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 automatic 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. Existing techniques such as sequential pattern and episode minings only address some of the issues arisen from the above phenomena. Our iterative pattern mining leverages on these existing mining techniques to discover frequent iterative patterns which occur repetitively both within a program trace and across multiple traces. We also extend the concept of closed sequential pattern to iterative patterns, thus enabling the derivation of an efficient mining algorithm. The efficiency of our mining algorithm and effectiveness of our pruning strategy is supported by a performance study on several simulated and real datasets. Our case study on JBoss Application Server confirms the usefulness of mined patterns in discovering interesting software behavioral specification.

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 [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, 16]. 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 [8].

The above factors have contributed to high software maintenance costs. It has been investigated that 90% of software cost is due to maintenance [10] and 50% of the maintenance cost is due to comprehending or understanding the code base [23]. 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.

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, 17], which mine temporal program properties or patterns rather than 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 documents. Some of these patterns are:

1. Resource Locking Protocol : \{lock, unlock\}

2. Telecommunication Protocol (c.f., [14]): \{off_hook, dial_on, dial_off, seizure, ring, answer, connection\}

3. Java Authentication and Authorization Service (JAAS) Authorization Enforcer Strategy Pattern (c.f., [24]): \{Subject.getPrincipal, PrivilegedAction.create, Subject.doAsPrivileged, JAAS_Module.invoke, Policy.getPermission, Subject.getPublicCredential, PrivilegedAction.run\}

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 → 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 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 [18]. 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 [28] or generated statically from the source code [26]. Secondly, important patterns for verification, such as, lock acquire and release or stream open and close (c.f. [28, 19]) 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) [14] and Live Sequence Chart (LSC) [7].

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., [14]). Abstracting caller and callee information, it can be represented as a pattern: ⟨ off_hook, dial_tone_on, dial_tone_off, seizure_int, ring_tone, answer, connection_on ⟩. 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 [12] and PrefixSpan [21]) 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., [27] & [25]). 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. Fortunately, the formulation of iterative pattern guarantees this property as described in Section 3.

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 be-
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. [27] and later improved by Wang and Han [25]. 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. [18] 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 [11]. 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. [28, 19]), 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 [29]. 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 [29] 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. Often, the appropriate value of this parameter n is not 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 [28]. 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 support values of reported patterns is made. It is also not complete since many of the more-than-two-events rules cannot be generated by simple concatenation of 2-events 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 [9]. 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. (lock, unlock)) the number of ‘insertions’ is irrelevant – events can be separated by an arbitrary number of events; iterative patterns capture such "behavior" well.

Algorithmically, interaction pattern mining appears to be incomplete, as some frequent patterns will not be generated, partly due to the absence of apriori property.

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 \( (e_1, e_2, \ldots, e_n) \) 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 \( 1 \leq i_1 < i_2 < i_3 < \ldots < i_n \leq m \) where \( e_1 = f_{i_1}, \ e_2 = f_{i_2}, \ldots, e_n = f_{i_n} \). Notation-wise, we write this relation as \( P_1 \subseteq P_2 \). 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).**

Conjugation 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 \( P_1(\langle a_1, \ldots, a_n, b_1, \ldots, b_m \rangle) \) and a suffix \( P_2(\langle b_1, \ldots, b_n \rangle) \) 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 $P(p_1, p_2, \ldots, p_n)$ and a string $S(s_1, s_2, \ldots, s_m)$, the
erasure of $S$ wrt. $P$, denoted by $\text{erasure}(S, P)$, is defined as a
new string $S_{\text{erased}}$ formed from $S$ where all events occurring
in $P$ are removed from $S$. Formally, $S_{\text{erased}}$ is defined as
$\langle s_1, s_2, \ldots, s_k \rangle$ such that (1) $\forall i, s_i \notin P$ and (2) there
exists a set of integers $\{i_1, \ldots, i_k\}$ with $1 \leq i_1 < i_2 < i_3 <
\ldots < i_k \leq m$ and $s_{i_1} = s_{i_2} = s_{i_3} = \cdots = s_{i_k}$ 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
language for specifying software behavioral require-
ment: Message Sequence Chart (MSC) (a standard of Inter-
national Telecommunication Union (ITU) [14]) and its
extension, Live Sequence Chart (LSC) [7].

MSC and LSC is a variant of the well known UML se-
quence 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 sim-
plified telephone switching protocol (c.f., [14]): abstract-
ing caller and callee information, the protocol can be repre-
sented as a pattern: $(\text{off\_hook}, \text{dia\_tone\_on}, \text{dia\_tone\_off},$ 
$\text{seizure\_int, ring\_tone, answer, connection\_on})$.

