# Identifying the Software Failure Mechanisms Using Data Mining Techniques

**Nadhem Sultan Ali Ebrahim, V.P Pawar**

***Abstract. Software is ubiquitous in our daily life. It brings us great convenience and a big headache about software reliability as well: Software is never bug-free, and software bugs keep incurring monetary loss or even catastrophes. In the pursuit of better reliability, software engineering researchers found that huge amount of data in various forms can be collected from software systems, and these data, when properly analyzed, can help improve software reliability. Unfortunately, the huge volume of complex data renders simple analysis techniques incompetent; consequently, Studies have been resorting to data mining for more effective analysis. In the past few years, we have witnessed many studies on mining for software reliability reported in data mining as well as software engineering forums. These studies either develop new or apply existing data mining techniques to tackle reliability problems from different angles. In order to keep data mining researchers abreast of the latest development in this growing research area, we propose this Paper on mining for software reliability.***

***Keywords- Reliability, Techniques***

## I. INTRODUCTION

Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. Although Software Reliability is defined as a probabilistic function, and comes with the notion of time, we must note that, different from traditional Hardware

Reliability, Software Reliability is not a direct function of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded.

Software Reliability is an important to attribute of software quality, together with functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. For example, large next-generation aircraft will have over one million source lines of software on-board; next-generation air traffic control systems will contain between one and two million lines; the upcoming international Space Station will have over two million lines on-board and over ten million lines of ground support software; several major life-critical defense systems will have over five million source lines of software.

**Manuscript received on September, 2013.**

**Nadhem Sultan Ali Ebrahim,** SRTM University Department of Computational Science Nanded Mahrashtra, India.

**Dr.V.PPawar,** SRTM University Department of Computational Science Nanded Mahrashtra, India.

While the complexity of software is inversely related to software reliability, it is directly related to other important factors in software quality, especially functionality, capability, etc. Emphasizing these features will tend to add more complexity to software.

## II. SOFTWARE FAILURE MECHANISMS

Software failures may be due to errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems. While it is tempting to draw an analogy between Software Reliability and Hardware Reliability, software and hardware have basic differences that make them different in failure mechanisms. Hardware faults are mostly physical faults, while software faults are design faults, which are harder to visualize, classify, detect, and correct. Design faults are closely related to fuzzy human factors and the design process, which we don't have a solid understanding. In hardware, design faults may also exist, but physical faults usually dominate. In software, we can hardly find a strict corresponding counterpart for "manufacturing" as hardware manufacturing process, if the simple action of uploading software modules into place does not count. Therefore, the quality of software will not change once it is uploaded into the storage and start running.

## III. SOFTWARE RELIABILITY IMPROVEMENT TECHNIQUES

Good engineering methods can largely improve software reliability. Before the deployment of software products, testing, verification and validation are necessary steps. Software testing is heavily used to trigger, locate and remove software defects. Software testing is still in its infant stage; testing is crafted to suit specific needs in various software development projects in an ad-hoc manner. Various analysis tools such as trend analysis, fault-tree analysis, Orthogonal Defect classification and formal methods, etc, can also be used to minimize the possibility of defect occurrence after release and therefore improve software reliability.

To achieve the preceding goal, developers often want to reuse existing frameworks or libraries instead of developing similar code artifacts from scratch. The challenging aspect for developers in reusing the existing frameworks or libraries is to understand the usage patterns and ordering rules (specifications) among Application Programming Interfaces (APIs) exposed by those frameworks or libraries, because many of the existing frameworks or libraries are not well documented. Incorrect usage of APIs may lead to violated API specifications, leading to security and robustness defects in the software. Furthermore, usage

*Published By:*
*Blue Eyes Intelligence Engineering*
*& Sciences Publication*

patterns and specifications might change with library re factorings, requiring changes in the software that reuse the library. To address these issues, we develop a technique (based on data mining) that automatically mine usage patterns and specifications, and detect re factorings from source code. Our techniques aid developers in productively reusing third party libraries to build reliable and secure software. We present three infrastructures based on mining source code to address the main issues faced by developers in reusing API libraries. The tracing infrastructure automatically mines API usage patterns and specifications from API client code in local source code repositories. The searching infrastructure expands the scope of mining to also include billions of lines of open-source API client code available on the web. The re factoring-detection infrastructure automatically detects re factorings in libraries by analyzing library API implementation code.

