Specification Mining: Methodologies, Theories and Applications

David Lo

(B.Eng. (Hons), Nanyang Technological University, Singapore)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2008

Specification Mining: Methodologies, Theories and Applications

Approved by:

A/P Siau-Cheng Khoo, Advisor

A/P Wei-Ngan Chin

A/P Stan Jarzabek

External Referee: Prof. Jiawei Han

Date Approved

ACKNOWLEDGEMENTS

I am grateful to my advisor, A/P Siau-Cheng Khoo, for his guidance, help, advice and encouragement. I would also like to thank the thesis committee members, A/P Wei-Ngan Chin, A/P Stan Jarzabek and Prof. Jiawei Han, for their advice, help and feedback. Many thanks to the co-authors of the papers I have written as part of this thesis: Dr. Chao Liu (UIUC; Microsoft Research, Redmond) and Shahar Maoz (The Weizmann Institute of Science, Israel) for their help and advice.

It has been a good experience to work with various members of Programming Language and Systems Lab. Their support and help is much appreciated. Also, this thesis work is built upon many things taught by various lecturers of both graduate and undergraduate courses that I have taken.

I would like to thank my parents and sister for their continual patience, support and understanding. Thank you for listening to me when I have been down and for giving practical advice.

The author would also like to thank anonymous reviewers of: ICSE'06, ASE'06, WCRE'06, FSE'06, ICDE'07, DASFAA'07, PODS'07, SIGMOD'07, AAAI'07, KDD'07, WCRE'07, ICSM'07, FSE-DS'07, PLDI-SRC'07, PCODA'07, CIKM'07, NGDM'07, OOPSLA-Poster'07, ASE'07, VMCAI'08, ICDE'08, DASFAA'08 and Encyclopedia of Data Warehousing and Mining for their valuable feedback.

Last but not least, the author would like to thank the following researchers: Dr. Glenn Ammons, Dr. Hugh Anderson, Dr. Peter Andreae, Prof. David Harel, A/P Jinyan Li, Prof. Beng Chin Ooi, Prof. Jon D. Patrick, Dr. Anand Raman, A/P Jianyong Wang and Prof. Limsoon Wong for their help, comments, advice and feedback.

TABLE OF CONTENTS

SUMMARY

Due to hard deadlines and short time-to-market requirements, software products often come with poor, incomplete, and sometimes without, documented specifications. This situation is further aggravated by software evolution – as software evolves, often the documented specification is not updated. This might render the original documented specification of little use after several cycles of program evolution. Hence, there is a need for tools to automatically mine specifications from programs.

To ensure the correctness of a software system, program verification tools have been proposed and shown useful. Program verification tools check for manually specified properties or specifications to find bugs and violations in a program. However, the difficulty and programmers' reluctance in writing formal specifications pose a challenge yet to be overcome. Automating the specification creation process can help to leverage the applications of formal verification tools further.

In this dissertation, we describe theories, methodologies and applications of mining expressive software specifications from program execution traces. By observing program execution traces, specifications in the formats of automata, frequent behavioral patterns, temporal rules expressed in Linear Temporal Logic (LTL) and Live Sequence Chart (LSC) can be mined. Our goal is to improve automation, accuracy and efficiency of mining processes. We build the work from the ground up by first describing evaluation measures, followed by properties and theorems, methodologies and finally applications of mined specifications. Mined specifications are useful to aid program understanding, reduce the cost of software maintenance and provide the set of formal specifications for program verification tools.

This work builds on the synergy of concepts and techniques from several domains of computer science including software engineering, programming languages, data mining, and machine learning. Some techniques are also adopted from simulation and modeling and bioinformatics. A multi-disciplinary approach enables one to use the best tool for the job and even overcome difficult obstacles in a particular domain.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

THESIS

My thesis is as follows:

> Expressive software specifications in diversified formats can be extracted with improved automation, accuracy and scalability from program execution traces.

In this dissertation, new approaches to mine software specifications are presented. We mine different target specifications including automata [112, 113], frequent software behavioral patterns [118], temporal rules expressible in Linear Temporal Logic (LTL) [111, 119, 120], and Live Sequence Charts (LSCs) [122, 123]. In addition to addressing novel mining problems, we expand the boundaries of existing research in specification mining by improving expressiveness and accuracy of mined specification, also automation and scalability of existing mining processes.

