Software Specification Discovery : A New Data Mining Approach

David Lo and Siau-Cheng Khoo
Department of Computer Science, National University of Singapore
dlo@comp.nus.edu.sg

Abstract

Software has been an ubiquitous component in our daily life. It ranges from large software systems like operating systems to small embedded systems like vending machines, both of which we frequently interact with. Software changes often during its lifespan; these cause difficulty in understanding existing systems. Program comprehension is estimated to take up to 45% of software costs which goes up to billions of dollars. One of the root causes of this problem is the fact that documented software specification is often missing, incomplete or outdated. This causes difficulties in software maintenance efforts especially when a software project involves many people or the project continues over a long period of time. Lack of documented software specification also causes difficulty in testing or verifying the correctness of a software system. Incorrect software has caused the loss of billions of dollars and even the loss of life. One solution to address the above problems is specification discovery, namely, automated extraction of software specification from program artifacts. In this paper, we describe our past and current work in the domains of data mining, software engineering and programming language in addressing the discovery of software specifications with the goal of reducing software costs and improving software dependability.

1 Introduction

It's best if all programs and software projects are developed with clear, precise and documented specifications. However, due to hard deadlines and 'short-time-to-market' requirement [8], software products often come with poor, incomplete and even no documented specification. This situation is further aggravated by the phenomenon termed as software evolution [6, 26]. As software evolves (*i.e.* changes) the documented specification is often not updated. This might render the original specification of little use after several cycles of program evolution [13].

Incomplete, outdated and missing specification has contributed to high software maintenance costs. It has been investigated that 90% of software cost is due to maintenance [16] and 50% of the maintenance cost is due to comprehending or understanding an existing code base [39]. Hence, approximately 45% of software cost is due to difficulty in understanding an existing system. This is especially true for software projects developed by many developers over a long period of time. A good indication on the amount associated with software costs is the US GDP's software component which amounts to $216.0 billion at the second quarter 2007 alone [5]. Considering the above factors, reducing maintenance cost by addressing the problem of lack, incomplete and outdated specification can potentially save a large amount of wasted resources.

On another related front, software dependability is a major concern of software vendors and users. Software dependability related issues have caused the loss of billions of dollars and even the loss of lives [17]. As reported by US National Institute of Standards and Technology (NIST) in 2002, incorrect or buggy software has caused US economy to suffer $59.5 billion dollars loss annually [34]. Another case with Ariane 5, a rocket project by European Space Agency that exploded on its maiden voyage due to specification and design errors, highlights the need of addressing software dependability [3].

To ensure correctness of a software system, program verification tools [10] have been proposed. However, program verifier can only check specified properties (or specification). If a property is not documented properly, incomplete, or unavailable, not much can be done in ensuring the correctness of a software system. Also, the difficulty in formulating a set of *formal* properties, which is the format required by standard program verifiers, has been a barrier to its wide-spread adoption in the industry [2].

One approach to address the problem of incomplete, outdated and missing specification is *specification discovery* – automated extraction of software specification from program artifacts. Having a complete and updated software specification is a great help to lower software maintenance costs and improve software dependability. Mined specification can be utilized for aiding program understanding. Furthermore, it can be converted to run-time tests and input as properties-to-verify to standard program verifiers.

Software specification can be represented in various formalisms. One common formalism is automata (*i.e.*, a transition system with start and end nodes) [22]. Another formalism is Linear Temporal Logic (LTL) [23] expressions which can be directly accepted by standard program verifiers [10]. Also, formal versions of UML sequence diagram, namely Message Sequence Chart (MSC) [24] and Live Sequence Chart (LSC) [12], have been active interests in the software modeling community. Not only are the above formalisms intuitive enough to aid program understanding, they are also formal enough to be verified via verification techniques. Hence, they address both program comprehension and dependability issues.

In this paper we describe briefly three threads of our past and current work in recovering specification (in the form of automata [28, 29], LTL [27, 31] and LSC [30, 32]) from program execution traces via data mining approaches. A program execution trace can be simply viewed as a series of method signatures (or names) of methods invoked when a system is executed.

