# Mining Iterative Generators and Representative Rules for Software Specification Discovery

David Lo[1], Jinyan Li[2], Limsoon Wong[3] and Siau-Cheng Khoo[3]
[1] School of Information Systems, Singapore Management University
[2] School of Computer Engineering, Nanyang Technological University
[3] School of Computing, National University of Singapore
davidlo@smu.edu.sg,jyli@ntu.edu.sg,{wongls,khoosc}@comp.nus.edu.sg

**Abstract**—Billions of dollars are spent annually on software related cost. It is estimated that up to 45% of software cost is due to difficulty in understanding existing systems when performing maintenance tasks (*i.e.*, adding features, removing bugs, etc.). One of the root causes is that software products often come with poor, incomplete or even without any documented specifications. This situation is further aggravated by a phenomenon termed software evolution – as software evolves or changes over time, the documented specifications often remain unchanged.

In an effort to improve program understanding, Lo et al. have proposed an algorithm for iterative pattern mining. The algorithm outputs patterns that are repeated frequently within a program trace, or across multiple traces, or both. These frequent iterative patterns reflect frequent program behaviors that likely correspond to software specifications. To reduce the number of patterns and improve the efficiency of the algorithm, Lo et al. have also introduced mining closed iterative patterns.

In this paper, to technically deepen research on iterative pattern mining, we introduce algorithms for mining *iterative generators* from program traces. Iterative generators can be paired with closed patterns to produce a set of *representative rules* having the shortest pre-conditions and longest post-conditions. Different from many rule formats proposed before, representative rules are able to specify forward, backward and in-between temporal constraints in one general representation.

A performance study on synthetic and real datasets shows the efficiency of our approach. A case study on traces of an industrial system shows how iterative generators and closed iterative patterns can be merged to form useful rules shedding light on software design.

## 1 Introduction and Motivation

It's best if software is developed with clear, precise and documented specifications. However, due to hard deadlines and 'short-time-to-market' requirement, software products often come with poor, incomplete and even without any documented specifications. This situation is further aggravated by a phenomenon termed software evolution [1]. As software evolves, the documented specifications often remain unchanged [2]. This might render the documented specification of little use after cycles of program evolution.

These factors have contributed to high software maintenance cost. It has been reported that up to 90% of software cost is due to maintenance [3] and up to 50% of the maintenance cost is due to the effort put in comprehending or understanding software code base [4]. Hence, approximately up to 45% of software cost is due to the difficulty in comprehending an existing code base. This is especially true for software projects developed by many developers over a long period of time.

These needs motivate work on automated tools to extract or mine specifications from programs. An interesting form of specifications to be mined is *patterns of software temporal behaviors*. These patterns are intuitive and commonly found in software documentations. Some examples are as follows:

1) Resource Locking Protocol : $\langle lock, unlock \rangle$
2) Telecommunication Protocol (*c.f.*, [5]): ⟨*off_hook*, *dial_tone_on*, *dial_tone_off*, *seizure_int*, *ring_tone*, *answer*, *connection_on*⟩
3) Java Authentication and Authorization Service (JAAS) Authorization Enforcer Strategy Pattern (*c.f.*, [6]): ⟨*Subject.getPrincipal, PrivilegedAction.create, Subject.doAsPrivileged, JAAS_Module.invoke, Policy.-getPermission*, *Subject.getPublicCredential, Privileged-Action.run*⟩
4) Java Transaction Architecture (JTA) Protocol (*c.f.*, [7]): ⟨*TxManager.begin, TxManager.commit*⟩, ⟨*TxManager.begin, TxManager.rollback* ⟩

Each of these patterns reflects an interesting program behavior. It can be mined by analyzing a set of program traces – each being a series of method invocations. These program traces can in turn be generated through running a test suite. From data mining 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.*, [8]) 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, Lo et al. proposed *iterative pattern min-*

*ing* [9] which extends sequential pattern mining and episode mining to address software specification mining.

Sequential pattern mining first addressed by Agrawal and Srikant in [10] 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.* propose episode mining to discover *frequent episodes within a sequence of events* [11]. An episode is defined as a series of events occurring *relatively close* to one another (*e.g.*, 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.

Frequent 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 [12] or generated statically from the source code. Secondly, important patterns for verification, such as lock acquire and release or stream open and close (*c.f* [12], [13]), 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 [10].

To support iterative pattern mining, we need a clear definition and semantics of iterative pattern different from that of episode and sequential pattern. Our definition of iterative pattern is inspired by common languages for specifying software behavioral requirements, namely Message Sequence Chart (MSC) [5] and Live Sequence Chart (LSC) [14].

MSC and LSC are variants of sequence diagram specifying how a system should behave. An example of such a chart is a simplified telephone switching protocol shown in Figure 1 – *c.f.*, [5]. 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 charts carry semantics that need to be obeyed e.g., the example chart disallows *ring_tone* to appear twice, one before and another after *answer* (see sub-section 3.2 for more details).

Pattern mining in general is an NP-hard problem. For it to be practical, efficient search space pruning strategies need to be employed. In [9], Lo *et al.* address the issue by mining a *closed* set of iterative patterns.

In this work, we investigate mining of iterative *generators*

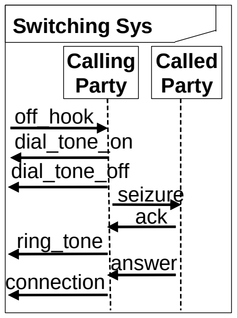


Fig. 1. **Simplified Telecommunication Protocol.**

from program execution traces. Generators are the minimal members of an equivalence class, while closed patterns are the maximal members. An equivalence class in turn is a set of frequent patterns with the same support and corresponding pattern instances. Iterative generators is a new data mining concept, though generators for frequent itemsets [15], [16], [17] and for sequential patterns [18] have been introduced previously in the literature.

We find that the algorithms mining closed iterative patterns and sequential generators *can not be trivially extended* to mine iterative generators as *major properties* used for mining closed iterative patterns and sequential generators *no longer apply* for iterative generators (see sub-section 3.4 for details).

The set of closed patterns and generators are potentially of much smaller size than the full-set of frequent patterns. As generators are minimal members of equivalence classes, according to the Minimum Description Length (MDL) principle, generators are the preferred descriptions or representations of the classes. Also, as argued by Li *et al*, generators are better candidates than closed patterns for applications involving model selection and classification [16].

Furthermore, closed iterative patterns and generators can be merged to form a compact knowledge representation capturing an expressive form of temporal constraints. These constraints are in the form of rules with minimal pre-conditions and maximal post-conditions referred to as *representative rules*. Temporal constraints add more information to patterns and are useful in software domain as they correspond to temporal properties used to find bugs and ensure correctness of systems via formal verification tools [19].

Past studies on mining rules have focused on finding forward temporal constraints (*i.e.*, if a series of events occurs, eventually another series of events occurs) [20], [21] or backward temporal constraints (*i.e.*, if a series of events occurs, previously, another series of events has occurred before) [22]. A representative rule on the other hand captures a more general form of temporal constraints. It first describes a sequence of pre-condition events which, if satisfied, implies the occurrence of other events which can happen before, in-between or after the sequence of pre-condition events.

For example, consider the rule $\langle X, \{A\}, Y, \{B\}, Z\rangle$, where the ones marked with brackets correspond to the pre-condition events. This rule states that when the events $A$ and $B$ occur, $X$ must happen before $A$, $Y$ must happen in between $A$ and $B$ and $Z$ must happen after $B$. The rule captures not only forward temporal constraints but also backward and in-between temporal constraints. As a visual representation, one can imagine the horizontal arrows in Figure 1 be drawn in two different colors, one representing the pre-condition and the other the post-condition.

The utility of the general format of representative rules is readily evident in daily life. Consider an Automated Teller Machine (ATM), one important property is: "a user must be *authenticated* after an ATM card is *inserted* and before money is *dispensed*", *i.e.*, $\langle\{ins\_card\}, authenticate, \{dis\_money\}\rangle$. Also, consider a university, an important property is: "a student must *pass all the examinations* after he/she is *matriculated* and before he/she *graduates*", *i.e.*, $\langle\{matriculate\}, pass\_exams, \{graduate\}\rangle$. Consider an example temporal constraint used to verify correctness of Windows device drivers: a driver that has called *KeAcquireSpinLock*, *KeInsertDeviceQueue* and eventually *KeReleaseSpinLock*, must have called *IoMarkIrpPending* after the call to *KeInsertDeviceQueue*, but before the call to *KeReleaseSpinLock*, *i.e.*, $\langle\{KeAcquireSpinLock\}$, $\{KeInsertDeviceQueue\}$, *IoMarkIrpPending*, $\{$*KeRelease-SpinLock*$\}\rangle$ – *c.f.* [23]. These types of temporal constraints can be mined by mining representative rules but not by the previous studies on rule mining which only mine *either forward or backward temporal constraints separately*.

To mine iterative generators, we introduce the concept of *approximate* iterative pattern. An approximate pattern weakens the constraint that a regular pattern imposes on its instances. We introduce approximate patterns since we can obtain the set of frequent patterns with *correct* support from them and they have a good property which can be utilized to avoid redundant efforts when traversing the search space of frequent patterns. The support of an approximate pattern is an *upper bound* of the actual pattern support.

