Efficient Mining of Recurrent Rules from a Sequence Database

David Lo[1], Siau-Cheng Khoo[1], and Chao Liu[2]

[1] Department of Computer Science, National University of Singapore
[2] Department of Computer Science, University of Illinois at Urbana-Champaign
dlo@comp.nus.edu.sg, khoosc@comp.nus.edu.sg, chaoliu@cs.uiuc.edu

Abstract. We study a novel problem of mining significant recurrent rules from a sequence database. Recurrent rules have the form "whenever a series of precedent events occurs, eventually a series of consequent events occurs". Recurrent rules are intuitive and characterize behaviors in many domains. An example is in the domain of software specifications, in which the rules capture a family of program properties beneficial to program verification and bug detection. Recurrent rules generalize existing work on sequential and episode rules by considering repeated occurrences of premise and consequent events within a sequence and across multiple sequences, and by removing the "window" barrier. Bridging the gap between mined rules and program specifications, we formalize our rules in linear temporal logic. We introduce and apply a novel notion of rule redundancy to ensure efficient mining of a compact representative set of rules. Performance studies on benchmark datasets and a case study on an industrial system have been performed to show the scalability and utility of our approach.

1 Introduction

The information age has caused an explosive growth in the amount of data produced. Mining for knowledge from data has been shown useful for many purposes [12] ranging from finance, advertising, bio-informatics and recently software engineering [17,20]. Addressing the same issue of knowledge discovery from data, we study the problem of mining recurrent rules, each having the following form:

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

The above rule is intuitive and represent an important form of knowledge characterizing the behaviors of many systems appearing in various domains. Examples of rules in this format include:

1. Resource Locking Protocol: Whenever a lock is acquired, eventually it is released.
2. Internet Banking: Whenever a connection to a bank server is made and an authentication is completed and money transfer command is issued, eventually money is transferred and a receipt is displayed.

J.R. Haritsa, R. Kotagiri, and V. Pudi (Eds.): DASFAA 2008, LNCS 4947, pp. 67–83, 2008.

3. Network Protocol: Whenever an HDLC connection is made and an acknowledgement is received, eventually a disconnection message is sent and an acknowledgement is received.

Zooming into the domain of software specification and verification, recurrent rules correspond to a family of program properties useful for program verification (*c.f.* [9]). The first example given above corresponds to a program property. Research in software verification addresses rigorous approaches to check the correctness of a software system with respect to a formal specification which often corresponds to a set of program properties (*c.f.*, [26,6]). However, specification might often be outdated or missing due to software evolution, reluctance in writing formal specification and short-time-to-market cycle of software development (*c.f.*, [8,3,5]). Recovering or mining specifications expressed as rules and automata has been a recent interest in software engineering and programming language domain [27,29,3,19]. However, recent approaches on mining rules as specification [27,29] has only focused on two-event rules due to the exponential complexity associated with mining rules of arbitrary length.

To address the above issue, in this paper we propose a novel extension of work on pattern mining, in particular sequential pattern mining [2] and episode mining [22]. Sequential pattern mining first addressed by Agrawal and Srikant in [2] discovers patterns that are supported by a *significant number of sequences*. A pattern is supported by a sequence if the former is a sub-sequence of the later. On the other hand, Mannila *et al.* perform episode mining to discover *frequent episodes within a sequence of events* [22]. An episode is supported by a window if it is a sub-sequence of the series of events appearing in the window. Garriga later extends Mannila *et al.*'s work to replace a fixed-window size with a gap constraint between one event to the next in an episode [11]. In both cases, an episode is defined as a series of events occurring *relatively close* to one another (*i.e.* they occur in the same window). Episode mining focuses on mining from a single sequence of events.

Rules can be formed from both sequential patterns and episodes as proposed in [24,22]. Different from a pattern, a rule expresses a *constraint* involving its premise (*i.e.*, pre-condition) and consequent (*i.e.*, post-condition). These constraints are needed for potential uses of rules in filtering erroneous sequences, detecting outliers, etc.

However, rules from sequential patterns and episodes have different semantics from recurrent rules. A sequential rule $pre \rightarrow post$ states: "whenever a sequence is a super-sequence of *pre* it will also be a super-sequence of *pre* concatenated with *post*". An episode rule $pre \rightarrow post$ states: "whenever a window is a super-sequence of *pre* it will also be a super-sequence of *pre* concatenated with *post*". Recurrent rules generalize sequential rules where for each rule, *multiple* occurrences of the rule's premise and consequent both *within a sequence and across multiple sequences* are considered. Recurrent rules generalize episode rules by allowing precedent and consequent events to be separated by an *arbitrary* number

of events in a *sequence database*. Also, a set of sequences rather than a single sequence is considered during mining.

These generalizations are needed in many application areas, and an example is in mining program properties from execution traces. Because of loops and recursions, an execution trace can contain repeated occurrences of a particular property. Also, program properties are often inferred from a set of traces instead of a single trace (*c.f* [3,19]). Finally, important patterns for verification, such as, lock acquire and release or stream open and close (*c.f* [29]) often have their events occur at some arbitrary distance away from one another in a program trace. Hence, there is a need to "break" the "window barrier" or "gap constraints" in order to capture program properties of interest.

