REFINEMENT TECHNIQUES IN MINING SOFTWARE BEHAVIOR

ZHIQIANG ZUO

NATIONAL UNIVERSITY OF SINGAPORE

2015

REFINEMENT TECHNIQUES IN MINING SOFTWARE BEHAVIOR

ZHIQIANG ZUO

BEng., Shandong University (China), 2010

A DISSERTATION SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2015

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

Zhiqiang Zuo

February 4, 2015

To my parents, for their selfless and endless love

Acknowledgment

It is hard to believe that this poor boy from the countryside in northern China has got a PhD at such a world-level school of National University of Singapore. I still remember that at the first semester when I came to Singapore in 2010, I felt stressed and lost. But now I finish my dissertation. I could not have imagined what I would be today without the help and support from many people, some of whom it is possible to mention here.

First and foremost, I would like to express my heartfelt gratitude to my advisor Dr. Siau Cheng Khoo for his continuous guidance and support during my PhD study. His rigorous style lets me understand how to do a good research. His optimism is contagious and motivational to me, especially during the tough time in the pursuit of my PhD. It is he who teaches me, both consciously and unconsciously, that there is always a solution to a problem. This has furnished me with the patience, confidence and enthusiasm, still now and in the future.

I gratefully acknowledge Dr. Wei Ngan Chin, Dr. Mong Li Lee and Dr. Lingxiao Jiang for agreeing to serve in my thesis committee. I would also like to thank Dr. Wei Ngan Chin and Dr. Jin Song Dong who served in my qualifying committee. Their insightful and valuable feedback helps to improve this dissertation a lot.

My thanks also go to my research seniors: Dr. David Lo, Sandeep Kumar, Chengnian Sun, who set examples for me in terms of hard work and research productivity. I also thank the labmates in my group: Narcisa Andreea Milea, Anh Cuong Nguyen, Ta Quang Trung etc., for the stimulating and inspiring discussions, and for the great pleasure in an awesome lab.

I am grateful to my seniors: Jingbo Zhou, Jinyu Xu and Yugang Liu for their help and care especially at the beginning of my life in Singapore. I also thank my friends: Jiexin Zhang, Yukun Shi, Xingliang Liu, Yongzheng Wu, Nan Ye, Chengwen Luo, Zhuolun Li, Jianxing Wang, Kegui Wu, Jing Zhai, Shuang Liu, Tao Chen etc., for eating, playing games, watching and sharing movies together. They are my dear "fair-weather" friends. But I also saw them in the "bad weather". I also want to say thanks to all the friends playing basketball together for almost four years even though we do not know each other's name. I indeed got a lot of fun and health from the court with them.

Last but not the least, I would like to thank my parents, Jinliang Zuo and Xiuying Guan, who raised and educated me to be who I am today. It is their unworldliness, honesty, guilelessness, diligence, and thrift that teach me what is worthy and what I should really care about, how to deal with people, and how to deal with myself. I dedicate this dissertation to them. I also thank my grandparents Baozhen Zuo, Fengrong Zhao, and my younger sister Ruiping Zuo who have always been the source of love, support and motivation to me.

February 4, 2015

Contents

Summary

Mining software behavior has been well studied to assist in numerous software engineering tasks for the past two decades. Two research topics which received much attention are specification mining and statistical debugging. To tackle the lack of precise and complete specifications, specification mining is proposed to automatically infer software behavior from the execution traces as specifications. In order to support debugging activities, researchers have developed various statistical debugging approaches (e.g., statistical bug isolation and bug signature mining) which commonly collect two groups of execution traces and employ statistical techniques to discover the discriminative elements as bug causes or signatures.

Among the execution traces analyzed by both specification mining and statistical debugging, there exist a significant number of useless elements. Mining directly over the raw execution traces wastes many computing resources and possibly produces meaningless results due to the meaningless elements. To enhance the efficiency and effectiveness of software behavior mining, *refinement techniques* are required to remove unwanted elements from raw execution traces. However, currently there is a lack of systematic refinement techniques for both software behavior mining studies. This dissertation presents a specific systematic refinement technique for each of the above two studies.

For specification mining, we propose a *semantics-directed specification mining* framework which exploits a user-specified semantic analysis to filter out the semantically irrelevant events from execution traces before mining. Consequently, specifications mined are all semantically significant, and mining becomes far more efficient. Based on the framework, we present a particular *dataflow sensitive specification mining* system where dataflow semantics is taken into consideration. The experimental results show that our approach generates high-quality specifications and scales well to real-world programs. Moreover, the mined specifications can practically help program understanding and bug detection.

For statistical debugging, we devise a novel *hierarchical instrumentation* (HI) technique to refine the execution traces. Based on HI, we safely and effectively prune away unnecessary instrumentation, and thus greatly reduce the overhead of statistical debugging. We apply the HI technique to both in-house debugging and cooperative debugging for field failures. The empirical evaluation validates that our HI technique effectively refines the execution traces under analysis by pruning away unnecessary instrumentation. The efficiency of debugging is significantly strengthened.

Keywords: specification mining, dataflow, automated debugging, hierarchical instrumentation, statistical bug isolation, bug signature mining, field failures

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

As modern software systems grow in capability and complexity, they greatly increase the difficulties and challenges in software development and maintenance. Over the past two decades, in order to improve software productivity and quality, data mining techniques are widely applied to discover software behavior from a variety of software engineering data, e.g., source code, documentations, bug reports, and execution traces [75, 111, 97]. Plenty of such research and development studies have provided practical assistance in many software engineering tasks, such as program development [76, 64, 110, 99, 13], software understanding [28, 68, 53, 87, 54], fault detection [56, 62, 106, 14, 63, 81], testing [28, 39, 83, 24], and debugging [48, 58, 4, 15, 96, 22, 50]. Figure 1.1 provides an overview of the diverse applications of data mining and machine learning techniques on software engineering tasks.

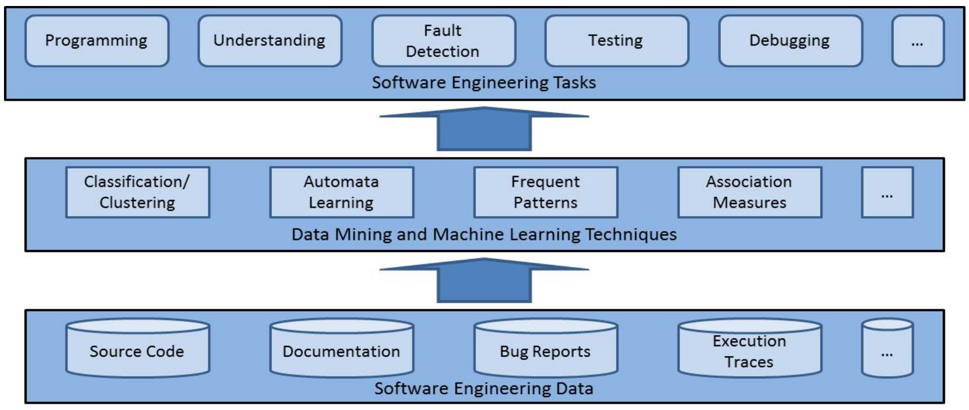


Figure 1.1: Overview of mining software behavior*

*This is partially borrowed from [111].

In this dissertation, we focus on the following two specific research topics: specification mining and statistical debugging.

Specification Mining. Software specifications play a crucial role in many software engineering tasks, e.g., program understanding, fault detection, and verification. However, due to the short time-to-market constraint, changing requirements, and poorly managed product evolution, the lack of precise and complete specifications is a common situation in practice. One approach to addressing this challenge is to automatically infer specifications of a system from its execution traces by a dynamic analysis process referred to as *specification mining* (see *e.g.*, [6]). Recently, various data mining and machine learning techniques have been adopted to discover software behavior as specifications in different formats [65]: automata [6, 87], patterns/rules [56, 105, 101] and value-based invariants [28, 21].

Statistical Debugging. Bugs are prevalent in software systems. As is well known, debugging is a notoriously painstaking and time-consuming task. To reduce developers' burden, researchers have proposed a wide variety of automated debugging approaches. Statistical debugging is one major family of these automatic approaches. The underlying rationale is that program elements which are frequently executed in the failing runs but rarely executed in the passing runs are quite likely to be faulty. These statistical debugging approaches collect failing and passing execution traces and apply statistical techniques to identify discriminative elements as potential bug causes [47, 58, 3] or signatures [42, 15, 96].

In both specification mining and statistical debugging, developers perform statistical techniques (frequent pattern mining or statistical analysis) on the execution traces[†] to discover significant software behavior. Unfortunately, there are a considerable number of insignificant or redundant elements (method call events or data predicates) occurring in these execution traces. As a consequence, mining directly over the raw execution data wastes enormous amount of computing resources due to these useless elements, and thus severely affects the efficiency and scalability of mining. Moreover, owing to the presence of the noise in the datasets, mining may sometimes produce meaningless or even erroneous results. These meaningless results will cause serious decline in effectiveness and practi-

[†]Someone also call them execution profiles. We use "traces" and "profiles" interchangeably in this dissertation.

cability of software behavior mining.

To enhance the performance of software behavior mining, *refinement techniques* are recommended to remove unwanted elements from the raw execution traces. However, currently there is a lack of systematic refinement techniques for both software behavior mining applications. In this dissertation, we investigate the above problem and develop systematic techniques so as to improve the efficiency and effectiveness of software behavior mining.

1.1 Thesis Statement

We give the following thesis statement explored by our research. It shall summarize the key contribution of this dissertation.

Thesis Statement

The efficiency and effectiveness of software behavior mining can be significantly improved by systematically refining the software execution data under analysis.

For each of the above two studies (specification mining and statistical debugging), we develop a specific systematic refinement technique, which guarantees safe removal of unwanted elements from the execution traces and significant reduction in mining overhead.

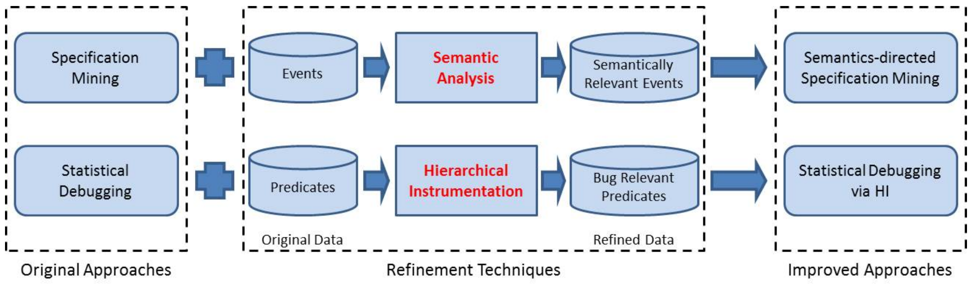


Figure 1.2: Overview of refinement techniques

Specifically, for specification mining, we propose a *semantics-directed specification mining* framework which exploits a user-specified semantic analysis to filter out the semantically irrelevant events from the raw data collected (i.e., the raw execution traces) before mining. Consequently, the mining dataset is effectively refined. The mined specifications are all semantically significant, and mining becomes far more efficient. For statistical debugging, a novel *hierarchical instrumentation* (HI) technique is devised. Based on HI, we effectively prune away the unnecessary program elements (i.e., predicates) for instrumentation. As a result, the execution traces collected and analyzed are significantly refined. The overhead of statistical debugging can thus be greatly reduced.

Figure 1.2 shows the overview of our refinement techniques. We briefly describe semantics-directed specification mining and statistical debugging via hierarchical instrumentation in the following two subsections, respectively.

1.2 Semantics-directed Specification Mining

To tackle the lack of precise and complete specifications, a great number of specification mining approaches have been studied to discover various software behavior as specifications. The majority of these specification mining approaches adopt a statistical technique, and share a common assumption: *significant program properties occur frequently*. They in general discover frequent patterns from the raw execution traces as specifications. Due to the presence of semantically insignificant events in the raw mining data (i.e., raw execution traces), a great number of meaningless specifications are produced. These meaningless specifications seriously undermine the performance of specification mining in terms of both efficiency and effectiveness. The underlying reason is that *statistical significance does not usually correlate to semantic significance.* Many statistically frequent specifications produced are semantically insignificant.

In order to enhance the efficiency and effectiveness of specification mining, we refine the execution traces before mining by introducing semantic information, and thus propose a *semantics-directed specification mining* framework (Chapter 3) to discover semantically significant specifications. The essential idea lies on the assumption that *semantically significant specification should be both semantically relevant and statistically significant.* We propose the respective semantic analysis according to user-specific semantics to extract semantically relevant sequences from raw execution traces, and then perform frequent pattern mining on these sequences to generate semantically significant specifications. Since all semantically irrelevant events are filtered out through the semantic analysis, the mined specifications are all semantically significant, and mining gets more efficient.

We develop a particular semantics-directed specification mining system called *dataflow sensitive specification mining* where dataflow semantics is considered (Section 3.3). The empirical evaluation confirms that insignificant specifications are effectively pruned away and the efficiency of mining is highly improved by our approach. Furthermore, the mined specifications capturing the essential program behavior can practically help program understanding and bug detection.

1.3 Statistical Debugging via Hierarchical Instrumentation

To assist in debugging, researchers have developed a wide variety of statistical bug isolation[‡] and bug signature identification systems. One problem with these statistical debugging systems is that they consider *every program statement* to be potentially relevant to the bug, and thus instrument the entire program to obtain the full-scale execution information before performing bug discovery task. However, most program code works well, and *only small portions of a program are relevant to a given bug* [16]. Such full-scale program instrumentation surely costs dearly due to the unnecessary execution data collection, storage and analysis.

In order to refine the execution data collected and analyzed and thus reduce the overhead of statistical debugging approaches, we propose a novel *hierarchical instrumentation* (HI) technique in Chapter 4. The core of HI is to only select essential program elements for instrumentation so that bug-relevant elements can be identified with much less instrumentation and analysis cost. The HI technique consists of two granularities of instrumentation: coarse-grained (e.g., function-level) and fine-grained (e.g., predicate-level). In brief, we first perform a lightweight coarse-grained instrumentation to acquire the execution information of coarse-grained elements (functions). We safely and effectively prune away fine-grained elements (predicates) for instrumentation according to the obtained coarse-grained information. In other words, we significantly refine the data collected and analyzed while producing the same results as original.

We apply the HI technique to two different types of statistical debugging approaches, namely in-house debugging and cooperative debugging for field fail-

[‡]We use "fault localization" and "bug isolation" interchangeably in this dissertation. Both refer to localizing defects based on execution traces.

ures. Specifically, we first employ the HI technique to predicated bug signature mining (MPS) [96] (which is an in-house debugging approach) and propose an efficient approach via HI, called HIMPS (Section 4.3). Secondly, we investigate the adoption of HI to cooperative bug isolation for field failures (CBI) [58] and propose an iterative approach via HI (Section 4.4). The empirical studies show that our HI technique safely and effectively prunes away unnecessary instrumentation, thus significantly refines the execution traces under analysis. Ultimately, the efficiency of debugging is substantially strengthened.

1.4 Organization

The organization of this dissertation is as follows.

Chapter 2, gives the preliminaries and literature review on specification mining and statistical debugging.

Chapter 3, presents a semantics-directed specification mining framework to efficiently discover semantically significant specifications. We first introduce the framework and then discuss a particular system called, *dataflow sensitive specification mining* which is built based on the framework.

Chapter 4, describes a systematic hierarchical instrumentation technique to improve the efficiency of statistical debugging. We first propose the systematic technique and then introduce two particular applications: an efficient predicated bug signature mining approach via HI and an iterative statistical bug isolation approach for field failures via HI.

Chapters 5, concludes this dissertation by discussing the contributions and future work.

1.5 Papers Appeared

The following lists my papers appeared in this dissertation.

- Zhiqiang Zuo and Siau-Cheng Khoo. **"Mining Dataflow Sensitive Specifications"**. In *Proceedings of the 15th International Conference on Formal Engineering Methods*, (ICFEM'13), pages 36-52, 2013. [122]

- Zhiqiang Zuo. **"Efficient Statistical Debugging via Hierarchical Instrumentation"**. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, (ISSTA'14 Doctoral Symposium), pages 457-460, 2014. [121]

- Zhiqiang Zuo, Siau-Cheng Khoo and Chengnian Sun. **"Efficient Predicated Bug Signature Mining via Hierarchical Instrumentation"**. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, (ISSTA'14), pages 215-224, 2014. [124]

- Zhiqiang Zuo and Siau-Cheng Khoo. **"Iterative Statistical Bug Isolation via Hierarchical Instrumentation"**. In *DSpace at School of Computing, NUS*, (TRC7-14), pages 1-13, 2014. [123]

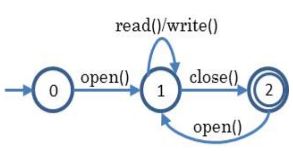
Chapter 2

Literature Review

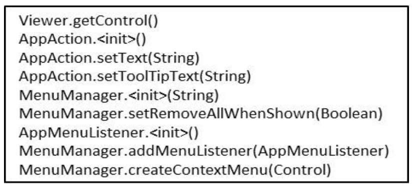
We present the preliminaries of specification mining and statistical debugging, as well as a brief overview of some existing work in the following.

2.1 Specification Mining

As mentioned earlier, specification mining is intended to automatically discover program specifications. In brief, specification mining takes source code or execution traces as input and applies data mining or machine learning techniques to generate specifications in various formats. The work of specification mining can be briefly categorized in terms of the formalism of their mined specifications, as follows: finite state machines, frequent patterns/rules, value-based invariants.

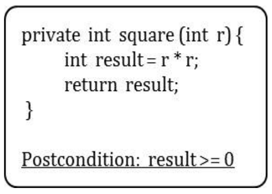


(a) finite state machine – file access



```
Viewer.getControl()
AppAction.<init>()
AppAction.setText(String)
AppAction.setToolTipText(String)
MenuManager.<init>(String)
MenuManager.setRemoveAllWhenShown(Boolean)
AppMenuListener.<init>()
MenuManager.addMenuListener(AppMenuListener)
MenuManager.createContextMenu(Control)
```

(b) frequent pattern – context menu in JFace



```
private int square (int r) {
    int result = r * r;
    return result;
}

Postcondition: result >= 0
```

(c) value-based invariant – square computation

Figure 2.1: Specification examples mined[*]

Each category has its own features and specific applications. Basically, finite state machines and frequent patterns/rules are event-based specifications. They are used to describe program behavior, system protocols and interactions of components. They usually take as input the sequences of events (e.g., method calls)

[*]These examples are taken from [65].

from execution traces or source code and apply automata learning or frequent pattern mining techniques to infer models or patterns. Figure 2.1(a) and Figure 2.1(b) show the example of finite state machines and patterns mined, respectively. On the other hand, value-based invariants describe relationships among various variables characterizing system's states, e.g., the precondition and/or postcondition of a method. They are state-oriented. An example is illustrated in Figure 2.1(c).

Overall, all these different forms of specifications present the functionalities of the program under analysis to some extent. They provide different focuses and therefore offer different usages. In the following, we will discuss several studies of each category in some detail.

Mining Finite State Machines. One of the pioneering works is presented by Cook and Wolf [20]. They explored methods for automatically deriving the formal model of a process from the event sequences collected on the process. They termed this analysis *process discovery*.

Ammons *et al.* [6] first proposed the term *specification mining*, which is a machine learning approach to discovering formal specifications of the protocols which the program must obey when interacting with an API or ADT. Based on the assumption that common behavior is often correct behavior, they collected the execution traces of the program and summarized the frequent interaction patterns as finite state machines. However, the statistical hypothesis could produce statistically frequent but semantically insignificant patterns which undermine the practicability of specification mining. To tackle this problem, Ammons *et al.* annotated the traces with intra-procedural dataflow dependency information, thus to improve the quality of specifications.

The work by Ammons *et al.* is extended by Lo and Khoo [64]. They developed an accurate, robust and scalable specification miner (SMArTIC) to discover the temporal API specifications. Based on the hypothesis: (1) filtering erroneous traces and (2) clustering related traces can improve the quality of specification mining, they introduced the trace filtering and clustering to eliminate the influence of incorrect traces and over-generalization in automata learning.

In [32], Gabel and Su introduced a symbolic algorithm based on binary decision diagrams (BDDs) to automatically mine resource usage patterns ($(ab^*c)^*$).

They later extended their approach to generate complex temporal properties from the execution traces by composing the simple generic patterns [31].

Lorenzoli *et al.* [70] enhanced the finite state models capturing interactions between software components with value-based invariants. They presented *GK-tail* approach to extract extended finite state machines (EFSMs) from interaction traces. Since the extended finite state machines capture constraints of both component interactions and data values, they can include more detailed and more precise information.

Pradel and Gross [87] presented a scalable approach to infer automaton-based specifications of method call sequences associated with multiple related objects from large volume of runtime traces. The essential idea behind the approach is to split the large number of runtime events into small sets of related objects and events, called *object collaborations.* Each object collaboration can be analyzed separately. Hence, the approach can analyze large volume of events in reasonable time. Moreover, since all the events in one object collaboration are semantically relevant due to the shared object, the mined specifications are semantically significant.

Lee *et al.* [54] proposed parametric specification mining. They distinguished different object interactions using parameters. A parametric trace slicing is first adopted to slice the parametric execution traces into a set of independent object interactions corresponding to different parameter bindings. An FSA learner is then employed to infer a finite state machine from a set of separate interactions generated by the trace slicing. The inferred finite state machines annotated with parameters are reported as the final parametric specifications.

Mining Patterns and Rules. PR-Miner is developed by Li and Zhou to automatically extract implicit programming rules from software code [56]. In this work, they adopted frequent itemset mining to efficiently mine programming rules in general forms. These rules can be composed of various program elements such as functions, variables and data types. These mined rules are then used to detect violations. As PR-Miner does not take semantic information into account, many semantically insignificant rules could be generated. These meaningless rules can produce considerable false positives when they are used to detect violations.

In [66], Lo *et al.* presented a novel technique to discover software temporal patterns from execution traces. They termed the particular pattern *iterative pattern*. In essence, iterative pattern is a sequence of events, which must satisfy total ordering and one-to-one correspondence properties. In their work, a depth-first search mining algorithm CLIPER is proposed to mine a closed set of iterative patterns. In subsequent work [67], they mined a non-redundant set of statistically significant temporal rules of arbitrary length from program execution traces. Again, due to the lack of semantic significance, their approach could produce meaningless specifications, which severely affect the efficiency and effectiveness of specification mining.

JADET is developed by Wasylkowski *et al.* to discover frequent usage patterns and detect anomalies via these patterns [105]. It constructs a set of object usage models which is composed of the associated method calls invoked by a specific object or use this object as a parameter. It then produces for each method M under analysis a set of call pairs appearing in an object usage model within M. JADET next discovers the frequent patterns as "normal" usage patterns over the sets of call pairs for all methods using frequent itemset mining [37]. If a pattern is supported by many methods but violated by few methods, it represents an anomaly. In essence, similar to [87], JADET considers the object sharing relation before mining to guarantee semantic significance of specifications.

Thummalapenta and Xie [101] mined exception-handling rules as sequence association rules to capture common exception-handling behavior. They first constructed a static trace database via a code search engine, next adopted a frequent sequence mining tool [104] to mine frequent closed subsequences from the database. Each mined subsequence is then transformed into an association rule. Note that the intra-procedural data-dependency is also analyzed to filter out unrelated calls before mining.

Lo and Maoz [69] integrated the scenario-based specification mining with inference of value-based invariants. This combination produces more expressive specifications, i.e., the scenario-based specifications annotated with value-based invariants. Briefly speaking, they first mined the frequent scenario-based specifications in the form of live sequence charts (LSC) [40]. They then inferred the scenario-specific invariants with respect to the mined LSCs using the scenario-based slicing technique.

Nguyen and Khoo also noticed the problem that many meaningless specifications could be produced by the mining process alone. Different from several work considering the semantic information before mining, they extracted significant specifications from the mined specification candidates through mutation testing [79, 80]. They first performed frequent pattern mining over the execution traces to discover the frequent patterns as specification candidates. Given a specification candidate, they next mutated the associated method body such that it violates the given candidate. The mutated method is run over the specific input to see if an exception is thrown. If the mutation indeed leads to misbehavior, the specification candidate is regarded as significant.

Mining Value-based Invariants. One pioneering work of discovering value-based invariants is Daikon [27, 28, 30] which infers likely program invariants from program execution traces. An invariant is a property which holds at a program point or points. These properties can be useful for many software tasks, from design to maintenance. Specifically, Daikon firstly runs the instrumented program over a set of test cases to obtain execution traces containing variable values of interest. It checks a set of possible invariants against the observed values in the trace. Once an invariant encounters a contradictory sample during checking, it will be discarded. Eventually, Daikon reports those invariants that are tested to a sufficient degree without falsification. Csallner *et al.* [21] applied dynamic symbolic execution for invariant inference. Different from Daikon that falsifies pre-set invariants by checking the observed values, they directly inferred the likely program invariants by dynamic symbolic execution.

These value-based invariants can be used for many applications, e.g., program understanding, verification, anomaly detection. They can also combined with the automaton- and rules-based specifications discussed above to further enhance the expression power of specifications, just like what have done by [70, 69].

All in all, mining specifications without considering semantic information could produce too many program properties, many of which are found to be semantically insignificant. The presence of these insignificant specifications severely affects the efficiency and effectiveness of specification mining. Several studies [62, 56, 66, 32, 31] mentioned above that do not consider semantic information have shown their drawbacks. On the other hand, some researchers [105, 101, 100, 87] have attempted to address this semantics-deficiency issue. However, none of these remedies proposed really incorporate in-depth semantic information into specification mining by refining the data under analysis beforehand. In Chapter 3, we develop a *semantics-directed specification mining* framework to refine the execution traces before mining by considering in-depth semantic information, finally to efficiently discover semantically significant specifications.

2.2 Statistical Debugging

Statistical debugging approaches in essence collect two groups of execution profiles[†] and apply statistical techniques to pinpoint discriminative elements as the potential bug cause or bug signature.

