION 3.2 (**Erasure Operator**). *Given a pattern P ($\langle p_1, p_2, \ldots, p_n \rangle$) and a string S ($\langle s_1, s_2, \ldots, s_m \rangle$), the erasure of S wrt. P, denoted by $erasure(S, P)$, is defined as a*

new string S_{erased} formed from S where all events occurring in P are removed from S. Formally, S_{erased} is defined as $(\langle se_1, se_2, \ldots, se_k\rangle)$ such that (1) $\forall i.se_i \notin P$ and (2) there exists a set of integers $\{i_1 \ldots i_k\}$ with $1 \leq i_1 < i_2 < i_3 < i_4 \ldots < i_k \leq m$ and $se_1 = s_{i_1}$, $se_2 = s_{i_2}$, $\cdots$, $se_k = s_{i_k}$ and $\forall j \notin \{i_1 \ldots i_k\}, s_j \in P$.

3.2 Semantics of Iterative Patterns

Our definition of iterative pattern is inspired by the common languages for specifying software behavioral requirement: Message Sequence Chart (MSC) (a standard of International Telecommunication Union (ITU) [16]) and its extension, Live Sequence Chart (LSC) [9].

MSC and LSC is a variant 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 specifies caller and callee information. An example of such charts is a simplified telephone switching protocol (*c.f.*, [16]): abstracting caller and callee information, the protocol can be represented as a pattern: $\langle$*off_hook*, *dial_tone_on*, *dial_tone_off*, *seizure_int*, *ring_tone*, *answer*, *connection_on*$\rangle$.

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 $k < i$ [16]. 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 [20]. 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 *ring-tone* event. The second doesn't satisfy the one-to-one correspondence requirement due to multiple occurrences of *answer* event.

The full language of MSC/LSC is complicated and it is not our intention to mine MSC/LSC. Iterative pattern abstracts away the caller and callee information but retains the uniqueness and total ordering requirements.

The pattern instance definition capturing the total-ordering and one-to-one correspondence between events in the pattern and its instance can be expressed unambiguously in the form of Quantified Regular Expression (QRE) [24]. 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 3.3 (**Pattern Instance - QRE**). *Given a pattern P $(p_1p_2 \ldots p_n)$, a substring SB $(sb_1sb_2 \ldots sb_m)$ of a sequence S in SeqDB is an instance of P iff it is of the following QRE expression*

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

Operationally we use an equivalent definition of pattern instance described using the erasure operation:

DEFINITION 3.4 (**Iterative Pattern Instance**). *Given a pattern P $(p_1p_2 \ldots p_n)$, a substring SB $(sb_1sb_2 \ldots sb_m)$ of a sequence S in SeqDB is an iterative pattern instance of P iff (1) $first(P) = first(SB)$, (2) $last(P) = last(SB)$ and (3) the following erasure constraint holds:*

$$erasure(SB, erasure(SB, P)) = P.$$

We use the term "pattern instance" and "iterative pattern instance" interchangeably in this paper. The operation $erasure(SB, erasure(SB, P))$ basically removes all events that occur in SB but not in P. An iterative pattern is thus identified by a set of iterative pattern instances, which can occur repeatedly in a sequence as well as across sequences. We also use the term "pattern" and "iterative pattern" interchangeably.

An instance is denoted compactly by a triple (s_{idx}, i_{start}, i_{end}) where s_{idx} refers to the sequence index of a sequence S in the database while i_{start} and i_{end} refer to the starting point and ending point of a substring in S. By default, all indices start from 1. With the compact notation, an instance is both a string and a triple – the representations are used interchangeably. The set of all instances of a pattern P in a database DB is denoted as $Inst(P, DB)$. Reference to the database is omitted if it refers to the input sequence database.

As an example, consider a pattern P ($\langle A, B\rangle$) and a database consisting of two sequences:

Identifier	Sequence
$S1$	$\langle D, B, A, B, A, B, C, E\rangle$
$S2$	$\langle D, B, A, B, B, B, A, B\rangle$

The set $Inst(P)$ is the set of triples {(1,3,4),(1,5,6),(2,3,4), (2,7,8)}.

There is a one-to-one ordered correspondence between events in the pattern and events in its instance. This one-to-one correspondence can be captured by the concept of pattern instance landmarks defined below.

DEFINITION 3.5 (**Pattern Instance Landmarks**). *Given a pattern P $(p_1p_2 \ldots p_n)$, an instance I $(s_1s_2 \ldots s_m)$ of pattern P has the following landmarks: $l_1, l_2, \ldots l_n$ where $1 \leq l_1 < l_2 < \ldots < l_n \leq m$ and $s_{l_1} = p_1, s_{l_2} = p_2, \ldots, s_{l_n} = p_n$. Due to erasure constraint, for each instance there is only one such set of landmarks. The landmarks of an instance I is denoted as Lnd(I). The ith member of the set Lnd(I) is called the ith landmark.*

The *support* of a pattern *wrt.* to a sequence database $SeqDB$ is the number of its instances in $SeqDB$. A pattern P is considered *frequent* when its support, $sup(P)$, exceeds a certain threshold ($min\_s\text{-}sup$).

