Mining Temporal Rules from Program Execution Traces

David Lo[†] and Siau-Cheng Khoo[†] and Chao Liu[‡]
[†]Department of Computer Science, National University of Singapore
[‡]Department of Computer Science, University of Illinois at Urbana-Champaign
{dlo,khoosc}@comp.nus.edu.sg, chaoliu@cs.uiuc.edu

Abstract

Software evolution incurs difficulties in program comprehension and software verification, and hence increases the cost of software maintenance. In this study, we propose a novel technique, to mine from program execution traces a sound and complete set of statistically significant temporal rules of arbitrary lengths. The extracted temporal rules reveal invariants that the program observes, and will consequently guide developers to understand the program behaviors, and facilitate all downstream applications like verifications. Different from previous studies that are restricted to mining two-event rules (e.g., $\langle lock \rangle \rightarrow \langle unlock \rangle$), our algorithm discovers rules of arbitrary lengths. Furthermore, in order to facilitate downstream applications, we represent the mined rules as temporal logic, so that existing model checkers or other formal analysis toolkits can readily consume our mining results. We performed case studies on JBoss Application Server (JBoss AS) and a buggy Concurrent Versions System (CVS) application, and the result clearly demonstrates the usefulness of our technique in recovering underlying program designs and detecting bugs.

1 Introduction

Software changes throughout its lifespan. Software maintenance deals with the management of such changes, ensuring that the software remains correct while features are added or removed. Maintenance cost can contribute up to 60%-80% of software cost [4]. A challenge to software maintenance is to keep documented specifications accurate and updated as program changes. Outdated specifications cause difficulties in program comprehension which account for up to 50% of program maintenance cost [4].

In order to ensure software correctness, model checking [5] has been proposed and shown useful in many cases. It accepts a model, often automatically constructed from the code, and a set of formal properties to check. However, the difficulty in formulating a set of formal properties has long been a barrier to its wide-spread application [2].

Addressing the above problems, there is a need for techniques to automatically mine formal specifications from program as it changes over time. Employing these techniques ensures specifications remain updated; also it provides a set of properties to be verified via formal verification tools like model checking. This family of techniques is commonly referred to as "specification mining" [2].

There have been a number of studies on specification mining, which relate to either program comprehension (*e.g.*, [20, 6, 15]) or verification (*e.g.*, [2, 22]). Most specification mining algorithms extract specifications in the form of an automaton (*e.g.*, [15, 2, 20, 6]) or two-event rules (*e.g.*, [22]). While a mined automaton expresses a global picture of a software specification, mined rules break this into smaller parts each expressing a program property which is easily understood. A mined automaton might be too complex to be comprehended manually. Also, methods mining automaton-based specifications from traces use automaton learners which suffer from the issue of *overgeneralization* (see [14, 15]), but this is not the case with mining rules. On the other hand, existing work on mining rules only mines two-event rules (*e.g.*, $\langle lock \rangle \rightarrow \langle unlock \rangle$) which are limited in their ability to express complex temporal properties.

The focus of this study is to automatically discover rules *of arbitrary lengths* from program execution traces. A trace can be viewed as a series of events, with each event corresponding to a method which is called when a program is executed. A multi-event rule is denoted by $ES_{pre} \rightarrow ES_{post}$, where ES_{pre} and ES_{post} are the premise/pre-condition and the consequent/post-condition, respectively. This rule means that "Whenever a series of events ES_{pre} occurs, eventually another series of events ES_{post} also occur."

The above rules can be expressed in temporal logic, and belong to two of the most used families of temporal logic expressions for verification (*i.e.*, response and chain-response) [7]. Examples of such rules include: *(Resource Locking Protocol)* Whenever a lock is acquired, eventually it is released; *(Internet Banking)* Whenever a connection to a bank server is made and an authentication is completed

and one transfers money, eventually money is transferred and a receipt is displayed.

From traces, many rules can be inferred, but not all are important. We therefore utilize the concept of support and confidence from data mining [8] to identify important rules. Rules satisfying user-defined thresholds of minimum support and confidence are referred to as *statistically significant*. A *non-redundant* set (see Section 3) of *statistically significant* rules are the output of our mining algorithm.

As with any program analysis tool, soundness and completeness are desirable goals to have [19]. Our algorithm is *sound* as all mined rules are statistically significant. It is *complete* as all statistically significant rules of the form $ES_{pre} \rightarrow ES_{post}$ are mined or represented.