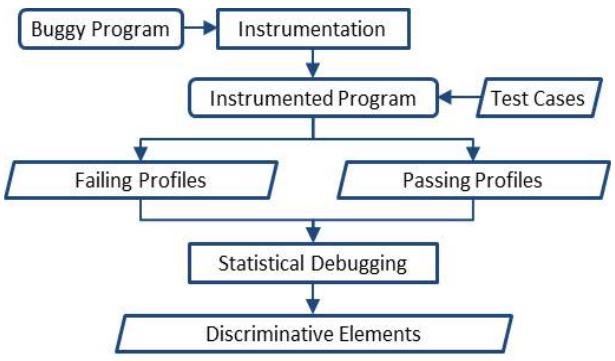


Figure 2.2: Workflow of statistical debugging

The procedure is as follows. Given a buggy program, we instrument it and then run the instrumented program over test cases. Thus each run will generate an execution profile recording the runtime execution information. Various types of coverage elements have been adopted to capture the execution information, e.g., the statements [47, 3], basic blocks [15] or functions [22] executed, def-use pairs [94], data predicates [58, 60] tracked during execution. The profile associated with the run behaving as expected (say, producing the correct output as oracle) is called *passing*, the profile associated with misbehavior is called *failing*. The rationale of statistical debugging is that program elements which appear frequently in failing executions but rarely in passing ones are highly correlated

[†]We use execution traces and profiles interchangeably in this dissertation.

to the failure and potentially faulty. Therefore, various statistical discriminative measures are utilized to quantify this correlation. We call these measures as *suspiciousness measures.* Different measures have been investigated [71], all based on the intuition that: if an element is highly discriminative from the failing and passing executions, then it is highly suspicious. Figure 2.2 shows the workflow of statistical debugging.

Statistical Bug Isolation. Statistical bug isolation is a family of approaches which isolate a single suspicious element as the root cause of failure by identifying the discriminative behavior between passing and failing execution profiles. As mentioned above, there are two essential issues to consider: type of coverage elements capturing program execution behavior and discriminative measure utilized to quantify the suspiciousness of elements.

Tarantula [48, 47] and Ochiai [4, 3] both use statement coverage information to represent the execution behavior and assess the suspiciousness of each statement based on their proposed suspiciousness measures. Instead of considering statement coverage information, Liblit *et al.* [58] collected runtime values of predicates and introduce *Importance* metric to measure each predicate.

Nainar *et al.* [9] introduced *complex predicates* composed of atomic predicates from [58] using logical operators (such as conjunction and disjunction). They demonstrated that complex boolean predicates are informative and useful. Gore *et al.* [33] also extended [58] by introducing the notion of *elastic* predicates such that statistical bug isolation can be tailored to a specific class of software involving floating-point computations and continuous stochastic distributions. Different from the uniform and static predicates capturing the negative and positive nature of a variable in [58], elastic predicates are constructed to capture the relations of a variable with its mean and standard deviation computed beforehand from all executions. Note that runtime predicates provide finer-grained execution information than statement coverage, therefore more precise results can be achieved. However, it suffers heavier instrumentation and higher analysis cost than statement coverage.

Furthermore, several researchers have noticed that the suspiciousness measures employed in the statistical debugging approaches could be susceptible to confounding biases [77, 86] so that the effectiveness of debugging is weakened.

To this end, causal inference has been recently applied to reduce the control and data flow dependence confounding biases in statement-level [11, 12] and also failure flow confounding bias in predicate-level [34] statistical bug isolation.

Bug Signature Identification. As is well known, debugging is an integral process of localizing the bug, understanding and then fixing it. Most of automated debugging approaches proposed focus only on the first bug localization phase, as we have reviewed many such works earlier. These studies commonly try to isolate the root cause of the bug, which is usually a single buggy statement. However, in practice, it is difficult to understand the bug by examining that single statement in isolation. The *perfect bug understanding* [85] does not hold. To better support debugging, more information than sole buggy statement or root cause is required. The "context" where the bug occurs is likely to provide more useful clue for identifying, understanding and correcting bugs. Hsu et.al [42] coined the term *bug signature* which comprises multiple elements providing bug context information. They adopted sequence mining algorithm to discover longest sequences in a set of failing executions as bug signatures. Cheng et.al [15] identified bug signatures using discriminative graph mining. They mined the discriminative control flow graph patterns from both passing and failing executions as bug signatures. Since only control flow transitions are considered in [15], bugs not causing any deviation in control flow transitions can not be identified. To enhance the predictive power of bug signatures, Sun and Khoo [96] proposed *predicated bug signature mining*, where both data predicates and control flow information are utilized. They devised the discriminative itemset generator mining technique to discover succinct predicated bug signatures.

In addition, Jiang and Su [44] proposed context-aware statistical debugging. Different from directly mining signatures, their approach first identifies suspicious bug predictors by feature selection, then groups correlated predictors by clustering and finally builds faulty control flow paths linking predictors to provide contextual information for debugging.

The problem with these statistical debugging approaches (statistical bug isolation [48, 47, 58, 4, 3, 33] and bug signature mining [15, 96]) is that they consider *every program statement* to be potentially relevant to the bug, and thus instrument the entire program to obtain the full-scale execution information for debugging. As a matter of fact, most program code works well. Such full-scale program instrumentation costs dearly in terms of wastage of execution time, storage space, CPU and memory usage due to the unnecessary execution data collection, storage and analysis. In this dissertation, we devise a hierarchical instrumentation technique to conduct selective instrumentation such that only the necessary program elements which are highly correlated to the failure are instrumented, ignoring irrelevant ones. The statistical debugging approaches can thus become more efficient. We will present it in detail in Chapter 4.

Cooperative Bug Isolation for Field Failures. In general, developers run the instrumented program in the laboratory to collect the execution profiles and then analyze them for debugging. We call these approaches *in-house debugging*, which is inherently suitable for debugging the failures occurring before software deployment. However, most software deployed remains buggy in spite of extensive in-house testing and debugging. These failures that occur after deployment on user machines are called *field failures* [18, 46]. To debug these field failures, developers are required to reproduce them in the laboratory according to the bug reports and then perform the same process shown by Figure 2.2 as in-house debugging. However in practice, it is hard for developers to reproduce field failures (especially client-side failures) in the laboratory due to the different environments, configurations, and/or nondeterminism. In [59, 58], Liblit *et al.* proposed a different execution data collection scheme for debugging field failures. They directly gathered the execution profiles from end-users. Specifically, they deployed the instrumented program to users and collected user's execution profiles for debugging with user's approval. They term this approach *cooperative bug isolation* [59]. In order to encourage active users' participation, the overhead for running the instrumented programs should be kept sufficiently low. To this end, sparse random sampling [57, 8] is adopted. Specifically, only a sparse and random subset of predicates are sampled to record their execution information while the instrumented program is running. Thus each user's overhead for running the instrumented program is sharply reduced. From the perspective of developers, the execution information of different predicates can be collected from different users. Based on the large user base, the complete execution information of all the predicates can be obtained so as to perform the statistical debugging.

This sampling technique amortizes the monitoring cost to a considerable number of end-users so that each user suffers a relatively low time overhead. Nevertheless, from the perspective of developers, the total monitoring overhead and the total size of execution data collected and analyzed remain unchanged. In this sense, the cooperative approach still suffers the same problem as mentioned earlier that: plenty of unnecessary execution data is collected and it consumes many resources such as network bandwidth, storage space, CPU time etc. due to the need for data transfer, storage and analysis.

As an extension of cooperative bug isolation, *iterative (cooperative) bug isolation* approaches [16, 10] have been proposed to ensure minimal effort spent by both end-users and developers. These approaches perform the instrumentation and statistical analysis in an iterative manner. Instead of the entire program, only partial code is instrumented and analyzed at each iteration. In these approaches, developers checked the bug predictors reported during each iteration and adjusted the instrumentation plan for the next iteration. The iterative debugging process proceeds until the bug is found or the entire program is explored. Specifically, Chilimbi *et al.* [16] monitored a set of functions, branches and paths to analyze whether these are strong predictors of the failure at each iteration. If so, they terminated the iterative process by returning these strong predictors. Otherwise, they expanded the search via a static analysis to monitor other parts of code closely interacting with the weak predictors. Similarly, Arumuga *et al.* [10] proposed an adaptive monitoring strategy based on the following principle of locality: *if a predicate is highly predictive of failure, then predicates in its vicinity are potentially good bug predictors as well.* To this end, the strategy monitors a few predicates at each iteration and adaptively adjusts the instrumentation plan to include predicates close to the highly suspicious predicate currently explored.

However, there are two main drawbacks of iterative approaches. Firstly, both iterative approaches [16, 10] make use of the principle of locality to guide their search for bugs; this principle however is not always effective in localizing bugs, as experiments have found [10]. Secondly, both iterative approaches require developers to check the predictors reported at each iteration, until the bug cause is found. As claimed in [85], developers are reluctant to go through a list of predictors, not to mention the need to repetitively perform this check at every iteration. In this dissertation, we will tackle the above drawbacks for cooperative bug isolation and propose an iterative statistical bug isolation approach via hierarchical instrumentation in Section 4.4.

Other Automated Debugging Approaches. Apart from the statistical approaches mentioned above, there also exist other automated debugging approaches. Program slicing [108, 72] is one commonly used technique for debugging. Recently, to improve the effectiveness, dynamic slicing [119] is adopted for debugging. Zeller *et al.* proposed *delta debugging* to isolate the failure-inducing difference in source code [114], inputs [116], and program states [115, 19] between one failing and one passing run. Gupta *et al.* [36] integrated dynamic slicing with delta debugging to narrow down the search for faulty code, while introducing the concept of *failure-inducing chops.* Similar to delta debugging for program states, Zhang *et al.* [117] forcibly switched the branch predicate's outcome in a failing run and localize the bug by examining the predicate whose switching produces correct result. In [93, 92], Renieris and Reiss selected from a large number of passing runs one passing run which most resembles the failing run using program spectra, and differentiated the program spectra of these two runs to help isolate the cause of the bug.

Chapter 3

Semantics-directed Specification Mining

Specification mining has become an attractive tool for assisting in numerous software development and maintenance tasks. The majority of these approaches directly mine frequent behavior as specifications from the raw execution traces. Unfortunately, directly mining frequent behavior from the raw execution traces containing semantically insignificant events produces too many frequent program properties, many of which are found to be semantically insignificant. The presence of these insignificant specifications severely undermines the efficiency and effectiveness of specification mining.

In this chapter, we develop a *semantics-directed specification mining* framework to refine the execution traces before mining by considering in-depth semantic information, finally to efficiently discover semantically significant specifications. On the basis of this framework, we take into account dataflow semantics and thus propose a particular specification mining system called *dataflow sensitive specification mining.* It is validated by the experiments that our approach effectively refines execution traces and efficiently produces semantically significant specifications.

In the following, Section 3.1 first gives the motivation of our work. The framework is then presented in Section 3.2, followed by an application in Section 3.3. We discuss the related work in Section 3.4. Section 3.5 finally summarizes this chapter.

3.1 Motivation

Program comprehension has been found to be a crucial and time-consuming component of software maintenance task. While the comprehension task can be made easier by the presence of program specifications, the short time-to-market constraints, changing requirements, and poorly managed product evolution reduce the availability of such specifications, causing them to be incomplete, incorrect and obsoletely documented. One approach to addressing this challenge, which has been gaining much recognition by the software development community, is to automatically infer specifications of a system from its execution traces by a dynamic analysis process referred to as *specification mining* (see *e.g.*, [6, 66]).

The majority of these specification mining approaches adopt a statistical approach, and share a common assumption: *significant program properties occur frequently*. Unfortunately, statistical inference of program properties remains unsatisfactory. A prevalent obstacle to these specification mining approaches is that a great number of meaningless specifications could be produced by directly mining over the raw traces. It is painful and laborious to separate them from those semantically meaningful specifications. Consequently, the presence of these meaningless specifications will seriously weaken the quality of inferred specifications, and diminish their value in use. Moreover, generating these meaningless patterns can consume enormous amount of mining time.

The underlying reason for these shortcomings is that *statistical significance does not usually correlate to semantic significance*. The events in the raw execution traces are not all semantically relevant. Some (in fact, many) semantically insignificant program specifications may be statistically significant, and get generated when we directly perform the mining over the raw execution traces. As a case study, we investigate the behavioral change resulted from a bug fix from Compress revision 922299 to 922309. We attempt to understand the bug fix from the significant patterns which discriminate the two versions. By mining over the raw traces via a traditional miner such as Iterative Pattern Miner [66], we obtain 63 discriminative patterns, many of which provide little value to our understanding of the change. On the other hand, by refining the traces such that events (i.e., method calls) in the patterns are dataflow related, we obtain just 4 discriminative patterns, all of which display the expected behavioral change of the revisions. We will elaborate on this case study in Section 3.3.5.3.

Several studies [62, 56, 66, 32, 31] not taking into account semantic information have shown their drawbacks. On the other hand, some researchers [105, 101, 100, 87] have attempted to address this semantics-deficiency issue. However, none of these remedies proposed really incorporate in-depth semantic information (e.g., dataflow semantics) into specification mining by refining the data under analysis beforehand.

3.2 Framework

To address the lack of semantic significance and further improve the efficiency of specification mining, we introduce semantic information, and perform the respective semantic analysis to refine the mining input. We propose a *semantics-directed specification mining* framework to discover semantically significant specifications from execution traces.

The essential idea lies on the following assumption.

Assumption

Semantically significant specification should be both semantically relevant and statistically significant.

Specifically, we first refine the mining input by extracting semantically relevant sequences from execution traces according to the user-specific semantics. We next employ frequent pattern mining algorithm on these sequences to generate semantically significant specifications. Figure 3.1 shows the workflow of our semantics-directed specification mining framework.

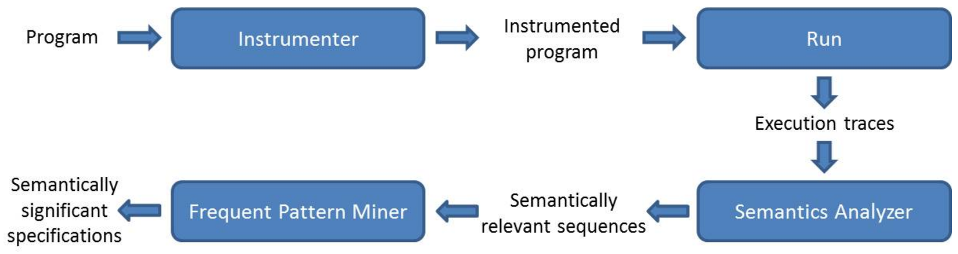


Figure 3.1: Framework of semantics-directed specification mining

Through the semantics analyzer (at the lower right corner of Figure 3.1), we only extract semantically relevant sequences satisfying the user-specific semantics. All the semantically irrelevant events are filtered out in advance before mining. Therefore, the mined specifications are all semantically significant. Moreover, the search space of frequent pattern mining is drastically reduced; hence mining becomes more efficient.

3.3 Mining Dataflow Sensitive Specifications

We demonstrate the application of our semantics-directed specification mining framework by considering a particular *dataflow semantics* and develop *dataflow sensitive specification mining*. Our experimental results validate that our approach can significantly refine the execution traces by filtering out dataflow irrelevant events and thus greatly improve the efficiency and effectiveness of mining. In addition, we also show that our mined specifications reflect the essential program behavior and can practically help program understanding and bug detection.

3.3.1 Introduction

Program specifications play a crucial role in program comprehension. In [66], Lo *et al.* proposed an automatic mining approach to discover specifications in a particular formalism termed *iterative pattern* from execution traces. These iterative patterns shed light on program behaviors. They can be interpreted as interaction diagrams between classes/objects, and be used to construct high-level scenario-based models such as live sequence chart (LSC) [23]. As usual, the mined specifications directly over the raw traces by [66] include numerous semantically insignificant ones, which severely undermines the practicability of this specification mining approach.

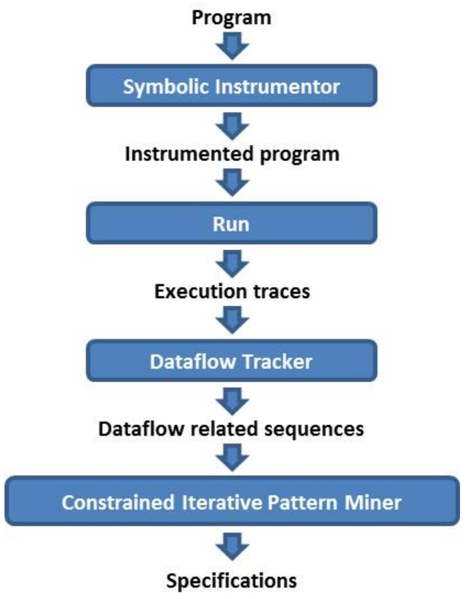


Figure 3.2: Workflow of dataflow sensitive specification mining

Based on our semantics-directed specification mining framework, we develop a particular system called *dataflow sensitive specification mining*. We investigate the introduction of *dataflow semantics* to extract from execution traces dataflow related sequences as the mining input, and demonstrate that mining specifications from these dataflow related sequences reduces a great number of meaningless specifications, resulting in a collection of specifications which are both semantically relevant and statistically significant.

Specifically, a particular *dataflow semantics* over the runtime events is taken into consideration. We propose a dynamic, inter-procedural dataflow tracking analysis, which analyzes execution traces to extract the dataflow related sequences. Lastly, we perform a novel constrained iterative pattern mining over these sequences to discover semantically significant iterative patterns as specifications. The workflow of our system is shown as Figure 3.2. We will discuss each component of this system in the following.

3.3.2 Symbolic Instrumentation

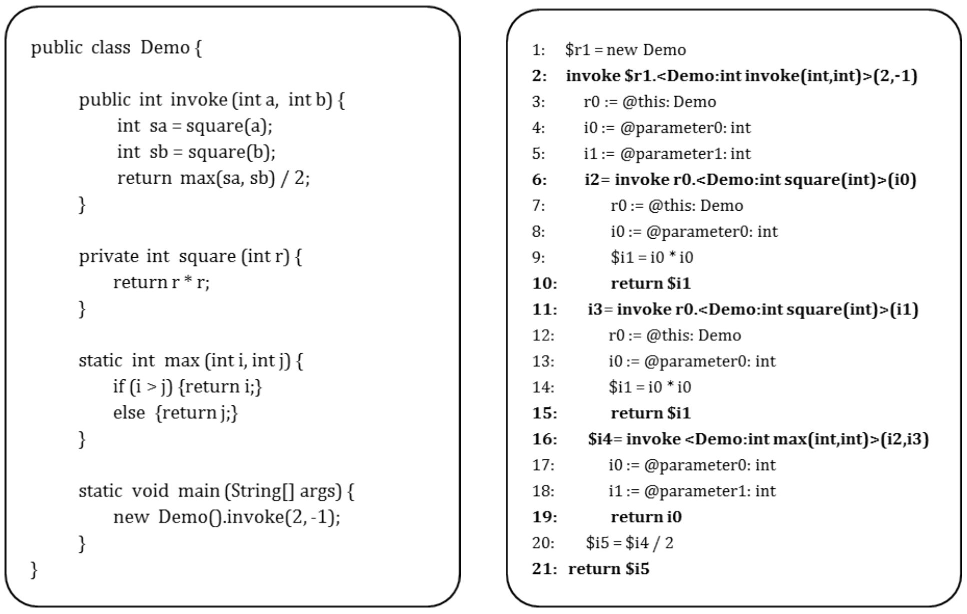
The symbolic instrumentor conducts static instrumentation on the programs under analysis. The footprint produced by the instrumentor during execution is a sequence of *symbolic statements*, which enables tracking and reasoning of dataflow relations. Our trace is in Jimple format (refer to [91] for the formal Jimple grammar), which is a 3-address intermediate representation of Java in Soot framework[*].

In this work, the events used to form iterative patterns are method calls and method returns. In order to capture precise and inter-procedural dataflow relations among events, we instrument five kinds of statements, namely IdentityStmt (e.g., statements 3, 4, 5), AssignStmt (9, 20), InvokeStmt (2, 6), ReturnStmt (10) and ThrowStmt, as they either contain dataflow information or represent the desired events. They are necessary to be instrumented and included in the execution traces. Figures 3.3(a) and 3.3(b) show a code example and its trace fragment, respectively.

3.3.3 Dataflow Tracking Analysis

Having the execution traces, we refine them via a semantic analysis. In this section, we introduce this semantic analysis, called *dataflow tracking analysis*

[*]http://www.sable.mcgill.ca/soot/



```
public class Demo {

        public int invoke (int a,  int b) {
              int sa = square(a);
              int sb = square(b);
              return max(sa, sb) / 2;
        }

        private int square (int r) {
              return r * r;
        }

        static int max (int i, int j) {
              if (i > j) {return i;}
              else {return j;}
        }

        static void main (String[] args) {
              new Demo().invoke(2, -1);
        }
}
```

```
1:   $r1 = new Demo
2:   invoke $r1.<Demo:int invoke(int,int)>(2,-1)
3:      r0 := @this: Demo
4:      i0 := @parameter0: int
5:      i1 := @parameter1: int
6:      i2= invoke r0.<Demo:int square(int)>(i0)
7:         r0 := @this: Demo
8:         i0 := @parameter0: int
9:         $i1 = i0 * i0
10:        return $i1
11:     i3= invoke r0.<Demo:int square(int)>(i1)
12:        r0 := @this: Demo
13:        i0 := @parameter0: int
14:        $i1 = i0 * i0
15:        return $i1
16:     $i4= invoke <Demo:int max(int,int)>(i2,i3)
17:        i0 := @parameter0: int
18:        i1 := @parameter1: int
19:        return i0
20:     $i5 = $i4 / 2
21:  return $i5
```

(a) Code (b) Trace

Figure 3.3: A running example

which takes as arguments the execution traces and produces a set of dataflow related sequences (i.e, the refined mining input).

3.3.3.1 Concepts

Consider the data dependence graph (Figure 3.4) of the example trace (Figure 3.3(b)). As defined in [41], a statement s_1 is *data dependent* on a statement s_2 iff[†] there is a variable x and a control flow path h from s_2 to s_1 such that x is defined at s_2, used at s_1, and not redefined along any subpath of h. In a data dependence graph, if s_1 is data dependent on s_2, then there is an edge (solid arrow line in Fig. 3.4) from s_2 to s_1. A *dataflow path* is a sequence of statements such that from each statement there is a data dependency edge to the next statement in the sequence. For instance, the graph (Figure 3.4) contains four dataflow paths. One of them is the sequence of statements $\langle 4, 8, 9, 17, 20 \rangle$

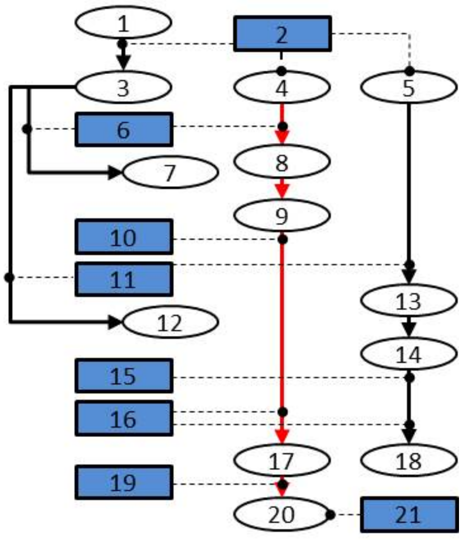


Figure 3.4: Dynamic data dependence graph

[†]iff means if and only if.

whose data dependency edges are marked as bold (and red).

While dataflow paths are defined over symbolic statements in the execution trace, our expected final results are sequences of events; i.e., method calls and method returns, shown as rectangles (with blue background) in the above graph. As an example, InvokeStmt 2 and ReturnStmt 21 represent a call event and a return event, respectively. We call the set of all events associated with a trace T an *event set*, and denote it by $E_{(T)}$. Considering the trace T in Figure 3.3(b), the event set $E_{(T)}$ contains those events which are highlighted in bold in Figure 3.3(b).

An event can have some arguments. For a call event, its arguments include the callee, its actual parameters, and all the class variables and instance variables used during method execution. Similarly, the arguments of a return event constitute the value returned by ReturnStmt, as well as all the defined or redefined class variables and instance variables during this method execution. The argument set of an event e is denoted by $A(e)$.

Definition 1 (Dataflow Association). *An event e is dataflow associated with a dataflow path H iff there exists a statement s in H such that at least one argument of e is defined or used at s.*

In Figure 3.4, the dashed line shows this association. For example, the event $\langle Demo : int\ invoke\ (int, int)\rangle\_2$[‡] in $E_{(T)}$ is dataflow associated with $H\ \langle 4, 8, 9, 17, 20\rangle$, since its first argument is used in statement 4.

Definition 2 ((Maximum) Dataflow Related Sequence). *Given a trace T, a dataflow path H of T, and the event set $E_{(T)}$, a sequence of events Q ($\langle e_1, e_2, \ldots, e_n\rangle$) is a dataflow related sequence with respect to H iff $\forall e_k (k \in [1, n]), e_k \in E_{(T)}$ and $\forall e_i, e_j (i, j \in [1, n] \wedge i < j)$, the following holds:*

- *Temporal relation: e_i appears before e_j in T;*
- *Dataflow relation: e_i and e_j are both dataflow associated with H.*

Furthermore, Q with respect to H is a maximum dataflow related sequence *iff there exists no dataflow related sequence Q' of H such that Q' is a super-sequence of Q.*

From the above example, we can derive that the event sequence $\langle Demo :$

[‡]It indicates the call event $\langle Demo : int\ invoke(int, int)\rangle$ represented by statement 2.

int invoke $(int, int)\rangle$_2, $\langle$*Demo* : *int square* $(int)\rangle$_6, *R* :$\langle$*Demo* : *int square* $(int)\rangle$_10[§], $\langle$*Demo* : *int max* $(int, int)\rangle$_16, *R* :$\langle$*Demo* : *int max* $(int, int)\rangle$_19, *R* :$\langle$*Demo* : *int invoke* $(int, int)\rangle$_21 is a maximum dataflow related sequence with respect to *H* $\langle 4, 8, 9, 17, 20\rangle$.