### *3.1 Tracing Infrastructure*

A software system interacts with third-party libraries through various APIs. Using these library APIs often needs to follow certain usage patterns (how to use a given set of APIs for a particular task?). Furthermore, ordering rules (specifications) exist between APIs, and these rules govern the secure and robust operation of the system using these APIs.

Unfortunately, API usage patterns and various API specifications are not well documented by the API-library developers. API patterns cut across procedural boundaries and an attempt to infer these patterns by manual inspection of source code (API client code) is often inefficient and inaccurate. Several problems exist even when the API specifications are known. API specifications (when known) can be formally written for third-party APIs and statically verified against a software system. But manually writing a large number of formal API specifications for static verification is often inaccurate or incomplete, apart from being cumbersome. Formal specifications are complicated and lengthy mainly due to the various API details (such as input/return type, error flags, and return values for APIs on success/failure) and language syntax considerations required for the specification to be accurate and complete. To address these issues, we present the tracing infrastructure that mines API details, patterns, and specifications by analyzing the source code (API client code). In this section, we present tracing infrastructure and the three tools based on the infrastructure,namely, API Pattern Miner, API Error Detector, and IDeaMiner (Section 3.2). The high-level overview of the tracing infrastructure is shown in Figure 1. The tracing infrastructure has four main components: trace generator, scenario extractor, miners, and pattern extractor. The trace generator uses compile-time push-down model-checking (PDMC) to generate inter-procedural static traces, which approximate run-time API behaviors. The PDMC process verifies a property specified in the form of Finite State Machine (FSM) over a given program. Using Triggers, a form of FSM, we adapt the PDMC process to output static traces in the program involving APIs of interest. A single static trace from the model checker might involve several API usage scenarios, being often interspersed. The scenario extractor separates different usage scenarios from a given trace, so that each scenario can be fed separately to the miners, our next component. The miner component employs various data mining techniques on these static traces to output frequent partial orders or frequent sequences (based on the employed data-mining technique) among APIs. The miner output is then processed by the pattern extractor to output API details, patterns, and specifications.

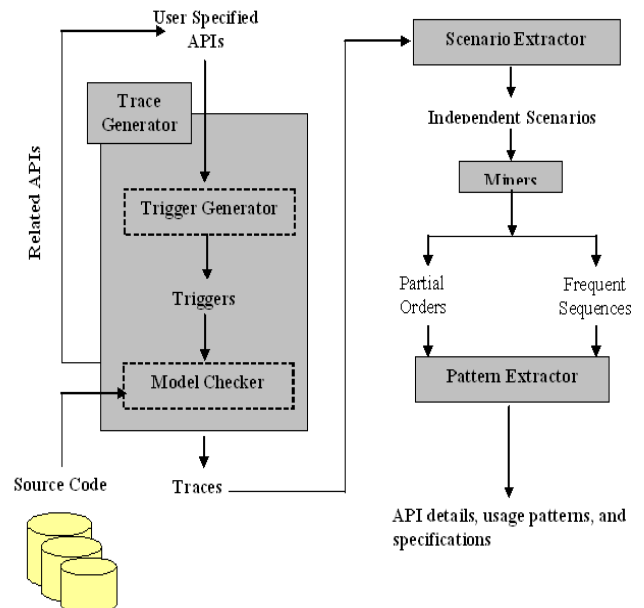


**Figure 1:- Tracing Infrastructure**

### *3.2 API Pattern Miners*

API Pattern Miner employs the tracing infrastructure to mine usage patterns and specifications that involve multiple-API sequences from the static traces. Previous approaches mine frequent association rules, item sets, or subsequences that capture API call patterns shared by API client code. However, these frequent API patterns cannot completely capture some useful orderings shared by APIs, especially when multiple APIs are involved across different procedures. API Pattern Miner summarizes API usage patterns as partial orders. Different API usage scenarios are extracted from the static traces by our scenario extraction algorithm and fed to a Frequent Closed Partial Order (FCPO) miner. The miner summarizes different usage patterns as compact partial orders. The usage patterns can be used as a recommender, which shows how to use a set of APIs for a particular task.