Our goal is to mine a set of rules satisfying given support and confidence thresholds. To reduce the number of mined rules and improve efficiency, we define a novel notion of *rule redundancy* and devise search space pruning strategies to detect redundant rules "early" in the mining process. The final output is a set of non-redundant rules satisfying the given support and confidence thresholds.

In order to bridge the gap between mined rules and program specifications, we formalize our rules using linear temporal logic (LTL) – a widely used formalism in program verification [6]. By mapping rules to LTL expressions, these rules can be directly consumed by existing program verifiers.

We carry out a performance study on several standard benchmark datasets to demonstrate the effectiveness of our search space pruning strategies. We also perform a case study on traces of JBoss Application Server – the most widely used J2EE server – to illustrate the usefulness of our technique in recovering the specifications that a software system obeys.

The contributions of this work are as follows:

1. We present a novel notion of recurrent rules along with its mining algorithm.
2. We introduce and utilize the definition of redundant rules to reduce the number of mined rules.
3. We employ two "apriori"-like properties and "early" detection of redundant rules effective in aiding the scalability of rule mining.
4. We bridge the gap between data mining and program verification by translating mined rules to useful LTL expressions.
5. We show a case study on the utility of our technique in recovering specifications of a large industrial programs.

The outline of this paper is as follows. In Section 2, we discuss related work. Section 3 contains important background information on LTL formalizing our definition of recurrent rules. Section 4 presents the principles behind mining recurrent rules and the pruning strategies employed. Section 5 presents our algorithm. Section 6 describes the study conducted to evaluate the performance of our mining framework and the benefits of various pruning strategies. Section 7 describes our case study, and Section 8 concludes this paper and presents some future work.

2 Related Work

Two related research threads are sequential pattern mining(*e.g.*, [2,28,25,24]) and episode mining (*e.g.*, [22,11]). This work can be viewed as an extension of both sequential rules and episode rules. Differences between our work and the above have been discussed in the introduction section.

In the area of specification mining, a number of studies on mining software temporal properties have been performed [29,27,3,19]. Most of them mine an automata (*e.g.*, [3,19]) and hence are very different from our work. Of the most relevance is the work on mining rule-based specification [29,27], where the rules have a similar semantics as ours but are limited to two-event rules (*e.g.*, $\langle lock \rangle \rightarrow \langle unlock \rangle$). Their algorithms do not scale for mining multi-event rules since they first list all possible two-event rules and then check the significance of each rules. For rules of arbitrary lengths, the number of possible rules is arbitrarily large. Our work generalizes their work by mining a complete set of rules of arbitrary lengths that satisfy given support and confidence thresholds. To enable efficient mining, we devise a number of search space pruning strategies.

In [20], we proposed iterative patterns to discover software specifications, which are defined based on the semantics of Message Sequence Charts (MSC) [15]. Different from [20], this work is based on a different formalism, namely the semantics of Linear Temporal Logic (LTL). LTL has a wider application area than MSC, ranging from software engineering [29] to security & privacy [4]. In the software domain, LTL (but not MSC) is one of the most widely-used formalism for program verification (*i.e.*, ensuring correctness of a software system) [6]. There are many standard verification tools readily taking software properties expressed in LTL as inputs. Since the underlying target formalisms and semantics are different, both the search space pruning strategies and the mining algorithm are very different from our previous work in [20].

3 Preliminaries

This section introduces preliminaries on LTL and its verification which dictate the semantics of recurrent rules. Also, notations used in this paper are described.

Linear-time Temporal Logic Our mined rules can be expressed in Linear Temporal Logic (LTL) [14]. LTL is a logic that works on possible program paths. A possible program path corresponds to a program trace. A path can be considered as a series of events, where an event is a method invocation. For example, (file_open, file_read, file_write, file_close), is a 4-event path.

There are a number of LTL operators, among which we are only interested in the operators 'G','F' and 'X'. The operator 'G' specifies that *globally* at every point in time a certain property holds. The operator 'F' specifies that a property holds either at that point in time or *finally (eventually)* it holds. The operator 'X' specifies that a property holds at the *next* event. Some examples are listed in Table 1.

Our mined rules state whenever a series of precedent events occurs eventually another series of consequent events also occurs. A mined rule denoted as $pre \rightarrow$

Table 1. LTL Expressions and their Meanings

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

Table 2. Rules and their LTL Equivalences

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

post, can be mapped to its corresponding LTL expression. Examples of such correspondences are shown in Table 2. Note that although the operator 'X' might seem redundant, it is needed to specify rules such as $\langle a\rangle \rightarrow \langle b, b\rangle$ where the 'b's refer to *different occurrences of 'b'*. The set of LTL expressions minable by our mining framework is represented in the Backus-Naur Form (BNF) as follows:

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

Checking/Verifying LTL Expressions. LTL expressions are originally developed for checking software systems expressed in the form of automata [13] (a transition system with start and end nodes). There are existing tools converting code to an automata (*e.g.*, [7]). Given an automata and an LTL property one can check for its satisfaction through a well-known technique of model checking [6].