3.3 Apriori Property and Closed Pattern

Iterative patterns possess the following *'apriori' property* used in PrefixSpan [25]:

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

In general, iterative pattern does not possess the apriori property used in GSP [1]: if a pattern is frequent so does its

sub-sequences. However, considering patterns having corresponding instances as described in Definition 3.6 below, the GSP apriori property holds. It is restated in Theorem 2.

DEFINITION 3.6 (**Corresponding Pattern Insts**). *Consider a pattern P and its super-sequence Q. Instance I_P (seq_P, $start_P$, end_P) of P corresponds to an instance I_Q (seq_Q, $start_Q$, end_Q) of Q iff $seq_P = seq_Q$ and $start_P \geq start_Q$ and $end_P \leq end_Q$.*

THEOREM 2 (**Apriori Property - GSP-Like**). *If a pattern Q is frequent and P is a sub-sequence of Q, then either P is frequent or every instance of Q do not correspond to any instance of P (and vice versa).*

DEFINITION 3.7 (**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 $end_P \leq end_Q$.

The second condition of the above definition is to prevent the following case from happening.

Identifier	Sequence
$S1$	$\langle A, B, B, A\rangle$
$S2$	$\langle A, B, A\rangle$

Consider the above sequence database. The only instance of the pattern $\langle A, B, A\rangle$ is (2,1,3), while the only instance of pattern $\langle A, B, B, A\rangle$ is (1,1,4). Both have the same support. However, since their instances match *different segments* of the sequences they should be reported separately. $\langle A, B, A\rangle$ is not "absorbed" by $\langle A, B, B, A\rangle$ and is thus closed.

Notation-wise, we denote the full set of closed iterative patterns mined from *SeqDB* by *Closed*. We consider the following problem: *Given a sequence database, find a closed set of iterative patterns.*

4. GENERATION OF ITERATIVE PATTERNS

Iterative pattern instances can be mined using depth first pattern growth and prune strategy (*c.f.*, FreeSpan [14] and PrefixSpan [25]). However, rather than using the usual projection that extracts sequential patterns, we perform a different type of projection outlined below.

DEFINITION 4.1 (**Projected-all**). *A database SeqDB* projected-all *on a pattern P results in a set of pairings and is denoted as $SeqDB_P^{all}$. It is defined recursively as follows.*

Base Case: if P is a single event ev
$\{(ev, sx) \mid \exists s \in SeqDB, ev{+}{+}sx$ *is a suffix of* $s\}$
Inductive Case: if P is multi-events
$\{(ox{+}{+}px{+}{+}last(P), sx) \mid$ $\exists(ox, (px{+}{+}last(P){+}{+}sx)) \in SeqDB^{all}_{P--last(P)}\cdot$ $((last(P) \notin erasure(ox, P--last(P))) \wedge$ $(\forall ev \in P, ev \notin px))\}$

The definition of projected-all database captures pattern instances that possibly occur repeatedly within a sequence and across multiple sequences. The first element of the pairings corresponds to pattern instances in string format. The second element corresponds to the remaining part of the sequence providing the context from which the pattern can still be extended. Support of a pattern P is equal to the number of instances supporting P, denoted as $|Inst(P, SeqDB)|$. In turn, $|Inst(P, SeqDB)|$ is equal to the size of the projected database $|SeqDB_P^{all}|$.

The instances of a length-1 pattern $\langle e_1\rangle$ is simply the occurrences of event e_1 throughout the sequences in $SeqDB$. The instances of a length-k $\langle e_1, \ldots, e_k\rangle$ pattern can be found from instances of length-(k-1) $\langle e_1, \ldots, e_{k-1}\rangle$ pattern.

Instances of a length-2 pattern $\langle e_1, e_2\rangle$ can be formed by extending instance pairings of $\langle e_1\rangle$, (ox, ss) in $SeqDB^{all}_{\langle e_1\rangle}$, on the condition: $\exists i.ss[i] = e_2 \wedge \forall j < i, ss[j] \notin \{p_1, p_2\}$. This condition corresponds to the second conjunctive clause of the inductive case of Definition 4.1. The first conjunctive clause in the definition is trivially satisfied since the erasure of a length-1 pattern instance is an empty string.

Similarly, instances of a length-3 pattern $\langle e_1, e_2, e_3\rangle$ can be formed by extending instance pairings of $\langle e_1, e_2\rangle$, (ox, ss) in $SeqDB^{all}_{\langle e_1, e_2\rangle}$, on the conditions:(1) $e_3 \notin erasure(ox, \langle e_1, e_2\rangle)$ and (2) $\exists i.ss[i] = e_3 \wedge \forall j < i, ss[j] \notin \{e_1, e_2\}$. The first and second conditions correspond respectively to the two conjunctive clauses of the inductive case. The first condition is necessary since a substring instance ox of a length-2 pattern $\langle e_1, e_2\rangle$ only obeys the erasure constraint for the original pattern – ox might contain e_3.