In verifying traces for conformance to an event sequence
specified in MSC/LSC, the sub-trace manifesting the event
sequence must satisfy the total-ordering property: Given an
event $ev$ in an MSC/LSC, the occurrence of $ev$ in the sub-
trace occurs before the occurrence of every $ev'$ where $j > i$
after $ev'$ where $k < i$ [14]. Kugler et al. strengthened the
above requirement to include a one-to-one correspon-
dence between events in a pattern and events in any sub-
trace satisfying it [15]. 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:

\[
\text{off\_hook, seizure\_int, ring\_tone, answer, ring\_tone, connection\_on}
\]

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
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 ab-
stracts 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 pat-
tern and its instance can be expressed unambiguously in the
form of Quantified Regular Expression (QRE) [20]. Quan-
tified regular expression is very similar to standard regul-
lar expression with ‘$\cdot$’ as concatenation operator, ‘[$P$]’
as exclusion operator (i.e. $\cdot [P]$ means any event except $P$ and $S$)
and * as the standard kleene-star.

3.3.3 (Pattern Instance - QRE). Given a pattern $P(p_1, p_2, \ldots, p_n)$, a substring $S(s_1, s_2, \ldots, s_m)$ of
a sequence $S$ in $\text{SeqDB}$ is an instance of $P$ if it is of the
following QRE expression

$\langle p_1, \ldots, p_n \rangle \cdot \ldots \cdot \langle p_1, \ldots, p_n \rangle \ast$.

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

Definition 3.4 (Iterative Pattern Instance). Given a
pattern $P(p_1, p_2, \ldots, p_n)$, a substring $S(s_1, s_2, \ldots, s_m)$
of a sequence $S$ in $\text{SeqDB}$ is an iterative pattern instance of $P$ if
(1) $\text{first}(P) = \text{first}(SB)$, (2) $\text{last}(P) = \text{last}(SB)$ and
(3) the following erasure constraint holds:

\[
\text{erasure}(SB, \text{erasure}(SB, P)) = P.
\]

We use the term "pattern instance" and "iterative pat-
tern instance" interchangeably in this paper. The operation
$\text{erasure}(SB, \text{erasure}(SB, P))$ 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" in-
terchangeably.

An instance is denoted compactly by a triple $(s_{i_1}, i_{start}, i_{end})$
where $s_{i_1}$ 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 in-
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 $\text{Inst}(P, DB)$. Reference
to the database is omitted if it refers to the input sequence
database.

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

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$(D,B,A,B,A,B,C,E)$</td>
</tr>
<tr>
<td>S2</td>
<td>$(D,B,A,B,B,B,A,B)$</td>
</tr>
</tbody>
</table>

The set $\text{Inst}(P)$ 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 $P(p_1, p_2, \ldots, p_n)$, an instance $I(s_1, s_2, \ldots, s_m)$
of pattern $P$ has the following landmarks: $l_1, l_2, \ldots, l_m$ where $1 \leq l_1 < l_2 < \ldots < l_m \leq m$ and $s_{i_1} = p_1, s_{i_2} = p_2, \ldots, s_{i_m} = p_n$.

Due to erasure constraint, for each instance there is only
one such set of landmarks. The landmarks of an instance $I$
is denoted as $\text{Lnd}(I)$. The $i$th member of the set $\text{Lnd}(I)$
is called the $i$th landmark.

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

3.3 Apriori Property and Closed Pattern

Iterative patterns possess the following ‘apriori’ property
used in PrefixSpan [21]:

\footnote{The proofs of the theorems in this section are available in
www.comp.nus.edu.sg/~edu/it-pat.pdf.}
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 Instances). 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 $Q$ (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 $Q$ have 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:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>(A, B, B, A)</td>
</tr>
<tr>
<td>S2</td>
<td>(A, B, A)</td>
</tr>
</tbody>
</table>

Consider the above sequence database. The only instance of the pattern (A, B, B, A) is (2,1,3), while the only instance of pattern (A, B, B, A) is (1,1,4). Both have the same support. However, since their instances match different segments of the sequences they should be reported separately. (A, B, B, A) is not “absorbed" by (A, B, B, A) 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 [12] and PrefixSpan [21]). 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 $\text{SeqDB}_{P}^{\text{all}}$. It is defined recursively as follows.