3.3.3.2 Approach

The objective of the dataflow tracking analysis is to obtain all maximum dataflow related sequences by analyzing the symbolic traces. A naive way is to first construct the dynamic data dependence graph as an intermediate data and then traverse all the paths in the graph to get all dataflow related sequences. However, such an intermediate data is usually quite big. Its generation can severely affect the scalability of our analysis. Instead, our tracking analysis eliminates the generation of such intermediate data by directly outputting the dataflow related sequences while tracking each dataflow path. Briefly, we keep track of each dataflow path through analyzing the *use-def pairs*[¶] statement by statement in chronological order. At the same time, we maintain one specific event list for each dataflow path. During the dataflow tracking, upon encountering an event which is associated with the currently tracked dataflow path, we append it to the end of the corresponding event list. When this dataflow path is completely tracked, the event list we maintain constitutes the desired maximum dataflow related sequence.

Our dataflow tracking analysis is called *stack-based scoped* because it dynamically maintains an "analysis stack" of "scopes" to help emulate the actual runtime execution. A scope mimics an activation record, which is pushed onto the stack during method invocation and popped out at call return. It contains *triples* representing dataflow paths which are currently tracked by the analysis. Specifically, a triple is denoted (v_s, L, v_c) , where L is an event list forming a dataflow related sequence spanning across procedures with respect to the dataflow path currently tracked, v_s refers to the start variable of this dataflow path, v_c refers to the currently arrived variable. For efficiency, each scope is represented by a hash set consisting of triples, with hash keys constructed from the currently arrived

[§]"R" denotes the corresponding return event.

[¶]A *use-def* pair (u, d), associates a statement in a program where variable u is used in defining variable d. For example, given the AssignStmt 14 in Figure 3.3(b), there is a *use-def* pair associated with this statement: $(i0, \$i1)$.

variable of the triple.

Algorithm 1: Scoped Dataflow Tracking Analysis

```
   Input: trace $T$
   Output: all the maximum dataflow related sequences
 1 foreach statement $s$ in chronological order in trace $T$ do
 2     switch $s$ do
 3         case InvokeStmt(s)
 4             $S \leftarrow \emptyset$;
 5             push($S, Stack$);
 6             break;
 7         case ReturnStmt(s)
 8             $S_u \leftarrow$ pop($Stack$);
 9             $S_d \leftarrow$ peek($Stack$);
10             KillAndGen($S_u, S_d, s$);        // the top element $S_d$ of $Stack$ is updated
11             for each $t(v_s, L, v_c) \in S_u$ do
12                 if isComplete($t$) then output $L$;
13             end
14             break;
15         case IdentityStmt(s)
16             $S_u \leftarrow$ peek2nd($Stack$);      // peek at the second top element of $Stack$
17             $S_d \leftarrow$ peek($Stack$);
18             KillAndGen($S_u, S_d, s$);
19             break;
20         case AssignStmt(s)
21             $S_u \leftarrow$ collapse($Stack$);
22             $S_d \leftarrow$ peek($Stack$);
23             KillAndGen($S_u, S_d, s$);
24         end
25     endsw
26 end
```

As shown in Algorithm 1, the analyzer handles the trace statements in chronological order. When encountering an InvokeStmt, the analyzer pushes a new scope S into $Stack$ (lines 4-5). On the other hand, ReturnStmt indicates the end of the existing scope, and the analyzer pours out all the dataflow related sequences associated with the complete dataflow paths (lines 11-13). $isComplete(t)$ determines if t represents a complete dataflow path (i.e., completely tracked dataflow path). For each kind of statements encountered, the analyzer handles each variable (used variable and defined variable) active at the statement in its legitimate scope. Specifically, for ReturnStmt, the legitimate scope S_u for used variables is the scope popped from Stack. The legitimate scope S_d for defined variable is the top scope of $Stack$ (lines 8-10). For IdentityStmt, S_u is the second top element of $Stack$ (line 16). The top element of $Stack$ corresponds to S_d (line 17). Similarly, as for AssignStmt, S_d is the currently top element of $Stack$ (line 22). S_u is likely to be any scope in the $Stack$ (due to the liveness of instance variables or class variables). The analyzer searches $Stack$ from top towards bottom to obtain the desired triples; this is performed by the function `collapse`(*Stack*) (Line 21).

Algorithm 2: KillAndGen(S_u, S_d, s)

```
   Input: statement $s$, scope for used variable $S_u$, scope for defined variable $S_d$
   Update: scope $S_d$ which is the currently top element of $Stack$
1  $Pairs \leftarrow$ get_UD_Pairs($s$);          // get the use-def pairs associated with $s$
2  foreach use-def pair $p(v_u, v_d) \in Pairs$ do
3      $GS \leftarrow \emptyset$;
4      if $v_u$ is a constant or a new instance then
5          $L' \leftarrow [\ ]$;
           // $A(e)$ is the set containing all the arguments of event $e$
6          if $\exists$ event $e$ associated with $s$, $v_u \in A(e)$ then
7              $L' \leftarrow L' +\!\!+ [e]$;
8          end
9          $GS \leftarrow GS \cup \{(v_u, L', v_d)\}$;            // generate the new triple
10     else
11         foreach $t'(v_s, L, v_u) \in S_u$ do   // the currently arrived variable of $t'$ is $v_u$
12             mark $t'$ as incomplete;
13             $L' \leftarrow L$;
14             if $\exists$ event $e$ associated with $s$, $v_u \in A(e)$ then
15                 $L' \leftarrow L' +\!\!+ [e]$;
16             end
               // extend the tracked dataflow path and generate the new triple
17             $GS \leftarrow GS \cup \{(v_s, L', v_d)\}$;
18         end
19     end
20     foreach $t(v_*, L_*, v_d) \in S_d$ do      // the currently arrived variable of $t$ is $v_d$
21         if isComplete($t$) then output $L_*$;
22         $S_d \leftarrow S_d - \{t(v_*, L_*, v_d)\}$;             // kill the old triple
23     end
24     $S_d \leftarrow S_d \cup GS$;
25 end
```

Given a statement s and the corresponding legitimate scopes S_u and S_d, the analyzer uses the typical "kill-and-gen" dataflow analysis mechanism to update the stack by modifying the scope S_d at its top. More specifically, for each use-def pair (v_u, v_d) contained in statement s (where v_d is defined in s), the algorithm tries to extend the relevant dataflow paths in S_d (or create new dataflow paths when necessary) by generating new triples or to remove some paths the ending variable of which is redefined by killing the corresponding triples. Going through the algorithm, Lines 4-9 handle the case where a new dataflow path is created due to a constant or new instance; Lines 11-18 extend the current dataflow paths through composing def-use chains. Lines 20-23 remove the triples whose currently arrived variable is redefined, and simultaneously output all the dataflow related sequences if t represents a complete dataflow path. Lastly, all generated triples are added into S_d (Line 24).

3.3.3.3 Challenges

To make our analysis more precise, we need to consider two important challenges namely object aliasing and scope mismatch.

Object Aliasing. As the execution traces contain symbolic variables, it is possible that more than one reference variables refer to or point to the same object in the program. Such situation is called object aliasing. Modifying the object through one reference variable implicitly affects the state of the object associated with other aliased variables. In turn, the manipulations of the same memory data can be operated through different variables. In order to keep track of the dataflow paths precisely, we have to determine whether two different variables in the trace refer to identical data. If two reference variables are alias of each other, we will treat them as the same one. Any field access on the object through different aliased variables should be always considered as the same field access.

To address this aliasing problem, we perform *dynamic alias tracking analysis*. Briefly, for each instantiated instance object, we maintain a unique reference variable which represents the absolute memory location of the object. All the aliased variables of the object will be mapped to this unique reference variable. All these mapping information is maintained in the environment of the analysis. When the statements containing field access occur in the trace, we first track the unique reference variable of the object, and then search or update the instance field of the object.

Scope Mismatch. In our analysis, we take advantage of InvokeStmt and ReturnStmt to identify the start and termination of each scope, respectively. However, due to the presence of uninstrumented code and exceptions, certain scopes are not explicitly represented by InvokeStmt and ReturnStmt. Failing to identify certain scopes leads to scope mismatch. Specifically, it is likely that a method call in uninstrumented code executes the instrumented method body. Therefore, no InvokeStmt representing the start of new scope appears in the trace. Besides, when an exception is thrown, the execution stack can be popped back arbitrary levels. Many scopes are terminated. Unfortunately, there is no explicit statement in the trace to denote it. If we do not terminate scopes appropriately, we may assign the subsequent statements into incorrect scopes.

To avoid scope mismatch, we perform *scope matching analysis* which guarantees we can start and terminate all the scopes correctly. In our analysis, we use an artificial statement DeclarationStmt to assist in identifying the start of a scope. Specifically, we instrument one DeclarationStmt at the beginning of each method body. When a method is executed, the corresponding DeclarationStmt will be printed out in the trace. Therefore, even though the InvokeStmt is missed, we can identify the start of method scope via the DeclarationStmt. To avoid the exception-caused mismatch, we determine the termination of scopes by checking method signatures. Specifically, we check each statement under analysis to see whether its method signature corresponds to the current scope's signature. If so, the scope is matching. Otherwise, we terminate the current scope, pop it and check again until scope matching succeeds.

Eventually, we integrate the above *dynamic alias tracking analysis* and *scope matching analysis* with the simplified analysis described in Algorithms 1 and 2. The pseudo code of complete scoped dataflow tracking analysis is shown in Appendix A.

3.3.4 Constrained Iterative Pattern Mining

We present a constrained iterative pattern mining algorithm which mines frequent patterns from the generated dataflow related sequences as final specifications in the following.

3.3.4.1 Background

The concept of *iterative pattern* is first introduced by Lo *et al.* [66] to capture program behaviors involving repeated event occurrences (possibly caused by loop iterations). It forms the basis for temporal rules, which have been used to formulate specifications such as Live Sequence Chart (LSC) [23]. An iterative pattern is a sequence of events which must satisfy total-ordering [43] and one-to-one correspondence [52] properties. It can be identified by a set of instances, and one sequence can contain multiple instances. The following definition proposed in [66] expresses an iterative pattern instance in the form of Quantified Regular Expression [82] with ';' as concatenation operator, '[-]' as exclusion operator ($[-p, q]$ means any event except p and q) and '*' as Kleene closure.

Definition 3 (Iterative Pattern Instance – QRE). *Given a pattern p^n ($\langle e_1 e_2 \ldots e_n \rangle$), a substring ($\langle f_1 f_2 \ldots f_m \rangle$) of a temporal sequence t ($\langle t_1 t_2 \ldots t_{end} \rangle$) in a sequence database SeqDB is an instance of p^n iff it can be expressed by the following QRE expression:*

$$e_1; [-e_1, \ldots, e_n]*; e_2; \ldots; [-e_1, \ldots, e_n]*; e_n.$$

Consider the following sequence representing a trace $\langle a,b,a,c,b,a,c,b \rangle$. The only three instances found in it for the iterative pattern $\langle a, b \rangle$ are substrings: $\langle a, b \rangle$ at index 1, $\langle a, c, b \rangle$ at index 3, and $\langle a, c, b \rangle$ at index 6. Note that $\langle a, b, a, c, b \rangle$ is not an iterative pattern instance, according to Definition 3.

The *support* of an iterative pattern P with respect to a sequence database $SeqDB$ is the number of instances of pattern P in $SeqDB$. A pattern P is *frequent* if its support $sup(P)$ exceeds a specified threshold min_sup.

To mine all the frequent patterns efficiently, instead of checking for all possible patterns in the search space, we actively prune those sub-spaces containing infrequent (aka., insignificant) patterns using *Apriori* property. This property was first presented by Agrawal *et al.* [5] as follows: *if a pattern is frequent, so do its sub-patterns.* As a simple example, consider the sequence lattice with *subsequence relation* as the partial order in Figure 3.5. By the contrapositive of the Apriori property, if the pattern $\langle A \rangle$ (red background) is found to be infrequent, then all its super-patterns (with gray background) cannot be frequent. Therefore, they need not be subject to checking.

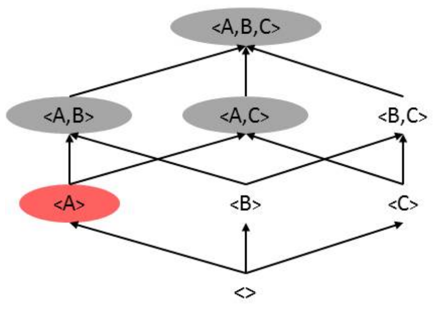


Figure 3.5: Sequence lattice

However, due to the additional constraints (*i.e.* total-ordering and one-to-one correspondence) on the event sequence, iterative pattern does not possess the traditional Apriori property in [5]. Here, we propose a particular Apriori property for the iterative patterns, which will be discussed later in Section 3.3.4.3.

3.3.4.2 Constrained Iterative Pattern

We analyze traces to derive dataflow related sequences which constitute a sequence database in last section. Our goal is to mine all *dataflow relevant* and *frequent* iterative patterns from the derived *dataflow related sequence database*.

For example, consider the trace (Fig. 3.3(b)), the original sequence of events occurring in it is shown in Table 3.1(a)[‖]. The generated dataflow related sequences by analyzing the trace are listed in Table 3.1(b). In order to track the origins of events in the dataflow sequences, we tag each event by the index with which the event is associated in the original event sequence (Table 3.1(a)). Let T be a trace. The sequence of all the events occurring in T is denoted by $L_{(T)}$. We refer to the ith event in $L_{(T)}$ as $L_{(T)}(i)$. We use $D_{(T)}$ to denote the dataflow related sequence database obtained by analyzing the trace T. There are two problems we need take into account, which are not present in the original iterative pattern mining context.

The first problem is *Duplication.* Since one event in the trace may be involved in multiple dataflow paths, it may occur multiple times in the sequence database $D_{(T)}$. These duplicated events confuse the miner when it attempts to calculate the support of an event. In Table 3.1, the support for the event a, as determined from the trace, is 1; however, it appears 4 times in $D_{(T)}$. A normal miner working on the sequence database will count it as 4, instead of 1.

Table 3.1: (a).Sequence (b).Database

(a)

Index	1	2	3	4	5	6	7	8
Event	a	c	d	c	d	e	f	b

(b)

Tid	Transaction					
0	1	2				
	a	c				
1	1	4				
	a	c				
2	1	2	3	6	7	8
	a	c	d	e	f	b
3	1	4	5	6		
	a	c	d	e		

Another problem is *Correspondence.* Since we focus solely on dataflow relation, some events in traces may be omitted in dataflow related sequences. Therefore, an iterative pattern instance detected from the se-

[‖]For brevity, we use a single character to represent an event.

quence database may not be a valid instance in the original trace. For instance, given the 2-pattern $\langle a, d\rangle$, we scan sequence transaction 3 in $D_{(T)}$ to obtain an iterative pattern "instance", $\langle 1, 5\rangle$. But in fact, this "instance" does not correspond to a valid instance due to the additional 3rd event d in the trace. To address the above two problems, we introduce a novel definition of iterative pattern instance, called *constrained iterative pattern instance.*

Definition 4 (Constrained Iterative Pattern Instance). *Given a trace T and its event sequence $L_{(T)}$, an ordering number subsequence $\langle o_1 o_2 \ldots o_n\rangle$ of a sequence in $D_{(T)}$ is a* constrained iterative pattern instance *of $p^n(\langle e_1 e_2 \ldots e_n\rangle)$ iff the following conditions hold:*

- $\forall q \in [1, n], L_{(T)}(o_q) = e_q$;
- $\forall i \in [1, n-1], \left(\forall j \in (o_i, o_{i+1}), L_{(T)}(j) \notin p^n\right)$.

Let's look at some examples. Consider sequence 3 in Table 3.1(b), the ordering number subsequence $\langle 1, 6\rangle$ is a constrained iterative pattern instance of pattern $\langle a, e\rangle$. Specifically, $n = 2, o_1 = 1, o_2 = 6$, $\forall q \in [1, 2], L_{(T)}(o_q) = e_q$ and $\forall j \in (1, 6), L_{(T)}(j) \notin \langle a, e\rangle$ hold. On the other hand, $\langle 1, 5\rangle$ is not a constrained iterative pattern instance of pattern $\langle a, d\rangle$, since for $3 \in (1, 5)$, $L_{(T)}(3) = d \in \langle a, d\rangle$, which contradicts the second condition in the definition. Every constrained iterative pattern instance is represented by a unique ordering number subsequence. Each unique number indexes one unique event occurrence in the trace. Therefore, we can determine the duplication by comparing the instances (sequences of ordering numbers). Besides, in our definition, we do not only check the events in sequences (i.e. the first condition), but also consider the events discarded by the sequences (i.e. the second condition). We can ensure that any constrained iterative pattern instance is an iterative pattern instance in the original trace. The correspondence problem is thus solved.

3.3.4.3 Apriori Property

Apriori property has been introduced to prune the search space of mining algorithm [5]. It states that if a pattern P is not frequent, then it is not possible for any super-pattern of P to be frequent. Thus, it is unnecessary to search for any frequent super-pattern of P. However, due to the additional constraints (*i.e.* total-ordering and one-to-one correspondence) on the event sequence, iterative pattern does not possess the traditional apriori property in [5]. Here, we propose a special Apriori property for iterative pattern. We first provide some definitions.

Definition 5 (Prefix_pattern, Suffix_pattern, Infix_pattern). *For a k-pattern* $p^k(\langle e_1, e_2, \ldots, e_k\rangle)$*, its prefix_pattern is defined as* $pre\_p^{k-1}(\langle e_1, e_2, \ldots, e_{k-1}\rangle)$*; its suffix_pattern as* $suf\_p^{k-1}(\langle e_2, e_3, \ldots, e_k\rangle)$*; and its infix_pattern as* $in\_p^{k-1}(\langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_k\rangle)$*, where* $i \in [2, k-1]$ *and* $e_i \notin in\_p^{k-1}$*.*

For example, given a 5-pattern $p^5(\langle A, B, C, D, B\rangle)$, it has the prefix_pattern $pre\_p^4(\langle A, B, C, D\rangle)$, suffix_pattern $suf\_p^4(\langle B, C, D, B\rangle)$, and infix_patterns $in\_p^4_1(\langle A, B, D, B\rangle)$ and $in\_p^4_2(\langle A, B, C, B\rangle)$. However, $\langle A, C, D, B\rangle$ is not an infix_pattern of p^5 since $B \in \langle A, C, D, B\rangle$.

Theorem 1 (Apriori Property). *If a pattern* p^k *is frequent, then its prefix_pattern, suffix_pattern and all infix_patterns are frequent.*

Having the above definitions, we arrive at the following specific Apriori property possessed by the (constrained) iterative pattern in Theorem 1. The proof is given by Appendix B.

3.3.4.4 Algorithm

Algorithm 3: CIPM($D_{(T)}, min\_sup, min\_den$)

```
Input: database $D_{(T)}$, support threshold $min\_sup$, density threshold $min\_den$
Output: set of all frequent closed patterns $F_{closed}$
1 $F_1 \leftarrow \{p^1 |\ sup(p^1) \geq min\_sup\}$;
2 for ($k \leftarrow 2$; $F_{k-1} \neq \emptyset$; $k$ ++) do
3     $C_k \leftarrow$ apriori_gen($F_{k-1}$);
4     $F_k \leftarrow$ apriori_count($C_k, min\_sup$);
5     $F_k \leftarrow$ prune_density($F_k, min\_den$);
6     $F_{closed} \leftarrow$ process_closed($F_k, F_{closed}$);
7 end
8 return $F_{closed}$;
```

Algorithm 3 gives the mining algorithm involving two main phases. The first phase simply scans the sequence database once to detect all frequent singleton patterns (Line 1). The second phase is an iterative phase (Lines 2-7), which consists of four subprocedures. Firstly, the set of frequent $(k-1)$-patterns F_{k-1} is used to generate candidate k-patterns C_k using the apriori_gen function. Secondly, an apriori_count function is called to count the support of each candidate pattern in C_k. A set of frequent patterns F_k is thus generated. Next, we apply a density-based pruning strategy to prune the search space further and finally discard all non-closed iterative patterns.

Apriori Candidate Generation. This procedure consists of two steps, namely *join* and *prune*. First, in the *join* step, we check each pair of frequent $(k-1)$-patterns $p_i^{k-1}(\langle e_1, e_2, \ldots, e_{k-1}\rangle)$ and $p_j^{k-1}(\langle f_1, f_2, \ldots, f_{k-1}\rangle)$ in F_{k-1} to see if the prefix_pattern $pre\_p_i^{k-2}\langle e_1, e_2, \ldots, e_{k-2}\rangle$ of p_i^{k-1} is same as suffix_pattern $suf\_p_j^{k-2}\langle f_2, f_3, \ldots, f_{k-1}\rangle$ of p_j^{k-1}. If so, the candidate k-pattern $\langle f_1, f_2, f_3, \ldots, f_{k-1}, e_{k-1}\rangle$ will be generated by joining the two frequent $(k-1)$-patterns p_i^{k-1} and p_j^{k-1}. Besides the prefix_pattern and suffix_pattern, all the infix_patterns of the given frequent pattern must be also frequent according to the Apriori Property. Next, in the *prune* step, we delete all candidate patterns whose infix_patterns are not all in F_{k-1}.

For instance, let F_3 be $\{\langle ABC\rangle, \langle BCD\rangle, \langle ACD\rangle, \langle BCB\rangle, \langle ABB\rangle\}$. After the *join* step, $\langle ABCD\rangle$ and $\langle ABCB\rangle$ are generated.($\langle ABCD\rangle$ is generated by joining $\langle ABC\rangle$ with $\langle BCD\rangle$, $\langle ABCB\rangle$ is composed of $\langle ABC\rangle$ and $\langle BCB\rangle$.). In the *prune* step, $\langle ABCD\rangle$ is deleted because one of its infix_patterns, $\langle ABD\rangle$ is not in F_3 (*i.e.* $\langle ABD\rangle$ is not frequent). Another candidate $\langle ABCB\rangle$ is retained since its only infix_pattern $\langle ABB\rangle$ is in F_3. Eventually, C_4 will be $\{\langle ABCB\rangle\}$.

Apriori Support Counting. In order to address the duplication problem described earlier, we maintain an instance set $V(p^k)$ for each candidate pattern p^k, which contains all the already generated instances of p^k. We scan each sequence transaction to find all the constrained iterative pattern instances $Ins(p^k, tid)$ of pattern p^k in sequence *tid*. We then check each instance in $Ins(p^k, tid)$ to see if it has already been in $V(p^k)$ (*i.e.* whether it is duplicate or not). If not, we add the instance to $V(p^k)$ and increment the support of p^k. After scanning all the transactions, we compare the support of each candidate pattern p^k with the threshold *min_sup*. Finally, the frequent pattern set F_k is returned.

Density Pruning. We observed that the patterns whose density are low do not contain much information. In addition, mining these patterns can be costly. We elect to perform pruning on the basis of patterns' density.

$$den(p^k) = |\text{distinct events in } p^k| \ / \ k$$

The pattern whose density is lower than a threshold *min_den* is pruned out.

Closed Iterative Pattern Processing. The closed iterative pattern processing addresses the "compactness" issue of mined patterns by substantially reducing the number of patterns discovered while preserving the complete information on the frequent patterns.

Definition 6 (**Closed Iterative Pattern**). *A frequent k-pattern p^k is closed iff there exists no super-pattern p^{k+1} such that:*

- *p^k and p^{k+1} has the same support;*
- *p^k is the prefix_pattern or suffix_pattern or infix_pattern of p^{k+1}.*

Based on the definition 6, we can identify all the non-closed iterative patterns and further discard them. In turn, from the closed frequent iterative patterns, we can readily derive all the frequent iterative patterns and their supports.

3.3.5 Empirical Evaluation

We conduct the experiments on five real-world programs using our implemented prototype. Table 3.2 shows the subjects used. All experiments are conducted on an Intel Quad 2.83GHz PC with 4GB main memory running Windows XP Professional.

Table 3.2: Characteristics of subject programs

Subject	Version	LoC	Class	Method	Description
JDepend	2.9.1	2,723	18	224	Java dependency analyzer
Libsvm	3.1	3,188	21	98	SVM implementation
Compress	1.3	9,629	59	502	Commons Compress library
PMD	4.2.5	66,881	720	4,991	Java source code analyzer
Fop	0.95	185,186	1,313	9,840	XSL-FO to PDF transformer

Table 3.3: Performance of dataflow tracking analysis

		Trace Generation					Dataflow Analysis		
Subject	Test	Trace	Stmt	Event	Size(MB)	Time(s)	Seq	AL**	Time(s)
JDepend	5	5	494k	93k	57.4	5	73k	6.9	5
Libsvm	5	5	854k	36k	75.3	7	8k	5.7	5
Compress	5	5	949k	254k	155	10	156k	4.7	10
PMD	4	8	2119k	498k	235	17	299k	26.8	20
Fop	5	5	3480k	621k	417	42	535k	10.6	53

**"AL" represents the average length of these generated dataflow related sequences.

3.3.5.1 Runtime Performance of Dataflow Tracker

Table 3.3 provides the detailed information on trace generation and dataflow tracking analysis. Even though the number of events (Event) or statements (Stmt) in traces is huge, our analysis managed to complete its task for each subject within a minute.