Generalizing the above, instances of a length-k pattern can be formed from instances of a length-(k-1) pattern, by following the inductive case of Definition 4.1.

A simple depth-first algorithm to generate a full-set of iterative patterns is as follows. First, generate a set of length-1 patterns where the support of each is greater than the *min_sup* threshold. A projected-all database can then be created from the set of frequent length-1 patterns according to the base case of Definition 4.1. Instances of a length-2 pattern can then be obtained by performing the inductive step of Definition 4.1 to the corresponding length-1 pattern projected database. Pattern not satisfying *min_sup* will be pruned. Since patterns obey *apriori* property, we can stop extending pruned patterns. Length-(i+1) patterns can be obtained from length-(i) patterns accordingly.

For ease of explanation, let's represent the inductive step of Definition 4.1 with the following *Projected-first* projection and the related *Seq* operator.

DEFINITION 4.2 (**Projected-first & Seq**). *A projected database $SeqDB_P^{all}$ can be* projected-first *on an event e resulting in a set of pairings and denoted as $(SeqDB_P^{all})_e^{fst}$. It is defined as the following set.*

$$\{(ox{+}{+}px{+}{+}e, sx) \mid \exists(ox, (px{+}{+}e{+}{+}sx)) \in SeqDB_P^{all}. \\ (e \notin erasure(ox, P)) \wedge (\forall ev \in (P{+}{+}e), ev \notin px))\}$$

We denote the size of $(SeqDB_P^{all})_e^{fst}$ as $Seq(e, SeqDB_P^{all})$.

The above operation locates the first instance of an event e in the projected database – hence the name *projected-first*. It computes the *sequences* in projected database supporting event e – hence the name *Seq* operator. However, constraints corresponding to the inductive step of Definition 4.1 is also added to ensure $(SeqDB_P^{all})_e^{fst} = SeqDB^{all}_{P{+}{+}e}$.

We also define the following two operations of equivalence of projected databases and inclusion of an event in a projected database.

DEFINITION 4.3 (**Operations on Projected DB**). *Projected databases DB_1 and DB_2 are equivalent (denoted*

as $DB_1 = DB_2$) iff $|DB_1| = |DB_2|$ and $\forall$ $(p_1, s_1) \in DB_1$. $\exists$ $(p_2, s_2) \in DB_2$. $s_1 = s_2$. Also, an event e is in a projected database DB (denoted as $e \in DB$) iff $\exists (p, s) \in DB$. e is an event in s.

Consider the following running example. Let us have the following sequence database $SeqDB$ shown in Table 1.

Identifier	Sequence
$S1$	$\langle A, B, A, B, A, B, C, D, E\rangle$
$S2$	$\langle A, B, B, B, B\rangle$
$S3$	$\langle A, B, C, A, D, E, B, C\rangle$
$S4$	$\langle A, B, C, C, A, B\rangle$

Table 1: Sample $SeqDB$

Support of pattern $\langle A, B, C\rangle$ can be found by first constructing the projected database of $\langle A\rangle$. This is shown below in Table 2

Instance	Remainder of Sequence
$(1, 1, 1)$	$\langle B, A, B, A, B, C, D, E\rangle$
$(1, 3, 3)$	$\langle B, A, B, C, D, E\rangle$
$(1, 5, 5)$	$\langle B, C, D, E\rangle$
$(2, 1, 1)$	$\langle B, B, B, B\rangle$
$(3, 1, 1)$	$\langle B, C, A, D, E, B, C\rangle$
$(3, 4, 4)$	$\langle D, E, B, C\rangle$
$(4, 1, 1)$	$\langle B, C, C, A, B\rangle$
$(4, 5, 5)$	$\langle B\rangle$

Table 2: Sample $SeqDB^{all}_{\langle A\rangle}$

The projected database $SeqDB^{all}_{\langle A,B\rangle}$ can then be constructed from $SeqDB^{all}_{\langle A\rangle}$ using the inductive step of Definition 4.1. Equivalently, we are applying the projected-first operation to the $SeqDB^{all}_{\langle A\rangle}$ with respect to event B. The result is shown below in Table 3.