The outline of this paper is as follows. Section 2 highlights our work in automaton-based specification mining. Section 3 highlights our work in mining Linear Temporal Logic (LTL) expressions. Section 4 describes our work in mining Live Sequence Charts (LSCs). Section 5 discusses related work, and finally Section 6 concludes and discusses future work.

2 Mining Automata

Initial studies on mining software specification often represent a mined specification in the form of an automaton (*e.g.*, [2, 11, 38, 4]). The work by Ammons *et al.* is one of the pioneer in automaton-based specification mining [2]. In [2], a machine-learning approach (*i.e.*, an automaton learner referred to as sk-strings [36]) is employed to discover program specification in the form of an automata by analyzing program execution traces. It is assumed that input traces must "reveal strong hints of correct protocols" although they can also contain errors.

Our work proposes novel data mining algorithms (outlier detection and clustering techniques) to improve the quality of work in [2].

First, we define an array of metrics to objectively measure the performance of an automaton-based specification miner (*c.f.*, [28]). The most important one is an *accuracy* metric measured by the notion of *recall* and *precision*. Precision and recall can then be defined as *the proportion of sentences in* S_{inf} *that is accepted by* S_{orig} and *the proportion of sentences in* S_{orig} *that is accepted by* S_{inf} where S_{orig} is the original specification and S_{inf} is the inferred specification.

Next, we devise a novel architectural framework (referred to as SMArTIC) that achieves specification mining

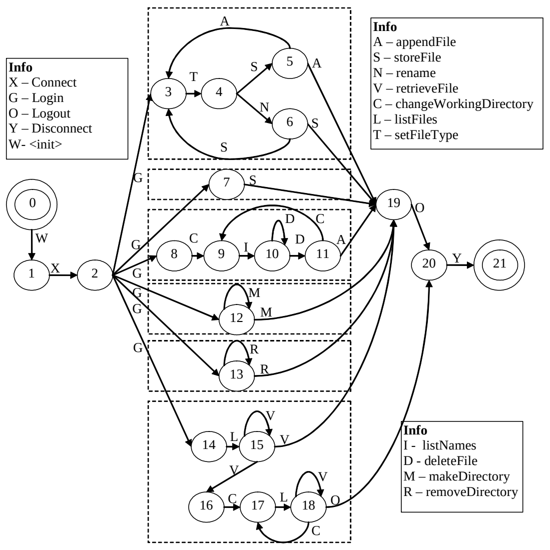

Figure 1. CVS Protocol

through pipelining of four functional components: Error-trace filtering, clustering, learning, and automata merging (*c.f.*, [29]). We demonstrate that such an architecture improves the quality of the mining result for two primary reasons:

1. Early identification and filtering of erroneous program execution traces can improve the quality of specification discovery.

2. Over-generalization which occurs at the learning stage can be mitigated by localization of learning process to groups of related program execution traces.

We conduct experiments to support our reasoning. Our experiments aim at deriving the API interaction protocol for a CVS application built on top of the Jakarta Commons Net FTP library [40]. There are six common FTP interaction scenarios in the CVS application: Initialization, multiple-file upload, download, and deletion, multiple-directory creation and deletion. All scenarios begin by connecting and logging-in to the FTP server. They end by logging-off and disconnecting from the FTP server. The client side only maintains a record of files backed-up in the FTP server.

All these scenarios are depicted in the automata shown in Figure 1. The dashed boxes, from top to bottom, represent file upload, repository initialization, file deletion, directory creation, directory deletion, and file download scenario respectively.

	Precs	**Recall**
SMArTIC	0.484	0.981
SMArTIC without filtering	0.426	1
SMArTIC without clustering	0.263	0.984
sk-strings	0.225	1.000

As shown in the table above, SMArTIC improves the precision while maintaining a good recall in the CVS protocol inference. The precision of SMArTIC are more than double the precision of sk-strings. Both filtering and clustering help in increasing precision while maintaining a good recall.