We first construct a compact lattice representing frequent approximate patterns by building it depth-first. Redundant search space traversal efforts are avoided by detection of equivalent sub-lattices. If a sub-lattice is equivalent to an existing sub-lattice there is no need to build it again. Next, we traverse the compact lattice breadth-first while performing: (1) Computation of real pattern support, (2) Identification of generators, and (3) Pruning of sub-search-spaces containing only non-generators.

After the steps above are performed, a set of generators would be mined. The generators are then merged with closed patterns to produce a set of representative rules capturing forward, backward and in-between temporal constraints.

A performance and a comparative study are conducted on synthetic and real benchmark datasets to evaluate the performance of the proposed mining algorithm. The proposed algorithm can run efficiently at low support thresholds, large number of sequences and reasonably long sequences on 10 synthetic and 2 real benchmark datasets. The experiments also show that iterative generators can be mined with similar efficiency as closed patterns. Since, *both* generators and closed patterns are needed to form representative rules capturing forward, backward and in-between temporal constraints we need to ensure that both algorithms run at least with similar efficiency. It is not our purpose to beat the efficiency of closed iterative pattern mining algorithm.

As a case study, we extend the study on JBoss Application Server (JBoss AS) in [9]. The mined generators combined with mined closed patterns form interesting rules that shed further light on the behavior of the software system expressing forward, backward and in-between temporal constraints that the system obeys.

The contributions of this work are as follows:

1) We propose iterative generators and representative rules as new data mining concepts for software specification discovery.
2) We present novel properties of iterative generators for: (i) generator identification, (ii) search-space compaction, and (iii) non-generator pruning.
3) We propose a new algorithm to mine iterative generators and generate representative rules. Different from past algorithms on mining from sequences of events, we introduce: (i) A merge between depth-first and breadth-first traversal of search space, (ii) A new lattice data structure, and (iii) New techniques to detect equivalences of projected databases, compute real support and utilize the properties of iterative generators for their efficient mining.

The outline of the paper is as follows: Section 2 presents related work. Section 3 provides preliminary information on notations, semantics of iterative pattern and definitions of closed pattern and generator. An analysis on the need for a new algorithm to mine iterative generators is also described in the section. Section 4 presents approximate iterative pattern and the approximate pattern lattice data structure. The section also discusses some properties of approximate patterns for effective search space compaction. Furthermore, some properties for efficient pruning of iterative generators are also discussed in this section. Section 5 describes our iterative generator mining algorithm and representative rule generation process. Section 6 presents the results of our performance study. Section 7 presents the case study on JBoss AS.

## 2 Related Work

Iterative pattern mining is an extension of sequential pattern mining, which was first proposed by Agrawal and Srikant [10]. To remove redundant patterns, closed sequential pattern mining was proposed by Yan *et al.* [24]. The approach was later improved by Wang and Han [25]. Different from sequential pattern, iterative pattern captures 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.* [11] 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 [26]. 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 an "episode". This difference is significant, since important program behavioral patterns, for example: lock acquire and release, or file open and close (*c.f* [12], [13]), 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 sequence, while iterative pattern mining operates over a set of sequences.

There are also other studies on mining patterns that repeat in sequences. In mining DNA sequences, Zhang *et al.* introduced the idea of "gap requirement" in mining periodic patterns from sequences [27]. Similar to ours, they detect repeated occurrences of patterns within a sequence and across multiple sequences. However, the gap requirement used in their work 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. Recently, Ding *et al.* mine for closed repetitive sub-sequences [28]. Different from their work, we consider the minimal patterns in equivalence classes (*i.e.*, generators) rather than the maximal patterns in equivalence classes (*i.e.*, closed patterns). Also, the semantics of Zhang *et al.*'s periodic pattern and Ding *et al.*'s repetitive subsequence differ from iterative pattern that follows MSC/LSC.

In [9], Lo *et al.* mine all frequent and closed iterative patterns. In this work, we mine for iterative generators. Closed iterative pattern mining algorithm proposed in [9] can not be trivially extended to mine for generators as the property used to detect non-closed iterative patterns no longer applies on iterative generators (more detail in sub-section 3.4).

There are also several studies on mining generators. Pasquier *et al.* mine generators of frequent itemsets [15]. In [16], Li *et al.* improve the algorithm in [15]. In [17], Li *et al.* extend their work by concurrently obtaining both closed itemsets and generators, and computing delta discriminative non-redundant equivalent classes. Lo *et al.* mine a set of sequential generators [18]. A concurrent study on mining sequential generators is also made by Gao *et al.* in [29]. Different from the above studies, we mine iterative generators which poses further challenges since repeated occurrences of patterns within a sequence need to be considered. Among the above studies, the closest to our proposed approach is the work mining sequential generators reported in [18]. However, even this work can not be trivially extended to mine iterative generators, as basic properties on sequential generators no longer apply on iterative generators. These include basic properties e.g., apriori property and detection on equivalence on projected databases (more detail in sub-section 3.4).

There are past studies on mining rules from sequences of events [20], [30], [31], [21], [22]. There are a number of differences between our work and each of them. *Most importantly*, the proposed *representative rules* are able to specify forward, backward and in-between temporal constraints in one representation and they can be mined efficiently.

## 3 Preliminaries

In this section, we define some notations, outline the semantics of iterative patterns and describe the definitions of closed iterative patterns and iterative generators. An analysis on the need for a new algorithm to mine iterative generators is also presented.

### 3.1 Basic Definitions

Let $I$ be a set of distinct events. Let a *sequence* $S$ be a series of events. We denote $S$ as $\langle e_1, e_2, \ldots, e_{end}\rangle$ where each $e_i$ is an event from $I$. The sequence database under consideration is denoted by $DB$. The $i$th sequence in the database $DB$ is denoted as $DB[i]$. Also the $j$th position of the sequence $DB[i]$ is denoted as $DB[i][j]$.

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 $P_1$. Concatenation of $P_1$ and $P_2$, denoted as $P_1{+\!+}P_2$, results in a longer pattern $P_3 = \langle e_1, \ldots, e_n, f_1, \ldots, f_m\rangle$.

We use the notation $|P|$ to refer to the length of pattern P. Given a series of events $evs$, $fst(evs)$ and $last(evs)$ refer to the first and last event of $evs$ respectively. An important concept of erasure operator is defined below.

*Definition 3.1 (***Erasure Operator):** Consider 2 series of events $S$ and $P$, the erasure of $S$ $wrt.$ $P$, denoted by $erasure(S, P)$, is defined as a new series of events formed by removing all events in $S$ that occurs in $P$.

### 3.2 Semantics of Iterative Patterns

Based on the above motivation of patterns in software, in [9], we define iterative patterns based on the semantics of commonly used software modeling languages. In particular we follow the semantics of Message Sequence Charts [5], a standard of International Telecommunication Union (ITU), and its extension, Live Sequence Charts [32].

MSC and LSC is a variant of the well-known UML sequence diagram describing behavioral requirement of software. Not only do they specify system interaction through ordering of method invocation, but they also specify caller and callee information. An example of such charts is a telephone switching protocol in [5]; abstracting caller and callee information, a simplified 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 a 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 $k < i$ [5]. 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 [33]. 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* |

| ID | Sequence |
|---|---|
| $S1$ | $\langle A, B, C, B, D, A, B, C, D\rangle$ |
| $S2$ | $\langle A, B, E, C, F, D\rangle$ |
| $S3$ | $\langle A, B, F, C, D, E\rangle$ |

TABLE 1
Running Example - ExDB

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

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) [34]. 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 3.2 (***Pattern Instance - QRE***):* Given a pattern $P$ $(p_1p_2 \dots p_n)$, a substring $SB$ $(sb_1sb_2 \dots sb_m)$ of a sequence $S$ in $SeqDB$ is an instance of $P$ iff it is of the following QRE expression

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

We use the term "pattern instance" and "iterative pattern instance" interchangeably in this paper. 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 $(s_x, i_{srt}, i_{end})$ where $s_x$ refers to the sequence index of a sequence $S$ in the database while $i_{srt}$ 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 database

Consider a pattern $P$ $(\langle A, B\rangle)$ and database $ExDB$ in Table 1. The set $Inst(P)$ is {(1,1,2),(1,6,7),(2,1,2), (3,1,2)}.

*Definition 3.3 (***Pattern Constraint Set***):* Iterative pattern defines a constraint to its instances corresponding to the exclusion operations (*i.e.*, $[-e_1, \dots, e_n]$) in Definition 3.2. We define the constraint set of $P$ to be $\cup_{e \in P}\{e\}$ and denote this by $Constr(P)$. If $Constr(P) \subset Constr(P')$, we say that $P$ $(P')$ has a weaker (stronger) constraint than $P'$ $(P)$.

Many specifications obey these negation-like constraints, for example, consider Windows device driver rule $CancelSpinLock$ in MSDN [35]:
"The $CancelSpinLock$ rule specifies that the driver calls $IoAcquireCancelSpinLock$ before calling $IoReleaseCancelSpinLock$ and that the driver calls $IoReleaseCancelSpinLock$ before any subsequent calls to $IoAcquireCancelSpinLock$."

The above rule (implicitly) expressed that no $IoAcquireCancelSpinLock$ or $IoReleaseCancelSpinLock$ appear in between subsequent calls of the two function calls.

The above definition of pattern instance guarantees that unless a pattern's prefix is the same as one of its suffix, instances of the pattern do not overlap. Also an instance is never contained in another instance of the same pattern. The pattern instance definition also allows the apriori property to hold (see sub-section 4.3).