Instance	Remainder of Sequence
$(1, 1, 2)$	$\langle A, B, A, B, C, D, E\rangle$
$(1, 3, 4)$	$\langle A, B, C, D, E\rangle$
$(1, 5, 6)$	$\langle C, D, E\rangle$
$(2, 1, 2)$	$\langle B, B, B\rangle$
$(3, 1, 2)$	$\langle C, A, D, E, B, C\rangle$
$(3, 4, 7)$	$\langle C\rangle$
$(4, 1, 2)$	$\langle C, C, A, B\rangle$
$(4, 5, 6)$	$\langle\rangle$

Table 3: Sample $SeqDB^{all}_{\langle A,B\rangle}$

Finally, performing the inductive step of Definition 4.1 to $SeqDB^{all}_{\langle A,B\rangle}$ will result in $SeqDB^{all}_{\langle A,B,C\rangle}$ from which support of $\langle A, B, C\rangle$ can be found. Equivalently, we apply the projected-first projection to $SeqDB^{all}_{\langle A,B\rangle}$ with respect to event C. The projected database is as shown below in Table 4.

Instance	Remainder of Sequence
$(1, 5, 7)$	$\langle D, E\rangle$
$(3, 1, 3)$	$\langle A, D, E, B, C\rangle$
$(3, 4, 8)$	$\langle\rangle$
$(4, 1, 3)$	$\langle C, A, B\rangle$

Table 4: Sample $SeqDB^{all}_{\langle A,B,C\rangle}$

The support of $\langle A, B, C\rangle$ is then given by the size of $SeqDB^{all}_{\langle A,B,C\rangle}$ which is 4: one from $S1$, two from $S3$ and another one from $S4$ in $SeqDB$.

Generating a full-set of iterative patterns results in many "redundancies". As all subsequences of a frequent iterative pattern P having corresponding instances are frequent, the number of frequent patterns is potentially exponential to the maximum length of the iterative patterns. Mining for closed patterns is an effective solution. Besides reducing the final number of patterns, closed pattern mining can usually reduce run-time by pruning the search space.

DEFINITION 4.4 (**Prefix Extension Events**). *For a pattern P, its set of prefix extension events is defined as the set of length-1 items e where $sup(e{+}{+}P) = sup(P)$.*

DEFINITION 4.5 (**Infix Extension Events**). *An event e is an infix extension of a pattern P iff $\exists$ a super-sequence Q where: (1) $SeqDB^{all}_P = SeqDB^{all}_Q$, (2) first(P) = first(Q), (3) $\forall$ event $ev1 \in erasure(Q, P)$. $ev1 = e$, (4) sup(P) = sup(Q), and (5) Every instance of P corresponds to a unique instance of Q.*

DEFINITION 4.6 (**Suffix Extension Events**). *For a pattern P, its set of suffix extension events is defined as the set of length-1 items e where $sup(P{+}{+}e) = sup(P)$.*

Prefix/ suffix extension events define events that can be added as prefix/ suffix (of length 1) to a pattern and results in another pattern having the same support[1]. Infix extension events define events that can be added as infix to a pattern and results in another pattern having the same support and corresponding instances.

As an example, consider the sample database in Table 1. For pattern $\langle D\rangle$, its set of prefix extension events is $\{\langle A, B, C\rangle\}$. For pattern $\langle A, C\rangle$, its set of infix extension events is $\{\langle B\rangle\}$. For pattern $\langle A\rangle$, its set of suffix extension events is $\{\langle B\rangle\}$.

The above definitions are used in the next two theorems, which are then used for incremental and early detection of closed patterns and early pruning of search space.

THEOREM 3 (**Extension Closure Checks**). *If there exists no prefix, infix and suffix extension event w.r.t. a pattern P, P must be a closed pattern; otherwise P must be non-closed.*

PROOF. Part 1: If there exists a prefix, infix or suffix extension event, then P must be non-closed.

Consider a pattern P (where $|P|$ = n). If there exists a suffix extension event e, there exists another pattern Q $(P{+}{+}e)$ having the same support and a corresponding set of instances as P.

Patterns P and Q have corresponding set of instances due to the following. The region from the 1st to the n^{th} landmark of an instance of Q is an instance of P. Hence, every instance of Q matches an instance of P. Also, if $sup(P) = sup(P{+}{+}e)$, we have every instance of P matches an instance of Q as well. They have corresponding instances.

Similarly, if there exists a prefix extension event e, there exists another pattern Q $(e{+}{+}P)$ having the same support and a corresponding set of instances as P. Hence, if there exists a prefix or suffix extension event for P, we can create a super-sequence of P having the same support and a corresponding set of instances (*i.e.* P is not closed).

[1]Patterns $e{+}{+}P$ and $P{+}{+}e$ will have corresponding instances as P iff $sup(e{+}{+}P) = sup(P)$ and $sup(P{+}{+}e) = sup(P)$ respectively – see proof of Theorem 3

Consider a pattern P. If there exists an infix extension event e, we can create another pattern Q super-sequence of P having the same support and corresponding instances. Hence, P is not closed.
Part 2: If there exists no prefix, suffix and infix extension event P must be closed.

We can only grow a pattern by adding prefix, infix and suffix to it. Hence, if we cannot find a prefix, infix and suffix extension event of a pattern P resulting in its super-sequence having the same support, P must be closed.