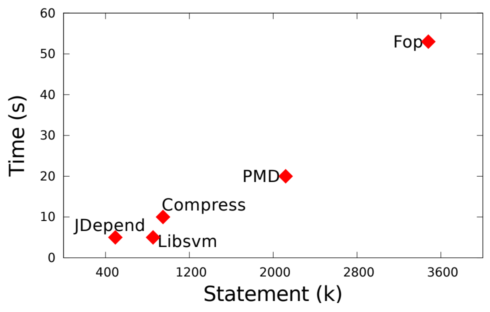


Figure 3.6: Execution time against the number of statements

Figure 3.6 shows the execution time of dataflow tracker against the number of statements analyzed in traces. It shows that the time required to track the dataflow is roughly linear with respect to the number of analyzed statements. Our dataflow tracking analysis is scalable to large traces.

3.3.5.2 Performance Comparison

Table 3.4 demonstrates the performance comparison of two different specification mining schemes in terms of number of patterns mined and time taken by mining. One is our dataflow sensitive specification mining (DSSM), which mines only semantically significant patterns over the generated dataflow related sequences using our constrained iterative pattern miner. Another directly performs the original iterative pattern mining (IPM) over the original sequence of events occurring in the trace. We further verified that *the set of patterns generated by DSSM is a proper subset of that generated by IPM.*

For each subject, we choose three absolute support thresholds to carry out the experiments. We choose the same density value (0.6) in all experiments. The asterisk "*" denotes a number which is more than 10 times bigger than the corresponding number of patterns generated by DSSM. The dash "-" denotes a time longer than 3600 seconds. As can be seen from Table 3.4, the number of patterns mined by IPM is much larger than that mined by DSSM. The ratios are

Table 3.4: Performance comparison

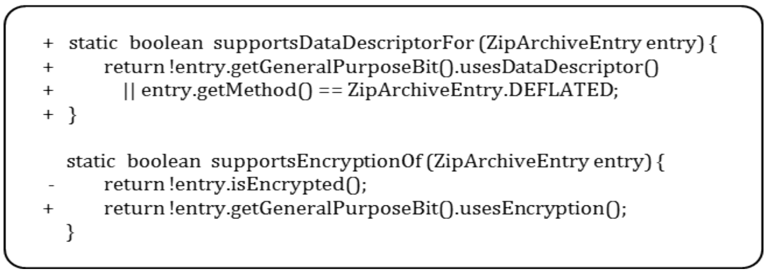
		DSSM		IPM		Ratio(IPM/DSSM)	
Subject	Support	Pattern	Time(s)	Pattern	Time(s)	Pattern	Time
JDepend	15	221	17	*	-	>10	
	30	181	15	*	-		
	50	54	9	*	-		
Libsvm	5	130	16	*	-	>10	
	10	115	15	*	-		
	30	48	7	175	99	3.6	>10
Compress	15	79	22	*	-	>10	
	50	44	19	*	-		
	100	32	18	*	-		
PMD	150	205	97	*	-	>10	
	250	100	45	*	-		
	450	32	35	149	-	4.7	>10
Fop	400	211	171	*	-	>10	
	1000	70	81	*	-		
	1500	21	60	636	1342		

mostly greater than 10. This indicates that our approach can effectively filter out dataflow irrelevant patterns. As a result, the effectiveness of the specifications will be substantially increased. We will further verify it in Section 3.3.5.3. In addition, the time cost of DSSM is much lower than that of IPM, at least 10 times faster in most cases. This shows that the efficiency of specification mining can be greatly improved.

3.3.5.3 Case Studies

Through two case studies, we shall demonstrate how dataflow sensitive specifications can be used to highlight changes in software evolution.

Compress Revision 922299-922309. Figure 3.7 illustrates the code changes between revision 922309 and 922299 in class ZipUtil in order to fix a bug (COMPRESS-100). "+" denotes the additional code in new revision. "-" denotes the code deleted from the old revision. Specifically, it primarily involves two methods. Firstly, method *supportsDataDescriptorFor* is added to determine whether an entry requires a data descriptor. As can be seen, it consists of two semantic scenarios. One is that *supportsDataDescriptorFor* first calls *getGeneralPurposeBit*, then returns a *GeneralPurposeBit* object which as a callee further calls *usesDataDescriptor*. Another is *supportsDataDescriptorFor* calls *getMethod* with the parameter *entry* as the callee. Secondly, method *supportsEncryptionOf* is changed. It originally invokes *isEncrypted*. In the new revision, *supportsEncryptionOf* calls *getGeneralPurposeBit* to acquire a *GeneralPurposeBit* object, and then *usesEncryption* is called to check if this entry is encrypted or not.



```
+  static  boolean  supportsDataDescriptorFor (ZipArchiveEntry entry) {
+       return !entry.getGeneralPurposeBit().usesDataDescriptor()
+           || entry.getMethod() == ZipArchiveEntry.DEFLATED;
+  }

   static  boolean  supportsEncryptionOf (ZipArchiveEntry entry) {
-       return !entry.isEncrypted();
+       return !entry.getGeneralPurposeBit().usesEncryption();
   }
```

Figure 3.7: Code changes between revision 922309 and 922299 in ZipUtil

We perform our dataflow sensitive specification mining (DSSM) on two revisions using the same input and derive 4 discriminative patterns shown in Table 3.5 (against the common patterns mined from both revisions). The first three are additional patterns mined from the new revision. The fourth pattern is deleted from the old revision. As can be seen, the first two additional patterns capture the added behavior of method *supportsDataDescriptorFor*. The third additional pattern and the deleted pattern correspond to the changes of *supportsEncryptionOf*.

Table 3.5: Discriminative patterns between revision 922309 and 922299

⟨ZipUtil: boolean supportsDataDescriptorFor(ZipArchiveEntry)⟩; ⟨ZipArchiveEntry: GeneralPurposeBit getGeneralPurposeBit()⟩; R: ⟨ZipArchiveEntry: GeneralPurposeBit getGeneralPurposeBit()⟩; ⟨GeneralPurposeBit: boolean usesDataDescriptor()⟩;
⟨ZipUtil: boolean supportsDataDescriptorFor(ZipArchiveEntry)⟩; ⟨ZipArchiveEntry: int getMethod()⟩; R: ⟨ZipArchiveEntry: int getMethod()⟩;
⟨ZipUtil: void checkRequestedFeatures(ZipArchiveEntry)⟩; ⟨ZipUtil: boolean supportsEncryptionOf(ZipArchiveEntry)⟩; ⟨ZipArchiveEntry: GeneralPurposeBit getGeneralPurposeBit()⟩; R: ⟨ZipArchiveEntry: GeneralPurposeBit getGeneralPurposeBit()⟩; ⟨GeneralPurposeBit: boolean usesEncryption()⟩;
⟨ZipUtil: void checkRequestedFeatures(ZipArchiveEntry)⟩; ⟨ZipUtil: boolean supportsEncryptionOf(ZipArchiveEntry)⟩; ⟨ZipArchiveEntry: boolean isEncrypted()⟩; R: ⟨ZipArchiveEntry: boolean isEncrypted()⟩;

To further assess the applicability of our approach (DSSM), we compare with the original iterative pattern mining (IPM). Similarly, we employ IPM to two revisions using the same input. With the same threshold setting, IPM generates much more discriminative patterns (48 additional and 15 deleted patterns) than

DSSM (3 additional and 1 deleted pattern). Although these patterns include the four semantically significant patterns, other semantically meaningless patterns would seriously weaken the efficacy of inferred specifications.

Compress Revision 911465-911467. Table 3.6 shows an additional pattern we discovered from revision 911467 for Compress. It describes a scenario that *fill* should call *count* to update the number of bytes read. It corresponds to a bug fixing. Specifically, in the previous revision 911465, *fill* missed calling *count* (COMPRESS-74).

Table 3.6: An additional pattern from revision 911467

⟨zip.ZipArchiveInputStream: void fill()⟩;
⟨ArchiveInputStream: void count(int)⟩;
⟨ArchiveInputStream: void count(long)⟩;
R: ⟨ArchiveInputStream: void count(long)⟩;
R: ⟨ArchiveInputStream: void count(int)⟩;
R: ⟨zip.ZipArchiveInputStream: void fill()⟩;

From another perspective, this case shows that our approach can assist in detecting bugs to some extent. When performing our approach on the older buggy revision (911465), we failed to discover the above pattern. Specifically, 7 dataflow related patterns are produced. None of them contain method *fill* or *count*. A programmer with knowledge of how *fill* and *count* interact will easily find this bug due to lack of the pattern by checking the mining results. Compared with our approach, IPM reports much more patterns (151 patterns) under the same setting. 11 of them involve method *fill*. Clearly, manually checking 11/151 patterns needs much more efforts than checking 0/7 patterns.

3.3.6 Discussion

Threats to Validity. The number of subjects tested remains small, possibly causing a threat to external validity of our experiments. To mitigate this, we ensure that they are all real-world programs from different domains with varying sizes. A potential threat to internal validity lies with the choice of the support and density thresholds used during mining. Here, we take into account the characteristics of the sequence data (e.g., number of sequences, average sequence length) while choosing these absolute support values, to limit the unnecessary randomness. Lastly, we note that the effectiveness of applying dataflow sensitive specifications to characterize program changes is prominent only when the program change is indeed dataflow related, and can be represented at call level.

Limitations. Firstly, we notice that symbolic instrumentation may suffer from high time and space overheads, especially for long-running programs. This can be circumvented by eliminating trace generation through fusing dataflow tracking analysis with instrumentation. In addition, we can restrict instrumentation activities only on entrances to basic blocks (in the sense of control flow graph), thus minimizing the cost. Secondly, we did not instrument JVM's library, and assume that there is no dataflow through JVM calls. On the other hand, our experiments do not indicate any loss of valuable specifications due to this approximation. Thirdly, duplication of events at multiple dataflow paths can affect the scalability of our approach, and we intend to apply incremental mining [74] to eliminate this limitation. Finally, the approach does not discover interactive behavior among multiple threads, which will remain one of the future work.

3.4 Related Work

Semantics-based Specification Mining. As mentioned before, several researchers have considered semantic information during specification mining. Ammons *et al.* [6] collected execution traces and annotated them with intra-procedural data dependency information. Their approach then infers call interaction patterns as finite state machines. Thummalapenta and Xie [101] also analyzed intra-procedural data dependency to filter out unrelated calls, while mining common exception-handling behavior. At the object-level semantics, Pradel and Gross [87] and Wasylkowski *et al.* [105] used object sharing relations to infer object usage models; Lee *et al.* [54] proposed specification mining parameterized by object interactions. At predicate-level semantics, Daikon [28] discovers from execution traces value-based invariants at specific program points. Our work differs from the above in the granularity of semantic information involved. Specifically, we track *fine-grained inter-procedural* dataflow information, and attempt to leverage complete dataflow relation to ensure that mined specifications are guaranteed to be dataflow relevant.

Dynamic Program Slicing. Our work is also related to dynamic data slicing

[118, 102] where only dynamic data dependences are considered ignoring control dependences. There are however differences: (1) Our mined patterns are both dataflow related and *statistically frequent*. (2) Mined patterns are viewed as program properties as they are derived from multiple traces, contrary to slices which are obtained from a single execution trace. (3) Mined patterns contain more abstract information than traditional slices as they only record method calls and returns. (4) Mined patterns capture more concrete runtime execution information than slices, because it can consist of multiple occurrences of the same statement in source code, whereas program slices record each statement uniquely. (5) Our result is a sequence or path of events instead of set or graph of statements for slicing.

3.5 Chapter Summary

In this chapter, we propose a novel semantics-directed specification mining framework to efficiently discover semantically significant specifications by exploiting semantic information to refine the execution traces. Based on this framework, we develop a *dataflow sensitive specification mining* system to mine frequent dataflow related iterative patterns from the execution traces as specifications. The empirical evaluation shows that our system is (1) effective in filtering off semantically irrelevant patterns, (2) efficient in generating semantically significant patterns, and (3) practical in program understanding and bug detection.

Chapter 4

Statistical Debugging via Hierarchical Instrumentation

Debugging is known to be a notoriously painstaking and time-consuming task. As one major family of automated debugging, statistical debugging approaches have been well investigated over the past decade. All these approaches instrument the entire buggy program to collect the full-scale execution profiles for debugging. Consequently, they often incur hefty instrumentation and analysis cost. As a matter of fact, major parts of the program code are error-free. The execution information collected corresponding to the error-free code is unnecessary for debugging. The instrumentation for these parts of code is wasteful.

In this chapter, a novel *hierarchical instrumentation* (HI) technique is devised to refine the execution profiles collected and analyzed by pruning away unnecessary instrumentation so as to make statistical debugging more efficient while upholding the debugging effectiveness. We apply the HI technique to two different categories of statistical debugging: in-house debugging and cooperative debugging for field failures. The experiments validate that the HI technique can greatly improve the efficiency of statistical debugging approaches without jeopardizing the debugging effectiveness.

We first provide the motivation in Section 4.1. Next, we present the details of our HI technique in Section 4.2, followed by two applications in Sections 4.3 and 4.4. Section 4.5 discusses the extension of our HI technique to multiple levels. Related work is given by Section 4.6 and Section 4.7 summarizes.

4.1 Motivation

Bugs are prevalent in software systems. As is well known, debugging is a notoriously painstaking and time-consuming task. To reduce developers' burden, researches have proposed a wide variety of automated debugging approaches. As one major family of automated debugging, statistical debugging approaches collect failing and passing executions and apply statistical techniques to identify discriminative elements as potential bug cause [47, 58, 3] or signature [42, 15, 96]. The rationale is that program elements which are frequently executed in the failing runs but rarely executed in the passing runs are quite likely to be faulty.

One problem with these statistical debugging approaches is that they consider *every program element* to be potentially relevant to the failure, and thus instrument the *entire program* for debugging. Such full-scale program instrumentation incurs hefty cost in terms of disk storage space usage, CPU time and memory consumption, etc, not just during instrumentation but also analysis thereafter. However, in fact, most parts of the program code work well, and *only small portions of a program are relevant to a given bug* [16]. As stated in [58], the majority (often 98-99%) of program elements (e.g., predicates) monitored are irrelevant to program failures. The execution information corresponding to these elements is not necessary to be collected in the execution profiles by instrumentation. This motivates us to devise a selective instrumentation technique such that only the necessary program elements which are highly correlated to the failure are instrumented, ignoring irrelevant ones. Therefore, the execution profiles collected and analyzed are refined. The statistical debugging approaches can thus become more efficient.

4.2 Methodology

In order to make statistical debugging approaches more efficient while not affecting their effectiveness, we devise a novel hierarchical instrumentation technique to refine the execution profiles collected and analyzed. The core of HI is to safely and effectively prune away unnecessary instrumentation and only instrument a set of prospective program elements such that bug-relevant elements can be discovered with much less instrumentation and analysis effort. HI is based on the following insight.

Insight

> *Information collected and measured by instrumenting composite syntactic constructs (e.g., functions) can be used to guide the selection of program elements (e.g., predicates) for subsequent instrumentation.*

We call the former instrumentation *coarse-grained* whereas the latter *fine-grained*. In brief, we first perform a lightweight coarse-grained instrumentation and obtain suspiciousness information of coarse-grained elements (e.g., functions). By means of such coarse-grained suspiciousness information, we safely and effectively prune away instrumentation of fine-grained elements (e.g., predicates[*]).

Specifically, given a fine-grained suspiciousness measure F which can quantitatively assess the suspiciousness of fine-grained elements (e.g., predicates) with respect to a failure, a statistical debugging approach analyzes two distinct groups of executions and returns the top-k suspicious elements. The goal of the HI technique is to make the statistical debugging approach more efficient by performing selective instrumentation, while upholding the original effectiveness of debugging (i.e., producing the same top-k elements with identical F values). To this end, our HI technique requires two coarse-grained measures: one C_p for pruning and another C_r for ranking.

4.2.1 Coarse-grained Measure for Pruning

A coarse-grained pruning measure C_p assigns a real to each coarse-grained element (such as function). This real number will be used to determine if the predicates within the corresponding function need to be instrumented during fine-grained instrumentation. We formalize this measure as follows:

Definition 7 (Coarse-grained Pruning Measure). *Given a fine-grained measure $F : \mathbb{N}^2 \rightarrow \mathbb{R}$ defined over the intervals $([0, X], [0, Y])$, a function $C_p : \mathbb{N}^2 \rightarrow \mathbb{R}$ under the same domain is defined as a coarse-grained pruning measure if it satisfies the following conditions:*

- *C_p is an upper bound of F, i.e., $C_p(x, y) \geq F(x, y)$*

[*]The predicates we considered in this dissertation are statement-level predicates [58].

- *C_p is nondecreasing, i.e., $C_p(x, y) \geq C_p(x-1, y) \land C_p(x, y) \geq C_p(x, y-1)$*
- *C_p is as close to F as possible, i.e., $\sum_{x,y=0}^{X,Y}\{C_p(x, y) - F(x, y)\}$ is kept minimal*

According to the above definition, given a fine-grained suspiciousness measure F in a close integer sub-interval domain ($x \in [0, X], y \in [0, Y]$), the best coarse-grained measure $C_p(x, y)$ can be computed by dynamic programming. Algorithm 4 gives the details. As is widely known, its complexity is linear to the size of its domain space, i.e., $O(XY)$.

Algorithm 4: Coarse-grained Pruning Measure Computation

```
Input: fine-grained suspiciousness measure $F$
Output: coarse-grained pruning measure $C_p$
// base case
$C_p(0,0) \leftarrow F(0,0)$
for $x \leftarrow 1$ to $X$ do
    $C_p(x,0) \leftarrow \max\{F(x,0), C_p(x-1,0)\}$
end
for $y \leftarrow 1$ to $Y$ do
    $C_p(0,y) \leftarrow \max\{F(0,y), C_p(0,y-1)\}$
end
// body
for $x \leftarrow 1$ to $X$ do
    for $y \leftarrow 1$ to $Y$ do
        $C_p(x,y) \leftarrow \max\{F(x,y), C_p(x-1,y), C_p(x,y-1)\}$
    end
end
```

4.2.2 Necessary Condition

Having the coarse-grained pruning measure, we can derive a necessary condition for a coarse-grained element to contain fine-grained elements of high suspiciousness.

Theorem 2 (Necessary Condition Derivation). *Let e denote a fine-grained element, m be the corresponding syntactic construct (i.e., coarse-grained element) encompassing e. $s(e)$ (or $s(m)$) and $n(e)$ (or $n(m)$) denote the number of passing and failing runs where e (or m) is executed, respectively. Given a threshold θ, fine-grained suspiciousness measure F and coarse-grained pruning measure C_p, we have the following implication.*

$$F(s(e), n(e)) \geq \theta \implies C_p(s(m), n(m)) \geq \theta$$

Proof.

$$
\begin{aligned}
& F(s(e), n(e)) \geq \theta && (4.1)\\
\Longrightarrow\ & C_p(s(e), n(e)) \geq \theta && (4.2)\\
\Longrightarrow\ & C_p(s(m), n(m)) \geq \theta && (4.3)
\end{aligned}
$$

According to Definition 7, we know that C_p is an upper bound of F. Therefore, (4.2) holds. Since m corresponds to a syntactic construct of e, whenever e is executed, m must be executed, i.e., $s(e) \leq s(m)$ and $n(e) \leq n(m)$. In addition, C_p is not decreasing, we can thus derive (4.3). □

Theorem 2 derives a necessary condition for instrumenting fine-grained elements in a coarse-grained element m: $C_p(s(m), n(m)) \geq \theta$. It asserts that, when the coarse-grained pruning value for m is less than certain threshold θ, we do not need to consider the fine-grained elements in m for instrumentation, as none of them has suspiciousness value greater than or equal to θ. We formalize this condition check as follows.

Corollary 1. *Given a threshold θ, a coarse-grained element m, let $s(m)$ and $n(m)$ denote the number of passing and failing runs where m is executed, respectively. If $C_p(s(m), n(m)) \geq \theta$ is false, then the fine-grained elements in m need not be instrumented.*

4.2.3 Coarse-grained Measure for Ranking

Note that a threshold θ is required when checking the above necessary condition. We have to consider the following issues when determining θ:

- **Safeness.** The value of θ ensures that the top-k elements that have been identified from the fully instrumented program are still derivable.
- **Effectiveness.** The value of θ enables many bug-irrelevant elements to be excluded during fine-grained instrumentation.

Specifically, on one hand, to guarantee the safeness of pruning, the threshold θ must be kept low – it must be no bigger than the F value of ultimate kth top element returned by the original statistical debugging approach. On the other hand, this threshold should be as high as possible so that more fine-grained elements could be pruned away.

To guarantee safeness, we perform the same statistical debugging process on partially instrumented subject programs. Specifically, only a small subset of elements in the entire subject program are instrumented. The debugging algorithm then produces the current top-k suspicious elements each with their respective suspiciousness values. We set the threshold θ to be the suspiciousness value of the current kth top element. Since this small set of elements chosen for instrumentation is a proper subset of the set of elements instrumented in the original approach, the current top-k elements returned here will also be returned by the original approach, except that their suspiciousness values might not all be within the range of the ultimate top-k elements. θ thus set is safe because it is a lower bound of the ultimate kth top suspiciousness value.

To ensure effectiveness, we introduce another coarse-grained measure C_r in HI such that the C_r value of a coarse-grained element is highly correlated with the F values of the enclosed fine-grained elements. In other words, if the C_r value of a coarse-grained element is high, then it is quite likely that the F values of the enclosed fine-grained elements are high as well. Thus, based on this correlation, and the fact that the value of θ is obtained from the F value of the kth top element obtained at previous iterations, we are likely to obtain a high threshold by giving the priority of performing fine-grained instrumentation to those functions with high C_r values.

There are several possible ways to obtain C_r. For instance, we can utilize the fine-grained suspiciousness measure F as C_r. We adopt this solution in our first application, and empirically validate our belief that the C_r value of coarse-grained element is highly correlated with the F values of their respective enclosed fine-grained elements. We will elaborate on it in Section 4.3.3.2. However, this solution does not work properly when the fine-grained measure depends on other variables, besides negative and positive supports, which are not meaningful at coarse granularity[†]. In that case, a possible solution is to adopt a modified version of fine-grained measure with reduced dimensions. We will discuss it in detail in Section 4.4.3.3.

[†]For example, *Importance* [58] depends on $s_t(e)$ and $n_t(e)$, which are not meaningful at function level.

4.3 Efficient Predicated Bug Signature Mining via Hierarchical Instrumentation

We first demonstrate the effectiveness of HI technique by applying it to predicated bug signature mining (MPS) described in [96]. MPS is an in-house debugging approach which requires developers to run the instrumented program in house to obtain execution profiles for analysis. For MPS as well as other in-house debugging approaches, we apply the HI technique to reduce the size of execution data collected and analyzed so as to make debugging more efficient. Here, we propose an efficient predicated bug signature mining approach via HI, called HIMPS. The empirical study reveals that our technique can achieve around 40% to 60% saving in disk storage usage, time and memory consumption, and performs especially well on large programs. It greatly enhances the efficiency of bug signature mining, making a step forward to painless debugging.

4.3.1 Introduction

As is well known, debugging is an integral process of localizing the bug, understanding and then fixing it. To assist in debugging, a considerable number of automatic approaches have been studied in the past decades. Most of them [48, 47, 58, 60, 4, 3, 33] focus only on the first phase which is termed as fault localization or bug isolation[‡]. These approaches commonly collect two groups (i.e., failing and passing) of executions and apply the statistical techniques to isolation the discriminative element as the potential failure cause.

However, Parnin and Orso [85] recently claimed that *perfect bug understanding* does not hold. It is difficult in practice to understand the bug by examining a single buggy statement. More contextual information where the bug occurs is likely to provide useful clue for identifying, understanding and correcting bugs. Hsu *et al.* [42] coined the term *bug signature*. Instead of a single suspicious element (statement or predicate) isolated by automated bug isolation, bug signature comprises multiple elements providing the bug *context* information. They adopted sequence mining algorithm to discover longest sequences in a set of failing executions as bug signatures. Subsequently, Cheng *et al.* [15] identified bug

[‡]We use these two terms "fault localization" and "bug isolation" interchangeably in this dissertation.

signatures using discriminative graph mining. They mined discriminative control flow graph patterns as bug signatures from both passing and failing executions. Since only control flow transitions are considered in [15], bugs not causing any deviation in control flow transitions cannot be identified. To enhance the predictive power of bug signatures, Sun and Khoo [96] proposed *predicated bug signature mining*, where both data predicates and control flow information are utilized. They devised a discriminative itemset generator mining technique to discover succinct predicated bug signatures. As usual, it requires instrumenting the entire program to produce full-scale execution profiles for mining. Such full-scale instrumentation is unnecessary and severely undermines the efficiency of debugging.

We here propose an efficient predicated bug signature mining approach via HI, called HIMPS, whose workflow is demonstrated by Figure 4.1. Specifically, HIMPS comprises two phases: one-pass coarse-grained phase followed by two-pass fine-grained phase. At coarse-grained phase, we instrument all the function entries (called coarse-grained elements) of the program and run the instrumented program over all the failing and passing test cases. We then capture the execution information of these function entries, which is then used in the fine-grained phase to guide the selective fine-grained instrumentation. The fine-grained phase comprises two passes, namely *boosting* and *pruning*. The boosting pass instruments and analyzes a selected subset of predicates (called fine-grained elements), and computes a fine-grained suspiciousness threshold from it. This threshold is then exploited in the pruning pass, for pruning away unnecessary predicates and returning a set of prospective predicates for fine-grained instrumentation. Only these prospective predicates will be instrumented during the pruning pass which finally returns the top-ranked bug signatures as output for our entire analysis.

4.3.2 Background

We explain some of the steps used in predicated bug signature discovery proposed by Sun and Khoo [96]. Specifically, we discuss in some details how a program is instrumented to produce predicates during execution, the metrics used in assessing the suspiciousness of signatures – in the form of itemset, and how preprocessing is done to reduce the size of database for signature mining.