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 3.4 (***Pattern Inst. Landmarks***):* Given a pattern $P$ $\langle e_1, \dots, e_n\rangle$, an instance I $\langle s_1, \dots, s_m\rangle$ of $P$ has the following landmarks: $l_1, \dots, l_n$ where $1 = l_1 < l_2 < \dots < l_n = m$ and $s_{l_1} = e_1, s_{l_2} = e_2, \dots, s_{l_n} = e_n$. Landmarks $l_1, l_2, \dots, l_n$ are referred to as the first, second, ..., $n$th landmark respectively. For each instance of $P$, there is only one possible set of landmarks.

As an example, consider a pattern $\langle A, B, C\rangle$ and its instance $\langle A, D, E, B, D, C\rangle$. The pattern instance has 3 landmarks: 1, 4, 6. They are called the first, second and third landmark respectively.

The *support* of a pattern *wrt.* a sequence database $DB$ is the number of its instances in $DB$. Repeated occurrences of a pattern within a sequence are taken into consideration for support calculation. A pattern $P$ is considered *frequent* when its support, $sup(P) \geq$ *min_sup* threshold.

## 3.3 Generators and Closed Patterns

Unless otherwise stated, in this paper, we refer to iterative generators, closed iterative patterns and representative rules as generators, closed patterns and rules respectively. Before defining generators and closed patterns, we describe corresponding pattern instances.

*Definition 3.5 (***Corresponding Pattern Insts***):* Consider a pattern P and its super-sequence Q. An 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 $end_P \leq end_Q$. If every instance of $P$ corresponds to an instance of $Q$ (and vice versa) we denoted that by $Inst(P) {\approx} Inst(Q)$.

*Definition 3.6 (***Generators and Closed Patterns***):* A frequent pattern $P$ is a *generator* if there exists no subsequence $Q$ s.t.:

1. $P$ and $Q$ have the same support
2. $Inst(P) {\approx} Inst(Q)$

Also, $P$ is a *closed pattern* if there exists no super-sequence $Q$ such that the above two conditions hold.

The set of generators and closed patterns mined from ExDB using *min_sup* = 3 are $\{\langle A\rangle : 4, \langle A, B, D\rangle : 3, \langle B\rangle : 5, \langle B, C, D\rangle : 3, \langle C\rangle : 4, \langle D\rangle : 4\}$ and $\{\langle A, B, C\rangle : 4, \langle A, B, C, D\rangle : 3, \langle A, C, D\rangle : 4, \langle B\rangle : 5, \langle B, D\rangle : 4\}$ respectively.

We consider the following problem: Given a sequence database, find a set of frequent iterative generators. We also describe how iterative generators and closed iterative patterns are merged to form representative rules capturing forward, backward and in-between temporal constraints.

### 3.4 Why a simple extension does not work?

Closed iterative pattern mining algorithm proposed in [9] can not be trivially extended to mine for generators. To detect non-closed patterns, the algorithm in [9] tries to extend instances of $P$ to form instances of a longer pattern $P'$. If all instances of $P$ can be extended, then $P$ is not closed. The approach works since, based on the apriori property, the longer pattern $P'$ has less or equal instances as compared to $P$. One might be tempted to perform a similar approach to detect non-generators by *removing* events from instances of $Q$ to form instances of a shorter pattern $Q'$. Unfortunately, this does not work since shorter patterns can have more instances. Some of these instances might not correspond to any instance of the longer pattern.

Similarly, the algorithm mining sequential generators in [18] can not be trivially extended to mine iterative generators since repetitions need to be addressed and the semantics of MSC/LSC need to be obeyed (which involves negation on the definition of pattern instance). A number of major properties of sequential patterns no longer apply as described in the following paragraphs.

First, the apriori property of sequential pattern is no longer true, *i.e.*, it is not always the case that if a pattern $P$ is a subsequence of another pattern $P'$, then sup($P$) $\geq$ sup($P'$). This is due to the negation involved in the definition of pattern instance following the semantics of MSC/LSC.

Second, it is no longer the case that if two patterns P and P' have the same projected database, *i.e.*, $DB_P == DB_{P'}$, then any extension of $P$ ($P{+\!+}evs$) will have the same projected database as the corresponding extension of $P'$ ($P'{+\!+}evs$), *i.e.*, $DB_{P{+\!+}evs} == DB_{P'{+\!+}evs}$. This property has been used in [18] to prune redundant search space. This necessitates us to propose *approximate* iterative patterns with approximated support where this property still holds. However our goal is still to mine iterative patterns with correct support. To do so, our approach first generates a compacted representation of approximate iterative patterns and then obtain iterative patterns with correct support from the compact representation of approximate iterative patterns.

Third, the approach to detect equivalences of projected databases used in sequential pattern mining [24], [18] no longer applies in iterative pattern mining. The work mining closed iterative patterns [9] does not require detection of equivalent projected database, while we need it in our approach. More details on this is given in sub-section 5.2.

Furthermore, different from the work in [18], there is no need to generate and filter a set of candidate patterns which can be expensive especially due to a large number of candidate patterns. Different from approximate iterative pattern, a candidate sequential pattern in [18] has the correct support, however might not be a sequential generator. Several subsequence checks need to be made to ensure that the candidate pattern is a generator (*i.e.*, there is no shorter patterns having the same support as itself). Elimination of this filtering step is especially useful when mining iterative generators, as the hash-based filtering approach used in [24], [18] is less effective when applied to iterative patterns. In essence, the hash-based approach tries to put patterns appearing in the same set of sequences in the same bucket (via a heuristic). Only patterns in the same bucket would need to be checked against one another. There would be more patterns in each bucket in the case of iterative patterns, as two patterns can occur in the same set of sequences, but repeat a different number of times.

## 4 Data Structure and Theorems

Our approach works by first mining frequent approximate patterns. The approximate patterns have a nice property that enables non-redundant traversal of search space. A compacted set of frequent approximate patterns can be represented as a lattice. This lattice is then used as the compacted search space for frequent iterative generators. Generator identification and search space pruning strategies are effectively employed to mine the set of frequent generators.

In this section, we describe approximate iterative patterns, mention our compacted lattice structure, and describe some properties for search space compaction and pruning of sub-search spaces only containing non-generators.

### 4.1 Approximate Patterns

An instance of an approximate pattern is defined by the following QRE expression shown in Definition 4.1.

*Definition 4.1 (***App. Pattern Instance - QRE):** Given an approximate pattern $P$ $\langle e_1, e_2, \ldots, e_n\rangle$, a substring $SB$ $\langle sb_1, sb_2, \ldots, sb_m\rangle$ of a sequence $S$ in $DB$ is an instance of $P$ iff it is of the following QRE expression

$$e_1; [-e_1, e_2]*; e_2; [-e_2, e_3]*; e3; \ldots; [-e_{(n-1)}, e_n]; e_n.$$

The *approximated support* of a sequence of events $P$ is the number of instances of the approximate pattern $P$. It is denoted as *sup-app* ($P$).

Approximate patterns, as we shall see in Theorem 1, have a nice property that enables avoidance of redundant search-space re-computation. To calculate the support of approximate patterns in the input database, we define two new projected database operations.

*Definition 4.2 (***Projected-approx-all):** A database *projected- approx-all* on a pattern $P$ is defined as:

$DB_P^{app-all} = \{(x, j, y) \mid$ the $j^{th}$ sequence in $DB$ is $s$, where $s = a{+\!+}x{+\!+}y$, and $x$ is an instance of an approximate pattern $P$ in $s$ $\}$

The approximate support of the pattern $P$ is then equal to the size of $DB_P^{app-all}$. Also, let us define the number of events in a projected database $PDB$ to be: $\Sigma_{(x,j,y)\in PDB)}$ $|y|$.

*Definition 4.3 (***Projected-approx-next):** A projected-approx-all database $PDB$ *projected-approx-next* on an event $e$ is defined as:

$PDB_e^{app-nxt} = \{(x', i, y') \mid (x, i, y) \in PDB,\ x' = x{+\!+}w,\ y = w{+\!+}y',\ w = u{+\!+}e$, and $last(x)$ & $e$ do not occur in $u\}$

The last term in the conjunction above is there to ensure that $x'$ will be an instance of the approximate pattern $P{+\!+}e$. Hence, $DB^{app-all}_{P{+\!+}e} = (DB^{app-all}_{P})^{app-nxt}_{e}$.

With projected-approx-next, a projected-approx-all database of a pattern $P$ can be built incrementally. The base case is projected approx-all databases of single events, each simply corresponds to all instances of the single event under projection, in the input database. Note that we only perform a pseudo-projection to reduce memory and runtime overhead similar to studies in [36], [24], [25], [9].

In this paper, unless otherwise stated, "projected" database refers to "projected approx-all" database. Also, in this paper, *approximate* pattern and pattern instance are stated explicitly. If not stated, "pattern" and "instance" refer to the regular iterative pattern and pattern instance defined in Definition 3.2.