We performed a case study on the transaction component of JBoss Application Server. The result shows the usefulness of our mining technique in recovering the underlying protocols that the code obeys, thus aiding program comprehension. Also, another case study has been performed on a buggy CVS application built on top of the Jakarta Commons Net [3] adapted from the one studied in [15, 14]. The result highlights the usefulness of our technique in mining bug-revealing properties, thus aiding program verification.

The rest of this paper is organized as follows. We first introduce the semantics of discovered rules in Section 2, and discuss the challenges of rule mining and our proposed solution in Section 3. Section 4 reports on the case studies. With related work discussed in Section 5, Section 7 concludes this study.

2 Semantics of Mined Rules

In this section, we discuss the semantics of mined rules and the computation of support and confidence values.

Temporal Logic Semantic. Our mined rules can be expressed in Linear Temporal Logic (LTL) [5]. There are a number of LTL operators, among which, we are 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 either a property holds at that point in time or *finally (eventually)* it holds. The operator 'X' specifies that a property holds at the *next* event. Let us consider two examples of LTL expressions below.

$F(unlock)$

Meaning: *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

Each of our mined rules states: whenever a series of premise events occurs eventually a series of consequent events also occurs. A mined rule denoted as $pre \rightarrow post$, can be mapped to its corresponding LTL expression. Examples of such correspondences are shown in the table below.

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

The set of LTL rules minable by our technique can be represented in the Backus-Naur Form (BNF) as follows:

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

By simple transformations, the mined rules can also be expressed in Computational Tree Logic (CTL) [5] and probabilistic CTL [9].

Support and Confidence. There are many possible rules, and we need to identify important ones. Statistics, such as support and confidence, adapted from data mining [8], can be used to distinguish important ones: *(Support)* The *number of traces* exhibiting the premise of the rule; *(Confidence)* The likelihood of the rule's premise being followed by its consequent in the traces.

The meaning of the above is best illustrated by an example. Consider the following set of simplified traces:

Trace 1	*lock,use,use,unlock,lock,use*
Trace 2	*lock,unlock,lock,unlock*

Let us consider the rule $lock \rightarrow unlock$. Its support value is two, as all two traces exhibit the premise of the rule. Its confidence is 0.75, as 75% of the time (3 out of 4) $lock$ is followed by $unlock$. Formal definitions of support and confidence are provided in the technical report [17].

Our technique will only output rules satisfying a user-defined thresholds of minimum support and confidence. Rules satisfying these thresholds are referred to as being *statistically significant*.

3 Mining Algorithm

This section discusses challenges of mining a sound and complete set of multi-event LTL rules, and presents our solution and mining algorithm.

Challenges and Solutions. The complexity of mining multi-event temporal rules is potentially exponential to the length of the longest trace in the trace-set. A naive approach is to check each possible rule of length two up to the maximum length of the traces. This simply does not work because a set of traces with a maximum length of 100 and containing 10 unique events will require 10^{100} checks.

To address the above challenge, we utilize an effective search space pruning strategy. In particular, the following 'apriori' properties are used to prune non statistically-significant rules from the search space:

1. If a rule $evs_P \rightarrow evs_C$ doesn't satisfy the support threshold, neither does any rule $evs_Q \rightarrow evs_C$ where evs_Q is a super-sequence of evs_P.
2. If a rule $evs_P \rightarrow evs_C$ doesn't satisfy the confidence threshold, neither does any rule $evs_P \rightarrow evs_D$ where evs_D is a super-sequence of evs_C.

Furthermore, we notice that many rules are redundant. A rule R_X is redundant if there exists another mined rule R_Y where:

1 The concatenation of R_X's premise and consequent is a proper subsequence of the concatenation of those of R_Y. Otherwise, if the concatenations are the same, R_X has a longer premise.
2 Both R_X and R_Y have the same support and confidence values.

To illustrate redundant rules, consider the following set of rules describing an Automated Teller Machine (ATM):

R1	*accept_card* → *enter_pin,display_goodbye,eject_card*
R2	*accept_card* → *enter_pin*
R3	*accept_card* → *display_goodbye*
R4	*accept_card* → *enter_pin,eject_card*
R5	*accept_card* → *display_goodbye,eject_card*

If the above rules have the same support and confidence values, rules R2-R5 are redundant since they are represented by rule R1. To keep the number of mined rules manageable, we remove redundant rules. This can drastically reduce the number of reported rules. Our performance study on data mining benchmark datasets shows the number of rules is reduced by more than 1,000 times lesser when *redundant* rules are removed – see our technical report [17].