### *3.3 API Error Detectors*

Incorrect handling of errors incurred after API invocations (in short, API errors) can lead to security and robustness problems, two primary threats to software reliability. Correct handling of API errors can be specified as formal specifications, verifiable by static checkers, to ensure dependable computing. But API error specifications are often unavailable or imprecise, and cannot be inferred easily by source code inspection. Based on our tracing infrastructure, we develop a technique called API Error Detector, for tactically mining API error specifications automatically from software package repositories, without requiring any user input. Similar to API Pattern Miner, API Error Detector employs the tracing infrastructure to approximate run-time API error behaviors with static traces. Frequent sequence mining is used on these static traces to mine specifications that define the correct handling of errors for relevant APIs used in the software packages. The mined specifications are then used to uncover API error-handling bugs.

### *3.4 IDeaMiner*

Manually writing formal specifications for static

verification can be cumbersome. Based on the tracing infrastructure, we implement IDeaMiner, which infers API details such as return values on success/failure, error flags, and return value type from the static traces. IDeaMiner implements simple data-flow extensions to the PDMC process to infer API details. Based on these inferred API details and the language syntax (user-provided, as a one-time AST database for a given language), Specifier tool translates user-specified generic API rules to concrete formal specifications verifiable by static checkers. Users can specify generic rules at an abstract level that needs no knowledge of the source code, system, or API details.

## IV. CONCLUSIONS

Software reliability is a key part in software quality. The study of software reliability can be categorized into three parts: modeling, measurement and improvement.

Software reliability measurement is naive. Measurement is far from commonplace in software, as in other engineering field. "How good is the software, quantitatively?" As simple as the question is, there is still no good answer. Software reliability can not be directly measured, so other related factors are measured to estimate software reliability and compare it among products. Development process, faults and failures found are all factors related to software reliability.

Software reliability improvement is hard. The difficulty of the problem stems from insufficient understanding of software reliability and in general, the characteristics of software. Until now there is no good way to conquer the complexity problem of software.

Complete testing of a moderately complex software module is infeasible. Defect-free software product can not be assured. Realistic constraints of time and budget severely limits the effort put into software reliability improvement.

As more and more software is creeping into embedded systems, we must make sure they don't embed disasters. If not considered carefully, software reliability can be the reliability bottleneck of the whole system. Ensuring software reliability is no easy task. As hard as the problem is, promising progresses are still being made toward more reliable software. More standard components and better process are introduced in software engineering field.

## REFERENCES

1. Breuker J. and Van Der Velde W: Common KADS Library for Expertise Modelling, Amsterdam: IOS Press, 1994.
2. Carroll J. M., Scenario-Based Design: Envisioning Work and Technology in System Development, New York: Wiley, 1995.
3. Fenton N. and Pfleeger S.L. Software Metrics: A Rigorous Approach. 2nd ed. London: International Thomson Computer Press, 1997.
4. Fenton N.: Applying Bayesian belief networks to critical systems assessment. Critical Systems. Club Newsletter, Vol 8, 3 Mar. 1999,pp. 10–13.
5. Fenton N., Maiden N.: Making Decisions: Using Bayesian Nets and MCDA. Computer Science Dept, Queen Mary and Westfield College, London, 2000
6. Frawley J, Piatetsky-Shapiro G, Matheus C. J.: Knowledge Discovery in Databases: An Overview, in Knowledge Discovery in Databases. Cambridge, MA: AAAI/MIT, 1991, pp. 1-27.
7. Goebel M. and Gruenwald L.: A Survey of Data Mining and Knowledge Discovery Software Tools. ACM SIGKDD, June 1999, Volume I, Issue 1-page20.
8. Gregoriades A., Sutclife A. and Shin J. E.: Assessing the Reliability of Sociotechnical Systems. 12th Annual Symposium INCOSE (Las Vegas, July 2002).
9. Hollnagel E.: Human Reliability Analysis-Context and Control. Academic Press, Inc., New York, NY, 1993.
10. Pearl, J., Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann, San Mateo, CA, 1988.
11. Reason J.: Human Error, Cambridge University Press. New York, NY,1990.
12. Reason J.: Managing the Risks of Organizational Accidents. Aldershot: Ashgate, 2000.
13. Sutcliffe A. G., Galliers J.: Human error and system requirements. 4Th International Symposium onon Requirements Engineering, RE’1999, 1999.
14. Sutcliffe A., Gregoriades A.: Validating Functional System Requirements with Scenarios. proceeding of the RE02 IEEE international Conference. Essen, Germany.


*Published By:*
*Blue Eyes Intelligence Engineering*
*& Sciences Publication*