As examples, projected-approx-all and projected-approx-next databases wrt. ExDB are shown in Tables 2 & 3.

| |
|---|
| $(\langle A,B\rangle,1,\langle C,B,D,A,B,C,D\rangle)$ |
| $(\langle A,B\rangle,1,\langle C,D\rangle)$ |
| $(\langle A,B\rangle,2,\langle E,C,F,D\rangle)$ |
| $(\langle A,B\rangle,3,\langle F,C,D,E\rangle)$ |

TABLE 2
$ExDB^{app-all}_{\langle A,B\rangle}$

| |
|---|
| $(\langle A,B,C\rangle,1,\langle B,D,A,B,C,D\rangle)$ |
| $(\langle A,B,C\rangle,1,\langle D\rangle)$ |
| $(\langle A,B,E,C\rangle,2,\langle F,D\rangle)$ |
| $(\langle A,B,F,C\rangle,3,\langle D,E\rangle)$ |

TABLE 3
$(ExDB^{app-all}_{\langle A,B\rangle})^{app-nxt}_{\langle C\rangle} = ExDB^{app-all}_{\langle A,B,C\rangle}$

## 4.2 Data Structure and Constraint Violators

The data structure we are using is an extension of prefix sequence lattice (PSL) first introduced in [24]. Each path in the tree corresponds to a frequent approximate pattern. Every projected database corresponding to a frequent approximate pattern is represented by a node in the lattice. We refer to this data structure as Equivalent Projected-database-based Approximate Pattern Lattice, shortened as *PAL*. In PSL, a projected database is possibly represented by multiple nodes, which is not the case with PAL.

Note that different patterns generated by traversing the PAL from the root to a node $n$ might have different supports since the lattice only captures approximated pattern support. As a further extension to the PSL introduced in [24], each node in PAL is linked with a table storing the ids and actual supports of various patterns ending at the node. Given a node $n$, its support table is denoted as $n$.SupTable. Each transition in the PAL is labelled. A transition $t$ from node $n$ to $m$ is given an integer label $i$ if it is the $i$th transition incoming to a sink node $m$ when $t$ is added to the PAL. As will be described in sub-section 5.1, PAL is built depth first in lexicographical order. Given a transition $t$ the label is denoted as $t$.Id. PAL built from $ExDB$ by considering $min\_sup$ threshold set at 3 is shown in Figure 2. The numbers next to the transitions denotes transition labels.

We next introduce the concept of forward constraint violators in Definition 4.4.

*Definition 4.4 (***Forward Constraint Violators***):* Consider a pattern $P = \langle e_1, e_2, \ldots, e_n\rangle$. Let the set $ext(P)$ = $\{Q|\ Q = P{+\!+}evs$, where $evs$ is a series of one or more events and sup-app$(Q) \geq min_sup\}$. It defines the set of extended patterns of $P$ whose approximate support are above the $min\_sup$ threshold.

The set of forward constraint violators of $P$ wrt. an extended pattern $Q$ in $ext(P)$ is the set of events, occurring in an instance of $Q$ that appear in between adjacent pairs of events in $last(P){+\!+}\ evs$ (*i.e.*, between each pair of the $(x-1)^{th}$ and $x^{th}$ landmarks of $Q$ (exclusive), where $x > |P|$). Let us denote this as $FV(P,Q)$.

The set of forward constraint violators of $P$, denoted as $FV(P) = \cup_{Q\in ext(P)}.FV(P,Q)$. Patterns having the same projected database will have the same set of forward constraint violators.

As examples of constraint violators, the constraint violators of the example database ExDB is provided on the table in Figure 2. Consider pattern $\langle A,B\rangle$ and the example database ExDB. The set $ext(\langle A,B\rangle)$ is the set $\{\langle A,B,C\rangle$, $\langle A,B,C,D\rangle$,$\langle A,B,D\rangle\}$. The set of $FV(\langle A,B\rangle,\langle A,B,C\rangle)$ is the set $\{E,F\}$. This is the case as there is an occurrence of event $E$ when we extend instance (2,1,2) of $\langle A,B\rangle$ to instance (2,1,4) of $\langle A,B,C\rangle$. Similarly, there is an occurrence of event $F$ when we extend instance (3,1,2) of $\langle A,B\rangle$ to instance (3,1,4) of $\langle A,B,C\rangle$. The set $FV(\langle A,B\rangle)$ is $\cup_{q\in ext(\langle A,B\rangle)}.FV(\langle A,B\rangle,q)$ which is the set $\{C,E,F\}$ shown in the table in Figure 2. In Theorem 3, the concept of constraint violators will be used to define a non-generator pruning property.

Several pieces of information are attached to a node $n$ in PAL, namely the corresponding event, projected database and forward constraint violators. They are denoted as $n$.Ev, $n$.PDB and $n$.FV respectively. Details on how this data structure is constructed and utilized are given in Section 5.

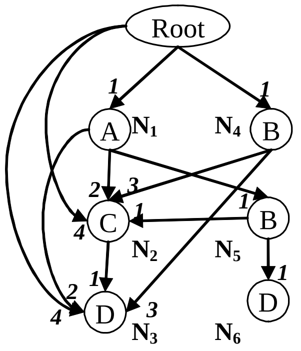
## 4.3 Properties and Theorems

Following are a property and theorems for efficient mining of iterative generators. Due to space limitation, the proofs of the property (and some others in the subsequent sections) and Theorem 1 are omitted.

*Property 1 (***Apriori Property***):* Consider a pattern $P$ and its supersequence $P'$ of the forms $P{+\!+}evs$ or $evs{+\!+}P$, where $evs$ is a series of events. It must be the case that $sup(P) \geq sup(P')$. Also, if they have the same support, $Inst(P){\approx}Inst(P')$

*Theorem 1 (***Generator Identification***):* Pattern $P$ is a generator iff $\forall Q$. if either one of the following conditions holds: 1. $P = ev{+\!+}Q$, 2. $P = Q{+\!+}ev$ or 3. $Q \in \cup_{e\in P} erasure(P,e)$, then $sup(Q){>}sup(P)$. Given a pattern $P$, all $Q$s of the above 3 formats are referred to as *sub-patterns* of $P$.

*Theorem 2 (***Search Space Compaction***):* Consider approximate patterns $P$ and $P'$, if $DB^{app-all}_{P} = DB^{app-all}_{P'}$, then for every arbitrary series of events $evs$, $DB^{app-all}_{P{+\!+}evs}$ = $DB^{app-all}_{P'{+\!+}evs}$.



| ID | FV | sup-app |
|---|---|---|
| $N_1$ | {B,C,E,F} | 4 |
| $N_2$ | {B,F} | 4 |
| $N_3$ | {} | 4 |
| $N_4$ | {B,C,E,F} | 5 |
| $N_5$ | {C,E,F} | 4 |
| $N_6$ | {} | 3 |

Fig. 2. PAL of ExDB (left) and its details (right)

*Proof:* From Definition 4.1, an approximate pattern instance has the following constraint: events occurring between two adjacent landmarks should not be any of the two landmark events. Since $P$ and $P'$ has the same projected database, it must be the case that $last(P) = last(P')$. Hence, whenever we can extend $P$ with an event $e$ to form $P{+}\!\!{+}e$ with support $x$, we can always extend $P'$ with $e$ to form $P'{+}\!\!{+}e$ with support $x$, *i.e.*, $DB^{app-all}_{P++e} = DB^{app-all}_{P'++e}$. Extending the argument, it can be shown that whenever we can extend $P$ with $evs$ we can also extend $P'$ with $evs$ and the resultant patterns have the same projected databases, *i.e.*, $DB^{app-all}_{P++evs} = DB^{app-all}_{P'++evs}$. □

*Theorem 3 (***Non-Generator Pruning***):* Suppose there exist patterns $P$ and $P'$, where $P' \in \cup_{ev \in P}\ erasure(P, ev)$, $sup(P) = sup(P')$, $DB^{app-all}_{P} = DB^{app-all}_{P'}$ and $\forall$ event $e \in Constr(P) - Constr(P')$. $e \notin FV(P)$. Then all patterns of the form $P{+}\!\!{+}evs$, where $evs$ is an arbitrary series of events, are not generators.

*Proof:* Since $P' \in \cup_{ev \in P}\ erasure(P, ev)$, every instance of $P$ is an instance of $P'$. Since $sup(P) = sup(P')$, $Inst(P) = Inst(P')$.

Since $DB^{app-all}_{P} = DB^{app-all}_{P'}$, $FV(P) = FV(P')$. $P'$ is weaker than $P$ by the set $Constr(P) - Constr(P')$. Since all events in this differential constraint set are not in $FV(P) \cup FV(P')$, $Inst(P)$=$Inst(P')$, and $DB^{app-all}_{P} = DB^{app-all}_{P'}$, it must be the case that whenever we can extend an instance of $P$ by $evs$ we can also extend the corresponding instance of $P'$ by $evs$ (and vice versa) such that $Inst(P{+}\!\!{+}evs)$=$Inst(P' {+}\!\!{+}\ evs)$. There will not be any constraint violation since all events in the differential constraint set is not in the set of forward constraint violators.

Hence, for any pattern $P{+}\!\!{+}evs$ there is another frequent pattern $P'{+}\!\!{+}evs$ which is its subsequence with the same set of instances. Hence, all patterns of the form $P{+}\!\!{+}evs$ are not generators. □

## 5 MINING ALGORITHM

To mine frequent generators of iterative patterns, we first compute frequent approximate patterns that can effectively be compacted and represented as a lattice. This lattice of approximate patterns can be computed fast by avoiding redundant traversal of search space of frequent patterns. An approximate pattern weakens the constraint that a regular pattern imposes on its instances. Hence, the approximated support is an upper bound to the actual support of an iterative pattern. This search space lattice of patterns with over approximated support is then traversed to mine for generators of iterative pattern. Several strategies are employed to effectively compute the actual support of patterns, check whether a pattern is a generator and prune the sub-search spaces only containing non-generators.