3 Mining Linear Temporal Logic

In this section, we describe mining Linear Temporal Logic (LTL) [23] expressions (or rules) satisfying given support and confidence thresholds. LTL is one of the two most commonly used formalisms accepted by standard program verifiers [10] – the other one is Computational Tree Logic (CTL) [23]. While an automata expresses a global picture of software specification (which might be complex), mined LTL expressions break the specification into smaller pieces each expressing *strongly observed behavior* which is easier to be understood.

Rules having the following format are the target of our mining algorithm:

’Whenever a series of events ES_{pre} occurs, eventually another series of events ES_{post} also occurs.’

The above can be denoted as $ES_{pre} \rightarrow ES_{post}$. ES_{pre} and ES_{post} are referred to as the *premise* and *consequent* of the rule respectively.

These set of rules can be expressed in LTL, and belong to two of the most frequently used families of temporal logic expressions for verification (*i.e.*, response and chain-response) according to a survey by Dwyer *et al.* [14]. Examples of such rules include:

1. Resource Locking Protocol: Whenever a lock is acquired, eventually it is released.
2. Network Protocol: Whenever an HDLC connection is made and an acknowledgement is received, eventually a disconnection message is sent and an acknowledgement is received.

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* point in time. The three examples listed in Table 1 illustrate the meaning and use of these operations.

$F(unlock)$ 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

Table 1. LTL Expressions and their Meanings

The set of LTL expressions minable by our mining framework is represented in the Backus-Naur Form (BNF) as follows:

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

To identify *strongly observed* rules, we introduce: *(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. Only rules satisfying user-defined thresholds of minimum support and confidence are mined.

An effective search space pruning strategy, inspired by closed pattern mining strategies [44, 41], is utilized to efficiently mine multi-event rules from traces. To prevent blow-up in the number of rules, only a representative sub-set of rules containing non-redundant ones is generated.

A case study is performed on traces from the security component of JBoss Application Server [37]. This shows the usefulness of our mining technique in recovering the underlying protocol that the system obeys, thus aiding program comprehension.

One of the mined rules (shown with abbreviated method signatures) is presented in Figure 2. It describes authentication using Java Authentication and Authorization Service (JAAS) for EJB within JBoss-AS. When authentication scenario starts, configuration information is first 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 in the consequent of the rule.

Premise $\longrightarrow$	**Consequent**

XLoginConfImpl.getConfEntry()
AuthenticationInfo.getName()
	ClientLoginModule.initialize()
	ClientLoginModule.login()
	ClientLoginModule.commit()
	SecAssocActs.setPrincipalInfo()
	SecAssocActs.pushSubjectCtxt()
	SimplePrincipal.toString()
	SecAssoc.getPrincipal()
	SecAssoc.getCredential()
	SecAssoc.getPrincipal()
	SecAssoc.getCredential()

Figure 2. A Rule from JBoss-Security

We have performed another case study to discover specifications of the transaction component of JBoss-AS. Also, a separate study on the integration of mined rules with a program verifier for bug discovery has been performed. Interested readers might refer to [27, 31] for details.

4 Mining Live Sequence Charts

Scenarios, depicted using variants of sequence diagrams, are popular means to specify the inter-object behavior of systems (see, e.g., [20]). They are included in the UML standard, and are supported by many modeling tools. In particular, we are interested in mining modal scenarios presented using a UML2 compliant variant of Live Sequence Chart (LSC) [12, 21]. LSC is an extension of International Telecommunication Union (ITU) standard on Message Sequence Chart [24]. Different from standard UML sequence diagram, MSC and LSC describe *constraints* on traces.

We address the mining problem in two steps. In [30], we propose iterative pattern, extending work on sequential pattern mining [1, 44, 41] and episode mining [33, 18]. An iterative pattern is not an LSC or MSC, however its semantics obeys the constraints of MSC and LSC. In [32], we extend iterative pattern mining to mine LSC.

Iterative pattern is a series of events supported by a *significant number of instances repeated within and across sequences (or traces)*. 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 closely together in a window.