Consider the example in Figure 1, the pseudo-code on the left corresponds to the automaton on the right. Given the property $\langle main, lock\rangle \rightarrow \langle unlock, end\rangle$, a model checking tool (*c.f*, [6]) will ensure that for all states in the model where *lock* preceded by a *main* occurs (marked by the red dashed arrows), eventually (whichever path is taken) *unlock and* then eventually *end* can be reached. For the above example, the property is violated. The *lock* immediately before *end* is not followed by an *unlock*. Note however, the property $\langle main, lock, use\rangle \rightarrow \langle unlock, end\rangle$ is satisfied. This is the case since the *lock* immediately before *end* is not followed by a *use*, i.e., the pre-condition of the rule is not satisfied and the rule vacuously holds.

In this paper, we map this to sequences. We consider a sequence as a form of automata (a linear one). An event is mapped to a state. A mined rule (or property)

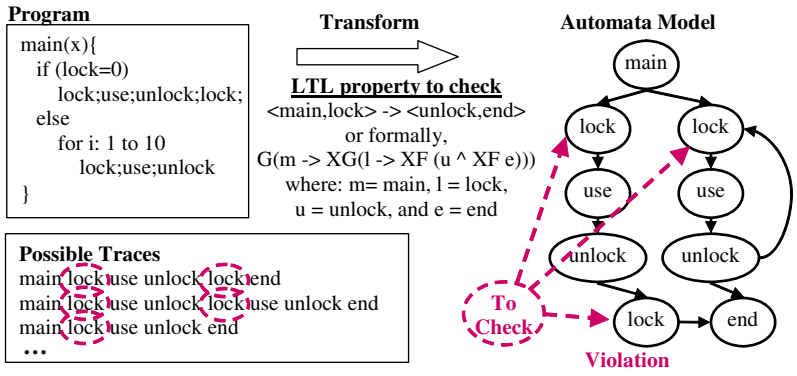

Fig. 1. Code -> Automata -> Verification

$pre \rightarrow post$ with a perfect confidence (*i.e.*, confidence=1) states that in the sequences from all states where the *pre* holds eventually *post* occurs. In the above example, for all points in the sequence (*i.e.*, temporal points) where $\langle main, lock\rangle$ occurs (marked with dashed red circle), one need to check whether eventually $\langle unlock, end\rangle$ occurs. Based on the definition of LTL properties and how they are verified, our technique analyzes sequences and captures *strong* LTL expressions that satisfy given support and confidence thresholds.

Basic Notations. Let I be a set of distinct events considered. The input to our mining framework is a sequence database referred to as $SeqDB$. Each sequence is an ordered list of events, and is denoted as $\langle e_1, e_2, \ldots, e_{end}\rangle$ where $e_i \in I$.

We define a pattern P to be a series of events. We use $last(P)$ to denote the last event of P. A pattern $P_1{+}{+}P_2$ denotes the concatenation of patterns P_1 and P_2. A pattern P_1 ($\langle e_1, e_2, \ldots, e_n\rangle$) is considered a *subsequence* of another pattern P_2 ($\langle f_1, f_2, \ldots, f_m\rangle$) if there exists integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $e_1 = f_{i_1}, e_2 = f_{i_2}, \cdots, e_n = f_{i_n}$ (denoted as $P_1 \sqsubseteq P_2$).

4 Generation of Recurrent Rules

Each recurrent rule has the form $P_1 \rightarrow P_2$, where P_1 and P_2 are two series of events. P_1 is referred to as the *premise* or *pre-condition* of the rule, while P_2 is referred to as the *consequent* or *post-condition* of the rule. The rules correspond to temporal constraints expressible in LTL notations. Some examples are shown in Table 2. We use the sample database in Table 3 as our running example.

Table 3. Example Database – $DBEX$

Seq ID.	Sequence
S1	⟨a, b, e, a, b, c⟩
S2	⟨a, c, b, e, a, e, b, c⟩

4.1 Concepts and Definitions

Mined rules are formalized as Linear Temporal Logic(LTL) expressions with the format: G(... → XF...). The semantics of LTL and its verification technique described in Section 3 will dictate the semantics of recurrent rules described here. Noting the meaning of the temporal operators illustrated in Table 1, to be precise, a recurrent rule expresses:

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

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

Definition 1 (Temporal Points). *Consider a sequence S of the form $\langle a_1, a_2, \ldots, a_{end}\rangle$. All events in S are indexed by their position in S, starting at 1 (e.g., a_j is indexed by j). These positions are called* temporal points *in S. For a temporal point j in S, the prefix $\langle a_1, \ldots, a_j\rangle$ is called the j-prefix of S.*

Definition 2 (Occurrences & Instances). *Given a pattern P and a sequence S, the* occurrences *of P in S is defined by a set of temporal points $\mathcal{T}$ in S such that for each $j \in \mathcal{T}$, the j-prefix of S is a super-sequence of P and $last(P)$ is indexed by j. The set of* instances *of pattern P in S is defined as the set of j-prefixes of S, for each $j \in \mathcal{T}$.*

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

We can then define a new type of database projection to capture events occurring after each temporal point. The following are two different types of projections and their associated support notions.

Definition 3 (Projected & Sup). *A database* projected *on a pattern p is defined as:*

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

Given a pattern P_X, we define $sup(P_X, SeqDB)$ to be the size of $SeqDB_{P_X}$ (equivalently, the number of sequences in $SeqDB$ containing P_X). Reference to the database is omitted if it is clear from the context.