### 5.1 Search Space Compaction into PAL

To generate the compact search space of frequent generators, we perform a depth-first search of the search space of frequent patterns. This is performed by growing single-event patterns. Events are added one by one to this pattern according to lexicographical order. Our goal is to produce PAL and avoid redundant traversal of search space.

Every time an event is added, a new pattern $P$ is formed. Correspondingly, a new node $n_P$ is added to the PAL. A check is performed if another pattern $P'$ corresponding to node $n_{P'}$ with the same projected database exists. If it does, according to Theorem 1, there is no need to extend $P$ anymore. The subtree rooted in $n_P$ will be the same as that rooted in $n_{P'}$. This is the case since for all sequence of events $evs$, the approximate projected databases of $P'{+}\!\!{+}evs$ and $P{+}\!\!{+}evs$ are the same. We simply need to add a link from $n_P$ to the subtree rooted at $n_{P'}$. Otherwise, if there does not exist such a $P'$ a new node is added to the PAL and the search space compaction process continues recursively.

The PAL data structure built from $ExDB$ at min_sup = 3 is shown in Figure 2. The algorithm performing the search space compaction into PAL is shown in Figure 3.

### 5.2 Detection of Equivalent Projected DBs

A basic operation needed during construction of PAL is the detection of equivalent projected databases. To avoid 're-inventing the wheel', we try to utilize the property used in [24] and [18] stated below.

*Property 2 (***Eq. Proj. DB - Sequential Pat.***):* Consider sequential patterns $P$ and $P'$ where $P \sqsubseteq P'$. It is the case that the projected databases of $P$ and $P'$ are equal iff the total number of items in the two projected databases are equal.

Unfortunately, the nice property above no longer holds for iterative patterns. To see this consider the following sample database.

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

Consider patterns $P$=$\langle A, B, B, C \rangle$ and $P'$=$\langle A, B, C \rangle$. The total number of items in $DB^{app-all}_{P}$ and $DB^{app-all}_{P'}$ are the same (*i.e.*, 1, corresponding to event $E$ in $S1$ and $D$ in $S2$ for $DB^{app-all}_{P}$ and $DB^{app-all}_{P'}$ respectively). However, they have different projected databases. Hence, the property no longer holds for iterative patterns.

To detect equivalences of projected databases, we need to compare each element of the projected databases one by one. To prevent unnecessary comparisons, we utilize the following property on *non-equivalence of projected databases*.

*Property 3 (***Non-Eq. of Proj. DB - Iterative Pat.***):* Let $T(P,DB)$ be defined as: $(\sum_{i\in\{sid|\exists(sid,x,y)\in Inst(P,DB)\}}\cdot\ i$ + $\sum_{(sid,x,j)\in Inst(P,DB)}\cdot\ j)$. Consider patterns $P$ and $P'$, if $T(P,DB)\neq T(P',DB)$, then $DB_{P}^{app-all}\neq DB_{P'}^{app-all}$.

At line 13 in Figure 3, to find if there exists another pattern having the same projected database, we first hash a newly formed pattern $P$'s projected database by $T(P,DB)$, where $DB$ is the input sequence database, and locate a bucket containing potential equivalent projected databases. Only projected databases (if any) belonging to this bucket need to be compared for equivalence. We find that the approach works well since the key (*i.e.*, $T(P,DB)$) is well distributed.

```
Procedure Generate_PAL
Inputs:   $DB$: A sequence database;
          $min\_sup$: Minimum support threshold;
Outputs:  $PAL$: Proj-db-based Approximate pattern Lattice ;
Method:
1: Let $FreqEvs = \{ev \mid (sup(ev,DB) \geq min\_sup)\}$
2: Let $root$ = Create new root node
3: Let $PAL$ = $root$
4: For each pattern $ev \in FreqEvs$
5:    Let $N_{new}$ = Create new node ($ev$,$DB_{ev}$)
6:    Append $N_{new}$ as child of $root$
7:    Call $Extend\_PAL$ ($DB_{ev}$,$N_{new}$,$PAL$,$min\_sup$)
8:    Append $FV(ev)$ information to $N_{new}$
9: Output $PAL$

Procedure Extend_PAL
Inputs:   $PDB$: A pseudo-projected approx-all database;
          $ParentNode$: Current node in $PSL$;
          $PAL$: Proj-db-based Approximate pattern Lattice;
          $min\_sup$: Minimum support threshold;
Method:
10: $FrNxEvs = \{ev \mid |PDB_{ev}^{app-nxt}| \geq min\_sup\}$
11: For each pattern $ev \in FrNxEvs$
12:    Let $N_{new}$ = Create new node ($ev$,$PDB_{ev}^{app-nxt}$)
13:    If ($\exists$ node $N_O \in PAL$. $N_O.ProjDB$ = $N_{new}.ProjDB$)
14:       Add $N_O$ as a child of $ParentNode$
15:       Label the link from $ParentNode$ to $N_O$ with the
           number of current parents of $N_O$.
16:    Else
17:       Add $N_{new}$ as a child of $ParentNode$
18:       Label the link from $ParentNode$ to $N_{new}$ with 1
19:       Call ExtendPAL ($PDB_{ev}^{app-nxt}$,$N_{new}$,$PAL$,$min\_sup$)
```

Fig. 3. Search Space Compaction into PAL

## 5.3 Mining Generators from PAL

To mine for generators from PAL, there are several challenges. First, we need to efficiently compute the actual support of iterative patterns and efficiently store this in PAL. The support of approximate patterns are only an over approximation of actual support of iterative pattern. Since for every node we store its corresponding projected approx-all database, we can always do backward traversal to find actual support of patterns. This is done by removing approximate instances of the pattern that does not satisfy the actual instance constraint defined in Definition 3.2.

Since a node can correspond to different patterns having the same projected approx-all database, there are potentially a number of actual support values corresponding to a node in PAL. There is a need for labels to uniquely identify different paths from root ending in a node $n$; each path corresponds to a different pattern with potentially different actual support.

A naive way is to label a path by all the transitions it traverses and use this label as the path identifier. However, this will be an overkill as the path can be long for long patterns. Our solution is to only include labels of important transitions in the lattice as the path identifier. When a path passes a transition $t$ sinking at a node $n$ with more than one incoming transitions, only then the transition label $t.Id$ will be appended as part of the path id.

Consider the example PAL data structure in Figure 2. Only transitions sinking in nodes having more than 1 incoming transitions are labeled. Considering the PAL structure for the example database $ExDB$ (see Table 1), we note that patterns $P$ $\langle A,B,C,D\rangle$ and $P'$ $\langle D\rangle$ has the same approximate projected database and share a node in the PAL, however their actual support is different – $sup(P)$ = 3 while $sup(P')$ = 4. To store and differentiate their support values, we use their path id. The path id of $P$ is $\langle 1,1\rangle$ (corresponding to transitions to node $N_2$ and $N_3$) while the path id of $P'$ is $\langle 4\rangle$ (corresponding to the transition to node $N_3$).

Traversal of the entire paths in PAL with computation of actual pattern supports will produce the full-set of frequent iterative patterns. However, this is not our goal as we want to mine the set of generators.

Hence, next we identify which patterns are generators. We try to avoid candidate generator generation. This is needed to avoid storage of candidate generators in the memory and expensive filtering of candidates which are not generators.

To detect and identify generators, we use Theorem 1. A pattern is a generator if none of its *sub-patterns* (defined in Theorem 1) have the same support as itself. We check the support of each sub-pattern by traversing the PAL from top to bottom. Since we will need the support of $P$'s sub-patterns to decide whether a pattern $P$ is a generator, we traverse the PAL breadth first. We first compute all patterns of length 1. We then continue to patterns of length 2, 3, .... Hence, when we check whether a pattern $P$ is a generator, we have the actual support of $P$'s sub-patterns.

As an example, consider the pattern $\langle C,D\rangle$ mined from $ExDB$. One of its *sub-patterns* $\langle D\rangle$ has the same support as itself. Hence, it is not a generator.

It is not profitable to check for every path in the PAL to see whether they are generators. The mining cost will then be larger than mining a full-set of patterns from the PAL. This would not be efficient. Rather, we employ several search space pruning strategies to prune the sub-search spaces containing non-generators.

We use Theorem 3 to prune the search space containing non-generators. For a pattern $P$, we look for $P'\in\cup_{ev\in P}$ $erasure(P,ev)$. If we detect that there is a $P'$ such that $sup(P)$= $sup(P')$, $DB_{P}^{app-all}$ = $DB_{P'}^{app-all}$ and $Cons-tr(P)$ − $Constr(P')\notin FV(P)$, according to Theorem 3, $P$ and its extensions (*i.e.*, $P$++$evs$, where $evs$ is an arbitrary series of events) are not generators. We can then prune the sub-search space in the PAL corresponding to patterns of the form $P$++$evs$. Since we explore PAL breadth-first, the supports of all $P'$s would have been computed before $P$ is evaluated.

For example, consider pattern $\langle A, C\rangle$ mined from $ExDB$. We note that $\langle C\rangle$ has the same support and projected database as $\langle A, C\rangle$. Also, $A$ is not in $FV(\langle C\rangle)$. From Theorem 3, there is no need to extend $\langle A, C\rangle$ further. The search space can be pruned.