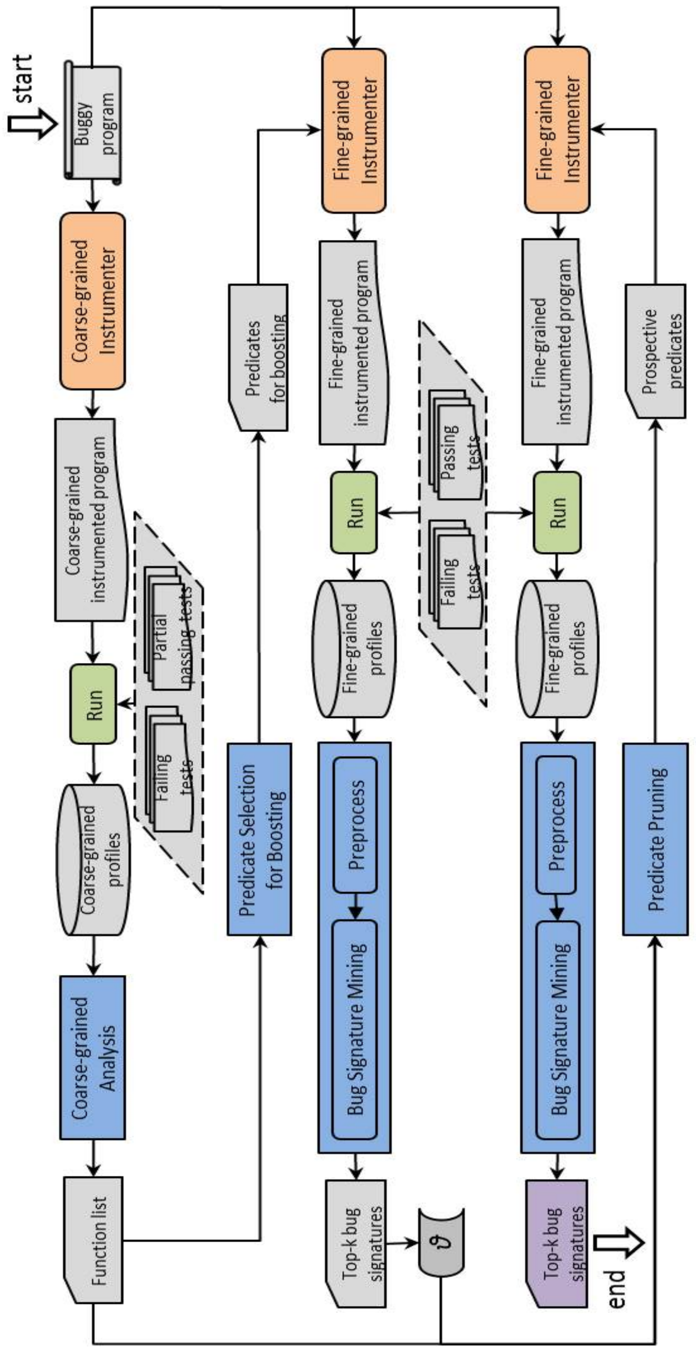


Figure 4.1: Workflow of predicated bug signature mining via HI

4.3.2.1 Predicated Bug Signature

An instrumentation scheme widely used in the statistical debugging community was developed by Liblit *et al.* [58]. It was also adopted by Sun and Khoo [96] in their design of predicated bug signature discovery. Here, a program is instrumented to collect the runtime values of *predicates* at particular program points. Each program point to be instrumented is called *instrumentation site.* At each instrumentation site, several *predicates* are tracked. There are three categories of instrumentation sites considered:

- **Branches.** For each conditional, two predicates are tracked to indicate whether the *true* or *false* branch is taken at runtime.
- **Returns.** At each scalar-returning call site, six predicates are created to capture whether the return value r is ever $> 0, \geq 0, < 0, \leq 0, = 0$, or $\neq 0$.
- **Scalar-pairs.** At each assignment of a scalar value, six predicates are considered: $x <, \leq, >, \geq, =, \neq y_i$ (or c_j), where x is the assigned value, y_i and c_j represent one of the other same-typed in-scope variables and one of the constant-valued integer expressions seen in the program, respectively.

A profile is obtained for each run of the instrumented program. It consists of a set of predicate counts which records the number of times each predicate is evaluated to true during the run. In [96], only those predicates whose counts are not less than 1 (i.e., the predicate is evaluated to true at least once), are retained. Each profile is thus regarded as a set of items, each of which is a predicate evaluated to true at least once during execution. In addition, each profile is labeled as passing or failing according to the oracle. All the profiles constitute a labeled itemset database, whose each transaction corresponds to one profile.

Formally, let $\mathcal{I} = \{e_1, e_2, \ldots, e_m\}$ be a set of items, $\mathcal{C} = \{+, -\}$ be the set of class labels, $\mathcal{D}$ be a class-labeled itemset database constituting n transactions, i.e., $\mathcal{D} = \{(T_1, c_1), \ldots, (T_n, c_n)\}$ where $\forall i \in [1, n], T_i \subseteq \mathcal{I} \wedge c_i \in \mathcal{C}$. As discussed above, in the context of predicated bug signature mining, $\mathcal{I}$ corresponds to the set of all the instrumented predicates. Each transaction T_i in $\mathcal{D}$ corresponds to a profile consisting of predicates evaluated to true during execution – a subset of $\mathcal{I}$. Each profile is generated by running the instrumented program using one test case. The class label (+) identifies the passing profile (which is associated with the correct execution behaving as expected), whereas (-) labels the failing profile (which is associated with the faulty execution). We call the corresponding transactions *positive* and *negative* transactions, respectively.

In [96], a bug signature is a set of predicate itemsets, which are observed together frequently in the failing executions but rarely in the passing ones, and thus regarded to be correlated to program failures. Given the class-labeled itemset database, predicated bug signature identification is formulated as a *discriminative itemset pattern mining* task. Sun and Khoo mined the highly discriminative predicate itemsets based on the discriminative measure discussed later in Section 4.3.2.2. Those itemsets with high discriminative significance values constitute the desired bug signatures.

4.3.2.2 Discriminative Significance

The discriminative significance of a pattern (itemset) is typically measured by the notion of *information gain* (IG) [90].

Let $\mathcal{D}$ be a class-labeled itemset database, $\mathcal{D}^+$ and $\mathcal{D}^-$ denote all the positive and negative transactions in $\mathcal{D}$, respectively. Given an itemset pattern P, the *support* of P *wrt.* an itemset database $\mathcal{D}$ is defined as the number of transactions in $\mathcal{D}$ containing P, i.e., $sup(P, \mathcal{D}) = |td(P, \mathcal{D})|$ where $td(P, \mathcal{D}) = \{(T, c) \in \mathcal{D} | P \subseteq T\}$. let $s = sup^+(P, \mathcal{D}) = |td(P, \mathcal{D}^+)|$ and $n = sup^-(P, \mathcal{D}) = |td(P, \mathcal{D}^-)|$ be the number of all the positive and negative transactions containing P, which are called *positive support* and *negative support* of P, respectively. The information gain of pattern P can be defined as follows:

$$\begin{aligned} IG(s, n) = H(|\mathcal{D}^+|, |\mathcal{D}^-|) - \frac{s+n}{|\mathcal{D}|} \times H(s, n) - \\ \frac{|\mathcal{D}| - (s+n)}{|\mathcal{D}|} \times H(|\mathcal{D}^+| - s, |\mathcal{D}^-| - n) \end{aligned} \tag{4.4}$$

where

$$H(a, b) = -\frac{a}{a+b} \times \log_2(\frac{a}{a+b}) - \frac{b}{a+b} \times \log_2(\frac{b}{a+b})$$

In [96], Sun and Khoo define the following *discriminative significance* measure based on IG:

$$DS(s,n) = \begin{cases} IG(s,n) & \text{if } \frac{n}{|D^-|} > \frac{s}{|D^+|} \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

Since all the itemsets within one bug signature possess the same positive and negative support, their DS values are also the same. Therefore, the DS value is used as the discriminative significance score of a signature.

4.3.2.3 Preprocessing and Bug Signature Mining

In [96], Sun and Khoo first performed preprocessing on the profiles to produce a dataset which is subsequently fed into the bug signature miner. To begin with, some unimportant or redundant predicates are filtered out in advance. Specifically, three filtering strategies are applied. Firstly, according to the definition of discriminative significance (Equation 4.5), all predicates such that $\frac{n}{|\mathcal{D}^-|} \leq \frac{s}{|\mathcal{D}^+|}$ are filtered as their DS values are always zero. Secondly, all predicates whose *Increase* [58] value not greater than zero are also filtered. Thirdly, all predicates with operators $\geq, \leq, \neq$ are filtered if they and their subsumed predicates[§] are both true in the same set of profiles. For instance, consider a predicate $a \geq b$, this predicate is true in all the same profiles where $a > b$ is true. That means $a \geq b$ does not capture additional execution information than its subsumed predicate $a > b$. The predicate $a \geq b$ is actually redundant and thus can be filtered. Note that preprocessing is essential as it constructs the database in a suitable format for the subsequent mining step; furthermore, it filters a great number of predicates so as to effectively reduce the scale of mining. However, it is also quite expensive especially if the profiles processed are of big size.

Sun and Khoo [96] devised the *discriminative itemset generator mining* algorithm. Given a predicate itemset database constructed from profiles through preprocessing, and the number of top discriminative signatures to mine k, the algorithm discovers the top-k discriminative bug signatures based on the *discriminative significance* measure discussed in Section 4.3.2.2. Specifically, they adopted a tree-based representation of the database as [55] and proposed a depth-

[§]$a \geq b$ subsumes $a > b$ and $a = b$. Similarly, $a \leq b$ subsumes $a < b$ and $a = b$, $a \neq b$ subsumes $a > b$ and $a < b$.

first search algorithm over the pattern space while effectively pruning the search space in a branch and bound fashion. They provided two modes of signature mining: *inter-procedural* and *intra-procedural*. In the first mode, a bug signature is identified over the whole program. The items in a signature can span across multiple functions. In the latter mode, the mining is employed to each function separately, and the items in a signature must reside in the same function. Since the *inter-procedural* signature mining is much more expensive than the *intra-procedural*, we focus on improving the efficiency of the *inter-procedural* mode in this work. We refer the readers to [96] for the detailed mining algorithm.

4.3.3 Approach

Given a buggy program and two groups (failing and passing) of test cases, the objective of our approach is to efficiently mine the top-k bug signatures which are highly correlated to the bug as measured by the DS values. The essence of our approach is a *safe pruning* of predicates instrumented and mined, making the bug signature mining more efficient. Briefly, we first capture the execution information of *functions* by a lightweight coarse-grained (function-level) instrumentation and analysis phase. Subsequently such information about functions is exploited to safely prune away unnecessary predicates. Algorithm 5 gives the detailed predicated bug signature mining algorithm via HI involving two phases, namely coarse-grained and fine-grained phase.

At the coarse-grained phase, only *function entries* of the subject program are instrumented (Line 1). This sparsely instrumented program is then run against all the failing and a small portion of passing test cases to collect the coarse-grained profiles (Line 2). Each coarse-grained profile consists of a set of functions which are executed at least once during execution. Having these coarse-grained profiles, a coarse-grained analysis is performed to produce a list of functions with their respective *negative* and *positive supports* (i.e., the number of failing and passing coarse-grained profiles containing the function, respectively) (Line 3). This function list will guide the ensuing fine-grained phase.

The fine-grained phase has two passes, each of which performs fine-grained instrumentation followed by bug signature mining. The first pass is called "threshold boosting". It aims to set a threshold which will be used as a lower bound of

Algorithm 5: Predicated Bug Signature Mining via HI

```
Input: buggy program $G$, test suite $T$, number of signatures mined $k$,
       percentage of predicates instrumented for boosting $\gamma$
Output: top $k$ suspicious signatures $PS$
// coarse-grained instrumentation and analysis
1  Instrument all function entries in the entire program $G$;
2  Run all the failing and passing tests in $T$ to collect coarse-grained profiles $CP$;
3  $list \leftarrow AnalyzeCoarseGrainedProfiles(CP)$;
// fine-grained instrumentation and analysis
// first pass: threshold boosting
4  $boost \leftarrow SelectPredicatesForBoosting(list, \gamma)$;
5  Instrument all predicates in $boost$;
6  Run all the failing and passing tests in $T$ to collect fine-grained profiles $BP$;
7  $BD \leftarrow Preprocess(BP)$;
8  $BS \leftarrow MineBugSignatures(BD, k)$;
9  $\theta \leftarrow k$th top $DS$ value of signatures;
// second pass: safe pruning
10 $prospect \leftarrow PrunePredicates(list, \theta)$;
11 Instrument all predicates in $prospect - boost$;
12 Run all the failing and passing tests in $T$ to collect fine-grained profiles $PP$;
13 $PD \leftarrow Preprocess(PP+BP)$;
14 $PS \leftarrow MineBugSignatures(PD, k)$;
15 return $PS$;
```

the ultimate kth top DS value of bug signatures mined by [96]. This is done by mining the kth top DS value of signatures having only a small fraction of highly suspicious predicates instrumented (Lines 4-9). The second pass will efficiently produce the top-k bug signatures through safely pruning considerable predicates whose DS values are less than the threshold determined in the first pass (Lines 10-15).

Specifically, in the first pass, we first select a few predicates which are likely to be of high DS values (Line 4). The selection detail will be expounded in Section 4.3.3.2. Next, we perform the fine-grained instrumentation to instrument all these selected predicates (Line 5), and then run the instrumented program using all the failing and passing test cases to acquire fine-grained profiles (Line 6). The profiles are preprocessed to create the mining dataset (Line 7), which is fed into the bug signature miner to discover the top-k bug signatures (Line 8). Note that Lines 5-8 indicate the traditional procedure of predicated bug signature mining stated by [96], making it possible for modular plug-in of new debugging algorithm. After mining, we attain a threshold θ, which is the kth top DS value of signatures mined during boosting (Line 9). We prove that this boosted threshold θ is a lower bound of the ultimate kth top DS value of signatures mined in [96]. We will discuss this in Section 4.3.3.3. In the second pass, predicates with DS value lower than θ are pruned away, leaving behind a set of prospective predicates constituting the top-k bug signatures (Line 10), which will be discussed in Section 4.3.3.4. Only these prospective predicates are considered in the pruning pass. Lines 11-14 perform the bug signature mining as usual. Finally, the identical top-k bug signatures as mined by [96] are returned (Line 15). Note that since we have obtained the fine-grained profiles corresponding to the predicates in *boost* during the first pass, we only need to instrument other predicates in $prospect - boost$ in the second pass so as to further reduce execution time and storage space for profile collection (Line 11). However, we have to preprocess all the profiles (i.e., $PP{+}BP$) in order to perform *inter-procedural* signature mining (Line 13).

4.3.3.1 Instrumentation

A salient feature of the HI technique is to have multiple levels of instrumentation, where instrumentation at higher/coarser-grained level can help prune unnecessary instrumentation at lower/finer-grained level, resulting in big saving in performance cost. In this work, two levels of instrumentation are applied, namely coarse-grained (Line 1) and fine-grained (Lines 5 and 11).

At the coarse-grained phase, only the function entries across the program are instrumented (Line 1). Each function entry corresponds to one instrumentation site. After running the coarse-grained instrumented program over all the failing and a portion of passing test cases, we obtain a set of coarse-grained profiles each for one test case. Each profile records a set of functions which are executed during the run.

The instrumentation scheme discussed in Section 4.3.2.1 is used for the fine-grained instrumentation (Lines 5 and 11). At each of the two passes, different parts of the program are instrumented, and the instrumented programs are executed using *all* the failing and passing test cases to generate two groups of fine-grained profiles, marked as failing and passing respectively. As mentioned in Section 4.3.2.1, each fine-grained profile is a set of predicates evaluated to true during the run. It corresponds to an itemset transaction in the fine-grained profiles database. Notice that our technique is also orthogonal to the instrumentation scheme. More types of predicates can be introduced without affecting our framework.

4.3.3.2 Predicate Selection for Boosting

Recall that the objective of the boosting pass is to generate a sufficiently high DS threshold of signatures for use in the pruning pass. Operationally, the threshold is generated by performing signature mining (*aka.*, MPS) on the subject with a small selected set of predicates being instrumented. As the whole mining process is involved, we wish to instrument as few predicates as possible so as to reduce overhead incurred and yet discover bug signatures with as high DS value as possible so as to prune more predicates away in the ensuing pruning pass.

Here we adopt the DS measure as C_r. We believe that the DS value computed during coarse-grained phase for each function, as an approximation to the DS values of the enclosed predicates, plays an important role in selecting predicates for boosting (i.e., the predicates instrumented during boosting pass). Specifically, we have the following hypothesis:

Hypothesis

If the DS value of a function is high, then it is quite likely that the DS values of predicates within this function are high as well.

In other words, there is a high correlation between DS values of a function and the predicates within. We test this hypothesis empirically by measuring the correlation coefficient (*i.e.*, Pearson's r [73, 7]) between the DS value of a function and the average DS value of all predicates within that function using 102 faulty versions in 5 buggy programs. For each version, we compute a correlation coefficient. Table 4.1 shows the averaged correlation coefficient among all the versions in each subject, excluding the statistically insignificant ones with p-value [73, 7] bigger than 0.05. The results indicate a strong positive correlation between DS values of a function and their predicates.

Table 4.1: Correlation Coefficient

Subject	replace	space	grep	sed	gzip	Overall
CC	0.70	0.71	0.63	0.45	0.64	0.63

Based on the above hypothesis, we elect to choose the predicates within functions of high DS values as the predicates for boosting. These predicates will be instrumented in the boosting pass for boosting a threshold. Specifically, we rank the functions in descending DS values, and select the predicates within the top few functions, until the total number of predicates selected reaches a predetermined percentage γ of the total number of predicates in the entire program (Line 4). In our experiments, to guarantee low overhead, we set γ to be 5%.

4.3.3.3 Safeness of Threshold Boosting

We obtain a threshold θ, i.e., the kth top DS value of signatures mined during boosting pass (Line 9). In the ensuing pruning pass, we will prune away those predicates whose DS values are less than θ, preventing them from being instrumented and thus saving the instrumentation and mining effort. To guarantee the safety of mining results (i.e., the ultimate top-k bug signatures mined by the original predicated signature mining [96] having all the predicates in the program instrumented, continue to appear in the results of pruning pass), we need to ensure that this boosted threshold is indeed a lower bound of the ultimate kth top DS value (as stated by Theorem 4), such that no real top-k signatures will be erroneously missed. Before the proof, we first introduce a definition and a theorem, which will be used in pruning Theorem 4.

Definition 8 (**Projected Database**). *Given a set of distinct items $\mathcal{I}$, a subset $\mathcal{I}' \subseteq \mathcal{I}$, a set of class labels $\mathcal{C}$, and a class-labeled itemset databases $\mathcal{D}$ constituting n transactions, i.e., $\mathcal{D} = \{(T_1, c_1), \ldots, (T_n, c_n)\}$ where $\forall i \in [1, n], T_i \subseteq \mathcal{I} \wedge c_i \in \mathcal{C}$, a same-sized itemset database $\mathcal{D}' = \{(T_1', c_1'), \ldots, (T_n', c_n')\}$ is said to be the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$ if and only if the following condition holds:*

$$\forall i \in [1, n], c_i' = c_i \wedge T_i' = T_i \cap \mathcal{I}'$$

Theorem 3 (**Pattern Preservation**). *Given a set of items $\mathcal{I}'$, two class-labeled itemset databases $\mathcal{D}$ and $\mathcal{D}'$ such that $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$, for an itemset pattern $P \subseteq \mathcal{I}'$, the following holds:*¶

$$DS(sup^+(P, \mathcal{D}'), sup^-(P, \mathcal{D}')) = DS(sup^+(P, \mathcal{D}), sup^-(P, \mathcal{D}))$$

¶The proof is provided in Appendix C.

Theorem 4 (Lower Bound). *Let θ be the boosted threshold, i.e., the kth top DS value of signatures mined by boosting pass, ds_k be the ultimate kth top DS value of signatures mined by the original predicated signature mining with all the predicates in the program instrumented, then we can derive that θ is a lower bound of ds_k, formally $\theta \leq ds_k$.*

Proof. Consider the original predicated signature mining, first instruments all the predicates in the entire program, then runs all the failing and passing test cases to collect profiles which are then constructed to a class-labeled itemset database for bug signature mining. As mentioned in Section 4.3.2.1, let $\mathcal{I}$ denote the set of all the instrumented predicates by the original signature mining, $\mathcal{D}$ denotes the class-labeled itemset database thus constructed. Accordingly, let $\mathcal{I}'$ correspond to the set of predicates instrumented at the boosting pass which is a subset of $\mathcal{I}$. $\mathcal{D}'$ is the itemset database derived during boosting. We can derive that $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$. Since for each executed test case, we will attain the same labeled profile in both databases (i.e., let n be the number of test cases executed, then $\forall i \in [1, n], c_i' = c_i$). Meanwhile, for those predicates instrumented during boosting (i.e., $e \in \mathcal{I}'$), they are evaluated to true if and only if they are also evaluated to true in the original mining for the same test case. Therefore, these predicates will identically appear in both transactions $T_i \in \mathcal{D}$ and $T_i' \in \mathcal{D}'$, i.e, $\forall i \in [1, n], T_i' = T_i \cap \mathcal{I}'$.

Since $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$, based on Theorem 3, we can conclude that for any itemset pattern mined in boosting pass, its DS value is identical to that computed in the original mining.

As mentioned before, a signature is a pattern consisting of itemsets with the same DS values. Therefore, given any signature mined in boosting, we will discover the same signature with the same DS value in the original mining. As a consequence, the ultimate kth top DS value of signatures mined in the original signature mining ds_k will be at least the boosted threshold θ, i.e., the kth top DS value of signatures mined during boosting pass, i.e., $\theta \leq ds_k$. □

4.3.3.4 Predicate Pruning

In the pruning pass, we aim to discover the actual top-k bug signatures. We leverage the results from the coarse-grained phase (which is a list of functions associated with their respective negative and positive supports) and the boosting pass (which is the boosted threshold θ) to safely prune away predicates for instrumentation whose DS value is less than θ, leaving behind a list of prospective predicates (Line 10 in Algorithm 5). According to the methodology presented in Section 4.2.1, we derive a specific coarse-grained pruning measure and exploit it for pruning based on a safe necessary condition.

Concretely, given the fine-grained suspiciousness measure DS which can be regarded as a function of two variables in natural number domain, i.e., $DS = F(s,n)$ where $s \in [0, |\mathcal{D}^+|], n \in [0, |\mathcal{D}^-|]$. (Please refer to Section 4.3.2.2 for notations.) The coarse-grained pruning measure can be defined recursively as Equation 4.6. We can thus employ dynamic programming (Algorithm 4) to derive our coarse-grained pruning measure.

$$C_p(s,n) = \begin{cases} DS(0,0) & \text{if } s = 0, n = 0 \\ \max\{DS(s,0), C_p(s-1,0)\} & \text{if } s \in (0, |\mathcal{D}^+|], n = 0 \\ \max\{DS(0,n), C_p(0,n-1)\} & \text{if } s = 0, n \in (0, |\mathcal{D}^-|] \\ \max\{DS(s,n), C_p(s-1,n), \\ \qquad C_p(s,n-1)\} & \text{if } s \in (0, |\mathcal{D}^+|], n \in (0, |\mathcal{D}^-|] \end{cases} \tag{4.6}$$

Conceptually, the predicates within functions whose C_p values fall below θ can be safely exempted from fine-grained instrumentation, since they cannot contribute a bug signature whose DS value is no less than θ. Specifically, this safe pruning condition can be formalized as follows.

Theorem 5 (Necessary Condition Derivation). *Let e_i denote a predicate, m_i be the function in which e_i is located, $s(m)$ and $n(m)$ be the positive and negative supports of function m, respectively. Given a predicate itemset database $\mathcal{D}$, a bug signature P comprising a set of predicate itemsets all having the same negative and positive supports, for ease of presentation, let $P = \{\{e_1, e_2, ..., e_k\}\}$ and a threshold θ, the following implication holds:*

$$DS(sup^+(P,\mathcal{D}), sup^-(P,\mathcal{D})) \geq \theta \implies \forall i \in [1,k], C_p(s(m_i), n(m_i)) \geq \theta$$

Proof.

$$
\begin{aligned}
& DS(sup^{+}(P,\mathcal{D}), sup^{-}(P,\mathcal{D})) \geq \theta && (4.7)\\
\Longrightarrow\ & C_p(sup^{+}(P,\mathcal{D}), sup^{-}(P,\mathcal{D})) \geq \theta && (4.8)\\
\Longrightarrow\ & \forall i \in [1,k], C_p(sup^{+}(\{e_i\},\mathcal{D}), sup^{-}(\{e_i\},\mathcal{D})) \geq \theta && (4.9)\\
\Longrightarrow\ & \forall i \in [1,k], C_p(s(m_i), n(m_i)) \geq \theta && (4.10)
\end{aligned}
$$

Inequality (4.8) holds because C_p is an upper bound of DS. Since C_p is non-decreasing and we have $s(m_i) \geq sup^{+}(\{e_i\},\mathcal{D}) \geq sup^{+}(P,\mathcal{D})$ and $n(m_i) \geq sup^{-}(\{e_i\},\mathcal{D}) \geq sup^{-}(P,\mathcal{D})$ hold, Inequalities (4.9) and (4.10) are thus derived. As a result, we have proved Theorem 5. □

Based on the above necessary condition derived, we prune away all the predicates within the functions whose C_p values are less than θ. Predicates that are not pruned away are called *prospective predicates*. As mentioned earlier, these prospective predicates will be instrumented and mined in the pruning pass.

From Theorem 5, we know that any bug signatures with DS value no less than θ, have to have their constituent predicates coming from these prospective predicates. Moreover, we have proved that this boosted threshold θ is a lower bound of the ultimate kth top DS value of signatures in Section 4.3.3.3. Thus, mining done at the pruning pass can discover all the actual top-k bug signatures.