Definition 4 (Projected-all & Sup-all). *A database* projected-all *on a pattern p is defined as:*
$SeqDB^{all}_P = \{(j, sx) \mid$ the j^{th} sequence in SeqDB is s, where $s = px{+}{+}sx$, and px is an instance of p in s $\}$

Given a pattern P_X, we define $sup^{all}(P_X, SeqDB)$ to be the size of $SeqDB^{all}_{P_X}$. Reference to the database is omitted if it is clear from the context.

Definition 3 is a standard database projection (*c.f.* [28,25]) capturing events occurring after the first *temporal point*. Definition 4 is a new type of projection capturing events occurring after *each temporal point*.

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

Table 4. $(a); DBEX_{\langle a,b\rangle}$ & $(b); DBEX^{all}_{\langle a,b\rangle}$

(a)

Seq ID.	Sequence
S1	⟨e, a, b, c⟩
S2	⟨e, a, e, b, c⟩

(b)

Seq ID.	Sequence
$S1_1$	⟨e, a, b, c⟩
$S1_2$	⟨c⟩
$S2_1$	⟨e, a, e, b, c⟩
$S2_2$	⟨c⟩

The two projection methods' associated notions of *sup* and sup^{all} are different. Specifically, sup^{all} reflects the number of occurrences of P_X in $SeqDB$ rather than the number of sequences in $SeqDB$ supporting P_X.

Example. Consider the example database, $sup(\langle a, b\rangle, DBEX) = |DBEX_{\langle a,b\rangle}| = 2$. On the other hand, $sup^{all}(\langle a, b\rangle, DBEX) = |DBEX^{all}_{\langle a,b\rangle}| = 4$.

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

Definition 5 ((S-/I-)Support & Confidence). *Consider a recurrent rule R_X ($pre_X \rightarrow post_X$). The [prefix-]*sequence-support *(s-support) of R_X is defined as the number of sequences in $SeqDB$ where pre_X occurs, which is equivalent to sup(pre_X, $SeqDB$). The* instance-support *(i-support) of R_X is defined as the number of* occurrences *of pattern $pre_X ++ post_X$ in $SeqDB$, which is equivalent to sup^{all}($pre_X ++ post_X$, $SeqDB$). The confidence of R_X is defined as the likelihood of $post_X$ happening after pre_X. This is equivalent to the ratio of sup($post_X$,$SeqDB^{all}_{pre_X}$) to the size of $SeqDB^{all}_{pre_X}$.*

Example. Consider $DBEX$ and a recurrent rule R_X, $\langle a, b\rangle \rightarrow \langle c\rangle$. From the database, the s-support of R_X is the number of sequences in $DBEX$ supporting (or is a super-sequence of) the rule's pre-condition – $\langle a, b\rangle$. There are 2 of them – see Table 4(a). Hence s-support of R_X is 2. The i-support of R_X is the number of *occurrences* of pattern $\langle a, b, c\rangle$ in $DBEX$ – *i.e.*, the number of *temporal points* where $\langle a, b, c\rangle$ occurs. There are also 2 of them. Hence, i-support of R_X is 2. The confidence of the rule R_X ($\langle a, b\rangle \rightarrow \langle c\rangle$) is the likelihood of $\langle c\rangle$ occurring after each *temporal point* of $\langle a, b\rangle$. Referring to Table 4(b), we see that there is a $\langle c\rangle$ occurring after each temporal point of $\langle a, b\rangle$. Hence, the confidence of R_X is 1.

Strong rules to be mined must have their [prefix-] sequence-supports greater than the *min_s-sup* threshold, their instance-supports greater than the *min_i-sup* threshold, *and* their confidences greater the *min_conf* threshold.

In mining program properties, the confidence of a rule (or property), which is a measure of its certainty, matters the most (*c.f.*, [29]). Support values are considered to differentiate high confidence rules according to the frequency of their

occurrences in the traces. Rules with confidences <100% are also of interest due to the imperfect trace collection and the presence of bugs and anomalies [29]. Similar to the assumption made by work in statistical debugging (*e.g.*, [10]), simply put, if a program behaves in one way 99% of the time, and the opposite 1% of the time, the latter is a possible bug. Hence, a high confidence and highly supported rule is a good candidate for bug detection using program verifiers.

We denote recurrent rules, expressible in LTL template $G(pre \rightarrow post)$, as $pre \rightarrow post$, where *pre* and *post* correspond to an event or a series of events. We added the notions of [prefix-] sequence-support, instance-support, and confidence to the rules. The formal notation of recurrent rules is defined below.

Definition 6 (Recurrent Rules). *A recurrent rule R_X is denoted by pre $\rightarrow$ post (s-sup,i-sup,*conf*). The series of events pre and post represents the rule pre- and post-condition and are denoted by R_X.Pre and R_X.Post respectively. The notions s-sup, i-sup and* conf *represent the sequence-support, instance-support and confidence of R_X respectively. They are denoted by s-sup(R_X), i-sup(R_X), and conf(R_X) respectively.*

Example. Consider $DBEX$ and the rule R_X, $\langle a, b\rangle \rightarrow \langle c\rangle$ shown in the previous example. It has s-support value of 2, i-support value of 2 and confidence of 1. It is denoted by $\langle a, b\rangle \rightarrow \langle c\rangle(2, 2, 1)$.

