Mining Temporal Rules for Program Comprehension and Verification

David Lo
PhD Candidate
National University of Singapore

Project Advisors:
Siau-Cheng Khoo, National University of Singapore
Chao Liu, Microsoft Research, Redmond

---

1. Problem & Motivation

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 [CC02]. 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 Outdated specifications cause difficulties in program comprehension which account for up to 50% of software maintenance cost [CC02]. On another angle, to ensure software correctness, model checking [CGP99] has been proposed and shown useful in many cases. It accepts a model, often automatically constructed from source code, and a set of formal specifications or properties to check. However, the difficulty in formulating a set of formal properties has been one of the challenges to its wide-spread industrial adoption [ABL02].

Addressing the above problems, there is a need for techniques to automatically mine formal specifications or properties from programs as the latter change 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. In this study, we propose a novel technique to mine from program execution traces *(statistically) significant* temporal rules, denoted as pre->post, of the format:

``Whenever a series of events pre occurs, eventually another series of events post also occurs."

The above rules are intuitive and can be commonly found in software documentation. Also, according to a survey in [DAC99], temporal rules defined above belong to two of the most frequently used families of properties for verification (i.e., response and chain-response). Hence, mining temporal rules is an important problem to be addressed with potentially many applications for program comprehension and verification.

2. Background & Related Work

Temporal Rules: Examples & Formalism. There are many examples of temporal rules in real life. Some examples of such rules are as follows:

(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.

Temporal rules can also be *formalized in Linear Temporal Logic* (LTL) [HR04]. For example, <lock>-><unlock> can be translated to G(lock->XF(unlock)), where G (Globally), X (neXt) and F (Finally) are LTL operators. LTL in turn can be fed directly to many formal verification tools including model checkers [CGP99].

Significant Rules. From traces, many rules can be inferred, but not all are important. To distinguish important rules, statistics, such as support and confidence, adapted from data mining [HK06] are employed:

(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>-><unlock>. Its support is two, as both traces exhibit the premise of the rule. Its confidence is 0.75, as 75% of the time (3 out of 4 times) *lock* is followed by *unlock*. Rules satisfying user-defined thresholds of minimum support and confidence are referred to as being *(statistically) significant*.

Related Work. There are existing studies on mining temporal rules [YEBBD06, WN05]. However, these studies are restricted to mining two-event rules (e.g., <lock>-><unlock>). In this work, we extend the above studies to discover rules of *arbitrary lengths*. In other studies on mining specifications such as in [ABL02, LK06], an automaton is mined instead of a set of rules from program execution traces. 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 significantly observed. The two families of specification mining techniques producing automata and rules are complementary to each other. There are other work in data mining domain on mining patterns & knowledge from data e.g., [AS95]. However, the formats and semantics of the mined representations are different from temporal rules. Temporal rules are intuitive, easily understandable and are useful for program comprehension & verification purposes. Hence, it deserves a separate consideration and mining algorithm.

3. Uniqueness of The Approach

The current algorithms mining temporal rules only mine significant rules of length 2 [YEBBD06, WN05]. These 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 rule. For mining all significant rules of length K, the number of possible rules to be checked is n^K, where n is the alphabet size. For mining significant rules of arbitrary lengths, this K is not known in advance and is only bounded by the length of the trace (which can be of hundreds or thousands of events). Note that simply concatenating significant 2-event rules will not work as it might miss some significant multi-event rules or introduce superfluous rules that are not significant.

To address the above challenge, we develop a novel rule mining algorithm. We model mining as a search space exploration and utilize effective search space pruning strategies to make mining significant temporal rules of arbitrary lengths feasible. In particular, the following `a priori' properties are used to prune sub-search spaces containing non-significant rules:

(Prop. 1) If a rule evsP -> evsC doesn't satisfy the support threshold, neither does any rule evsQ -> evsC where evsQ is a super-sequence of evsP.
(Prop. 2) If a rule evsP -> evsC doesn't satisfy the confidence threshold, neither does any rule evsP -> evsD where evsD is a super-sequence of evsC.

With the above properties, after checking and finding that a rule, say <a>-><b>, is not significant, one does not need to check for <a,b>-><c> or <a>-><b,c> because they will also not be significant. A large sub-search space can then be pruned.

Furthermore, we notice that many significant rules are redundant. For example, for an Automated Teller Machine (ATM) the rule: <accept_card>-><enter_pin, eject_card> is rendered redundant by <accept_card>-> <enter_pin, display_goodbye, eject_card> Hence, we only retain non-redundant rules. This can drastically reduce the number of reported rules as a non-redundant rule can cover a potentially exponential (wrt. the length of the non-redundant rule) number of rules.