The algorithm to mine iterative generators from PAL is shown in Figure 4.

```
Procedure Mine_Gen
Inputs:  $PAL$: Proj-db-based Approximate pattern Lattice;
         $min\_sup$: Minimum support threshold;
Result: Generators are outputted to a file
Method:
1:  Let $root$ = Get the root node of $PAL$
2:  Let $Trans$ = Get transitions from $root$ to its children
3:  For each $t$ in $Trans$
4:      Let $chd$ = Sink node of $t$
5:      Let $eid$ = ($chd$ has $>$ one parent) ? $chd$.Id : ‘’
6:      Call $Extend\_Gen$ ($chd$, $chd$.Event, $eid$, $min\_sup$)

Procedure Extend_Gen
Inputs:  $CNode$: Current node in PAL;
         $Pat$: Pattern considered;
         $pid$: Path identifier;
         $min\_sup$: Minimum support threshold;
Method:
7:  Let $realsup$ = Compute actual support of $Pat$ (see text)
8:  Add ($pid, realsup$) to $CNode$.SupTable
9:  If ($realsup \geq min\_sup$)
10:     If ($Pat$ is a generator acc. to Thm. 1)
11:         Output $Pat$
12:     If $\neg$($Pat$++$evs$ can be pruned acc. to Thm. 3)
13:         Let $Trans$ = Get trans. from $CNode$ to its children
14:         For each $t$ in $Trans$
15:             Let $chd$ = Sink node of $t$
16:             Let $npat$ = $Pat$++($chd$.Ev)
17:             Let $nid$ = $pid$++($chd$ has $>$ one parent)?$t$.Id:‘’
18:             Call $Extend\_Gen$ ($chd$, $npat$, $nid$, $min\_sup$)
```

Fig. 4. Mine Generators

## 5.4 Generating Representative Rules

We would like to generate rules with minimum pre-condition and maximum post-condition satisfying a minimum confidence threshold *min_conf*. Different from past studies on rule mining, we would like to discover rules representing not only forward temporal constraints but also backward and in-between temporal constraints.

To create a representative rule, we pair an iterative generator $G$ and a closed iterative pattern $C$. From the pairings, one will be able to see what events are pre-pended, inserted and appended to the generators. These events correspond to the events that is likely to happen before, in-between and after a precursor series of events. The challenge is that one needs to ensure that every instance of a closed pattern $C$ corresponds to an instance of the generator $G$ it pairs with. To do so, we use this property:

*Property 4:* Given a generator $G$ and closed pattern $C$ where $C \sqsupset G$, every instance of $C$ corresponds to an instance of $G$ iff $G$ is a substring of a pattern $P$ formed by erasing all events not in $G$ from $C$.

We keep the rule generation step simple by simply checking each pair of generator $G$ and closed pattern $C$ where $|C|$ $>$ $|G|$ and $sup(C)/sup(G) \geq$ *min_conf* for the satisfaction of Property 4. Each pair of $C$ and $G$ satisfying the property corresponds to a representative rule. We assume the number of closed patterns and generators are reasonable enough for this step to be performed scalably. In our case study, we find that this final step incurs the minimum computation cost. A rule composed of a generator $G$ and a closed pattern $C$ is denoted by $rule(G, C)$.

# 6 Performance and Comparative Study

We conducted extensive experiments for the performance evaluation and a comparative study of our algorithm. In this section, we report the performance/scalability results on benchmark synthetic data sets, and present comparative results on 3 data sets including two real-life benchmark data sets.

## 6.1 Performance Study

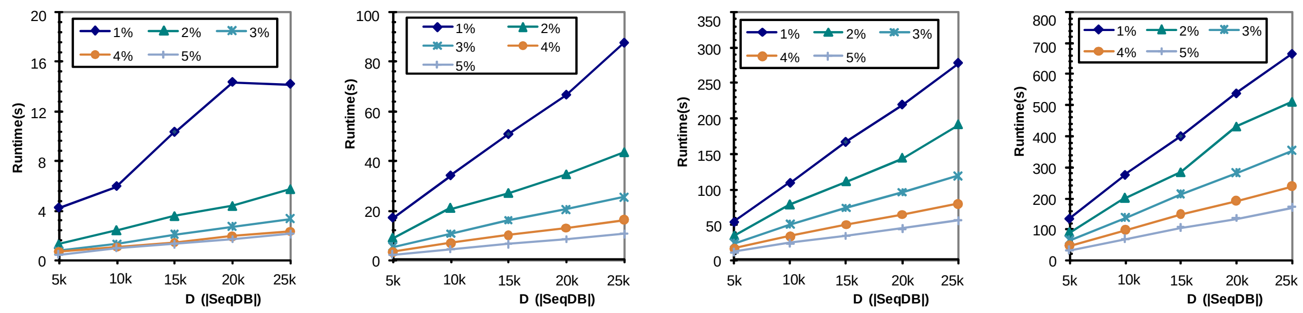
In this sub-section, the performance of our mining algorithm when the support values and database size (number of sequences, and average length of sequences) are varied is studied. We use a synthetic data generator provided by IBM (the one used in [10]) with slight modification to generate sequences of *events*. The data generator accepts 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 patterns. In these set of experiments, we set the repetition factor parameter of IBM data generator to 0.5 out of 1. Experiments were conducted on a Pentium M 1.6GHz IBM X41 tablet PC with 1.5GB main memory, running Windows XP Tablet PC Edition 2005. Algorithms were written using C#.Net compiled using “Release” mode of VS.Net 2005.

The results of the scalability experiments are shown as line graphs in Figures 5, 6, and 7.The Y-axis corresponds to the runtime taken or the number of generated patterns. The X-axis corresponds to the number of sequences in the database ($|SeqDB|$) or the average sequence length – these are parameters of the synthetic data generators. For each graph we plot 5 different 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.

Figure 5 shows the runtime (in linear scale)[1] when the number of sequences in the database is increased. From the figure we can note that generally the runtime grows linearly with the number of sequences in the database. Figures 6 & 7 show the runtime (in log scale) and number of patterns (in log scale) when the average sequence length is increased.[2] From the figure we note that the runtime corresponds to the

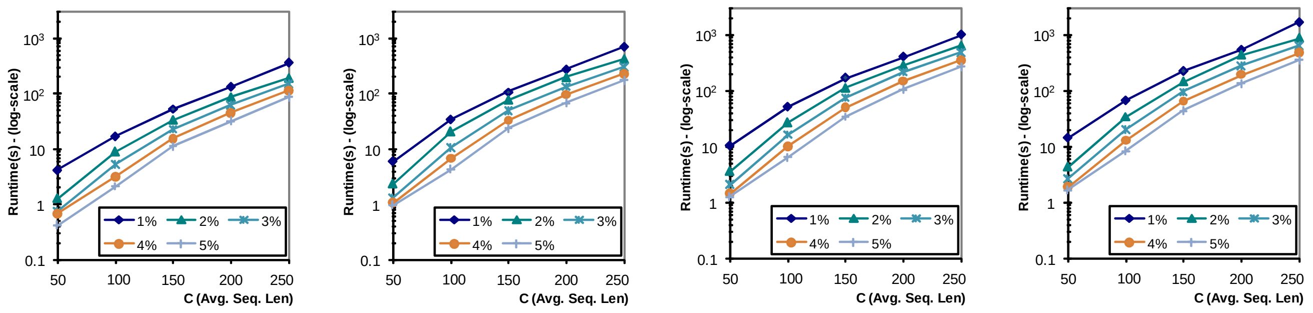
1. Due to space limitation, we omit the plot of the number of patterns vs. variation of $min\_sup$ and $|SeqDB|$.

2. Note that we set parameters C and S of the synthetic data generator to be of the same value. Hence, the patterns are more likely to be longer and harder to mine.



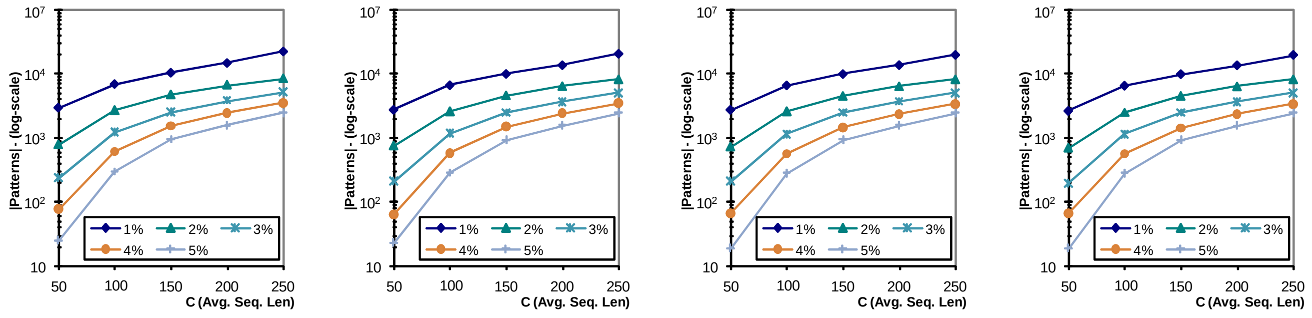
$(a) D[5-25]C50N10S50$ $(b) D[5-25]C100N10S100$ $(c) D[5-25]C150N10S150$ $(d) D[5-25]C200N10S200$

Fig. 5. **Varying $min\_sup$ and D ($|SeqDB|$) for Synthetic Datasets - Runtime**



$(a) D5C[50-250]N10S[=C]$ $(b) D10C[50-250]N10S[=C]$ $(c) D15C[50-250]N10S[=C]$ $(d) D20C[50-250]N10S[=C]$

Fig. 6. **Varying $min\_sup$ and C (Avg. Seq. Len.) for Synthetic Datasets - Runtime**



$(a) D5C[50-250]N10S[=C]$ $(b) D10C[50-250]N10S[=C]$ $(c) D15C[50-250]N10S[=C]$ $(d) D20C[50-250]N10S[=C]$

Fig. 7. **Varying $min\_sup$ and C (Avg. Seq. Len.) for Synthetic Datasets - $|$Patterns$|$**

number of mined generators. The runtime is more sensitive to the length of sequences than the number of sequences in the database. In general, this is the case with pattern mining algorithms when the size of transactions (for itemset mining) or the length of sequences is increased.