4.2 Apriori Properties and Non-redundancy

Apriori properties have been widely used to ensure efficiency of many pattern mining techniques (*e.g.*, [1,2]). Fortunately, recurrent rules obey the following apriori properties:

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

Theorem 2 (Apriori Property – Confidence). *If a rule $evs_P \rightarrow evs_C$ does not satisfy the min_conf threshold, neither will all rules $evs_P \rightarrow evs_D$ where evs_D is a super-sequence of evs_C.*

To reduce the number of rules and improve efficiency, we define a notion of rule redundancy defined based on *super-sequence relationship* among rules having the same support and confidence values. This is similar to the notion of *closed* patterns applied to sequential patterns [28,25].

Definition 7 (Rule Redundancy). *A rule R_X ($pre_X \rightarrow post_X$) is redundant if there is another rule R_Y ($pre_Y \rightarrow post_Y$) where:*
(1); R_X is a sub-sequence of R_Y (i.e., $pre_X + post_X \sqsubset pre_Y + post_Y$)
(2); Both rules have the same support and confidence values

Also, in the case that the concatenations are the same (i.e., $pre_X + post_X = pre_Y + post_Y$), to break the tie, we call the one with the longer premise as being redundant (i.e., we wish to retain the rule with a shorter premise and longer consequent).

A simple approach to reduce the number of rules is to first mine a full set of rules and then remove redundant ones. However, this "late" removal of redundant rules is inefficient due to the exponential explosion of the number of intermediary rules that need to be checked for redundancy. To improve efficiency, it is therefore necessary to identify and prune a search space containing redundant rules "early" during the mining process. The following two theorems are used for 'early' pruning of redundant rules. The proofs are available in our technical report [21].

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

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

Utilizing Theorems 3 & 4, many redundant rules can be pruned 'early'. However, the theorems only provide sufficient conditions for the identification of redundant rules – there are redundant rules which are not identified by them. To remove remaining redundant rules, we perform a post-mining filtering step based on Definition 7.

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

Step 1. Leveraging Theorems 1 & 3, we generate a *pruned* set of pre-conditions satisfying *min_s-sup*.
Step 2. For each pre-condition *pre*, we create a *projected-all* database $SeqDB^{all}_{pre}$.
Step 3. Leveraging Theorems 2 & 4, for each $SeqDB^{all}_{pre}$, we generate a *pruned* set containing such post-condition *post*, such that the rule $pre \rightarrow post$ satisfies *min_conf*.
Step 4. Checking the rules' instance-supports, we remove rules from step 3 that do not satisfy *min_i-sup*.
Step 5. Using Definition 7, we filter any remaining redundant rules.

In the next section, we describe our algorithm in detail.

5 Algorithm

In the previous section, the process of mining non-redundant rules has been divided into 5 steps. Steps 1 and 3 sketch how a pruned set of pre- and post- conditions are mined. The following paragraphs will elaborate them in more detail.

Before proceeding, we first describe a set of patterns called *projected database closed* (or LS-Set) first mentioned in [28]. A pattern is in the set if *there does not exist any super-sequence pattern having the same projected database*. Patterns having the same projected database must have the same support, but not vice versa. *Projected database closed* patterns is of special interest to us, as explained in the following paragraphs.

At step 1, a pruned set of pre-conditions is generated from the input database $SeqDB$. From Theorem 3, a pattern is in the pruned pre-condition set if *there does not exist any super-sequence pattern having the same projected database*. Comparing with the definition of *projected database closed* patterns in the previous paragraph, we note that this pruned set of pre-conditions corresponds to the *projected database closed set* (or LS-Set) mined from $SeqDB$.

At step 3, starting with a projected-all database $SeqDB^{all}_{pre}$, we generate a pruned set of post-conditions. From Theorem 4, a pattern is in the pruned post-condition set if *there does not exist any super-sequence pattern having the same projected database*. Again, this set of pruned post-condition corresponds to the *projected database closed set* (or LS-Set) mined from $SeqDB^{all}_{pre}$.

Our mining algorithm (NR^3-Miner: Non-Redundant Recurrent Rule Miner) is shown in Figure 2. First, a pruned set of pre-conditions satisfying the minimum sequence-support threshold (*i.e.*, *min_s-sup*) is mined using an LS-Set miner modified from BIDE [25], the state-of-the-art closed sequential pattern miner.[1] Next, for each pre-condition mined, a database projected-all on it is formed. Consequently, another LS-Set Generator is run on each projected-all database to mine the set of post-conditions of the corresponding candidate rules having enough sequence-support and confidence values. Next, each candidate rule is further checked for the