4.3.4 Empirical Evaluation

We have conducted an empirical evaluation of our approach using 102 faulty versions in 5 buggy programs on an Intel Core 2 Quad 3.0GHz PC with 16GB main memory running 64-bit Fedora 19. Table 4.2 lists the subjects used, number of faulty versions, lines of code, number of functions, number of instrumentation predicates, and the size of test suite used. Note that in our experiments, all the results displayed for each subject are the average values computed across all the faulty versions in that subject. Moreover, the number of top discriminative signatures to mine k is set to 1, and γ which is the percentage of predicates instrumented during the boosting pass is set to 5%.

The original predicated signature mining [96] is composed of four main steps, namely instrumentation, profile collection, preprocessing and mining. Since each

Table 4.2: Characteristics of subject programs

Subject	Versions	LoC	Functions	Predicates	Tests
replace	31	564	21	22,412	5,542
space	34	6,199	136	459,030	1,248
grep	12	10,068	129	1,412,055	809
sed	16	14,427	169	2,377,612	363
gzip	9	5,680	90	3,741,611	213

instrumentation is only performed once and then the instrumented program can be run forever, its cost is not significant compared with the other steps (i.e., profile collection, preprocessing and mining). In the following, we mainly discuss the performance improvement during the other three steps. Specifically, we measure the improvement of our approach in reducing execution time and storage space during profile collection in Section 4.3.4.1. In Section 4.3.4.2, we compare with [96] in terms of time and memory consumption for preprocessing and signature mining. We abbreviate the original predicated bug signature mining [96] as MPS (Mining Predicated Bug Signatures), whereas our approach as HIMPS (MPS via HI).

4.3.4.1 Profile Collection

In this step, we run the failing and passing test cases to collect profiles. This corresponds to Lines 2, 6 and 12 in Algorithm 5. Here we consider two aspects of performance cost, namely the execution time for running the instrumented programs and the disk storage space used for profiles.

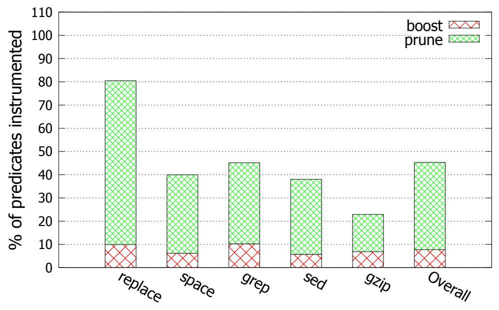


Figure 4.2: Percentage of predicates instrumented

As both performance costs are to some extent dependent on the number of predicates instrumented, we first present the percentage of predicates which are instrumented in our approach. Figure 4.2 depicts the percentage of predicates instrumented during boosting and pruning in HIMPS. As can be seen, compared with MPS where all the predicates (100%) in the whole program are instrumented, HIMPS manages to prune away considerable predicates. It only needs to instrument less than 45% of predicates for all subjects except for replace. Overall, more than half of the predicates are exempted from instrumentation compared with MPS. Note that HIMPS performs quite well especially on large programs. For gzip, even more than 75% of predicates are safely pruned away. For replace, it is of small size and most of the functions are executed during each run. HIMPS can hardly prune away predicates based on the coarse-grained information (i.e., negative and positive supports). As a consequence, it can only reduce 20% of predicates instrumented. We believe that the larger the program, the higher percentage of predicates our approach can prune away.

We have discussed the effectiveness of our approach in pruning unnecessary predicates in Figure 4.2. This provides an empirical evidence that HIMPS incurs relatively less time in executing instrumented programs and also utilizes less disk storage space for profiles than MPS. We further validate this hypothesis by directly measuring the execution time and the storage space for profiles.

Table 4.3: Execution time (in seconds) for profile collection

	MPS	HIMPS				Ratio
Subject	original	coarse	boost	prune	total	total/original
replace	12,023	6,272	6,603	9,811	22,686	188.70%
space	26,942	2,258	2,709	11,392	16,359	60.72%
grep	156,843	1,837	8,823	18,763	29,423	18.76%
sed	69,453	839	3,717	38,583	43,139	62.11%
gzip	665,144	4,463	113,129	57,217	174,809	26.28%
Overall	186,081	3,134	26,996	27,154	57,283	71.31%

Table 4.3 illustrates the execution time spent for running the instrumented program to collect profiles. We compare HIMPS including the coarse-grained phase (Column *coarse*) and two fine-grained passes (Columns *boost* and *prune*) with MPS adopting full instrumentation. As mentioned earlier, the coarse-grained instrumentation is lightweight. That is why Column *coarse* is quite small. Column *total* indicates the total execution time, which is the sum of all three columns in HIMPS. As can be seen, for grep and gzip, HIMPS only takes less than 30% of the execution time that MPS takes. But for replace, our approach costs more time than MPS due to running the fine-grained instrumented

program twice. Note that we employ *sampler-cc* [57] as the instrumenter in this experiment. Moreover, in order to ensure credible and stable results, we run each instrumented program four times, ignore the first run and compute the average execution time of the other three runs.

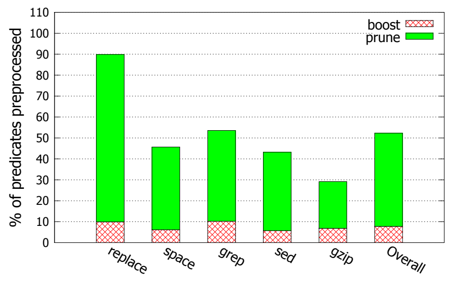
Table 4.4: Disk storage space used (in KB) for profile collection

	MPS	HIMPS				Ratio
Subject	original	coarse	boost	prune	total	total/original
replace	125,883	903	13,819	89,136	103,857	82.50%
space	567,025	522	35,089	195,290	230,901	40.72%
grep	1,141,582	318	116,325	395,454	512,096	44.86%
sed	864,367	191	49,465	288,836	338,491	39.16%
gzip	821,573	66	57,187	130,588	187,840	22.86%
Overall	704,086	400	54,377	219,861	274,637	46.02%

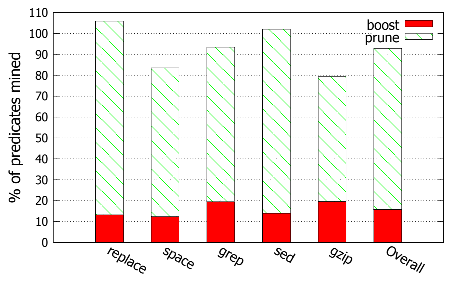
Table 4.4 presents the storage space used for profiles in kilobytes for HIMPS and MPS. Three groups of profiles were collected for HIMPS: the coarse-grained profiles (Column *coarse*), the fine-grained profiles during boosting (Column *boost*) and pruning (Column *prune*). It shows that HIMPS only requires less than 45% of the profile storage space required by MPS for all the subjects except for replace.

4.3.4.2 Preprocessing & Mining

Having fine-grained profiles, we perform preprocessing to construct the mining dataset (Lines 7 and 13), and then mine bug signatures (Lines 8 and 14). Here, we compare HIMPS with MPS in terms of time cost and memory consumption for preprocessing and mining. Note that in our experiment, both HIMPS and MPS perform the same procedure as stated in Section 4.3.2.3 under the same setting.



(a) preprocessing



(b) mining

Figure 4.3: Percentage of predicates analyzed

We first demonstrate the percentage of predicates preprocessed and mined in HIMPS against that of MPS as the base (100%). Figures 4.3(a) and 4.3(b) plot the percentage during preprocessing and mining, respectively. As mentioned earlier, the profiles collected during boosting have to be preprocessed again in the pruning pass for the inter-procedural mining (Line 13 in Algorithm 5). That is why the percentage of predicates preprocessed (Figure 4.3(a)) is slightly bigger than that of instrumented (Figure 4.2). Nevertheless, around 50% of the total predicates are preprocessed during boosting and pruning together. Compared with 50% reduction in the number of predicates for preprocessing, overall only about 10% of predicates are reduced for mining shown as Figure 4.3(b). The underlying reason is that a certain number of predicates have been pruned through the filtering strategy during preprocessing discussed in Section 4.3.2.3, which weakens the pruning effectiveness of our technique. Again, HIMPS can prune away higher percentage of predicates for larger programs than for smaller ones. In the following, we directly provide the time cost and memory consumption for preprocessing and mining.

Table 4.5: Time (in seconds) and memory consumption (in KB) for preprocessing

	MPS		HIMPS						Ratio	
	original		boost		prune		total		total/original	
Subject	Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
replace	65.10	141,455	1.90	39,112	38.89	128,448	40.79	128,448	62.66%	90.81%
space	155.43	427,286	3.52	57,590	55.09	201,124	58.61	201,124	37.71%	47.07%
grep	220.94	867,907	12.11	166,308	79.94	452,685	92.05	452,685	41.66%	52.16%
sed	147.17	696,303	5.34	135,928	65.23	374,397	70.57	374,397	47.95%	53.77%
gzip	115.19	731,132	6.14	184,280	28.41	312,901	34.55	312,901	29.99%	42.80%
Overall	140.77	572,816	5.80	116,644	53.51	293,911	59.32	293,911	44.00%	57.32%

Table 4.6: Time (in seconds) and memory consumption (in KB) for mining

	MPS		HIMPS						Ratio	
	original		boost		prune		total		total/original	
Subject	Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
replace	28.88	240,061	0.07	16,120	26.39	230,307	26.46	230,307	91.62%	95.94%
space	77.24	260,895	0.25	12,085	65.03	240,507	65.28	240,507	84.52%	92.19%
grep	335.07	351,248	1.88	25,009	197.32	273,248	199.19	273,248	59.45%	77.79%
sed	24.67	65,754	0.05	5,595	19.51	59,170	19.56	59,170	79.29%	89.99%
gzip	56.80	70,730	0.13	7,259	35.13	49,478	35.27	49,478	62.09%	69.95%
Overall	104.53	197,738	0.48	13,214	68.68	170,542	69.15	170,542	75.39%	85.17%

Tables 4.5 and 4.6 show the time cost in seconds and peak memory used in kilobytes for preprocessing and mining, respectively. Column *total* gets the sum of *boost* and *prune* for *Time* and the maximum for *Memory*. For preprocessing, HIMPS only takes less than 45% of time that MPS takes on average. The peak

memory consumed is also smaller than 55% of that used by MPS for all the subjects except for replace. As for mining, HIMPS can also save around 25% of time and 15% of peak memory consumption in general.

Table 4.7: Time (in seconds) and memory consumption (in KB) for preprocessing and mining together

	MPS		HIMPS						Ratio	
	original		coarse		boost & prune		total		total/original	
Subject	Time	Memory	Time	Memory	Time	Memory	Time	Memory	Time	Memory
replace	93.98	240,061	0.28	60,918	67.25	230,307	67.53	230,307	71.86%	95.94%
space	232.67	427,286	0.33	57,221	123.90	240,507	124.23	240,507	53.39%	56.29%
grep	556.01	867,907	0.85	101,256	291.24	452,685	292.09	452,685	52.53%	52.16%
sed	171.84	696,303	1.39	128,319	90.13	374,397	91.52	374,397	53.26%	53.77%
gzip	171.99	731,132	2.16	184,506	69.82	312,901	71.98	312,901	41.85%	42.80%
Overall	245.30	592,538	1.00	106,444	128.47	322,159	129.47	322,159	54.58%	60.19%

Table 4.7 demonstrates the total time cost and memory consumption for preprocessing and mining together. In addition, HIMPS also includes the time and memory used by the coarse-grained analysis, shown as Column *coarse*. We can see that the coarse-grained analysis is quite cheap compared with the fine-grained analysis (Column *boost & prune*). Overall, HIMPS can save more than 45% of total time and near 40% of memory consumption compared with MPS for the whole analysis.

4.4 Iterative Statistical Bug Isolation via Hierarchical Instrumentation

We illustrate the employment of the HI technique to cooperative bug isolation (CBI) for field failures [58] in the following. Different from in-house debugging where developers run the instrumented program to obtain execution data, CBI directly gathers execution profiles from end-users running the deployed instrumented program. Therefore, besides considering the scale of execution data collected and analyzed by developers, we have to consider another important performance aspect - user's overhead for running the instrumented programs. To this end, we propose a novel iterative (and cooperative) statistical bug isolation approach for field failures via HI. Our experiments show that this new approach via HI saves significant instrumentation effort and sharply reduces runtime overhead, while upholding the accuracy of bug isolation.

4.4.1 Introduction

Most software deployed around the world remains buggy in spite of extensive in-house testing. Hitherto, debugging continues to be a tedious and painstaking effort for developers. As an essential and yet expensive process in debugging, bug isolation (or fault localization) aims to isolate or locate program faults [103]. It has spun off many research activities aiming to automate this process. One automated bug isolation approach for field failures which has received much attention recently is *cooperative (statistical) bug isolation* [57, 58]. This approach applies the idea of crowd-sourcing in sampling classes of program runtime behavior from a large pool of end-users running instrumented programs for bug tracking. The gathered program traces enable developers to apply statistical techniques to pinpoint the likely causes of failures. The success of this approach hinges on the availability of a sufficiently large user base to run the instrumented programs. In order to encourage a great number of users to participate, the user's overhead for running the instrumented programs should be kept sufficiently low. To this end, Liblit *et al.* adopted a sparse random sampling technique [57]. This technique amortizes the cost of monitoring user-end executions to a large number of users so that each user suffers a relatively low time overhead. Nevertheless, from the perspective of developers, this approach does not really reduce the monitoring cost or the total size of execution data, which consumes many resources such as network bandwidth, storage space, CPU time, etc, due to the need for data transfer, storage and analysis. This constrains the practicability of post-deployment bug hunting, especially for large applications.

To ensure minimal effort spent by both end-users and developers, *iterative bug isolation* approaches have been proposed [16, 10]. They perform the instrumentation and statistical analysis in an iterative manner. Instead of the entire program, only partial code is instrumented and analyzed at each iteration. They check the bug predictors reported during each iteration and adjust the instrumentation plan for the next iteration based on the following principle of locality [10]: *if a program element (e.g., predicate) is highly predictive of failure, then the elements in its vicinity are potentially good bug predictors as well.* The iterative debugging process proceeds until the bug is found or the entire program is explored.

However, there are two main drawbacks of iterative approaches. Firstly, both iterative approaches [16, 10] make use of the principle of locality to guide their search for bugs; this principle however is not always effective in localizing bugs, as experiments have found [10]. Secondly and also more importantly, both iterative approaches require developers to check the predictors reported at each iteration, until the bug cause is found. As claimed in [85], developers are reluctant to go through a list of predictors, not to mention the need to repetitively perform this check at every iteration.

In order to tackle the above drawbacks, we employ HI to perform selective instrumentation and propose an iterative bug isolation approach for field failures via HI. Our approach via HI consists of two phases: a one-pass coarse-grained phase followed by an iterative fine-grained phase guided by results obtained from the first phase. The process runs as follows: first we instrument the entire program at a coarse granularity (e.g., function entry-level) and deploy the instrumented program to users for execution; such instrumentation is guaranteed to be lightweight. Once a sufficient number of user executions are collected, we calculate the coarse-grained execution information for each function. This information is then exploited to guide the iterative fine-grained (predicate-level) instrumentation. In the iterative phase, we use the coarse-grained information and feedback from previous iterations to help select a function at each iteration and instrument all the predicates in that function. This fine-grained instrumented program is then deployed. As we only instrument one function, the runtime overhead incurred at user side is kept minimal. We again collect the execution data and measure the suspiciousness value for each predicate in the function. We iterate the process until all the selected functions are exhaustively examined. Eventually, the globally top-k predictors are reported to the developers. Figure 4.4 illustrates the workflow of our system.

4.4.2 Background

Before presenting our work, we give a brief overview of cooperative statistical bug isolation approach and its extension to adaptive bug isolation. We also briefly associate our work with these approaches.

4.4.2.1 Cooperative Statistical Bug Isolation

Cooperative bug isolation (CBI) is a dynamic analysis for locating the causes of program field failures. It collects execution information of an instrumented program from both failing and passing end-user runs, and employs statistical techniques to pinpoint the likely root causes of software failures [59, 58].

Specifically, a program is instrumented to collect the runtime values of *predicates* at particular program points. Each program point to be instrumented is called *instrumentation site*. At each instrumentation site, several *instrumentation predicates* are tracked. There are three categories of instrumentation sites considered:

- **Branches**: For each conditional, two predicates are recorded to indicate whether the *true* or *false* branch is taken at runtime.
- **Returns**: At each scalar-returning call site, six predicates are tracked to capture whether the return value r is ever $> 0, \geq 0, < 0, \leq 0, = 0$, or $\neq 0$.
- **Scalar-pairs**: At each assignment of a scalar value, six relationships between the assigned value x and each other same-typed in-scope variable y_i are considered. Specifically, for each y_i, six predicates are created:

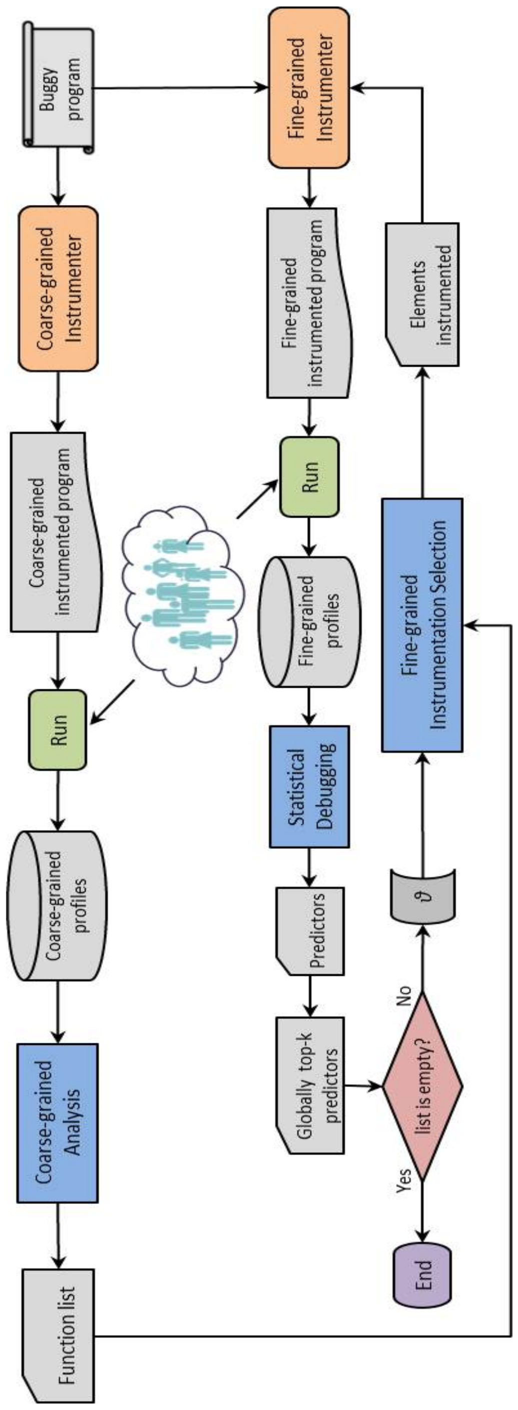

Buggy program
Coarse-grained Instrumenter
Coarse-grained instrumented program
Run
Coarse-grained profiles
Coarse-grained Analysis
Function list
Fine-grained Instrumenter
Fine-grained instrumented program
Run
Fine-grained profiles
Statistical Debugging
Predictors
Globally top-k predictors
list is empty?
Yes
No
End
ϑ
Fine-grained Instrumentation Selection
Elements instrumented

Figure 4.4: Workflow of iterative bug isolation for field failures via HI

$x <, \leq, >, \geq, =, \neq y_i$.

A profile is obtained for each run of the instrumented program. It consists of a set of predicate counts which records the number of times each predicate is evaluated to be true during this execution. In CBI [58], these counts are simplified to binary values (i.e., they only distinguish whether the predicate is true at least once or never). In addition, each profile is labeled as passing or failing. Having these profiles, the statistical technique is applied to compute a suspiciousness value for each predicate. The top scored predicate is regarded as the best predictor of the failure.

The following measure called *Importance* (Equation 4.11) has been used to assess the suspiciousness of predicates [58]. *Importance* is actually the harmonic mean of *Increase* (Equation 4.12) and *Sensitivity* (Equation 4.13). Notation-wise, for a predicate e, $s_t(e)$ and $n_t(e)$ are the number of passing and failing runs where e is observed to be true, respectively. $s(e)$ and $n(e)$ are the number of respective passing and failing runs in which e is observed, regardless of whether e is evaluated to be true or false. Note that *Importance* is actually a function under the natural number domain. Specifically, $s \in [0, S], n \in [0, N], s_t \in [0, s], n_t \in [0, n]$ where S and N are the total number of passing and failing runs, respectively.

$$Importance(e) = \frac{2}{\frac{1}{Increase(e)} + \frac{1}{Sensitivity(e)}} \tag{4.11}$$

$$Increase(e) = \frac{n_t(e)}{n_t(e) + s_t(e)} - \frac{n(e)}{n(e) + s(e)} \tag{4.12}$$

$$Sensitivity(e) = \frac{\log n_t(e)}{\log N} \tag{4.13}$$

In order to reduce the performance impact of instrumentation, CBI adopts sparse random sampling [57] derived from the work of Arnold and Ryder [8]. This technique generates instrumentation which samples a sparse and random subset of predicate counts during the execution. This helps to protect privacy and diminish performance overhead. However, the total monitoring and computational cost is not reduced as a whole. Moreover, sampling [8, 57] doubles the size of the executable, which constrains the practicality especially for large applications.

In this work, *we adopt CBI as our fine-grained statistical model*: we employ the instrumentation scheme discussed above to perform our fine-grained instrumentation, and *Importance* measure to assess the suspiciousness of predicates. However, instead of sampling, we conduct iterative monitoring. We select one function to be monitored at each iteration and instrument all the predicates within this function.

4.4.2.2 Adaptive Bug Isolation

Based on the observation that *only small portions of a program are relevant to a given bug* [16], two adaptive statistical bug isolation approaches [16, 10] have been proposed. HOLMES employs an iterative, locality-based instrumentation scheme to address both time and space overheads [16]. At each iteration, it identifies predictors of the reported failure. Based on the suspiciousness values of these predictors, HOLMES either decides that these predictors are the root causes, or expands the search by monitoring code in other functions that closely interact with these weak predictors.

Adaptive bug isolation presented by Arumuga *et al.* [10] is a fine-grained adaptive monitoring system which adaptively monitors the program at the granularity of predicates. It monitors a fraction of predicates at each iteration and adaptively adjusts the instrumentation plan to include predicates closer to the highly suspicious predicate currently explored. It is formulated as a search on the control dependence graph and presents several heuristics to guide the search.

Similarly, we also perform statistical bug isolation in an iterative way. Different from them, we propose a novel hierarchical instrumentation to facilitate iterative monitoring.

4.4.3 Approach

Based on the systematic HI technique discussed in Section 4.2, we propose an iterative bug isolation approach via HI. In brief, our approach aims to discover the top-k suspicious predictors at less instrumentation effort, while ensuring lightweight instrumentation throughout.

Algorithm 6 depicts the pseudo code of our approach which consists of a coarse-grained phase followed by an iterative fine-grained phase. At the first

Algorithm 6: Iterative Statistical Bug Isolation via HI

```
Input: buggy program $G$, number of predicates returned $k$
Output: top $k$ suspicious predicates $P_k$
$P_k \leftarrow \emptyset$;
$\theta \leftarrow 0$;
// coarse-grained instrumentation and analysis
Instrument all function entries in the entire program $G$;
Deploy the instrumented program;
CollectSufficientData();
list ← AnalyzeCoarseGrainedProfiles();
// fine-grained instrumentation and analysis
while $list \neq \emptyset$ do
    $m \leftarrow$ function with the highest $C_r$ value from $list$;
    $list \leftarrow list - m$;
    if NecessaryCondition($m, \theta$) is false then
        continue;
    end
    Instrument all predicates in function $m$;
    Deploy the instrumented program;
    CollectSufficientData();
    Update($P_k$, predictors in $m$);
    $\theta \leftarrow$ getTopKthImportance($P_k$);
end
return $P_k$;
```

phase, a coarse-grained instrumentation is performed to instrument all the function entries of the entire program, and then the instrumented program is deployed in the field (Lines 3-4). We collect sufficient execution data from a large number of end-users running the instrumented program (Line 5). We analyze the collected execution data (i.e., profiles) and eventually obtain a list of functions with their respective execution information (i.e., the number of failing and passing profiles containing the function) (Line 6). This list provides the coarse-grained execution information which will be utilized to guide the iterative fine-grained instrumentation in the second phase. The second phase is an iterative phase. At each iteration, we instrument at most one function at fine granularity. We select the function with the highest ranking value (i.e., C_r value) in the current list (Line 8). Based on the feedback θ which is the highest suspiciousness value (*Importance* value) explored so far, we verify the necessary condition for function m (by comparing its C_p value against θ) to contain predicates whose *Importance* value can be greater than or equal to θ. If the necessary condition is not satisfied, we skip this function m and proceed to the next iteration (Lines 10-12). Otherwise, we instrument all predicates within m according to the instrumentation scheme of CBI discussed in Section 4.4.2.1, and redeploy the instrumented program (Lines 13-14). We wait for sufficient execution data to be collected (Line 15). We next compute the *Importance* value of predicates in m and update the top-k predicates P_k as well as the kth top *Importance* value θ (Lines 16-17). If the current list of functions is not empty, we proceed to the next iteration. It is worth noticing that each function in the list can be examined at most once as it is removed from the list after it is considered (Line 9). We will provide the detailed discussion for each step in the remaining section.

4.4.3.1 Instrumentation and Deployment

The HI technique involves two different levels of instrumentation. At the coarse-grained instrumentation level (Line 3), we only instrument the function entries in the whole program. There is only one instrumentation site (function entry) per function, thus fairly lightweight. For each function entry, we count the number of times this function is called during a run. After a sufficient number of end-users running the instrumented program, we successfully collect the required number of coarse-grained profiles. Each profile consists of a set of functions which are called at least once, as well as a label marking this run as failing or passing. For fine-grained instrumentation (Line 13), we adopt the instrumentation scheme of CBI discussed in Section 4.4.2.1. Obviously, the number of predicates is much larger than that of function entries in coarse-grained instrumentation. However, here we only instrument the predicates within *one* function at each iteration. The performance overhead is much lower than that of sampling (cf. [57]), as validated in our experimental evaluation (Section 4.4.4.3).