To mine from a set of very long sequences, we plan to follow a similar approach with Lo and Maoz [31] by embedding intuitive constraints familiar to software engineers and providing support for user-guidance during the mining process. In this study, we focus on a general purpose algorithm which is fully automated without the need of additional user input aside from the support threshold.

The memory requirement of the algorithm for the largest dataset D25C250N10S250 at the lowest support threshold considered (at 1%) is capped at 690MB. The memory requirement of the algorithm for the smallest dataset D5C50N10S50 at the highest support threshold considered (at 5%) is capped at 33MB. The memory requirement when the application starts is around 18MB (it comes with a Graphical User Interface). The memory requirement is reasonable considering the main memory of current off-the-rack PCs. A lower memory consumption can also be obtained by performing minor modifications to data structures used, e.g., by using ushort (16 bit) rather than int (32 bit).

## 6.2 Comparative Study

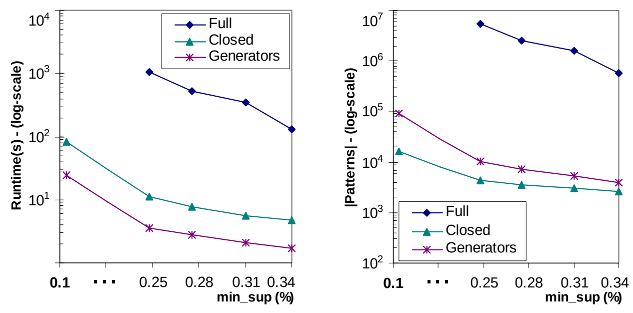
In this sub-section, we compare the performance of our algorithm with two algorithms developed in [9]: one of them for mining a complete set of frequent patterns, the other for mining frequent closed iterative patterns. Similar to other studies on mining from sequences [9], [24], [25], low support thresholds are utilized to test for scalability. For comparison purpose, the same datasets, thresholds and environment used in [9] are reused. Our experiments are on the synthetic

D5C20N10S20 dataset, and two real datasets: a click stream dataset and TCAS program trace dataset.

The click stream dataset (*i.e.*, Gazelle dataset) is from KDDCup 2000 [37] which was previously examined in [9], [24], [25]. It contains 29369 sequences with an average length of 3 and a maximum length of 651. Though many of the sequences are short, there are also some long sequences (*i.e.*, the maximum length is 651). Since we consider very low support thresholds, these long sequences actually are considered significantly during mining.

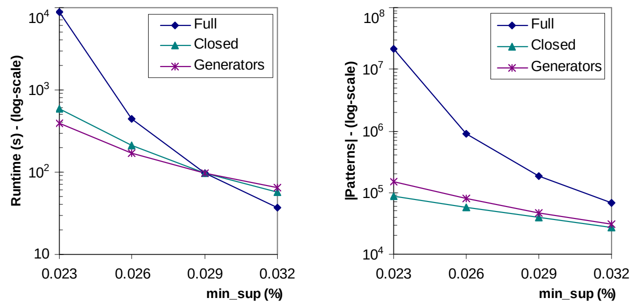
To evaluate the performance of our algorithm on real program traces, we generate traces from the Traffic alert and Collision Avoidance System (TCAS) of the Siemens Test Suite [38], which has been used as one of the benchmarks for research in software testing and error localization. This test suite comes with 1578 correct test cases. We run these test cases and obtained 1578 traces. The sequences are of average length of 36 and maximum length of 70. In total, it contains 75 different events - the events corresponding to the basic block ids of the control flow graph of TCAS. We call this dataset the TCAS dataset.

The result of the experiments on the D5C20N10S20, Gazelle and TCAS dataset performed using algorithms mining generators (Generators), closed patterns (Closed) and all frequent patterns (Full) are shown in Figure 8, 9 & 10 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.



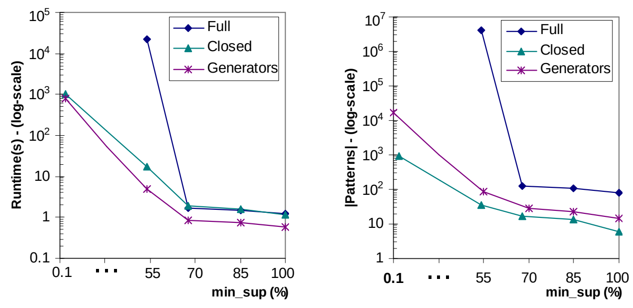
$(a) Runtime$ $(b) No.\ of\ Patterns$

Fig. 8. **Varying** $min\_sup$ **for D5C20N10S20 dataset. Note: We need both closed patterns and generators to form representative rules.**



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

Fig. 9. **Varying** $min\_sup$ **for Gazelle dataset**



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

Fig. 10. **Varying** $min\_sup$ **for TCAS dataset**

From the results, for most cases generators can be mined with similar or better efficiency than closed patterns. This is despite the fact that the number of generators is larger than the number of closed patterns. Note that it is not the case that a larger number of patterns always corresponds to a longer running time (see also study in [16]). Although there are more generators than closed patterns, due to the effectiveness of the pruning strategy generator mining algorithm can complete faster than closed pattern mining algorithm does. Generator and closed pattern mining are aimed at different types of patterns, hence it is not our primary purpose here to beat the efficiency of mining closed iterative pattern by a large factor.

The more important result that we can observe from Figures 8, 9 & 10 is that the runtime sum of mining iterative generators and closed patterns is much less than the time of mining the full set of frequent patterns. This result is significant because we can gain much efficiency to form representative rules by mining both iterative generators and closed patterns, instead of mining the full set of frequent patterns. That is, to eventually find representative rules, mining both iterative and closed patterns is effective, otherwise one would need to mine the full set of frequent patterns which might be very large in size and which might take a prohibitively long time.

For example, on the TCAS dataset, the iterative generator miner was able to run even at the lowest possible support threshold (at 1 instance) and finished within 14 minutes. On the other hand, the algorithm mining all frequent patterns did not stop with excessive runtime ($>$ 6 hours) even at a relatively high support threshold of 867. In general for all datasets, even at very low support, generator miner was able to complete within less than 14 minutes.

## 7 Case Study

For comparison and continuity purposes, we experimented with the case study on the transaction component of JBoss Application Server (JBoss AS) originally reported in [9]. The purpose of this case study is to show the usefulness of the mined generators in supplementing closed patterns in generating useful rules describing behaviors of the transaction sub-component of JBoss AS.

The JBoss trace set contains 28 sequences of a total of 2551 events and an average of 91 events. The longest trace

| Connection Set Up |
|---|
| 1. TransactionManagerLocator.getInstance |
| 2. TransactionManagerLocator.locate |
| 3. TransactionManagerLocator.tryJNDI |
| 4. TransactionManagerLocator.usePrivateAPI |

| Tx Manager Set Up |
|---|
| 5. TxManager.begin |
| 6. XidFactory.newXid |
| 7. XidFactory.getNextId |
| 8. XidImpl.getTrulyGlobalId |

| Transaction Set Up |
|---|
| 9. TransactionImpl.associateCurrentThread |
| 10. TransactionImpl.getLocalId |
| 11. XidImpl.getLocalId |

| Transaction Set Up (Con't) |
|---|
| 12. LocalId.hashCode |
| 13. TransactionImpl.equals |
| 14. TransactionImpl.getLocalIdValue |
| 15. XidImpl.getLocalIdValue |
| 16. TransactionImpl.getLocalIdValue |
| 17. XidImpl.getLocalIdValue |

| Transaction Commit |
|---|
| 18. TxManager.commit |
| 19. TransactionImpl.commit |
| 20. TransactionImpl.beforePrepare |
| 21. TransactionImpl.checkIntegrity |
| 22. TransactionImpl.checkBeforeStatus |

| Transaction Commit (Con't) |
|---|
| 23. TransactionImpl.endResources |
| 24. TransactionImpl.completeTransaction |
| 25. TransactionImpl.cancelTimeout |
| 26. TransactionImpl.doAfterCompletion |
| 27. TransactionImpl.instanceDone |

| Transaction Dispose |
|---|
| 28. TxManager.releaseTransactionImpl |
| 29. TransactionImpl.getLocalId |
| 30. XidImpl.getLocalId |
| 31. LocalId.hashCode |
| 32. LocalId.equals |

Fig. 11. **Longest mined rule (32 events) from JBoss transaction component – read from top-to-bottom, left-to-right**

is of 125 events. There are 64 unique events. Each event corresponds to an invocation of a method call. Using min_sup of 65%, the closed iterative pattern and iterative generator mining algorithms take the same time (29s) to complete. Using min_conf of 95%, the rule generation process is completed within 1s. The algorithm mining all frequent patterns does not complete even after running for *more than 8 hours* and produces more than *5 GB of patterns*.