It is enough to consider a single event extension since *apriori* property holds for patterns having corresponding instances. □

As an example, consider the sample database in Table 1. For pattern $\langle A, B, C\rangle$, its sets of prefix, suffix and infix extension events are empty. We can conclude that the pattern $\langle A, B, C\rangle$ is closed. On the other hand, for pattern $\langle A\rangle$, its set of suffix extension events is not empty. Hence it is not closed since there exists a pattern $\langle A, B\rangle$ which is a super-sequence of $\langle A\rangle$ with the same support.

THEOREM 4 (**InfixScan Search Space Pruning**). *Given a pattern P, if there exists an infix extension event e w.r.t. a pattern P and $e \notin SeqDB_P^{all}$, we can safely stop growing pattern P.*

PROOF. From Definition 4.5, if a pattern P has an infix extension event e, there exists a super-sequence pattern Q where: (1) $SeqDB_P^{all} = SeqDB_Q^{all}$, (2) $\forall$ event $ev1 \in erasure(Q, P)$. $ev1 = e$, (3) $\sup(P) = \sup(Q)$, and (4) Every instance of P corresponds to a unique instance of Q.

Since $SeqDB_P^{all} = SeqDB_Q^{all}$, if we can extend an instance s_x in $Inst(P)$ (and also in $Inst(Q)$) with a substring s_{ext} where $erasure(s_x{+\!+}s_{ext}, erasure(s_x{+\!+}s_{ext}, P{+\!+}s_{ext}))$ $= P{+\!+}s_{ext}$, $erasure(s_x{+\!+}s_{ext}, erasure(s_x{+\!+}s_{ext}, Q{+\!+}s_{ext}))$ will also be equal to $Q{+\!+}s_{ext}$.

Since e is not in $SeqDB_P^{all}$, whenever $P{+\!+}s_{ext}$ violate erasure constraint so does $Q{+\!+}s_{ext}$.

Thus, given an arbitrary series of events s_{ext}, if $P{+\!+}s_{ext}$ is frequent, there exists another pattern $Q{+\!+}s_{ext}$ having the same support and corresponding instances. Hence, any pattern having P as prefix will not be closed. We can stop growing pattern P. □

As an example, consider the sample database in Table 1. For pattern $\langle A, C\rangle$, its set of infix extension events is $\{B\}$. There is no point extending pattern $\langle A, C\rangle$ further. Take for example pattern $\langle A, C, D\rangle$ of support 1. It is not closed since, there exists pattern $\langle A, B, C, D\rangle$ which is a super-sequence and has the same support and corresponding instances as the pattern $\langle A, C, D\rangle$.

The next section outlines our algorithm utilizing the above closure checks and InfixScan search space pruning for efficient memory and time utilization and for pruning of redundant search space.

5. ALGORITHM

Our CLIPER (CLosed Iterative Pattern minER) algorithm is shown in Figure 1. The main procedure to compute the closed set of iterative patterns: **MinePatterns**, is shown at the top of the figure. It will call a recursive procedure **MineRecurse** shown at the bottom of the figure.

Procedure **MinePatterns** will first find patterns of length one whose instances are more than or equal to *min_sup* threshold. For all frequent length-1 patterns, it will then call the procedure **MineRecurse** to recursively grow each patterns.

Procedure MinePatterns
Inputs:
$SeqDB$: Sequence Database
min_sup: Minimum Support Threshold
Outputs:
$Closed$: Closed Iterative Patterns
Methods:
1: Let $Freq = \{p | (|p|{=}1) \wedge (|Inst(p, ProjDB)| \geq min\_sup)\}$
2: Let $Closed = \{\}$
3: For every f_ev in $Freq$
4: Call MineRecurse $(f\_ev, SeqDB_{f\_ev}^{all}, min\_sup, Closed, Freq)$
5: End For
Output $Closed$

Procedure MineRecurse
Inputs:
Pat : Pattern so far
$SeqDB_{Pat}^{all}$: Sequence Database
min_sup: Minimum Support Threshold
$Closed$: Current Set of Closed Iterative Patternss
EV: Set of Frequent Events
Methods:
6: Let $Freq = \{e | e \in EV \wedge (Seq(e, SeqDB_{Pat}^{all}) \geq min\_sup)\}$
7: If $(PreExt(Pat) = \{\} \wedge SufExt(Pat) = \{\})$
8: Add Pat to $Closed$
9: End If
10: For every f_ev in $Freq$
11: Let $NxtPat = Pat{+\!+}f\_ev$
12: Let $ProjDB = (SeqDB_{Pat}^{all})_{f\_ev}^{fst}$
13: If $(\nexists\ e.\ (e \in InfixExt(NxtPat) \wedge e \notin ProjDB))$
14: Call MineRecurse $(NxtPat, ProjDB, min\_sup, Closed, EV)$
15: End If
16: End For

Figure 1: CLIPER Algorithm

The recursive algorithm **MineRecurse**, shown at the bottom of Figure 1, will have as inputs the pattern prefix computed so far (Pat), the projected-all sequence database ($SeqDB_{Pat}^{all}$), the support threshold, the data structure containing current set of closed patterns ($Closed$) and the set of frequent events.