During each iteration of fine-grained instrumentation, developers need to instrument the program and deploy the instrumented program to users (Lines 13-14). In practice, it is not feasible to recompile and reinstall the entire program each time, especially for large-scale programs. To this end, we adopt a technique called *weak recompilation* [89]. Instead of recompiling the entire program, we only compile the changed component (i.e., the function instrumented at each iteration) as a patch to a shared library. During execution phase, we replace the original function with the patch corresponding to the instrumented version of this function through utilizing function wrapping mechanism provided by runtime instrumentation tools such as Valgrind [78]. In other words, we wrap a function such that the calls to this function are intercepted and rerouted to the specific instrumented version of this function included in the shared library. As such, we are only required to recompile the instrumented function as a patch and deploy it to users at each iteration. The empirical evaluation conducted by [89] shows that the weak recompilation technique saves tremendous compilation cost and is scalable and practical.

4.4.3.2 Pruning Measure Calculation & Necessary Condition Derivation

By employing the systematic approach presented in Section 4.2.1, we derive a specific coarse-grained pruning measure and use it to derive a necessary condition for the purpose of pruning (Line 10).

Concretely, given the fine-grained suspiciousness measure *Importance* which can be regarded as a function of four variables in natural number domain, i.e., $Importance = F(s, n, s_t, n_t)$ where $s \in [0, S], n \in [0, N], s_t \in [0, s], n_t \in [0, n]$. (Please refer to Section 4.4.2.1 for notations.) The coarse-grained pruning measure can be defined recursively as follows:

$$C_p(s,n) = \begin{cases} M(0,0) & \text{if } s = 0, n = 0 \\ \max\{M(s,0), C_p(s-1,0)\} & \text{if } s \in (0,S], n = 0 \\ \max\{M(0,n), C_p(0,n-1)\} & \text{if } s = 0, n \in (0,N] \\ \max\{M(s,n), C_p(s-1,n), \\ \qquad C_p(s,n-1)\} & \text{if } s \in (0,S], n \in (0,N] \end{cases}$$

and

$$M(s,n) = \max\{F(s,n,s_t,n_t) \mid s_t \in [0,S], n_t \in [0,n]\} \tag{4.14}$$

Notice that in this particular case, the fine-grained suspiciousness measure *Importance* is of four dimensions whereas the coarse-grained measure only has two‖. We reduce the dimensions just by computing the maximum over the entire domain of additional variables s_t and n_t, as shown in Equation (4.14). Conse-

‖ *Importance* distinguishes whether a predicate is evaluated to true or false (i.e., $s_t(e)$ and $n_t(e)$). However, such values are not meaningful at function level.

quently, we are able to employ dynamic programming (Algorithm 4) to compute our coarse-grained suspiciousness measure.

Having the coarse-grained measure C_p, let e be a predicate, m be the function where e is located, and a threshold value θ, we can readily deduce the following implication.

$$Importance(e) \geq \theta \implies C_p(s(m), n(m)) \geq \theta$$

In other words, $C_p(s(m), n(m)) \geq \theta$ is a necessary condition for function m to contain predicates with *Importance* value at least θ. Consequently, when this necessary condition is *invalid*, we can safely skip function m for fine-grained instrumentation.

4.4.3.3 Ranking Measure Calculation

In order to effectively prune away unnecessary instrumentation, a coarse-grained ranking measure is required (Line 8). As we mentioned earlier, *Importance* measure has additional variables (s_t and n_t). Here, we propose to adopt the pruning measure derived above as the ranking measure. We will validate its effectiveness in the following.

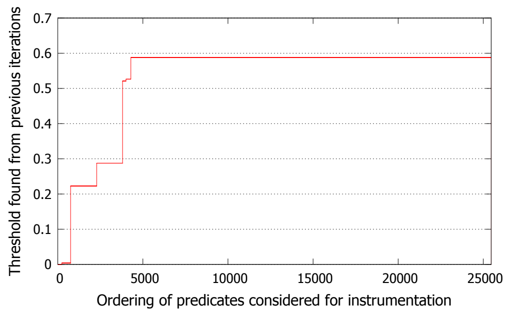


Figure 4.5: Threshold used for pruning versus ordering of predicates considered for instrumentation

Let's consider the case where we are looking for top-1 suspicious element. We order each predicate by the iteration in which it was considered for instrumentation; i.e., predicates with lower order are considered for instrumentation at earlier iteration. Predicates considered in the same iteration are randomly ordered. Figure 4.5 plots the threshold used when a predicate is considered for instrumentation versus the predicate order for a faulty version of subject *space*. Ideally, we want threshold to be set sufficiently high for as many predicates as possible, because this will increase the chance for necessary condition to be falsified, and the instrumentation of predicates be pruned. As such, it is desirable to have a plot in which the threshold raises very quickly and stabilizes at a high level.

To quantify the effectiveness of ranking measure, we represent the curve in Figure 4.5 by a function $\theta = f(x)$ under the integer domain ($x \in [1, X]$) and real number codomain ($\theta \in [0, \theta_{max}]$) where X and θ_{max} denote the total number of predicates and the maximum threshold found in the entire program, respectively. Given this function, we introduce a metric called *threshold mass* which is defined as follows:

$$\textit{threshold mass} = \sum_{x=1}^{X} f(x)$$

Note that the value of *threshold mass* is exactly the area under the curve. We then define the effectiveness of ranking measure as the following ratio:

$$\textit{ratio} = \frac{\textit{threshold mass}}{\theta_{max} \times X}$$

The higher the ratio, the more effective the ranking measure is. To test the practicality of our adopted ranking measure (which is the same as the pruning measure, as mentioned earlier in this section), we compute this ratio for multiple subject programs. Table 4.8 shows the averaged ratio over all the versions in each subject. On average, the ratio is about 83%. This indicates that, in practice our ranking measure can guide the iterative process to raise the threshold to a relatively high level at very early stage.

Table 4.8: Ratio of threshold mass

Subject	replace	space	grep	sed	gzip	Overall
Ratio	0.64	0.77	0.83	0.97	0.94	0.83

4.4.3.4 Sufficient Data Collection

Statistical bug isolation relies on a huge amount of data to ensure the stability of results. In other words, given a sufficiently large number of execution profiles, we can obtain the same result (e.g., the same ranked list of functions or predictors) even if the profiles under analysis are different.

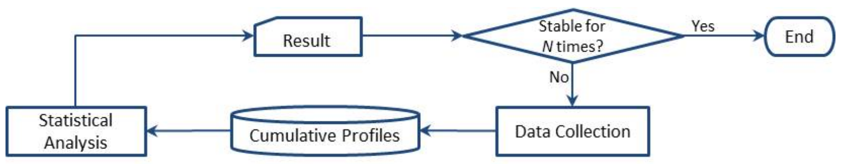


Figure 4.6: Sufficient data collection strategy

In practice, we can wait for sufficient user executions until a stable result is achieved. Specifically, once we have collected a good number of profiles, we can start generating a result. Next, we can continue to wait for the arrival of additional number of profiles and analyze all the profiles collected so far to get a new result. We keep this iterative process until we always achieve the same result in the recently consecutive iterations. Figure 4.6 illustrates the procedure. Although this is not the main focus of our work, we will adopt this strategy in our experiments discussed later in Section 4.4.4.1.

4.4.4 Empirical Evaluation

The effectiveness of statistical bug isolation has been validated by lots of prior work [57, 58, 120, 60, 44]. We thus mainly focus on validating the performance of our technique in reducing instrumentation effort in Section 4.4.4.1 and runtime overhead in Section 4.4.4.3 compared with non-iterative sampling-based statistical approach [58]. In addition, we also experimentally compare our approach with adaptive bug isolation [10] in Section 4.4.4.4.

Table 4.9: Characteristics of subject programs

Subject	Versions	LoC	Functions	Predicates	Tests
replace	31	564	21	2,920	5,542
space	34	6,199	136	25,449	1,248
grep	12	10,068	129	226,286	809
sed	16	14,427	168	96,389	363
gzip	9	5,680	91	179,408	213

We have conducted an empirical evaluation using the subject programs from SIR repository [25]. Table 4.9 lists our subjects used, number of faulty versions, lines of code, number of functions, number of instrumentation predicates, and the size of test suite used. The *Importance* measure we adopted as the fine-grained suspiciousness measure is not applicable if the faulty version fails in less than two test cases. Such versions were omitted and not included in Table 4.9. All experiments are conducted on an Intel Core 2 Quad 3.0GHz PC with 16GB main memory running 64-bit Fedora 19.

To mimic a real deployment, we randomly choose a subset of test cases at each iteration of the fine-grained phase according to the data collection strategy proposed in Section 4.4.3.4. Test cases used at different iterations are mostly different. This is quite similar to the practical situation where no two user executions are exactly same. For the coarse-grained phase, the entire test suite is exploited in our experiments.

4.4.4.1 Instrumentation Effort

We take the traditional, non-iterative statistical bug isolation (Liblit *et al.* [58]) as the baseline approach and identify top-k suspicious predictors using the entire test suite. In order to verify the performance of our approach, we measure the total instrumentation effort required, i.e., the percentage of predicates instrumented to obtain the same top-k predictors. The smaller the percentage, the better the performance of our approach is. Here we set k to 1.

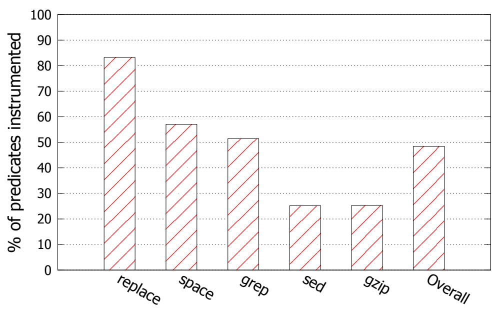


Figure 4.7: Percentage of predicates instrumented

Figure 4.7 plots the percentage of predicates instrumented to find the top predictor for each subject (averaged across all versions). Overall, we can guarantee that more than 50% of instrumentation predicates can be pruned away using our approach even without developers' effort. Our iterative approach performs better for larger programs. For sed and gzip, only about 25% of predicates need to be examined, 75% are skipped. For replace, it is of small size and most of the functions are executed in both failing and passing runs. It is hard to distinguish them based on the coarse-grained information. Therefore, more than 80% of predicates have to be instrumented. In more detail, Table 4.10 shows the average number of iterations required (Column “Iterations”) to find the top predictor and the average number of predicates instrumented during an iteration (Column “Predicates_per”).

Table 4.10: Average number of iterations and average number of predicates instrumented per iteration

Subject	Funs_total	Iterations	Predicates_total	Predicates_per
replace	21	15.9	2,920	157.8
space	136	69.1	25,449	206.0
grep	129	41.3	226,286	2952.5
sed	168	55.6	96,389	532.4
gzip	91	20.8	179,408	2516.8

4.4.4.2 Stability of Results

According to the data collection strategy stated in Section 4.4.3.4, we randomly select a subset of test cases at each iteration. Such a data collection strategy is effective in practice due to a large pool of data available from users. In our experiment though, the number of test cases can be limited, and there might be a challenge to obtain stable experiment result; i.e., our iterative approach might not be able to attain the same top predictor as the baseline approach (i.e., the traditional, non-iterative approach). In this section, we discuss the success rate of our approach in attaining the desired result using the data collection strategy in our experiment.

Table 4.11: Average number of successful runs and average number of test cases used per iteration

Subject	Runs	Tests_total	Tests_per(#)	Tests_per(/)
replace	84.03	5,542	2,993	0.54
space	79.71	1,248	998	0.80
grep	82.50	809	542	0.67
sed	83.06	363	229	0.63
gzip	69.67	213	185	0.87
Overall	79.79	1,635	989	0.70

Specifically, we ran our approach for 100 times and measured how many times our approach can successfully obtain the same top predictor as identified by the baseline approach. Table 4.11 shows the average number of successful runs (Column “Runs”) among all 100 runs, and the average number of test cases used at each iteration (Column “Tests_per(#)”). Column “Tests_per(/)" indicates the ratio of the number of selected tests at each iteration (Column “Tests_per(#)”) to the total number of tests (Column “Tests_total”) we have.

As can be seen, we can get the same top predictor as the baseline approach in most of runs, overall around 80%. The test suite available for gzip is very small and thus it is difficult to achieve the really stable result for each iteration. That is why gzip has a relatively low success rate.

4.4.4.3 Performance Overhead

We compared the user’s time overhead of our (non-sampling) fine-grained iterative instrumentation against that of the sampling scheme [58] with different sampling rates (1/1, 1/100, 1/10000). In addition, we measured the time overhead of our coarse-grained instrumentation scheme (shown as Column “coarse”), where only function entries are instrumented without sampling. We ran each subject four times, ignored the first run and computed an average execution time of the other three runs.

Table 4.12: Time overhead

	Sampling			HI	
Subject	1/1	1/100	1/10000	coarse	fine
replace	0.721	0.690	0.671	0.393	0.540
space	1.617	1.517	1.460	0.294	0.384
grep	26.475	15.954	11.482	0.197	0.395
sed	8.006	5.328	4.628	0.230	0.286
gzip	4.678	2.346	1.745	0.021	0.057
Overall	8.299	5.167	3.997	0.227	0.332

Table 4.10 shows that only a few predicates instrumented during each iteration (shown as Column “Predicates_per”) with respect to the total number of predicates (Column “Predicates_total”) in the program. This provides an empirical evidence that our approach has rather low user’s time overhead. We further validated this expectation by assessing the average execution time for executing the fine-grained instrumented program at each iteration, as well as the execution time for coarse-grained instrumentation in the first phase of our approach. Table 4.12 presents the time overhead for running the instrumented programs of our approach (Column “HI”) and that of the sampling (Column “Sampling”). All the instrumentation in our experiments is performed via *sampler-cc* developed by Liblit *et al.* [57] using two different schemes: sampling and non-sampling.

Overall, our fine-grained iterative instrumentation (Column "fine") suffers a much lower overhead than sampling. Especially for grep, sed and gzip, the overhead of our approach is at least an order of magnitude smaller than that of sampling. For replace, the overhead difference between our instrumentation and the sampling schemes is not so significant as other subjects. This is because the program is of small size and we instrument most of the predicates at each iteration. As for our coarse-grained instrumentation (Column "coarse"), its overhead is even smaller than that of the fine-grained iterative instrumentation.

4.4.4.4 Performance Comparison with Adaptive Bug Isolation

We also compared our approach with adaptive bug isolation [10]. Recall that the adaptive approach requires developers to check and verify the identified predictors for bug at each iteration. This is in stark contrast to our approach, where developers only need to check the top-k predictors *at the end of all iterations*. For instrumentation effort saved, without developers' intervention, the adaptive approach can not save any instrumentation effort. In this sense, our approach (50% saving) definitely outperforms theirs. For performance overhead, the adaptive approach in general performs better than ours. At each iteration, it only instruments a small fraction of predicates on the control flow graph, while our approach instruments all the predicates within one function. As a result, our approach could suffer from higher time overhead than theirs. However, on the other hand, their approach requires running more iterations than ours. This further increases the developers' burden for checking predictors. The debugging process will also take longer time. In fact, the trade-off between the overhead and debugging time can be tuned in our approach by considering additional levels in the HI technique. For example, we can add the block level between function level and predicate level such that only the predicates within a block are instrumented during each iteration. The performance overhead can thus be further reduced. We will expound this idea later in Section 4.5.

4.4.5 Discussion

As an iterative approach, Algorithm 6 needs to run the bug isolation process multiple iterations. Consequently, it relieves users of running instrumented programs with high overhead, but increases the waiting time for results. The productivity trade-off between users and developers is common in approaches that adopt crowd-sourcing. In what follows, we suggest several ways to better balance this trade-off.

Multiple Deployment. Since we have a tremendous number of end-users, we can simultaneously deploy multiple different instrumented programs to different users for execution. As a result, the waiting time for execution data can be significantly reduced. Specifically, we can identify n functions which are of high ranking values and can not be pruned away. We separately instrument the predicates in each of n functions and eventually obtain totally n instrumented programs which are then deployed at the same time. As a consequence, developers can collect execution profiles for debugging with less waiting time.

Multi-function Instrumentation. While only the predicates in one function are instrumented in one instrumented program, it does not prevent us from doing fine-grained instrumentation on multiple functions in one instrumented program. This particularly works well for functions of small size, where such additional performance overhead remains tolerable to users. We can thus reduce the number of iterations. Developers can obtain the final top-k predictors earlier.

4.5 Multiple Levels in Hierarchical Instrumentation

So far, we only consider two levels of instrumentation in the hierarchy of a program: function-level and predicate-level. Our HI technique needs not to be restricted to these two particular levels. According to practical requirements, different levels and different numbers of levels can be taken into account in our technique.

For example, if a program is very large-scale and has a great number of functions, the cost saved from having full-scale function-level instrumentation and analysis may not be sufficient. In this case we can add one more level in the hierarchy, e.g., class-level. We exploit the class-level information to prune away unnecessary instrumentation of functions at the first place. Thus only some of the functions in the whole program are instrumented at the function-level. The gain compared to the two levels will be the saved cost at function level instrumentation and analysis minus the additional cost at class level. Moreover, if a function has a big piece of code, the execution information of this function may not be a sufficiently precise approximation to the information of enclosed predicates. In this case, the pruning effectiveness can be further improved by introducing another level, like block-level. Therefore, more predicates can be pruned away. But notice that we need to pay the extra cost for this added block-level instrumentation and analysis. Similarly, the gain here will be the saved cost at predicate level minus the additional cost at block level.

Specifically, let's assume four levels considered in hierarchical instrumentation: class-level, function-level, block-level, predicate-level. All levels of instrumentation except for the most fine-grained one (predicate-level) are regarded as coarse-grained ($1st$ coarse-grained:class-level, $2nd$ coarse-grained:function-level, and $3rd$ coarse-grained:block-level). We perform the instrumentation level by level from the most coarse-grained to the fine-grained. The coarse-grained pruning and ranking measures presented earlier in Section 4.2 are applied to each coarse-grained level. Algorithms 7 and 8 show the pseudo code for the extended versions of Algorithms 5 and 6 respectively, handling multiple levels. The number of levels considered are specified by a parameter n where $n \geq 2$. Note that the algorithms of multi-level versions are mostly the same as the original versions except recursive procedures: *recBoosting* and *recPruning* in Algorithm 7, *recIML* in Algorithm 8. Once an additional level is involved in the hierarchy, the code fragments: Lines 10-13 and 24-27 in Algorithm 7, Lines 17-26 in Algorithm 8, will be invoked one more time.

Algorithm 7: Predicated Bug Signature Mining via HI (Multi-level)

```
Input: buggy program $G$, test suite $T$, number of signatures mined $k$,
       percentage of elements instrumented for each level boosting $\gamma$, number of
       levels $n$ ($n \geq 2$)
Output: top $k$ suspicious signatures $PS$
Instrument the entire program $G$ at level 1;
Run all the failing and passing tests in $T$ to collect all the profiles $CP$;
$L_1 \leftarrow$ AnalyzeCoarseGrainedProfiles($CP$);
// threshold boosting
boost $\leftarrow$ SelectElementsForBoosting($L_1$, $\gamma$);
$\theta \leftarrow$ recBoosting($2, n, boost, T, k, \gamma$);
// safe pruning
prospect $\leftarrow$ PruneElements($L_1$, $\theta$);
$PS \leftarrow$ recPruning($2, n, prospect, T, k, \theta$);
return $PS$;

Procedure: recBoosting($i, n, boost, T, k, \gamma$)
Input: current level $i$, instrumentation candidate boost, $n$, $T$, $k$, $\gamma$
Output: threshold for pruning $\theta$
if $i < n$ then
    Instrument boost at level $i$;
    Run all the failing and passing tests to in $T$ collect all the profiles $CP$;
    $L_i \leftarrow$ AnalyzeCoarseGrainedProfiles($CP$);
    boost $\leftarrow$ SelectElementsForBoosting($L_i$, $\gamma$);
    return recBoosting($i + 1, n, boost, T, k, \gamma$);
else if $i == n$ then
    Instrument all predicates in boost;
    Run all the failing and passing tests in $T$ to collect all the profiles $BP$;
    $BD \leftarrow$ Preprocess($BP$);
    $BS \leftarrow$ MineBugSignatures($BD$, $k$);
    $\theta \leftarrow k$th top $DS$ value of signatures;
    return $\theta$;
end

Procedure: recPruning($i, n, prospect, T, k, \theta$)
Input: current level $i$, instrumentation candidate prospect, $n$, $T$, $k$, threshold $\theta$
Output: top $k$ suspicious signatures $PS$
if $i < n$ then
    Instrument prospect at level $i$;
    Run all the failing and passing tests in $T$ to collect all the profiles $CP$;
    $L_i \leftarrow$ AnalyzeCoarseGrainedProfiles($CP$);
    prospect $\leftarrow$ PruneElements($L_i$, $\theta$);
    return recPruning($i + 1, n, prospect, T, k, \theta$);
else if $i == n$ then
    Instrument all predicates in prospect;
    Run all the failing and passing tests in $T$ to collect all the profiles $PP$;
    $PD \leftarrow$ Preprocess($PP$);
    $PS \leftarrow$ MineBugSignatures($PD$, $k$);
    return $PS$;
end
```

Algorithm 8: Iterative Statistical Bug Isolation via HI (Multi-level)

```
Input: buggy program $G$, number of predicates returned $k$, number of levels $n$ ($n \geq 2$)
Output: top $k$ suspicious predicates $P_k$
$P_k \leftarrow \emptyset$;
$\theta \leftarrow 0$;
Instrument the entire program $G$ at level 1;
Deploy the instrumented program;
CollectSufficientData();
$L_1 \leftarrow$ AnalyzeCoarseGrainedProfiles();
while $L_1 \neq \emptyset$ do
    $e \leftarrow$ element with the highest $C_r$ value from $L_1$;
    $L_1 \leftarrow L_1 - e$;
    if NecessaryCondition($e, \theta$) is false then
        continue;
    end
    recIML($2, n, e, k, P_k, \theta$);
end
return $P_k$;

Procedure: recIML($i, n, e, k, P_k, \theta$)
Input: current level $i$, instrumentation candidate $e$, $n$, $k$
Update: $P_k$, threshold $\theta$
if $i < n$ then
    Instrument $e$ at level $i$;
    Deploy the instrumented program;
    CollectSufficientData();
    $L_i \leftarrow$ AnalyzeCoarseGrainedProfiles();
    while $L_i \neq \emptyset$ do
        $e \leftarrow$ element with the highest $C_r$ value from $L_i$;
        $L_i \leftarrow L_i - e$;
        if NecessaryCondition($e, \theta$) is false then
            continue;
        end
        recIML($i + 1, n, e, k, P_k, \theta$);
    end
else if $i == n$ then
    Instrument all the predicates in $e$;
    Deploy the instrumented program;
    CollectSufficientData();
    Update($P_k$, predictors in $e$);
    $\theta \leftarrow$ getTopKthImportance($P_k$);
end
return;
```

4.6 Related Work

There are quite a few related studies which refine the data under analysis for debugging by performing selective instrumentation. HOLMES [16] and Adaptive Bug Isolation [10] are the most related work to our iterative approach presented in Section 4.4. HOLMES employs an iterative, bug-directed instrumentation scheme to isolate the causes of failures. Specifically, at each iteration, it monitors a set of functions, branches and paths to analyze whether they are strong predictors of the failure. If so, it terminates by returning these strong predictors. Otherwise, it expands the search via a static analysis to monitor other parts of code that closely interact with the weak predictors. Similarly, adaptive bug isolation adopts an adaptive monitoring strategy based on the following principle of locality [10]: *If a predicate is highly predictive of failure, then predicates in its vicinity are potentially good bug predictors as well.* It monitors a few predicates at each iteration and adaptively adjusts the instrumentation plan to include predicates close to the highly suspicious predicate currently explored. This iterative process proceeds until the bug cause is found.

Both studies perform selective instrumentation in the sense that only partial program code is instrumented at each iteration. However, their performance strongly relies on the principle of locality. They also need developers' intervention to terminate the process. In contrast, we devise hierarchical instrumentation and exploit the coarse-grained information to safely and effectively prune away unnecessary fine-grained instrumentation in an automatic way.

4.7 Chapter Summary

We devise a novel hierarchical instrumentation technique performing selective instrumentation to refine the execution data collected and analyzed, finally enhancing the efficiency of statistical debugging. We apply the HI technique to two different types of statistical debugging approaches: in-house debugging and cooperative debugging for field failures. The results reveal that our technique safely and effectively prunes away unnecessary instrumentation. The execution profiles are significantly refined, and therefore the efficiency of statistical debugging is improved. Specifically, we first employ the HI technique to predicated bug signature mining (MPS) and propose an approach called HIMPS. The empirical study shows that HIMPS can achieve around 40% to 60% saving in time and space compared with MPS while discovering the same top-k bug signatures. Secondly, we investigate the adoption of HI to cooperative bug isolation for field failures and propose an iterative approach via HI. The experimental results validate that our approach not only greatly saves the total monitoring and computational costs, but also sharply reduces the end-user's runtime overhead.

Chapter 5

Conclusion

For the past two decades, mining software behavior has been well investigated to assist in various software engineering tasks. Two topics which received much attention are specification mining and statistical debugging.

Among the execution data analyzed by both specification mining and statistical debugging, there exist a significant number of useless elements. Mining directly over the raw execution data wastes a great amount of computing resources and possibly produces unexpected results due to the redundant and meaningless elements. This severely undermine the efficiency and effectiveness of software behavior mining. In this dissertation, we investigate, explore and validate the thesis statement that: *the efficiency and effectiveness of software behavior mining can be significantly improved by systematically refining the software execution data under analysis.* We devise the specific refinement technique for each of two studies to refine the data collected and analyzed, thus improving the efficiency and effectiveness of software behavior mining.