Without the application of the 'apriori' properties and the removal of redundant rules, the case studies considered are infeasible as the naive approach requires an exponential number of checks. A different 'apriori' property and redundancy identification have enabled practical use of pattern mining which would otherwise require exponential running time [8, 21].

Algorithm Sketch. Inputs of our mining algorithm comprise a set of traces and the support and confidence thresholds. The output of our algorithm is a set of rules, each of which expresses: whenever a series of events occurs at a point in time (*i.e.*, temporal point), another series of events will eventually occurs. To generate these rules, we need to identify interesting temporal points, and for such points, note what series of events are likely to occur next. Our mining process is composed of three steps – for full details see [17]:

Step 1 Mine a set of premises where each has a support value greater than the minimum support threshold.

Step 2 For each premise pre, do the following:
- **a** Find all temporal points where the premise pre occurs
- **b** Extract all rules of the form $pre \rightarrow post$, where the consequence $post$ happens with the likelihood greater than or equal to the minimum confidence threshold.

Step 3 Remove remaining redundant rules.

We perform an aggressive pruning strategy to remove redundant rules. Sub search spaces containing redundant rules are identified "early" and pruned. Rather than generating *all* statistically significant rules and removing redundant ones at Step 3 (*i.e.*, "late" pruning), we *avoid* generating redundant rules in the first place (*i.e.*, "early" pruning). At step 3, we only remove the remaining redundant rules not identified by our aggressive pruning strategy. "Early" pruning of redundant rules greatly improves the performance of our algorithm. In our performance study, unless redundant rules are pruned "early", several experiments on a real-life benchmark dataset at various minimum support and confidence thresholds are practically infeasible – see our technical report [17].

At the end of the above process, a complete set of non-redundant multi-event LTL rules of the form $ES_{pre} \rightarrow ES_{post}$ satisfying the support and confidence thresholds are mined.

4 Case Studies

We performed a case study on the transaction component of JBoss AS to show the applicability of our method in providing insight into the protocol that the code obeys – hence aiding program comprehension. Another case study on a buggy CVS (Concurrent Versions System) application adapted from the one previously studied in [15, 14] shows the utility of mined rules in identifying bugs via model checking.

JBoss AS Transaction Component. We instrumented the transaction component of JBoss-AS using JBoss-AOP and generated traces by running the test suite that comes with JBoss-AS distribution. In particular, we ran the transaction manager regression test of JBoss-AS. Twenty-eight traces of a total size of 2603 events, with 57 unique events, were generated. Running the algorithm with the minimum support and confidence thresholds set at twenty-five traces and ninety-percent respectively, 163 non-redundant rules were mined. The algorithm completed in 35.1 seconds.

A sample of the mined rules is shown in Figure 1. The 19-event rule in Figure 1 describes that the series of events ⟨connection to a server instance events, transaction manager and implementation set up event⟩ (event 1-10) at the start of a transaction is always followed by the series of events ⟨transaction completion events and resource release events⟩ (event 11-19) at the end of the transaction. The above rule describing the temporal relationship and constraint between the 19 events is hard to identify manually. The rule sheds light into the *implementation details* of JBoss AS which are implemented at various locations in (*i.e.*, crosscuts) the JBoss AS large code base.

CVS on Jakarta Commons Net. A case study was performed on a buggy CVS application built on top of the FTP library of Jakarta Commons Net to show the usefulness of

Premise	Consequent
TxManLocator.getInstance()	TransactionImpl.instanceDone()
TxManLocator.locate()	TxManager.getInstance()
TxManLocator.tryJNDI()	TxManager.releaseTransImpl()
TxManLocator.usePrivateAPI()	TransactionImpl.getLocalId()
TxManager.getInstance()	XidImpl.getLocalId()
TxManager.begin()	LocalId.hashCode()
XidFactory.newXid()	LocalId.equals()
XidFactory.getNextId()	TransactionImpl.unlock()
XidImpl.getTrulyGlobalId()	XidImpl.hashCode()
LocalId.associateCurrentThread()	
TransactionImpl.lock()	

Figure 1. A Rule from JBoss-Transaction

mined rules in verification and bug detection. The CVS interaction protocol with the underlying FTP library can be represented as a 33-state automaton partially drawn in Figure 2 (see [15, 14] for a more detailed diagram).