The algorithm will first find length-1 event e where $Pat{+\!+}e$ is frequent. Given the input pattern Pat and an event e, the number of instances of $Pat{+\!+}e$ is equivalent to the number of pairings (px, sx) in $SeqDB_{Pat}^{all}$ where we can extend px to an instance of $Pat{+\!+}e$. The above is equivalent to $Seq(e, SeqDB_{Pat}^{all})$.

A set of prefix extension events of Pat is the set of such event e where $sup(e{+\!+}Pat) = sup(Pat)$. A set of suffix extension events of Pat is the set of such event e where $sup(Pat{+\!+}e) = sup(Pat)$.

Only such pattern Pat without any infix extension events will be an input to the recursive algorithm. Hence, it is only necessary to check for the existence of any suffix and prefix extension events. If there isn't any, by Theorem 3, we can add the pattern Pat to the set $Closed$.

Next, for any frequent pattern $Pat{+\!+}e$, following Theorem 4, we check for its infix extension events. If there is an infix extension event which does not appear in $SeqDB_{Pat{+\!+}e}^{all}$, we do not need to grow the pattern $Pat{+\!+}e$ anymore.

Growing patterns is performed recursively. At each step, given an extension event e, the *projected-all* database of $SeqDB^{all}_{Pat++e}$ need to be computed. It can be computed incrementally by taking the projected-first database of $SeqDB^{all}_{Pat}$ (*i.e.* $(SeqDB^{all}_{Pat})^{fst}_{e}$).

6. PERFORMANCE STUDY

Experiments had been performed on both synthetic and real datasets to evaluate the scalability of our mining algorithm and the effectiveness of our pruning strategy. Similar to work in closed sequential pattern mining [31, 29], low support thresholds are utilized to test for scalability.

Datasets. We use three datasets in our experiments: a synthetic and two real datasets. 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.

We also experimented on click stream dataset (*i.e.*, Gazelle dataset) from KDD Cup 2000 [19] which was also used to evaluate CloSpan [31] and BIDE [29]. It contains 29369 sequences with an average length of 3 and a maximum length of 651.

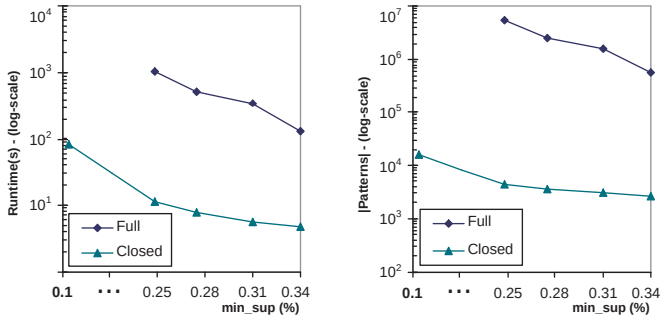
To evaluate our algorithm performance on mining from program traces, we generate traces from a simple Traffic alert and Collision Avoidance System (TCAS) from the Siemens Test Suite [15], which has been used as one of the benchmarks for research in error localization (*e.g.*, [8]). The test suite comes with 1578 correct test cases. We run these test cases to obtain 1578 traces.

To test for scalability, instead of tracing method invocations, we trace executions of basic blocks of TCAS's control flow graph. A basic block is a maximal sequence of statements such that the execution of one statement will always results in the execution of the subsequent statements in the sequence. Each trace of basic block ids is treated as a sequence. The sequences are of average length of 36 and maximum length of 70. It contains 75 different events – the events are the basic block ids of the control flow graph of TCAS. We call this dataset the TCAS dataset.

Environment and Pattern Miners. All experiments were performed on a Pentium 4 3.0GHz PC with 2GB main memory running Windows XP Professional. Algorithms were written using Visual C#.Net running under .Net Framework 2.0 with generics compiled with the release option using Visual Studio.Net 2005.

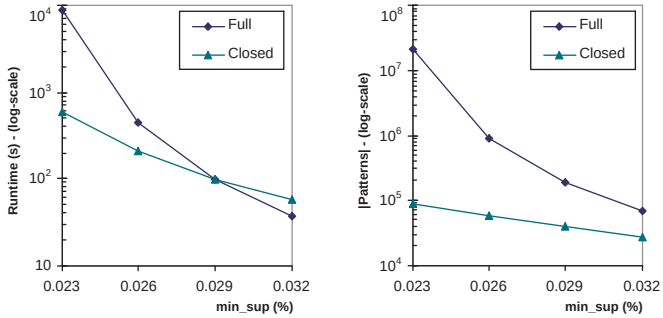
For the experiments we tested our pattern miner on two configurations to test the effectiveness of our pruning strategy. The first mines a closed set of iterative patterns while another mines a full set of iterative patterns. Let's refer the earlier as *closed* iterative pattern miner and the latter as *full-set* iterative pattern miner.