In the following, we summarize our research discussed above by giving the main contributions, and talk about the potential future work.

5.1 Summary and Contributions

We have the following contributions for specification mining.

Semantics-directed Specification Mining. We propose a *semantics-directed specification mining* framework which injects semantic information into mining. We propose the respective semantic analysis to extract semantically relevant sequences from execution traces. We then perform frequent pattern mining on these sequences to generate semantically significant specifications. Since all semantically irrelevant events are pruned away through the semantic analysis, the mined specifications are all semantically significant, and mining becomes more efficient.

Dataflow Sensitive Specification Mining. Based on the semantics-directed specification mining framework, we present a particular *dataflow sensitive specification mining* system where dataflow semantics is taken into consideration. We present a *dynamic, inter-procedural dataflow tracking analysis* to extract all the dataflow related sequences from execution traces. A novel Apriori-like *constrained iterative pattern mining* algorithm is particularly developed to discover frequent patterns from a set of dataflow related sequences. We conduct experiments on five real-world subjects using our implemented prototype. The results show that our approach produces high-quality semantically significant specifications and scales to large real-world programs.

For statistical debugging, the contributions are as follows.

Hierarchical Instrumentation. We devise a novel *hierarchical instrumentation* (HI) technique to refine the execution data by pruning away unnecessary instrumentation. In brief, we first perform a lightweight coarse-grained instrumentation and obtain the execution information of coarse-grained elements (e.g., functions). By means of such coarse-grained execution information, we safely and effectively prune away instrumentation of fine-grained elements (e.g., predicates). Therefore, the execution traces collected and analyzed are significantly refined. We formalize the underlying principles of the hierarchical instrumentation technique and provide a general and systematic approach to applying it. Various statistical debugging approaches can thus benefit from this technique.

Efficient Predicated Bug Signature Mining via HI. We propose HIMPS, an efficient predicated bug signature mining via HI to safely and effectively prune away considerable unnecessary predicates for instrumentation. We conduct experiments to compare HIMPS against the MPS system developed in [96]. The experiments validate that HIMPS can achieve around 40% to 60% saving in disk storage space usage, time and peak memory consumption compared with MPS without jeopardizing the effectiveness of mining for top-ranked signatures.

Iterative Statistical Bug Isolation via HI. We propose an iterative statistical bug isolation approach for field failures via HI. We iteratively perform fine-grained instrumentation as guided by the coarse-grained information, and dynamically prune unnecessary instrumentation away. We conduct the experiments on real-world programs. The experimental results validate that our approach not only saves the total monitoring and computational costs, but also sharply diminishes the performance overhead for end-users.

5.2 Future Work

For specification mining, we talk about the following potential directions for investigation.

Semantics & Specification Formalisms. We would like to introduce other semantic relations among events such as calling relation [35], object collaborative relation [87], and resource acquire-release relation [109], into our semantics-directed specification mining framework just like the dataflow semantics we have studied. Based on the user-specific semantics, we develop the respective semantics analysis to filter our irrelevant data from execution traces, and then mine the semantically significant specifications with respect to the specific semantics. Moreover, apart from the iterative pattern, we are also interested in mining other formalisms of specifications, such as graphs. Specifically, we will construct a set of graphs capturing certain semantic relation among events. We then employ frequent subgraph mining algorithm [112] to discover frequent subgraphs as specifications.

At the technical level, while we have provided a framework for introducing semantic information into the specification mining process, we have not considered how such introduction of semantic information can be fully automated. To this end, we can consider the design of a domain specific language for describing (1) what semantic information should be introduced, and (2) where and how they should be introduced, and collected during program execution. Moreover, we will also investigate the application of program transformation techniques, such as fusion and partial evaluation [17, 49], to automatically merge the domain-specific program with the specification miner to obtain an automated semantics-based specification miner.

Value-based Invariants. As an extension, we also plan to integrate value-based invariants with our iterative patterns [69]. The value-based invariants [27] capture the properties among variables. They can enrich our event-based iterative patterns. Therefore, the final specifications will be more expressive and can be used in more applications. Specifically, we can extract the runtime variable values which are relevant to the events in the mined patterns. Such variable information forms execution traces. The invariant inference algorithm [29] can then be employed to detect value-based invariants from these traces. Finally, we produce the event-based patterns enriched by value-based invariants as the final specifications.

For statistical debugging, we would like to investigate the following potential studies in the future.

Semantic Hierarchy. Our HI technique is currently dependent on the syntactic hierarchy of program source code. On the basis of the syntactic containment relation between program elements (e.g., functions and the predicates contained within), we exploit the execution information of syntactic coarse-grained elements (e.g., functions) to guide the instrumentation of fine-grained elements (e.g., predicates). In future, we plan to investigate the semantic hierarchy. For example, complex predicates can be constructed to form a hierarchy by means of logical operators. In this sense, a path condition [51] can be regarded as a semantic coarse-grained element of branch conditionals. Program slices [107] are also a kind of semantic coarse-grained elements of code statements. Similar to the syntactic hierarchy, we can also utilize various program semantic hierarchies in our HI technique.

Automated Debugging for Performance Problems. As we mentioned earlier, automated debugging has been studied for decades, for sequential bugs [47, 58] and concurrency bugs [84], in the context of one version program [115] or multiple versions [88]. Most of these studies focus on the *functional bugs* which refer to the bugs leading to functional misbehavior (e.g., incorrect outputs, crashes, and exceptions). Quite recently, several studies on debugging performance problems (i.e., performance degradation and energy waste [45, 61]) have been investigated [38, 95, 113]. These approaches achieve some useful results, but are far away from maturity. Profilers [1, 2] as the most commonly used and available tool for diagnosis, can tell where and how many computation resources are spent, but not whether or why they are wasted. They still require enormous manual effort for performance debugging. Several non-profiling, trace analysis debugging approaches have been recently proposed [38, 113]. However, all of them aim to identify the performance-causality relation among system events and components, none on fine-grained source code level. Song and Lu [95] applied predicate-based statistical debugging (originally proposed for functional bugs [58]) to performance problems. It isolates predicates highly correlated to the root cause, but it is still difficult for developers to locate and understand the bug by only examining the single predicate in isolation [85]. To better support debugging for performance problems, we would like to explore the combination of both profiling [98] and contrast data mining [26]. Specifically, we can represent the program behavior as control flow graph annotated with runtime information (e.g., execution cost) via profiling tools and then adopt contrast data mining techniques to identify the discriminative patterns, which aid in localizing and understanding the root cause of performance problems. Our HI technique can be applied to achieve better scalability.

Bibliography

[1] `http://sourceware.org/binutils/docs-2.21/gprof/`.

[2] `http://oprofile.sourceforge.net`.

[3] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[5] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 3–14, Washington, DC, USA, 1995. IEEE Computer Society.

[6] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

[7] G. Argyrous. *Statistics for Research: With a Guide to SPSS*. SAGE Publications Ltd; Third Edition edition (February 9, 2011), 2011.

[8] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA. ACM.

[9] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 5–15, New York, NY, USA, 2007. ACM.

[10] P. Arumuga Nainar and B. Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 255–264, New York, NY, USA. ACM.

[11] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 73–84, New York, NY, USA, 2010. ACM.

[12] G. K. Baah, A. Podgurski, and M. J. Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *SIGSOFT FSE*, pages 146–156, 2011.

[13] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. ACM.

[14] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: A new approach to revealing neglected conditions in software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 163–173, New York, NY, USA, 2007. ACM.

[15] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 141–152, New York, NY, USA, 2009. ACM.

[16] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44. IEEE Computer Society, 2009.

[17] W.-N. Chin. Towards an automated tupling strategy. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '93, pages 119–132, New York, NY, USA, 1993. ACM.

[18] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

[19] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[20] J. E. Cook and A. L. Wolf. Automating process discovery through event-data analysis. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 73–82, New York, NY, USA, 1995. ACM.

[21] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.

[22] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. Technical report, Jerusalem, Israel, Israel, 1998.

[24] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *ISSTA*, pages 233–244, 2006.

[25] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[26] G. Dong and J. Bailey, editors. *Contrast Data Mining: Concepts, Algorithms, and Applications.* CRC Press, 2013.

[27] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[28] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 27:213–224, 2001.

[29] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 449–458, New York, NY, USA, 2000. ACM.

[30] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, Dec. 2007.

[31] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International*

Symposium on Foundations of software engineering, SIGSOFT '08/FSE-16, pages 339–349, New York, NY, USA, 2008. ACM.

[32] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 51–60, New York, NY, USA, 2008. ACM.

[33] R. Gore, P. F. Reynolds, and D. Kamensky. Statistical debugging with elastic predicates. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE'11, pages 492–495, Washington, DC, USA. IEEE Computer Society.

[34] R. Gore and P. F. Reynolds, Jr. Reducing confounding bias in predicate-level statistical debugging metrics. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 463–473, Piscataway, NJ, USA. IEEE Press.

[35] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, Nov. 2001.

[36] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272, 2005.

[37] J. Han. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[38] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 145–155, Piscataway, NJ, USA, 2012. IEEE Press.

[39] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 60–71, Washington, DC, USA, 2003. IEEE Computer Society.

[40] D. Harel and S. Maoz. Assert and negate revisited: modal semantics for uml sequence diagrams. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, SCESM '06, pages 13–20, New York, NY, USA, 2006. ACM.

[41] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[42] H.-Y. Hsu, J. A. Jones, and A. Orso. Rapid: Identifying bug signatures to support debugging activities. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 439–442. IEEE Computer Society, 2008.

[43] ITU-T. Itu-t recommendation z.120: Message sequence chart (msc). 1999.

[44] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 184–193, New York, NY, USA. ACM.

[45] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[46] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 474–484, Piscataway, NJ, USA, 2012. IEEE Press.

[47] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282. ACM, 2005.

[48] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA. ACM.

[49] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[50] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[51] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[52] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps. Temporal logic for scenario-based specifications. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 445–460, Berlin, Heidelberg, 2005. Springer-Verlag.

[53] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Mining message sequence graphs. In *ICSE*, pages 91–100, 2011.

[54] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 591–600, New York, NY, USA, 2011. ACM.

[55] J. Li, H. Li, L. Wong, J. Pei, and G. Dong. Minimum description length principle: Generators are preferable to closed patterns. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 409–414. AAAI Press, 2006.

[56] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 306–315, New York, NY, USA, 2005. ACM.

[57] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 141–154, New York, NY, USA. ACM.

[58] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 15–26, New York, NY, USA. ACM.

[59] B. R. Liblit. *Cooperative Bug Isolation.* PhD thesis, University of California, Berkeley, Dec. 2004.

[60] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *Proceedings of the 2005 Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2005, pages 286–295, 2005.

[61] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[62] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM.

[63] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *KDD*, pages 557–566, 2009.

[64] D. Lo and S.-C. Khoo. Smartic: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 265–275, New York, NY, USA, 2006. ACM.

[65] D. Lo, S.-C. Khoo, J. Han, and C. Liu. *Mining Software Specifications: Methodologies and Applications.* CRC Press; 1 edition (May 24, 2011), 2011.

[66] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 460–469, New York, NY, USA, 2007. ACM.

[67] D. Lo, S.-C. Khoo, and C. Liu. Mining temporal rules for software maintenance. *J. Softw. Maint. Evol.*, 20(4):227–247, July 2008.

[68] D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. In *ASE*, pages 387–396, 2010.

[69] D. Lo and S. Maoz. Scenario-based and value-based specification mining: better together. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 387–396, New York, NY, USA, 2010. ACM.

[70] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 501–510, New York, NY, USA, 2008. ACM.

[71] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[72] J. R. Lyle and W. M. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computer and Applications*, pages 877–883, 1987.

[73] R. D. Mason, D. A. Lind, and W. G. Marcha. *Statistics: An Introduction.* Duxbury Press, 5 Sub edition (1998), 1998.

[74] F. Masseglia, P. Poncelet, and M. Teisseire. Incremental mining of sequential patterns in large databases. *Data Knowl. Eng.*, 46(1):97–121, July 2003.

[75] M. Mendonca and N. L. Sunderhaft. Mining software engineering data: A survey, 1999.

[76] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 167–176, New York, NY, USA, 2000. ACM.

[77] S. L. Morgan and C. Winship. *Counterfactuals and Causal Inference: Methods and Principles for Social Research*. Cambridge University Press; 1 edition, 2007.

[78] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[79] A. C. Nguyen and S. Khoo. Extracting significant specifications from mining through mutation testing. In *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, pages 472–488, 2011.

[80] A. C. Nguyen and S. Khoo. Discovering complete API rules with mutation testing. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pages 151–160, 2012.

[81] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[82] K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. Softw. Eng.*, 16:268–280, March 1990.

[83] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag.

[84] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

[85] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA. ACM.

[86] J. Pearl. *Causality: Models, Reasoning, and Inference.* Cambridge University Press, 2000.

[87] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[88] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach for debugging evolving programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 33–42, New York, NY, USA, 2009. ACM.

[89] Y. Qi, X. Mao, and Y. Lei. Making automatic repair for large-scale programs more efficient using weak recompilation. In *ICSM*, pages 254–263. IEEE Computer Society, 2012.

[90] J. R. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[91] V.-R. Raja. Soot: A java bytecode optimization framework. Master's thesis, School of Computer Science, McGill University, Montreal, 2000.

[92] E. Renieris. *A research framework for software-fault localization tools.* PhD thesis, Providence, RI, USA, 2005. AAI3174662.

[93] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 30–39, 2003.

[94] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 56–66, Washington, DC, USA, 2009. IEEE Computer Society.

[95] L. Song and S. Lu. Statistical debugging for real-world performance problems. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, New York, NY, USA, 2013. ACM.

[96] C. Sun and S.-C. Khoo. Mining succinct predicated bug signatures. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 576–586, New York, NY, USA, 2013. ACM.

[97] Q. Taylor and C. Giraud-Carrier. Applications of data mining in software engineering. *Int. J. Data Anal. Tech. Strateg.*, 2(3):243–257, July 2010.

[98] J. Thiel. An overview of software performance analysis tools and techniques: from gprof to dtrace, 2006. `http://www1.cse.wustl.edu/~jain/cse567-06/ftp/sw_monitors1/index.html`.

[99] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[100] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 283–294, Washington, DC, USA, 2009. IEEE Computer Society.

[101] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 496–506, Washington, DC, USA, 2009. IEEE Computer Society.

[102] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[103] I. Vessey. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Trans. Syst. Man Cybern.*, 16(5):621–637, Sept. 1986.

[104] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 79–, Washington, DC, USA, 2004. IEEE Computer Society.

[105] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 35–44, New York, NY, USA, 2007. ACM.

[106] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[107] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[108] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, July 1982.

[109] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Iterative mining of resource-releasing specifications. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 233–242, 2011.

[110] T. Xie and J. Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 54–57, New York, NY, USA, 2006. ACM.

[111] T. Xie, J. Pei, and A. E. Hassan. Mining software engineering data. In *ICSE Companion*, pages 172–173, 2007.

[112] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 433–444, New York, NY, USA, 2008. ACM.

[113] X. Yu, S. Han, D. Zhang, and T. Xie. Comprehending performance from real-world execution traces: A device-driver case. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 193–206, New York, NY, USA, 2014. ACM.

[114] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC / SIGSOFT FSE*, pages 253–267, 1999.

[115] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '02, pages 1–10, 2002.

[116] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[117] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281, 2006.

[118] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.

[119] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADEBUG*, pages 33–42, 2005.

[120] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 1105–1112. ACM, 2006.

[121] Z. Zuo. Efficient statistical debugging via hierarchical instrumentation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 457–460, New York, NY, USA, 2014. ACM.

[122] Z. Zuo and S.-C. Khoo. Mining dataflow sensitive specifications. In *Proceedings of the 2013 International Conference on Formal Engineering Methods*, ICFEM '13, pages 36–52, 2013.

[123] Z. Zuo and S.-C. Khoo. Iterative statistical bug isolation via hierarchical instrumentation. Technical Report TRC7/14, School of Computing, National University of Singapore, July 2014. `https://dl.comp.nus.edu.sg/jspui/handle/1900.100/4666`.

[124] Z. Zuo, S.-C. Khoo, and C. Sun. Efficient predicated bug signature mining via hierarchical instrumentation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 215–224, New York, NY, USA, 2014. ACM.

Appendices

Appendix A

Complete Scoped Dataflow Tracking Analysis

Algorithm 9: Complete Scoped Dataflow Tracking Analysis

```
Data: trace $T$
Result: output all the maximum dataflow related sequences
foreach statement $s$ in chronological order in trace $T$ do
    while peek($Stack$).$methodSignature \neq s.methodSignature$ do
        if $DeclarationStmt(s)$ then
            $S \leftarrow \emptyset$;
            $S^* \leftarrow \emptyset$;
            push(($S, S^*$), $Stack$);
        else
            $(S_u, {S_u}^*) \leftarrow$ pop($Stack$);
            foreach $t(v_s, L, v_c) \in S_u$ do
                if isComplete($t$) then output $L$;
            end
        end
        if peek($Stack$).$isDeclaration = false$ then
            $(S_d, {S_d}^*) \leftarrow$ peek($Stack$);
            KillAndGen($S_u, S_d, {S_u}^*, {S_d}^*, throw$);
        end
    end
    switch $s$ do
        case $InvokeStmt(s)$
            $S \leftarrow \emptyset$;
            $S^* \leftarrow \emptyset$;
            push(($S, S^*$), $Stack$);
            break;
        case $ReturnStmt(s)$
            $(S_u, {S_u}^*) \leftarrow$ pop($Stack$);
            for each $t(v_s, L, v_c) \in S_u$ do
                if isComplete($t$) then output $L$;
            end
            if peek($Stack$).$isDeclaration = false$ then
                $(S_d, {S_d}^*) \leftarrow$ peek($Stack$);
                KillAndGen($S_u, S_d, {S_u}^*, {S_d}^*, s$);
            end
            break;
        case $IdentityStmt(s)$
            $(S_u, {S_u}^*) \leftarrow$ peek2nd($Stack$);
            $(S_d, {S_d}^*) \leftarrow$ peek($Stack$);
            KillAndGen($S_u, S_d, {S_u}^*, {S_d}^*, s$);
            break;
        case $AssignStmt(s)$
            $(S_u, {S_u}^*) \leftarrow$ collapse($Stack$);
            $(S_d, {S_d}^*) \leftarrow$ peek($Stack$);
            KillAndGen($S_u, S_d, {S_u}^*, {S_d}^*, s$);
        end
    endsw
end
```

Algorithm 10: KillAndGen($S_u, S_d, {S_u}^*, {S_d}^*, s$)

```
$Pairs \leftarrow$ get_UD_Pairs($s, {S_u}^*$);
foreach use-def pair $p(v_u, v_d) \in Pairs$ do
    $GS \leftarrow \emptyset$;
    if $v_u$ is a constant or a new instance then
        $L' \leftarrow [\ ]$;
        if $\exists$ event $e$ associated with $s$, $v_u \in A(e)$ then
            $L' \leftarrow L' +\!\!+ [e]$;
        end
        $GS \leftarrow GS \cup \{(v_u, L', v_d)\}$;
    else
        foreach $t'(v_s, L, v_u) \in S_u$ do
            mark $t'$ as incomplete;
            $L' \leftarrow L$;
            if $\exists$ event $e$ associated with $s$, $v_u \in A(e)$ then
                $L' \leftarrow L' +\!\!+ [e]$;
            end
            $GS \leftarrow GS \cup \{(v_s, L', v_d)\}$;
        end
    end
    foreach $t(v_*, L_*, v_d) \in S_d$ do
        if isComplete($t$) then output $L_*$;
        $S_d \leftarrow S_d - \{t(v_*, L_*, v_d)\}$;
    end
    $S_d \leftarrow S_d \cup GS$;
    // dynamic alias tracking analysis
    if isAliasingType($p(v_u, v_d)$) then
        $GS^* \leftarrow \emptyset$;
        if $v_u$ is a constant or a new instance then
            $GS^* \leftarrow GS^* \cup \{(v_u, v_d)\}$;
        else
            $GS^* \leftarrow GS^* \cup \{(v_s, v_d) | (v_s, v_u) \in {S_u}^*\}$;
        end
        foreach $(v_*, v_d) \in {S_d}^*$ do
            ${S_d}^* \leftarrow {S_d}^* - \{(v_*, v_d)\}$;
        end
        ${S_d}^* \leftarrow {S_d}^* \cup GS^*$;
    end
end
```

Appendix B

Proof of Apriori Property

Definition 9 (**Closure Subpattern**). *Given a pattern p^k, a subpattern p^{k-1} is its closure subpattern iff for each $inst(p^k)$, there exists a subsequence of $inst(p^k)$, which is an instance of p^{k-1}, $inst(p^{k-1})$.*

Lemma 1 (**Closure Subpattern Lemma**). *Given a (constrained) iterative pattern p^k, its prefix_pattern pre_p^{k-1}, suffix_pattern suf_p^{k-1} and all infix_patterns in_p^{k-1} are all closure subpatterns of p^k.*

Proof. Given a trace T and its event list $L_{(T)}$, a pattern p^k ($\langle e_1, e_2, \ldots, e_k \rangle$), and any one of its constrained iterative pattern instances $inst(p^k)$ ($\langle o_1, o_2, \ldots, o_k \rangle$). Firstly, we can prove that the subsequence ($\langle o_1, o_2, \ldots, o_{k-1} \rangle$) is an instance of the prefix_pattern pre_p^{k-1}. Since $\forall q \in [1,k], L_{(T)}(o_q) = e_q$ and $\forall i \in [1,k-1] \left(\forall j \in (o_i, o_{i+1}), L_{(T)}(j) \notin p^k\right)$ hold according to the definition 4, we can easily reach $\forall q \in [1,k-1], L_{(T)}(o_q) = e_q$ and $\forall i \in [1,k-2]$ ($\forall j \in (o_i, o_{i+1})$, $L_{(T)}(j) \notin pre\_p^{k-1}$). Similarly, it can be proved that the subsequence ($\langle o_2, o_3, \ldots, o_k \rangle$) is an instance of the suffix_pattern suf_p^{k-1}. Next, consider an infix_pattern in_p^{k-1} ($\langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_k \rangle$). We will prove that the subsequence ($\langle o_1, \ldots, o_{i-1}, o_{i+1}, \ldots, o_k \rangle$) is an instance of in_p^{k-1} where $L_{(T)}(o_i) = e_i$. Since the only change is the absence of o_i, we just need to prove that $L_{(T)}(o_i) \notin in\_p^{k-1}$. According to the definition of the infix_pattern, $i \in [2, k-1]$ and $e_i \notin in\_p^{k-1}$ hold. Moreover, $L_{(T)}(o_i) = e_i$. We can conclude that $L_{(T)}(o_i) \notin in\_p^{k-1}$. All in all, we proved that the prefix_pattern, suffix_pattern and all infix_patterns are closure subpatterns. $\square$

Theorem 6 (**Downward Closure Property**). *If a pattern p^k is frequent, then all of its closure subpatterns c_p^{k-1} are frequent.*

Proof. Without loss of generality, consider one closure subpattern c_p^{k-1} of the given pattern p^k. According to the definition of closure subpatterns, each instance of p^k corresponds to an instance of the closure subpattern c_p^{k-1}. Therefore, it is easy to conclude that the support of p^k is not greater than the support of c_p^{k-1}, that is $sup(c\_p^{k-1}) \geq sup(p^k)$. Note that if p^k is frequent, $sup(p^k) \geq min\_sup$, thus $sup(c\_p^{k-1}) \geq min\_sup$, the closure subpattern c_p^{k-1} is frequent. $\square$

Appendix C

Proof of Pattern Preservation

The following provides the proof of Theorem 3.

Proof. Since $\mathcal{D}'$ is the projected database from $\mathcal{D}$ wrt. $\mathcal{I}'$, we can derive that $\forall i \in [1,n], c'_i = c_i \wedge T'_i = T_i \cap \mathcal{I}'$ according to Definition 8.

Given an pattern $P \subseteq \mathcal{I}'$ and $\forall i \in [1,n], T'_i = T_i \cap \mathcal{I}'$, we have the following deduction:

$$\forall i \in [1,n], P \subseteq T'_i \tag{C.1}$$

$$\iff \forall i \in [1,n], P \subseteq T_i \cap \mathcal{I}' \tag{C.2}$$

$$\iff \forall i \in [1,n], P \subseteq T_i \wedge P \subseteq \mathcal{I}' \tag{C.3}$$

$$\iff \forall i \in [1,n], P \subseteq T_i \tag{C.4}$$

Thus we proved that $\forall i \in [1,n], P \subseteq T'_i \iff P \subseteq T_i$.

Further, recall that the positive support of P *wrt.* an itemset database $\mathcal{D}'$, $sup^+(P,\mathcal{D}') = |td^+(P,\mathcal{D}')|$ where $td^+(P,\mathcal{D}') = \{(T',c') \in \mathcal{D}' | P \subseteq T' \wedge c' = +)\}$. Since $\forall i \in [1,n], P \subseteq T'_i \iff P \subseteq T_i$ as proved above and given $\forall i \in [1,n], c'_i = c_i$, we can derive that $td^+(P,\mathcal{D}') = td^+(P,\mathcal{D})$ where $P \subseteq \mathcal{I}'$. Therefore, $sup^+(P,\mathcal{D}') = |td^+(P,\mathcal{D})| = sup^+(P,\mathcal{D})$. Similarly, we can get $sup^-(P,\mathcal{D}') = sup^-(P,\mathcal{D})$. According to Equation 4.5, $DS(sup^+(P,\mathcal{D}'), sup^-(P,\mathcal{D}'))$ will be equal to $DS(sup^+(P,\mathcal{D}), sup^-(P,\mathcal{D}))$. $\square$