**Base Case:** If $P$ is a single event ev

| $(ev, sx)$ | $sx \in \text{SeqDB}$, $ev+e=sx$ |

**Inductive Case:** If $P$ is multi-events

| $(oz+ev+last(P), sx)$ | $3(oz, (px+ev+last(P), sx)) \in \text{SeqDB}_{P}^{\text{all}}$ |


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 $|\text{Inst}(P, \text{SeqDB})|$. In turn, $|\text{Inst}(P, \text{SeqDB})|$ is equal to the size of the projected database $|\text{SeqDB}_{P}^{\text{all}}|$.

The instances of a length-1 pattern $(p_1)$ is simply the occurrences of event $p_1$ throughout the sequences in SeqDB. The instances of a length-k $(p_1, . . . , p_k)$ pattern can be found from instances of length-(k-1) $(p_1, . . . , p_{k-1})$ pattern.

Instances of a length-2 pattern $(p_1, p_2)$ can be formed by extending instance pairings of $(p_1, p_2)$, $(px, sx)$ in $\text{SeqDB}_{P_1, P_2}^{\text{all}}$, on the conditions: (1) $px \notin \text{erase}(px, (p_1, p_2))$ and (2) $3.s[i] = p_2 \land \forall j < i, s_j \notin \{p_1, p_2\}$. 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 $(p_1, p_2, p_3)$ can be formed by extending instance pairings of $(p_1, p_2, p_3)$, $(px, sx)$ in $\text{SeqDB}_{P_1, P_2, P_3}^{\text{all}}$, on the conditions: (1) $px \notin \text{erase}(px, (p_1, p_2, p_3))$ and (2) $3.s[i] = p_3 \land \forall j < i, s_j \notin \{p_1, p_2, p_3\}$. 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 $px$ of a length-2 pattern $(p_1, p_2)$ only obeys the erasure constraint for the original pattern – $px$ might contain $p_1$.

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 $\text{min\_sup}$ 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 $\text{min\_sup}$ 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 $\text{Seq}$ operator.

DEFINITION 4.2 (Projected-first & Seq). A projected database $\text{SeqDB}_{P}^{\text{first}}$ can be projected-first on an event $e$ resulting in a set of pairings and being denoted as $\{\text{SeqDB}_{P}^{\text{first}}(e)\}$.

It is defined as the following set:

\[
\{ (ox+ev+e, sx) \mid 3(ox, (px+ev+e+sx)) \in \text{SeqDB}_{P}^{\text{all}} \land (e \notin \text{erase}(ox, P) \land (ev \in (P+e), ev \notin px)) \}
\]

We denote the size of $\{\text{SeqDB}_{P}^{\text{first}}(e)\}$ as $\text{Seq}(e, \text{SeqDB}_{P}^{\text{first}})$. 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 $\text{Seq}$ operator. However, constraints still be extended. Support of a pattern $P$ is equal to the number of instances supporting $P$, denoted as $|\text{Inst}(P, \text{SeqDB})|$. In turn, $|\text{Inst}(P, \text{SeqDB})|$ is equal to the size of the projected database $|\text{SeqDB}_{P}^{\text{all}}|$. It is added to as $|\text{seqDB}_{P}^{\text{first}}| = |\text{seqDB}_{P}^{\text{all}}|$. 