We perform the following post-processing steps:

1. Density. Filter away rules whose number of unique events is $<$ 80% of its length.
2. Ranking. Order the rules according to their lengths and support values
3. Rule subsumption. Only report a mined rule $R$ = $rule(G,C)$, if there does not exist another mined rule $R'$ = $rule(G',C')$ where $G' \sqsubseteq G$ and $C' \sqsupseteq C$.

Steps 1 and 2 is performed early before the patterns are composed into rules by filtering and sorting generators and closed patterns. After the rules are generated, step 3 is performed. The rules are grouped into rule-groups. A rule group corresponds to a closed pattern and a set of generators related to it. There are 87 rule groups mined.

After running the closed pattern mining algorithm, we found at least 5 interesting specifications corresponding to software behavioral patterns:

| | |
|---|---|
| C1 | ⟨Connection Set Up Evs, TxManager Set Up Evs, Transaction Set Up Evs, Transaction Commit Evs, Transaction Disposal Evs⟩ |
| C2 | ⟨Connection Set Up Evs, TxManager Set Up Evs, Transaction Set Up Evs, Transaction Rollback Evs, Transaction Disposal Evs⟩ |
| C3 | ⟨Resource Enlistment Evs, Transaction Execution Evs, Transaction Commit Evs, Transaction Disposal Evs⟩ |
| C4 | ⟨Resource Enlistment Evs, Transaction Execution Evs, Transaction Rollback Evs, Transaction Disposal Evs⟩ |
| C5 | ⟨Lock-Unlock Evs⟩ |

These patterns correspond to closed patterns of longest length and highest support. Note that the each "Evs" corresponds to a series of events. The longest pattern $C1$ is composed of 32 events. In addition to the closed patterns, with the mining of generators, we are also able to describe pre-condition events that underlie the long pattern hence forming a rule. Important minimal pre-condition events for each of the 5 patterns ($Ci$ is paired with $Gi$) are as follows:

| | |
|---|---|
| G1 | ⟨TxManager.Commit⟩ |
| G2 | ⟨TransactionImpl.rollbackResources⟩ |
| G3 | ⟨TransactionImpl.enlistResource, TxManager.Commit⟩ |
| G4 | ⟨TransactionImpl.enlistResource,TransactionImpl.-rollbackResources⟩ |
| G5 | (a) ⟨lock⟩ and (b) ⟨unlock⟩ |

With these minimal pre-conditions, the mined specifications are more complete. Each of them is a rule stating what series of events happen *after*, *before* and *in-between* the minimal pre-conditions. For example, for the shortest rule involving lock and unlock, the rule group mined is able to say dually that: (1) when lock is acquired, lock must be released, (2) when lock is released, lock must be acquired before. Also, consider the longest rule mined. These are 5 minimal pre-conditions. These are marked with red dashed-boxes in Figure 7.

Furthermore, since generators can be a combination of events, as are the cases with pre-conditions G3 and G4 above, mined specifications can describe the series of events that must happen in between the pre-condition's constituent events. The specification $rule$(G4,C4) describes 24 other events that are bound to occur with likelihood more than 95% when two events $enlistResource$ and $rollbackResources$ are called. One of these events occur before $enlistResource$, nine occur in between the two events, and 14 others occur after $rollbackResources$.

## 8 Conclusion

In this paper, we propose a novel algorithm to mine iterative generators as a step forward in developing data mining tools addressing the needs of software engineers. We also introduce a novel concept of representative rules expressing forward, backward and in-between temporal constraints. These rules characterize many real-world constraints including those found in software specifications. These rules can be mined scalably by pairing iterative generators with closed iterative patterns.

To mine generators, we first construct a novel compact lattice representing frequent approximate patterns referred to as Equivalent $\underline{P}$rojected-database-based $\underline{A}$pproximate Pattern

_L_attice, shortened as *PAL*. We propose a novel method involving a synergy of depth- and breadth-first search to mine generators.

We build PAL depth-first and employ redundant search space traversal efforts by detection of equivalent sub-lattices considering the semantics of iterative patterns. Next, we utilize a breadth-first traversal strategy on the compact lattice while performing: (1) Computation of real pattern support, (2) Identification of generators, and (3) Pruning of sub-search-spaces containing non-generators via a novel pruning property.

The extensive performance study conducted shows that the algorithm can run efficiently at low support thresholds, large number of sequences and reasonably long sequences on 10 synthetic and 2 real benchmark datasets with similar or better efficiency as closed iterative patterns. Since we need both generators and closed patterns to form representative rules, it is important that the two algorithms' efficiencies are similar. A case study on traces of JBoss AS shows how mined iterative generators can be merged with closed iterative patterns to form representative rules shedding light on software design.

As future work, we plan to further improve the efficiency of the mining algorithm and conduct more case studies. We also plan to supplement our mining engine with a tool for visualization and editing of mined patterns and rules, integrated to a standard software development platform e.g. Eclipse [39]. It would be interesting to extend the study to mine for representative rules involving disjunction, to mine from a database of sequences of transactions and to consider incremental mining.

**Acknowledgement.** We wish to thank Blue Martini Software for contributing the KDD Cup 2000 data.

## References

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

[2] S. Deelstra, M. Sinnema, and J. Bosch, "Experiences in software product families: Problems and issues during product derivation," in *Proceedings of International Software Product Line Conference*, 2004.

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

[4] T. Standish, "An essay on software reuse." *IEEE Transaction on Software Engineering*, vol. 10, pp. 494–497, 1984.

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

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

[7] Java Transaction API (JTA), "http://java.sun.com/javaee/technologies/jta/index.jsp (accessed 18 Dec 2008)."

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

[9] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery." in *Proceedings of SIGKDD Conference on Knowledge Discovery and Data Mining*, 2007.

[10] R. Agrawal and R. Srikant, "Mining sequential patterns." in *Proceedings of IEEE International Conference on Data Engineering*, 1995.

[11] H. Mannila, H. Toivonen, and A. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, pp. 259–289, 1997.

[12] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das, "Perracotta: Mining temporal API rules from imperfect traces." in *Proceedings of International Conference on Software Engineering*, 2006.

[13] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. Nguyen, "Verifying safety policies with size properties and alias controls," in *Proceedings of International Conference on Software Engineering*, 2005.

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

[15] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *Proceedings of Symposium on Principles of Database Systems*, 1999.

[16] J. Li, H. Li, L. Wong, J. Pei, and G. Dong, "Minimum description length principle: Generators are preferable to closed patterns." in *Proceedigns of AAAI Conference on Artificial Intelligence*, 2006.

[17] J. Li, G. Liu, and L. Wong, "Mining statistically important equivalence classes and delta-discriminative emerging patterns." in *Proceedings of SIGKDD Conference on Knowledge Discovery and Data Mining*, 2007.

[18] D. Lo, S.-C. Khoo, and J. Li, "Mining and ranking generators of sequential rules," in *Proceedings of SIAM International Conference on Data Mining*, 2008.

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

[20] M. Spiliopoulou, "Managing interesting rules in sequence mining." in *Proceedings of European Conference on Principles of Data Mining and Knowledge Discovery*, 1999.

[21] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of recurrent rules from a sequence database." in *Proceedings of International Conference on Database Systems for Advanced Applications*, 2008.

[22] ——, "Mining past-time temporal rules from execution traces," in *Proceedings of International Workshop on Dynamic Analysis*, 2008.

[23] "MarkingQueuedIrps," msdn.microsoft.com/en-us/library/aa469118.aspx (accessed 18 Dec 2008).

[24] X. Yan, J. Han, and R. Afhar, "CloSpan: Mining closed sequential patterns in large datasets." in *Proceedings of SIAM International Conference on Data Mining*, 2003.

[25] J. Wang and J. Han, "BIDE: Efficient mining of frequent closed sequences." in *Proceedings of IEEE International Conference on Data Engineering*, 2004.

[26] G. Garriga, "Discovering unbounded episodes in sequential data." in *Proceedings of European Conference on Principles of Data Mining and Knowledge Discovery*, 2003.

[27] M. Zhang, B. Kao, D. Cheung, and K. Yip, "Mining periodic patterns with gap requirement from sequences," in *Proceedings of SIGMOD International Conference on Management of Data*, 2005.

[28] B. Ding, D. Lo, J. Han, and S.-C. Khoo, "Efficient mining of closed repetitive gapped subsequences from a sequence database (to appear)." in *Proceedings of IEEE International Conference on Data Engineering*, 2009.

[29] C. Gao, J. Wang, Y. He, and L. Zhou, "Efficient mining of frequent sequence generators," in *Proceedings of International Conference on World Wide Web (poster)*, 2008.

[30] D. Lo, S. Maoz, and S.-C. Khoo, "Mining modal scenario-based specifications from execution traces of reactive systems," in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*, 2007.

[31] D. Lo and S. Maoz, "Mining scenario-based triggers and effects." in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*, 2008.

[32] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[33] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps, "Temporal logic for scenario-based specifications," in *Proceedings of International Conference on the Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[34] K. Olender and L. Osterweil, "Cecil: A sequencing constraint language for automatic static analysis generation." *IEEE Transaction on Software Engineering*, vol. 16, pp. 268–280, 1990.

[35] "Windows Driver Kit: Driver Development Tools – CancelSpinLock," msdn.microsoft.com/en-us/library/aa469115.aspx (accessed 18 Dec 2008).

[36] 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 *Proceedings of IEEE International Conference on Data Engineering*, 2001.

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

[38] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria," in *Proceedings of International Conference on Software Engineering*, 1994.

[39] Eclipse.org home, "www.eclipse.org (accessed 25 Dec 2008)."