Procedure Mine Non-Redundant Recurrent Rules
Inputs: $SeqDB$: Sequence DB; *min_s-sup* : Min. S-Support Thresh.; *min_i-sup* : Min. I-Support Thresh.; *min_conf* : Min. Conf. Thresh.
Output:
Rules: Non Redundant Set of Recurrent Rules
Method:
1: Let *PreCond* = Generate an LS-Set from $SeqDB$ with the threshold set at *min_s-sup*
2: For every *pre* in *PreCond*
3: Let $SeqDB^{all}_{pre}$ = $SeqDB$ *projected-all* by pattern *pre*
4: Let *bthd* = *min_conf* × $|SeqDB^{all}_{pre}|$
5: Let *PostCond* = Generate an LS-Set from $SeqDB^{all}_{pre}$ with the threshold set at *bthd*
6: For every *post* in *PostCond*
7: Add (*pre* → *post*) to *Rules*
8: For every *rx* in *Rules*
9: If (i-sup(*rx*) < *min_i-sup*)
10: Remove *rx* from *Rules*
11: If (*rx* is redundant according to Def. 7)
12: Remove *rx* from *Rules*
13: Output *Rules*

Fig. 2. Mining Algorithm – NR^3-Miner

[1] BIDE, in effect prunes all search sub-spaces containing patterns not in LS-Set. To mine LS-Set using BIDE, we keep the search space pruning strategy but remove the closure check. The details are available in our technical report [21].

satisfaction of the minimum instance-support threshold (*i.e.*, *min_i-sup*). Provided that the pattern *pre*++*post* is in the pruned set of pre-conditions computed in the first step of the mining process (*i.e.*, line 1 of the algorithm), the instance-support of a rule $pre \rightarrow post$ has been computed during the second step of the mining process (*i.e.*, line 3 of the algorithm). Otherwise, an additional database scan need to be made to compute the rule's instance-support value. Finally, a filtering step to remove any remaining redundant rules based on Definition 7 is performed. To perform the final filtering step scalably, each remaining rule is first hashed based on its support and confidence values. Only rules falling into the same hash bucket need to be checked for super-sequence relationship.

The algorithm can be adapted easily to generate a full set of recurrent rules. This is performed to serve as a point of reference for investigating the benefit of the early identification and pruning of redundant rules. To generate the full set we can simply: (1); Generate a full set of pre- and post- conditions of rules satisfying the s-support and confidence thresholds at lines 1 and 5 of the algorithm respectively (we use PrefixSpan [23] for this purpose), and (2); Skip the final redundancy filtering step (*i.e.*, lines 11-12 of the algorithm in Figure 2).

6 Performance Evaluation

Experiments have been performed on both synthetic and real datasets on low support thresholds to evaluate the *scalability of our mining framework*. The lower the thresholds, the more difficult it is to mine the rules. Our algorithms are the *first* algorithms mining recurrent rules, hence we compare and contrast the runtime required and the number of rules mined when full and non-redundant sets of recurrent rules are mined to evaluate the *effectiveness of our pruning strategies*.

Datasets. We use 2 datasets in our experiments: one synthetic and another real. Synthetic data generator provided by IBM was used with modification to ensure generation of single-event sequences (*i.e.*, all transactions are of size 1). We also experimented on a click stream dataset (*i.e.*, Gazelle dataset) from KDD Cup 2000 [16]. It contains 23639 sequences with an average length of 3 and a maximum length of 651.

Environment and Configuration. All experiments were performed 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 Visual C#.Net.

Experiment Methodology & Presentation. Experiments were performed by varying *min_s-sup*, *min_i-sup* & *min_conf* thresholds. The results are plotted as line graphs. 'Full' and 'NR' correspond to the full set and non-redundant set of rules respectively. The x-axis of the graph corresponds to the thresholds used while the y-axis represents the runtime required, or the number of mined rules.

For the experiment with the Gazelle dataset, only the results for mining 'NR' rules are plotted. The 'Full' set is not mine-able even at the highest *min_s-sup* threshold shown in Figure 5 & 6 – our attempt produced a gigantic 51 GB file before we had to stop the process.

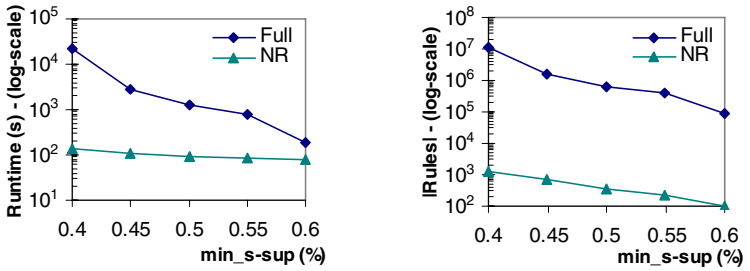

Fig. 3. Varying *min_s-sup* (at *min_conf*=50%,*min_i-sup*=1) for D5C20N10S20 dataset

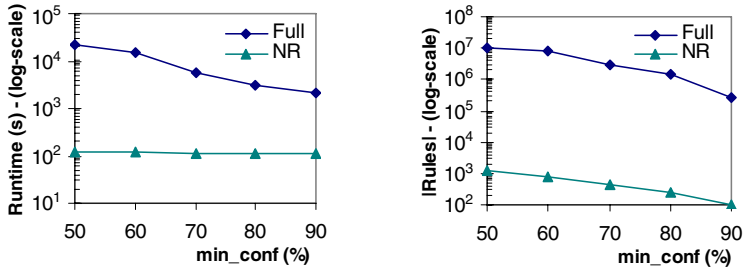

Fig. 4. Varying *min_conf* (at *min_s-sup*=0.4%,*min_i-sup*=1) for D5C20N10S20 dataset

Synthetic Dataset Result. The experiment results for the synthetic dataset are shown in Figure 3 & 4. We produce a synthetic dataset by running the IBM synthetic data generator with the following parameter setting: D (number of sequences - in 1000s) = 5, C (average sequence length) = 20, N (number of unique events - in 1000s) = 10 and S (average number of events in maximal sequences) = 20.