Our algorithm first computes a pruned set of significant pre-conditions obeying the minimum support threshold. Next, for each pre-condition, we find a pruned set of post-conditions forming rules obeying minimum support and confidence thresholds. The resultant set of rules are then subjected to one more check to remove *remaining* redundant rules. Most of the redundant rules have been removed when the pruned set of pre- and post-conditions are computed, thus further improving the efficiency of our algorithm. The detail of redundant rule pruning strategy is complicated; we describe it together with a more complete description of our algorithm and a performance study in [TR]. With the `a priori' pruning strategies, the number of rules checked by our algorithm is close to the number of *significant* rules. Hence, the algorithm runtime can be approximated to O(|SR|*|T|), where |SR| is the number of significant rules and |T| is the size of the input trace set. Usually, the number of rules checked is even much less due to the pruning of search spaces containing *redundant* rules. In our case studies, we show that rather than checking the significance of more than 64^{100} rules (by a simple extension of two-event rule mining algorithm to mining rules of arbitrary lengths, which takes an exorbitant amount of time), our algorithm completes in less than 30 seconds.

In this work, we provide a guarantee that all rules mined are significant (sound) and all significant rules are mined (complete). We refer to this property as ``statistical" soundness and completeness. Hypothetically, if the input traces are sound and complete, the resultant mined specifications after employing our dynamic analysis technique will also be sound and complete.

4. Results & Contribution

Results. To evaluate our approach, we first performed a case study on the transaction component of JBoss Application Server (JBoss AS). The purpose is to show the applicability of our method in providing insight into the protocol that a code obeys -- hence aiding program comprehension. Another case study on a buggy CVS (Concurrent Versions System) application adapted from the one previously studied in [LK06] 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 2551 events, with 64 unique events, were generated. Running the algorithm with the minimum support and confidence thresholds set at twenty-five traces and ninety-percent respectively, 182 non-redundant rules were mined. The algorithm completed within 30 seconds.

A sample of the mined rules is shown in Figure 1. The 20-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 1-11) at the start of a transaction is always followed by the series of events <transaction completion events and resource release events> (event 12-20) at the end of the transaction. The above rule describing the temporal relationship and constraint between the 20 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 transaction component code base.

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

Figure 1. A 20-Event Rule Mined from JBoss-Transaction. Read Top-to-Bottom Left-to-Right.

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 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 [LK06] for a more detailed diagram). We focus on the two scenarios of multiple-file upload and deletion. 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 the system log file. Multiple versions of the same file stored in the CVS are maintained by adding timestamps to old versions of the file.