5
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).** Two projected database $DB_1$ and $DB_2$ are equivalent (denoted as $DB_1 = DB_2$) iff $|DB_1| = |DB_2|$ and $\forall (p_1, s_1) \in DB_1$. 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.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$\langle A, B, A, B, A, B, C, D, E \rangle$</td>
</tr>
<tr>
<td>S2</td>
<td>$\langle A, B, B, B \rangle$</td>
</tr>
<tr>
<td>S3</td>
<td>$\langle A, B, C, A, D, E, B, C \rangle$</td>
</tr>
<tr>
<td>S4</td>
<td>$\langle A, B, C, A, B \rangle$</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Remainder of Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1, 1)</td>
<td>$\langle B, A, B, A, B, C, D, E \rangle$</td>
</tr>
<tr>
<td>(1, 3, 3)</td>
<td>$\langle B, A, B, C, D, E \rangle$</td>
</tr>
<tr>
<td>(1, 5, 5)</td>
<td>$\langle B, C, D, E \rangle$</td>
</tr>
<tr>
<td>(2, 1, 1)</td>
<td>$\langle B, B, B \rangle$</td>
</tr>
<tr>
<td>(3, 1, 1)</td>
<td>$\langle B, C, A, D, E, B, C \rangle$</td>
</tr>
<tr>
<td>(3, 4, 4)</td>
<td>$\langle D, E, B, C \rangle$</td>
</tr>
<tr>
<td>(4, 1, 1)</td>
<td>$\langle B, C, C, A, B \rangle$</td>
</tr>
<tr>
<td>(4, 5, 6)</td>
<td>$\langle B \rangle$</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Instance</th>
<th>Remainder of Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1, 2)</td>
<td>$\langle A, B, A, B, C, D, E \rangle$</td>
</tr>
<tr>
<td>(1, 3, 4)</td>
<td>$\langle A, B, C, D, E \rangle$</td>
</tr>
<tr>
<td>(1, 5, 6)</td>
<td>$\langle C, D, E \rangle$</td>
</tr>
<tr>
<td>(2, 1, 2)</td>
<td>$\langle B, B, B \rangle$</td>
</tr>
<tr>
<td>(3, 1, 2)</td>
<td>$\langle C, A, D, E, B, C \rangle$</td>
</tr>
<tr>
<td>(3, 4, 7)</td>
<td>$\langle C \rangle$</td>
</tr>
<tr>
<td>(4, 1, 2)</td>
<td>$\langle C, C, A, B \rangle$</td>
</tr>
<tr>
<td>(4, 5, 6)</td>
<td>$\langle \rangle$</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>Instance</th>
<th>Remainder of Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 5, 7)</td>
<td>$\langle D, E \rangle$</td>
</tr>
<tr>
<td>(3, 1, 3)</td>
<td>$\langle A, D, E, B, C \rangle$</td>
</tr>
<tr>
<td>(3, 4, 8)</td>
<td>$\langle \rangle$</td>
</tr>
<tr>
<td>(4, 1, 3)</td>
<td>$\langle C, A, B \rangle$</td>
</tr>
</tbody>
</table>

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

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) $SeqDB_{Q}^{P} = SeqDB_{Q}^{P_{1}}$, (2) $first(P) = first(Q)$, (3) $\forall$ event $ev \in erasure(Q, P)$, $ev1 = e$, (4) $sup(P + e) = sup(Q)$, 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. 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 $sup(P) = sup(Q)$, then $sup(P + e)$ will have corresponding instances as $P$. Hence, $P + e$ will have sufficient support.
The procedure to compute the closed set of iterative patterns: **MinePatterns**, is shown at the top of Figure 1. It will call a recursive procedure **MineRecurse** shown at the bottom of Figure 1.

**Figure 1: Mining Algorithm**

Procedure **MinePatterns** will first find patterns of length one whose instances are more than or equal to \( \text{min}_\text{sup} \) threshold. For all frequent length-1 patterns, it will then call the procedure **MineRecurse** to recursively grow each pattern.

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

The algorithm will first find length-1 event \( e \) where \( \text{Pat}++e \) is frequent. Given the input pattern \( \text{Pat} \) and an event \( e \), the number of instances of \( \text{Pat}++e \) is equivalent to the number of pairings \( (p_1, e) \) in \( \text{SeqDB}_{\text{proj}} \), where we can extend \( p_1 \) to an instance of \( \text{Pat}++e \). The above is equivalent to \( \text{Seq}(e, \text{SeqDB}_{\text{proj}}) \).

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

Only such pattern \( \text{Pat} \) 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 Patent to the set Closed.

Next, for any frequent pattern Patent+e, following Theorem 4, we check for its infix extension events. If there is an infix extension event which does not appear in SeqDBPat++, we do not need to grow the pattern Patent+e anymore.

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

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 [27, 25], 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 KDDCup 2000 which was also used to evaluate CloSpan [27] and BIDE [25]. It contains 29069 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 [13], which has been used as one of the benchmarks for research in error localization (e.g., [6]). 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 result 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.
egy 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). 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 over JUnit and Ant on a regression test of the JBoss AS transaction manager in the testsuite that comes along with the JBoss AS source distribution. Tracing all method invocations while running the regression test produces 28 traces of a total of 7279 events with 265 unique events.

We focus on invocations of methods within the transaction sub-component of JBoss AS (i.e., org.jboss.tm package). This reduces the traces to a total of 2551 events and an average of 91 events. The longest trace is of 125 events. There are 87 unique events. Using min_sup of 60%, 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 proposed a novel algorithm to mine iterative patterns from a sequence database. 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.

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., [28, 19]) 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 pat-
9. REFERENCES


Figure 5: Longest Iterative Pattern Mined from JBoss Transaction Component