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

Both the runtime and the number of mined rules are significantly increased when the *min_s-sup* threshold is lowered. When a full set of rules is mined, lowering the *min_conf* threshold from 90% to 50% significantly increases the runtime and the number of mined rules. Reducing the confidence threshold has less effect when non-redundant rules are mined.

Varying the i-support threshold does not affect the runtime because we do not have any apriori property involving the instance support of mined rules. Hence, i-support threshold is not used to prune the search space. However, the i-support threshold still affects the number of mined rules: the number decreases as the threshold increases. Due to the space limitation, experimental results on varying the i-support threshold is moved to the technical report [21].

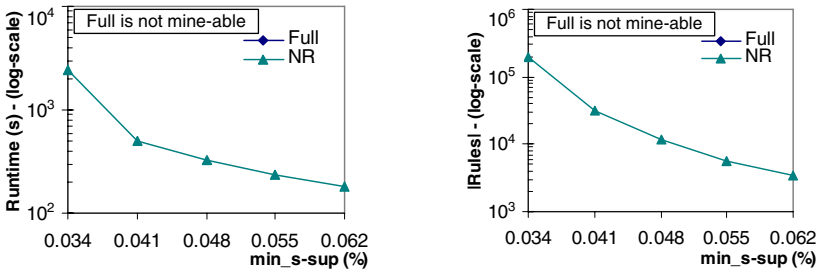

Fig. 5. Varying *min_s-sup* (at min_conf=50%, min_i-sup=1) for Gazelle dataset

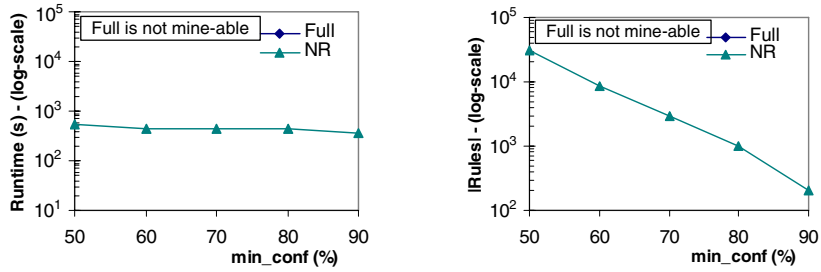

Fig. 6. Varying *min_conf* (at min_s-sup=0.041%, min_i-sup=1) for Gazelle dataset

Gazelle Dataset Result. The experiment results of mining non-redundant rules from the Gazelle dataset are shown in Figure 5 & 6. The runtime is significantly increased when the *min_s-sup* threshold is lowered. Lowering the *min_conf* threshold does not affect the runtime much. However, we note that the number of mined rules sharply reduces when the *min_conf* threshold is increased from 50% to 90%. The results also show that we can efficiently mine recurrent rules from real data set even at a low *min_s-sup* support of 0.034%.

Summary. The experiment results show the effectiveness of our pruning strategies in reducing both the runtime and the number of mined rules. A non-redundant set of recurrent rules can be mined efficiently from both real and synthetic datasets even at low *min_s-sup* and *min_i-sup* thresholds. We did not experiment with low *min_conf* thresholds as we believe the usefulness of low confidence rules (if any) is minimal.

7 Case Study

A case study was performed on the security component of JBoss Application Server (JBoss AS). JBoss AS is the most widely used J2EE application server. It contains over 100,000 lines of code and comments. The purpose of this study is to show the usefulness of the mined rules to describe the behavior of a real software system.

Premise	⟶ Consequent
XLoginConfImpl.getConfEntry() AuthenticationInfo.getName()	ClientLoginModule.initialize() ClientLoginModule.login() ClientLoginModule.commit() SecAssocActs.setPrincipalInfo() SetPrincipalInfoAction.run() SecAssocActs.pushSubjectContext() SubjectThreadLocalStack.push() SimplePrincipal.toString() SecAssoc.getPrincipal() SecAssoc.getCredential() SecAssoc.getPrincipal() SecAssoc.getCredential()

Fig. 7. A Rule from JBoss-Security

We instrumented the security component of JBoss-AS using JBoss-AOP and generated traces by running the test suite that comes with the JBoss-AS distribution. In particular, we ran regression tests testing Enterprise Java Bean (EJB) security implementation of JBoss-AS. Twenty-three traces of a total size of 4115 events, with 60 unique events, were generated. Running the algorithm with the minimum support and confidence thresholds set at 15 and 90% respectively, ten non-redundant rules were mined. The algorithm completed within three seconds.

A sample of the mined rules (with abbreviated class and method names) is shown in Figure 7. The rule, read from top to bottom, left to right, describes authentication using Java Authentication and Authorization Service (JAAS) for EJB within JBoss-AS. When authentication scenario starts, first configuration information is checked to determine authentication service availability – this is described by the premise of the rule. This is followed by: invocations of actual authentication events, binding of principal information to the subject being authenticated, and utilizations of subject's principal and credential information in performing further actions – these are described by the consequent of the rule.