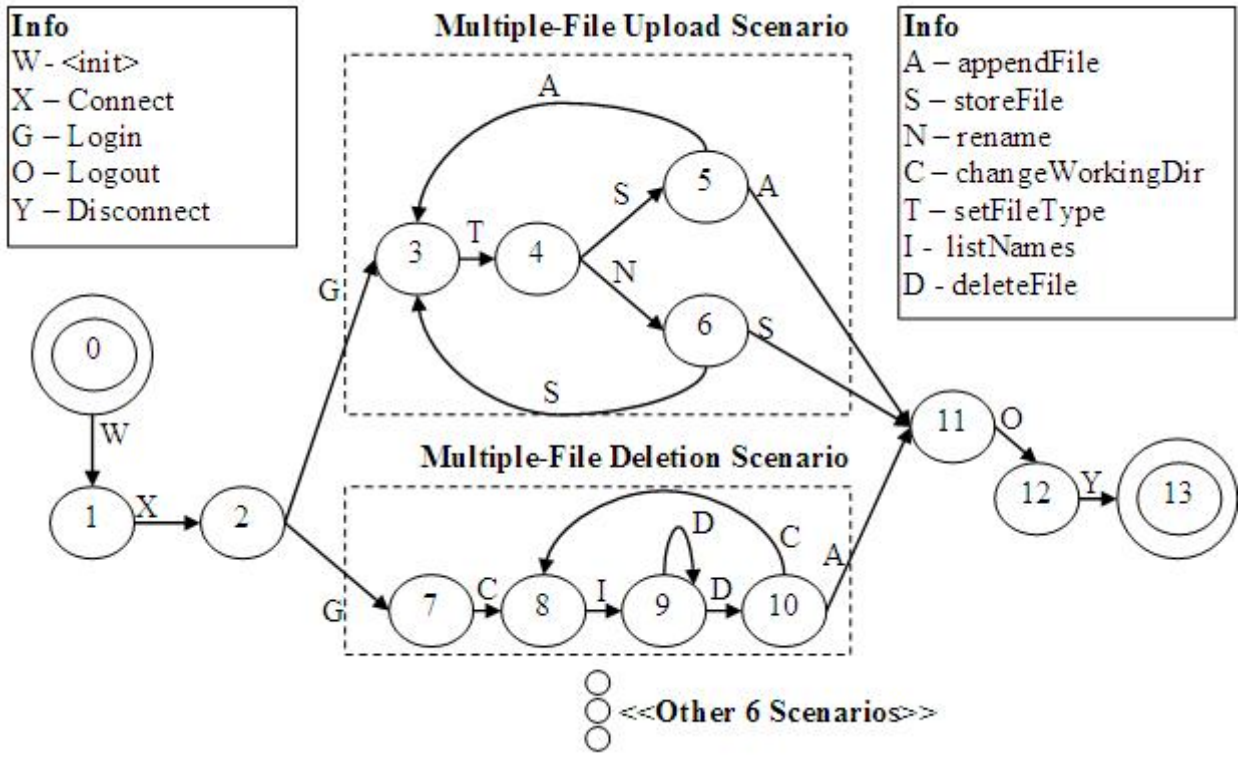

Figure 2. CVS Protocol

The CVS application is buggy, there are 4 bugs, each of them causes *inconsistent system log file* error. 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 [LK06]. 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 than 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*: <W,X,G,T,N> -> <S,O,Y>.
2. <W,X,G,C,I,D> -> <A,O,Y>

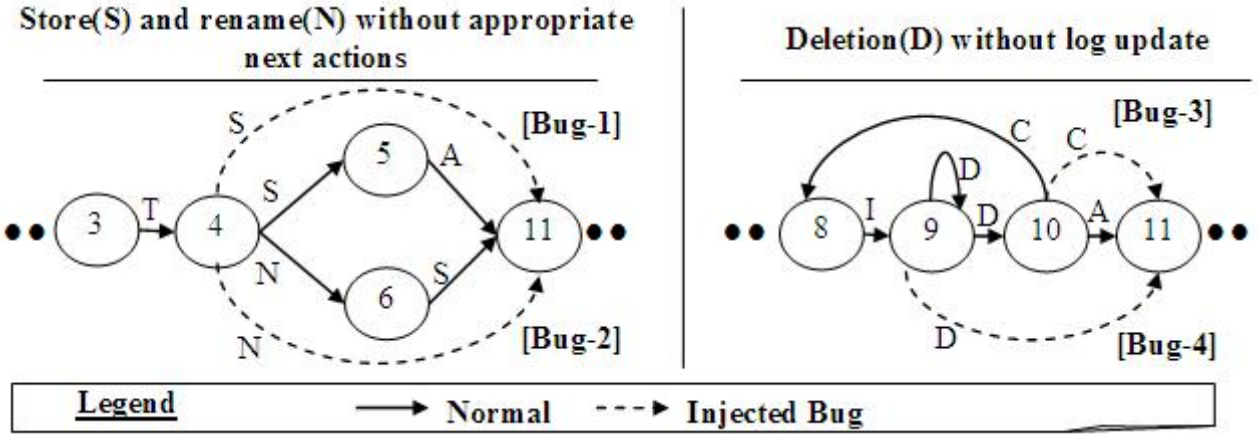

Figure 3. Injected Bug

We used the model checker described in [HKNP06]. 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.

Contribution. In this work, a novel method to mine a set of *non-redundant statistically significant* rules *of arbitrary lengths* of the form: ``Whenever a series of events *pre* occurs, eventually another series of events *post* also occurs'' is proposed. The problems of exponential runtime cost and huge number of reported rules have been effectively mitigated by employing search space pruning strategies and by eliminating redundant rules. Case studies have been conducted to demonstrate the usability of the proposed technique for program comprehension and verification.

References

[ABL02] G. Ammons, R. Bodik and J.R. Larus. Mining Specifications. In *Symposium on Principles of Programming Languages*, 2002.
[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *IEEE International Conference on Data Engineering*, 1995.
[CC02] G. Canfora and A. Cimitile. Software maintenance. *Handbook of Software Engineering and Knowledge Engineering*, 2002.
[CGP99] E.M. Clarke, O. Grumberg and D.A. Peled. *Model Checking*. MIT Press, 1999.
[DAC99] M. Dwyer, G. Avrunin, and J.Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*.1999
[HK06] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*, 2nd Ed. Morgan Kauffman, 2006.
[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2006.
[HR04] M. Huth and M. Ryan. *Logic in Computer Science.* Cambridge. 2004.
[LK06] D. Lo and S-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miners. In *International Symposium on Foundations of Software Engineering*, 2006.
[TR] D. Lo, S-C. Khoo, and C. Liu. Mining Temporal Rules for Software Maintenance. Technical Report at: www.comp.nus.edu.sg/~dlo/SRC-TR.pdf
[YEBBD06] J. Yang, D. Evans, D Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules form imperfect traces. In *International Conference on Software Engineering*, 2006.
[WN05] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.