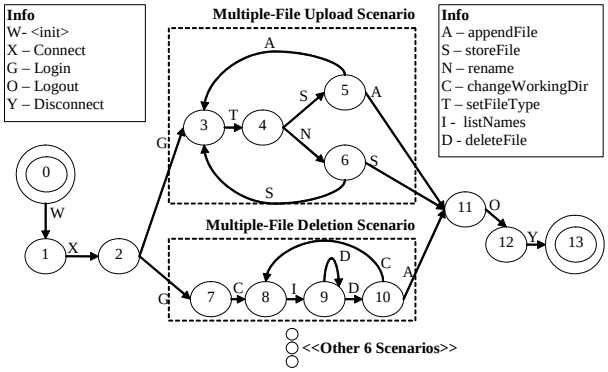

Figure 2. CVS Protocol

We focus on the two scenarios of multiple-file upload and deletion scenario. The scenarios start with connecting and logging into the FTP server and end by logging off and disconnecting from the server. Whenever a new file is added or a file is deleted a record is made to a system log file. Multiple versions of the same file are maintained by adding timestamp to old versions of the file.

The CVS application is buggy, there are 4 bugs that causes *inconsistent system log file*. The system log file describes the state of the CVS repository and should be kept consistent with the stored files. The 4 bugs are illustrated by the error transitions (in dashed lines) shown in Figure 3. Due to the bugs, a file can be added or deleted without a proper log entry being made. Also, an old version of a file can be renamed by appending a time-stamp without the new version being stored to the CVS.

The bugs occur because scenarios are not executed atomically. Each invocation of a method of `FTPClient` of the FTP library may generate exceptions, especially `ConnectionClosed` and `IO` exception. Hence the code accessing `FTPClient` methods need to be enclosed in a `try..catch..finally` block. Every time such an exception happens the program simply logout and disconnect from the FTP server.

To generate traces, we follow the process discussed in [15]. First, we instrument the CVS application byte

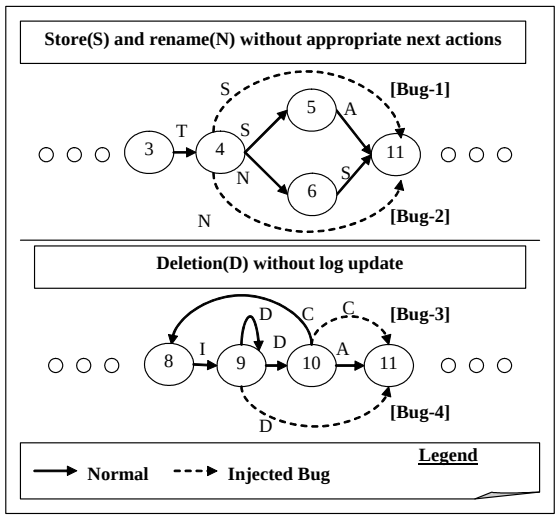

Figure 3. Injected Bug

code using an adapted version of Java Runtime Analysis Toolkit [12]. Next, we ran the instrumented CVS application over a set of test cases to generate traces. Via a trace post-processing step, we then filter events in the traces not corresponding to the interactions between the CVS application and the Jakarta Commons Net FTP library. Thirty-six traces of a total size of 416 events were generated.

We ran our mining algorithm on the generated traces. It ran in less in 1.1 second and mined 5 rules with minimum support and confidence thresholds set at fifteen traces and ninety percent respectively. Among the mined rules, the following two rules are the bug-revealing program properties:

1 Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, file type is set (T) and an old file is renamed(N), *then eventually* a new file is stored(S), followed by a logout (O) and a disconnection from server (Y). This is *denoted as*: $\langle W, X, G, T, N\rangle \rightarrow \langle S, O, Y\rangle$

2 $\langle W, X, G, C, I, D\rangle \rightarrow \langle A, O, Y\rangle$

We used the model checker described in [10]. We converted an abstract model of the CVS application to the format accepted by the model checker and checked against the above two properties. The model checker reported violations of the above properties. These violations correspond to 3 out of the 4 bugs (Bug-2,3,4) in the model.

5 Related Work

In [22], Yang et al. present an interesting work on mining two-event temporal logic rules (i.e., of the form $G(a \rightarrow XF(b))$, where G, X, and F are LTL operators [11]), which are statistically significant with respect to a user-defined 'satisfaction rate'. The algorithm presented, however, does not scale to mine multi-event rules of arbitrary length. To handle longer rules, Yang et al. suggest a partial solution based on concatenation of mined two-event

rules. Yet, the method proposed might miss some multi-event rules or introduce superfluous rules that are not statistically significant – it is neither sound nor complete. In contrast, we mine LTL rules of arbitrary size; scalability is accomplished by utilizing search space pruning strategies adapted from the data mining domain. The method is sound and complete as all mined rules are statistically significant and all statistically significant rules are mined.