The above differences are required for analyzing program traces and inferring specifications or properties. Due to loops and recursions, a trace can contain repeated occurrences of interesting patterns. Also, program properties are often inferred from a set of traces instead of a single trace. Finally, important patterns for verification, such as lock acquire and release or stream open and close (*c.f* [45, 9]), 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' in order to capture these patterns of interest.

Iterative pattern obeys the following apriori property utilized by depth-first search sequential pattern miners (*e.g.*, FreeSpan [19] and PrefixSpan [35]) which states:

If P is not frequent then $P{+\!\!+}evs$ (where evs is a series of events) is also not frequent.

Due to possibly combinatorial number of frequent subsequences of a long pattern, it's best to mine a closed set of patterns (*c.f.*, [44] & [41]). Closed pattern mining discovers patterns without any super-sequence having corresponding set of instances. In [30], we mine a *closed* set of iterative patterns. A search space pruning strategy employed by *early identification and pruning* of non-closed patterns is used to mine a closed set of iterative patterns efficiently. Our performance study on synthetic and real-world datasets shows the success of our closed pattern miner: it runs with over an order of magnitude speedup (over mining a full set of frequent patterns) especially on low support thresholds or when frequent patterns are long.

In [32], we extend the algorithm to mine LSCs. To demonstrate and evaluate our approach, we present the results of a case study we have conducted using traces from various components of Jeti [25], an open-source Java based full featured instant messaging application. The results demonstrate the effectiveness of our mining technique in recovering non-trivial and expressive underlying interactions.

One of the mined LSCs involving sending of messages when one client starts communicating with another is shown in Fig. 3. The scenario starts whenever a user uses the roster tree to select a party to communicate with. Then, the roster tree will initiate the chat and set up a chat window. After several resources and identifiers of communicating parties are obtained, eventually, an initial message is sent via the `Backend`/`Connect`/`Output` channel.

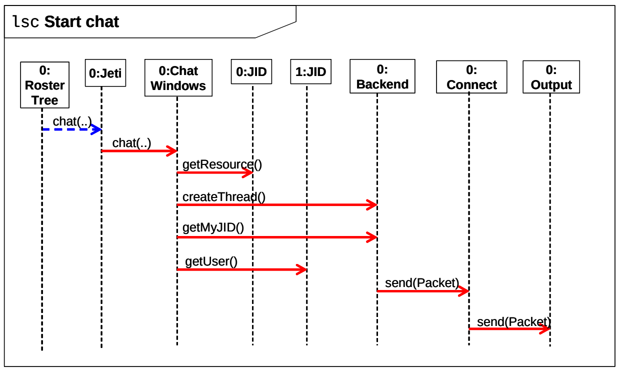

Figure 3. A Mined LSC: `Start chat`

5 Related Work

For a complete description of related work, interested readers can refer to the related work sections of our past studies [29, 28, 27, 31, 30, 32]. In this section, only some related studies will be highlighted.

Mining Automata. In [29], we extended the specification miner described in [2]. In another work, Whaley *et al.* extract object-oriented component interface sequencing constraints to form multiple finite state automatons [43]. Reiss *et al.* encode program execution traces as directed acylic graphs to aid visualization and understanding of programs [38]. Arts *et al.* dynamically extract program models from Erlang programs as state graph models for model checking and visualization [4]. We believe that these and other similar miners can benefit from our data-mining based architecture of filtering errors and clustering traces, with minimal changes.

Mining LTL. Of the most relevance is the work on mining rule-based specification [45, 42], where the rules have a similar semantics as our work [27] 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.

Mining LSCs. Several studies reverse engineers object interactions from program traces and visualize them using sequence diagrams (see, e.g., [15, 7]). These might seem similar to our work in [32]. However, different from our work, these studies simply represent the *entire trace* as a sequence diagram. On the other hand, we mine *strongly observed* LSCs expressed in the traces.

6 Conclusion & Future Work

In this paper, we have proposed a new data mining approach, namely specification discovery, which is a process for automated extraction of software specification from program artifacts. Three separate threads of work mining different form of specifications (automata, LTL and LSC) from program execution traces have been briefly described. Not only can these specification formalisms be easily understood by software developers, they are also formal enough to be verified by program verifiers. Hence, mining the above specification formalisms can aid both program comprehension and dependability.