Experiment Results and Analysis. The results of experiments performed on the D5C20N10S20, Gazelle and Siemens dataset using closed and full-set iterative pattern miners are shown in Figures 2, 3 & 4 respectively. 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. Note that, different from sequential patterns, due to repeated patterns within a sequence this number can exceed 1.



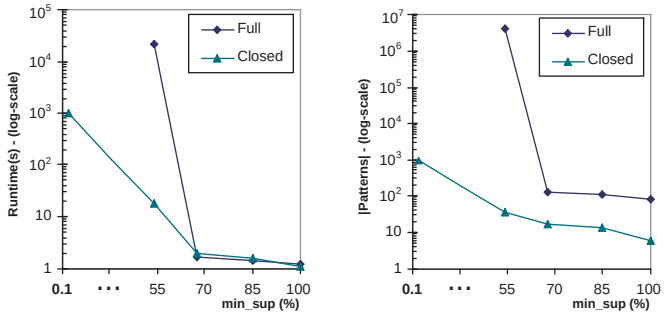
(a)*Runtime* (b)*No. of Patterns*

Figure 2: Performance results of varying *min_sup* for D5C20N10S20 dataset



(a)*Runtime* (b)*No. of Patterns*

Figure 3: Performance results of varying *min_sup* for Gazelle dataset



(a)*Runtime* (b)*No. of Patterns*

Figure 4: Performance results of varying *min_sup* for TCAS dataset

From the plotted results it is noted that the pruning strategy significantly reduces the runtime and the number of patterns mined especially on low support threshold and when the reported patterns are long. Admittedly, when the numbers of closed and full-set of patterns differ by only a small factor, the overhead of mining closed patterns may result in longer runtime as compared to mining a full-set of patterns.

However, when the length of the patterns is long, the number of closed patterns is likely to be much less than that of a full-set of patterns.

For all datasets, even at very low support, closed pattern miner is able to complete within less than 17 minutes. TCAS dataset especially highlights performance benefit of our pruning strategy. Closed iterative pattern miner is able to run even at the *lowest possible support threshold* (at 1 instance) within less than 17 minutes. On the other hand, full-set iterative pattern miner runs with excessive runtime ($>$ 6 hours) even at a relatively high support threshold of 867 instances.

The above shows that our miner can efficiently perform its task on various benchmark data. Comparison of performance results of closed and full-set pattern miner highlights the benefit and effectiveness of our pruning strategy.

7. CASE STUDY: JBOSS APP. SERVER

A case study was performed on the transaction component of JBoss Application Server (JBoss AS) [17]. JBoss AS is the most commonly used J2EE application server. It contains over 100,000 lines of code and comments. The transaction component alone contains over 5,000 lines of code and comments. The purpose of this case study is to show the usefulness of the mined patterns by discovering iterative patterns describing behavior of the transaction sub-component of JBoss AS.

Traces are obtained by running JBoss-AOP [18] over JUnit and Ant on a regression test of the JBoss AS transaction manager. We trace invocations of methods within the transaction component of JBoss AS (*i.e.*, org.jboss.tm package). This produces 28 traces of a total of 2551 events and an average of 91 events. The longest trace is of 125 events. There are 64 unique events. Using min_sup of 65%, the closed iterative pattern mining algorithm runs in less than a minute (29s). Full-set pattern mining doesn't terminate even after running for *more than 8 hours* and produces more than *5 GB of patterns*.

There are a total of 44 patterns resulting from the following post-processing step after iterative pattern mining:

1. Density. Only report patterns whose number of unique events is $>$ 80% of its length.
2. Subsumption. Only report pattern P if none of its super-sequences is frequent.
3. Ranking. Order them according to length and support values.

We found at least 5 interesting software patterns of behavior resulting from mining the traces. These correspond to the patterns of longest length and highest support. Their abstracted representations are as follows:

1. ⟨Connection Set Up Evs, TxManager Set Up Evs, Transaction Set Up Evs, Transaction Commit Evs, Transaction Disposal Evs⟩
2. ⟨Connection Set Up Evs, TxManager Set Up Evs, Transaction Set Up Evs, Transaction Rollback Evs, Transaction Disposal Evs⟩
3. ⟨Resource Enlistment Evs, Transaction Execution Evs, Transaction Commit Evs, Transaction Disposal Evs⟩
4. ⟨Resource Enlistment Evs, Transaction Execution Evs, Transaction Rollback Evs, Transaction Disposal Evs⟩
5. ⟨Lock-Unlock Evs⟩

The first 4 patterns correspond to the few of the longest patterns, the last pattern on lock and unlock events corresponds to the pattern with the highest support of 313. The actual mined pattern for the first pattern shown above, which is the longest pattern mined (of length 32), is shown in Figure 5.