There are many other work on mining frequent patterns [1, 21, 16], automaton [2, 20, 6, 15], Live Sequence Charts [18], etc. Technique wise, this work is similar to the family of pattern mining algorithms and has similar worst case complexity. By employing a pruning strategy, typical runtimes of pattern mining algorithms can be much less than the worst case complexity. However, different from the above existing studies, some of which are our own, in this work we mine LTL rules which have a different semantic and require different mining strategy than previous approaches.

6 Discussion and Future Work

Note that the time taken for mining is *much improved* with search space pruning strategy. Without employing a search space pruning strategy, the mining process will require at least E^L check operations, where E is the number of unique events and L is the maximum length of the trace. For traces from JBoss AS considered in Section 4, the mining process (without pruning) will require more than 50^{100} operations. Considering 1 picosecond per operation, it will only complete *in about* $2.501\text{x}10^{148}$ *centuries* whereas we simply need 35.1 seconds. This highlights the power and importance of search space pruning strategies in improving the scalability of the mining process.

In the second case study, Bug-1 cannot be revealed because the bug-revealing property is outside the bound of the LTL expressions minable by our algorithm. The bug-revealing property is: Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, a file type is set (T), *and there is no rename (N) until a new file is stored (S)*, then eventually a log entry is made (A), followed by a logout (O) and a disconnection from server (Y). To mine the property, we need to mine rules containing the LTL operators not($\neg$) and until(U) – this is a future work.

Another open issue is in improving the scalability of our method further. One direction we are investigating is to reduce the size of input traces while retaining the quality of the specification mined. One can do so by throwing away non-important or less important events. In [23], Zaidman *et al.* identify important key classes using a web-mining algorithm. Similar approach to that in [23] might be employed to identify and remove less important events from the traces. In [13], Kuhn and Greevy partition a trace into different phases. Rather than mining specifications for the entire trace, one can separately mine each phase in the trace. This can be more efficient than mining the entire trace. If a test-suite is available, one can also perform a divide-and-conquer strategy by generating traces for each distinct part of the test suite (*i.e.*, group those testing the same component together) and analyzing them separately. Another direction we are investigating is to incorporate latest research in data mining to improve the algorithm further where approximation in the mining algorithm result can improve mining speed (*e.g.*, [24]).

7 Conclusion

In this paper, a novel method to mine a *non-redundant* set of *statistically significant* rules *of arbitrary lengths* of the form: "Whenever a series of events ES_{Pre} occurs, eventually another series of events ES_{Post} also occurs" is proposed. The problems of potentially exponential runtime cost and huge number of reported rules have been effectively mitigated by employing a search space pruning strategy and by eliminating redundant rules. Case studies have been conducted to demonstrate the usability of the proposed technique for program comprehension and verification.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Int. Conf. on Very Large DataBases*, 1994.
[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *SIGPLAN-SIGACT POPL*, 2002.
[3] Apache Software Foundations. Jakarta Commons/Net. *http://jakarta.apache.org/commons/net/*.
[4] G. Canfora and A. Cimitile. Software maintenance. In *Handbook of Software Eng. and Knowledge Eng.*, 2002.
[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
[6] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. on Software Eng. and Methodology*, 1998.
[7] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.
[8] J. Han and M. Kamber. *Data Mining: Concepts and Techniques, 2nd ed*. Morgan Kaufmann, 2006.
[9] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, (6):512–535, 1994.
[10] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS*, 2006.
[11] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge, 2004.
[12] Java Runtime Analysis Toolkit. *jrat.sourceforge.net/*.
[13] A. Kuhn and O. Greevy. Exploiting analogy between traces and signal processing. In *ICSM*, 2006.
[14] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.
[15] D. Lo and S.-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.
[16] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proc. of SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2007.
[17] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of strong temporal rules from a sequence database. *Technical report at: www.comp.nus.edu.sg/∼dlo/pcoda07-techrep.pdf*, 2007.
[18] D. Lo, S. Maoz, and S.-C. Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *ASE (to appear)*, 2007.
[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.
[20] S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, 2001.
[21] X. Yan, J. Han, and R. Afhar. CloSpan: Mining closed sequential patterns in large datasets. In *Proc. of SIAM Int. Conf. on Data Mining*, 2003.
[22] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.
[23] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, 2005.
[24] F. Zhu, X. Yan, J. Han, P. Yu, and H. Cheng. Mining collosal frequent patterns by core pattern fusion. In *Int. Conf. on Data Eng.*, 2007.