8 Conclusion and Future Work

In this paper, we proposed a novel framework to mine *recurrent rules* from a sequence database. Recurrent rules have the form "whenever a series of precedent events occurs, eventually a series of consequent events occurs". Recurrent rules are intuitive and characterize behaviors in many domains. Support and confidence values are attached to recurrent rules to distinguish significant ones. Two apriori properties pertaining to the sequence-support and confidence values of rules have been used to prune the search space of possible rules. Also, we have proposed a novel definition of rule redundancy. Employing "early" pruning of redundant rules has further improved the efficiency of the mining process and reduced the number of mined rules. Our performance study shows the effectiveness of our pruning strategies in reducing runtime (up to 147 times less) and in removing redundant rules

(up to 8500 times less). Non-redundant recurrent rules can be efficiently mined even at low support thresholds by our proposed mining framework. A case study on JBoss Application Server shows the applicability of our rules in mining program properties.

As future work, we plan to apply our technique to analyze other real-life datasets from various domains not restricted to software data. A comparative study to compare recurrent rules with other forms of software specifications mined from execution traces [3,19,20] is another future work. Improving scalability of the mining process further and extending to mining from stream data are other possible future work.

Acknowledgement. We thank the anonymous reviewers for their valuable comments and advice. We thank judges of SIGPLAN Symposium on Programming Language, Design and Implementation 2007 student research competition for their advice and encouragement [18]. Our thanks to Jianyong Wang and Jiawei Han for the source code of BIDE. Also, we thank Jiawei Han and his group at UIUC for the binary of PrefixSpan bundled in the data mining package ILLIMINE. We wish to thank Blue Martini Software for contributing the KDD Cup 2000 data.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: VLDB (1994)
2. Agrawal, R., Srikant, R.: Mining sequential patterns. In: ICDE (1995)
3. Ammons, G., Bodik, R., Larus, J.R.: Mining specification. In: SIGPLAN-SIGACT POPL (2002)
4. Barth, A., Datta, A., Mitchell, J.C., Nissenbaum, H.: Privacy and contextual integrity: Framework and applications. In: S&P (2006)
5. Capilla, R., Duenas, J.C.: Light-weight product-lines for evolution and maintenance of web sites. In: CSMR (2003)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
7. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby,, Laubach, S., Zheng, H.-J.: Bandera: extracting finite-state models from java source code. In: ICSE (2000)
8. Deelstra, S., Sinnema, M., Bosch, J.: Experiences in software product families: Problems and issues during product derivation. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, Springer, Heidelberg (2004)
9. Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state verification. In: ICSE (1999)
10. Engler, D.R., Chen, D.Y., Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: SOSP (2001)
11. Garriga, G.C.: Discovering unbounded episodes in sequential data. In: Lavrač, N., Gamberger, D., Todorovski, L., Blockeel, H. (eds.) PKDD 2003. LNCS (LNAI), vol. 2838, Springer, Heidelberg (2003)
12. Han, J., Kamber, M.: Data Mining Concepts and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2006)
13. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Thoery, Language, and Computation. Addison-Wesley, Reading (2001)

14. Huth, M., Ryan, M.: Logic in Computer Science. Cambridge (2004)
15. ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC) (1999)
16. Kohavi, R., Brodley, C., Frasca, B., Mason, L., Zheng, Z.: KDD-Cup 2000 organizers report: Peeling the onion. SIGKDD Explorations 2, 86–98 (2000)
17. Liu, C., Lian, Z., Han, J.: How bayesians debug. In: Perner, P. (ed.) ICDM 2006. LNCS (LNAI), vol. 4065, Springer, Heidelberg (2006)
18. Lo, D.: A sound and complete specification miner. In: SIGPLAN PLDI Student Research Competition (awarded 2^{nd} position) (2007), `http://www.acm.org/src/winners.html`
19. Lo, D., Khoo, S.-C.: SMArTIC: Toward building an accurate, robust and scalable specification miner. In: SIGSOFT FSE (2006)
20. Lo, D., Khoo, S.-C., Liu, C.: Efficient mining of iterative patterns for software specification discovery. In: SIGKDD (2007)
21. Lo, D., Khoo, S.-C., Liu, C.: Mining recurrent rules from sequence database. In SoC-NUS Technical Report TR12/07 (2007)
22. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. DMKD 1, 259–289 (1997)
23. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.-C.: Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: ICDE (2001)
24. Spiliopoulou, M.: Managing interesting rules in sequence mining. In: Żytkow, J.M., Rauch, J. (eds.) PKDD 1999. LNCS (LNAI), vol. 1704, Springer, Heidelberg (1999)
25. Wang, J., Han, J.: BIDE: Efficient mining of frequent closed sequences. In: ICDE (2004)
26. Wing, J.M.: A specifier's introduction to formal methods. IEEE Computer 23, 8–24 (1990)
27. Weimer, W., Necula, G.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
28. Yan, X., Han, J., Afhar, R.: CloSpan: Mining closed sequential patterns in large datasets. In: SDM (2003)
29. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining temporal API rules from imperfect traces. In: ICSE (2006)