The first two patterns specify that a series of set up events is always followed by a series of termination events. The first pattern 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. The second pattern specifies a similar behavior except that the transaction is being roll-backed.

The third and fourth patterns specify the pattern observed when the actual work is being performed. A resource need to be enlisted to the transaction and the transaction execution then take place. At the end of the execution, the transaction can either be committed or roll-backed. Note that there can be one or more resource enlistments and transaction executions before a commit. Hence the pattern is not included in the body of the first 2 patterns.

The fifth pattern corresponds to a more fine grained iterative pattern occurring most often, namely lock and unlock.

8. CONCLUSION

In this paper, we propose iterative patterns – iterative patterns are commonly occurring series of events exhibited repeatedly within a sequence and across multiple sequences. We extend sequential pattern mining to consider repeated occurrences of pattern instances *within* sequences. We extend episode pattern mining by removing the constraint on *window size* and consider a *database of sequences* rather than a single sequence. To mine iterative pattern efficiently, we present CLosed Iterative Pattern MinER (CLIPER).

The motivation of our work comes from the emerging field of dynamic analysis where a set of program traces are analyzed to mine interesting software properties. Due to looping similar patterns occur within a sequence and across multiple sequences. Mining interesting patterns should take into account both multiple sequences, and multiple occurrences of patterns within a sequence. Also, since important patterns like lock acquire followed-by lock-release and file open followed-by file close (*c.f.*, [32, 7]) are often separated by a considerable number of events, we need to remove the window size constraint of frequent episode mining.

To reduce the number of reported patterns and improve efficiency, we mine for the set of closed iterative patterns. This reduces the run-time needed for mining patterns and aids user in analyzing important patterns by sifting out patterns "absorbed" by another.

Our performance study shows the efficiency of our method in both real-world and synthetic datasets. The effectiveness of our pruning strategy to mine closed patterns is evident by comparing the runtime and the number of patterns generated before and after the pruning strategy is employed. The set of interesting patterns mined from JBoss Application Server transaction component confirms the usefulness of our method in discovering software specifications in iterative pattern form.

Besides mining software behavioral pattern, we believe the proposed mining technique can potentially be applied to other knowledge discovery domains.

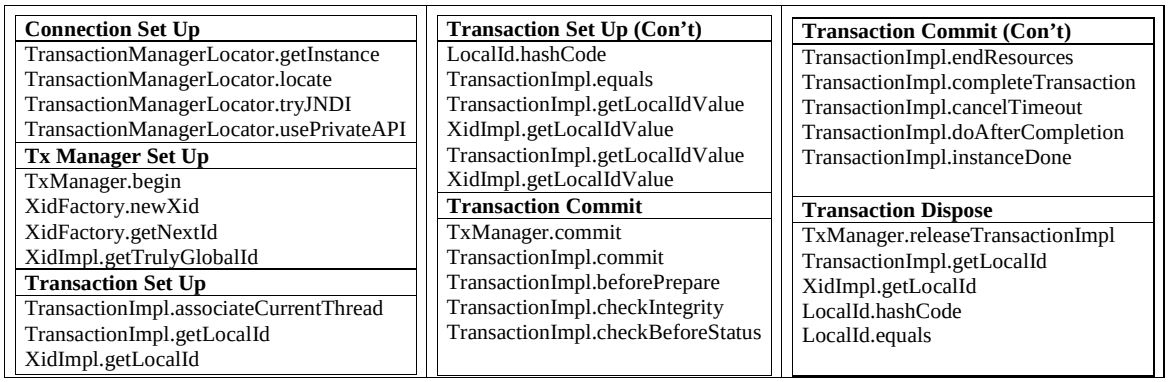


Figure 5: Longest Iterative Pattern Mined from JBoss Transaction Component

Acknowledgement. We would like to thank Jiawei Han and Shahar Maoz for their valuable comments. We wish to thank Blue Martini Software for contributing the KDD Cup 2000 data. This research is partially supported by an NUS research grant R-252-000-250-112, NSF ITR/CCR-0325603, IIS-05-13678, NSF BDI-05-15813, and IIS-02-42840.

9. 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] G Canfora and A Cimitile. *Software Maintenance*, volume 1 of *Handbook of Software Engineering and Knowledge Engineering*, pages 91–120. World Scientific, 2002.

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

[7] 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.

[8] C.Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff. SOBER: statistical model-based bug localization. In *SIGSOFT ESEC-FSE*, 2005.

[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] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M-C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *KDD*, 2000.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *ICSE*, 1994.

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

[17] JBoss. http://www.jboss.org.

[18] JBoss AOP. http://labs.jboss.com/jbossaop/.

[19] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2:86–98, 2000.

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

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

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

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

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

[25] 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.

[26] Java Trans. API Spec. java.sun.com/products/jta/.

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

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

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

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

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

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

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