Our future research directions include: improving the scalability of the mining processes further, incorporating user-defined constraints suitable for software developers, performing more case studies especially on more industrial systems, and exploring further intersection of techniques in data mining, software engineering and programming language to improve specification discovery process.

Acknowledgement. We would like to thank our co-authors of some of our past studies referred to in this paper, namely, Chao Liu and Shahar Maoz. Also we would like to thank Glenn Ammons, Jiawei Han and David Harel for their advice on some of our past studies referred to in this paper.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, 1995.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *SIGPLAN-SIGACT POPL*, 2002.

[3] ARIANE 5 Failure - Full Report. http://www.ima.umn.edu/∼arnold/disasters/ariane5rep.html, 1996.

[4] T. Arts and L. Fredlund. Trace analysis of erlang program. In *Proc. of Erlang Work.*, 2002.

[5] BEA: News Release: Gross Domestic Product. http://www.bea.gov/newsreleases/national/gdp/gdpnewsrelease.htm, Sep. 2007.

[6] B. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[7] L. C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed Java software. *IEEE Trans. Software Engineering*, 32(9):642–663, 2006.

[8] R. Capilla and J. Duenas. Light-weight product-lines for evolution and maintenance of web sites. In *CSMR*, 2003.

[9] W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. Nguyen. Verifying safety policies with size properties and alias controls. In *ICSE*, 2005.

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

[11] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[12] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 19(1):45–80, 2001.

[13] S. Deelstra, M. Sinnema, and J. Bosch. Experiences in software product families: Problems and issues during product derivation. In *SPLC*, 2004.

[14] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.

[15] Eclipse Test and Performance Tools Platform. http://www.eclipse.org/tptp/.

[16] L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Pro*, 17–23, 2000.

[17] GAO Report: Patriot Missile Defense – Software Problem Led to System Failure at Dhahran, Saudi. http://www.fas.org/spp/starwars/gao/im92026.htm, 1992.

[18] G. Garriga. Discovering unbounded episodes in sequential data. In *PKDD*, 2003.

[19] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *KDD*, 2000.

[20] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.

[21] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. *Software and System Modeling*, 2007.

[22] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Thoery, Language, and Computation*. Addison Wesley, 2001.

[23] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge, 2004.

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

[25] Jeti. Version 0.7.6 (Oct. 2006). http://jeti.sourceforge.net/.

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

[27] D. Lo. A sound and complete specification miner. In *SIGPLAN PLDI Student Research Competition (2^{nd} position). http://www.acm.org/src/winners.html*, 2007.

[28] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *IEEE WCRE*, 2006.

[29] D. Lo and S.-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.

[30] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD*, 2007.

[31] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules from program execution traces. *Int. Work. on Prog. Comprehension through Dynamic Analysis (to appear)*, 2007.

[32] D. Lo, S. Maoz, and S.-C. Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *IEEE/SIGSOFT ASE (to appear)*, 2007.

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

[34] Software Errors Cost U.S. Economy $59.5 Billion Annually. http://www.nist.gov/public_affairs/releases/n02-10.htm, 2002.

[35] 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 *ICDE*, 2001.

[36] A. V. Raman and J. D. Patrick. The sk-strings method for inferring pfsa. In *Proc. of the Work. on Automata Induction, Grammatical Inference and Language Acquisition at ICML*, 1997.

[37] Red Hat Middleware, LLC. JBoss.com - JBoss application server. *http://www.jboss.org/products/jbossas*.

[38] S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, 2001.

[39] T. Standish. An essay on software reuse. *IEEE Trans. on Software Engineering*, 5(10):494–497, 1984.

[40] The Apache Software Foundation. Jakarta Commons/Net. *http://jakarta.apache.org/commons/net/*.

[41] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *ICDE*, 2004.

[42] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005.

[43] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object oriented component interfaces. In *ISSTA*, 2002.

[44] X. Yan, J. Han, and R. Afhar. CloSpan: Mining closed sequential patterns in large datasets. In *SDM*, 2003.

[45] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.