CHAPTER II

INTRODUCTION

Documented software specification is an important piece of program artifacts. It states how a software is supposed to behave. With a clear, complete and proper specification, a piece of software can be developed well and maintained easily. Specifications can also be used as input to formal program verification tools to discover bugs or converted to runtime monitors to detect for violations of specifications or properties of interests during program execution. Hence, it is best if all programs and software projects are developed with clear, precise and well-documented specifications.

However, often documented specifications are inadequate or lacking in the industry. This problem can be solved by specification mining, which is an automated process to extract specification from a program.

2.1 The Specification Problem and Specification Mining

Due to hard deadlines, short time-to-market requirements [25] and emphasis on measuring productivity in terms of lines of code (LOC) written [18] or projects completed, software systems are often developed without clear, complete or proper documented specifications. The motto 'specify well first, code later' is often replaced by 'release as fast as possible, patch later'. This is well supported if we note the frequent releases of patches in the software industry.

Software is 'fluid' in nature: it changes often [17]. Different from many other engineering products that rarely changes after it is completed, software is likely to change frequently. A software project often involves a number of stakeholders [144, 135] and is affected by changes in market, business process, hardware, operating system, management, etc. These factors contribute to the diverseness of changes software experienced. The phenomenon of frequent changes of software has been well studied under the umbrella notion of software evolution [104]. Software evolution adds additional strain to the accuracy and relevancy of program specifications. As a system changes, often the original documented specification remains unchanged [40]. Results of multiple small changes can render the original specification invalid or even misguiding.

The above factors have contributed to high software maintenance cost. Software maintenance refers to the process of adding new features, fixing bugs and making changes to a software system after the initial development process has been completed. It has been investigated that up to 90% of total software cost is due to maintenance [50] and up to 50% of the maintenance cost is due to effort in comprehending or understanding the existing code base before changes are made [53, 158, 24]. Without properly documented specification, it is hard to understand and maintain an existing code base. This is especially true for software projects developed by many developers over a long period of time.

On another angle, to ensure software reliability, program verification and runtime monitoring tools, e.g., [29, 26], have been proposed and shown useful in many cases. Program verification and runtime monitoring tools accept specification in the form of a set of formal properties and check for their violations in a software system. However, the difficulty in formulating a set of formal properties has been a barrier to its wide-spread industrial application [7]. Adding software evolution to the equation, the verification process is further strained. First, ensuring correctness of a program as changes are made is not a trivial task: a change in one part of a program, might induce unwanted effects resulting in bugs in other parts of the program. Furthermore, as a system changes and features are added, there is a constant need to add new properties or modify outdated ones to render automated verification techniques effective in detecting bugs and ensuring the correctness of the system.

Addressing the above problems, there is a need for solutions to tackle or alleviate the impact of the lack of, or outdated specifications. Removing or alleviating the problems can result in more well-specified programs, aid software comprehension, reduce software maintenance cost and improve program reliability.

To tackle the problem of inadequate or lack of documented specification, one needs to look into the 'root' of the problem. Software developers are often pressured by time. Many (if not most) projects run late or over budgets [159]. Software developers are under continuous pressures to meet deadlines. The competition among software vendors is getting tighter due to globalization. These pressures result in reluctance of writing proper or even formal specifications.

To solve or alleviate the specification problem, techniques that can reduce the burden of software developers in formulating proper specification are needed. A viable solution is an automated or semi-automated method in extracting various specifications from either software traces or software code. These specifications can later be analyzed for various purposes: program understanding, bug and anomaly detection, verification, security threats detection and mitigation, and many more. In the literature, this automated or semi-automated process is often referred to as 'Specification Mining' [7].

2.2 Our Approach and Contributions

Software specifications can be mined from either code (i.e., static analysis) or traces (i.e., dynamic analysis). We focus our work on mining specifications from program execution traces.

Static and dynamic analysis complement each other and each has its own benefits. With dynamic analysis, run-time information, e.g., user inputs, environments, object instantiation, pointer points-to information, information on actual method called when polymorphism and dynamic linking is involved, etc., is available for analysis. Also, often third party vendors only have their binary code released to users [131]. In the literature, currently there are only a few studies working on static analysis on binary code. Furthermore, dynamic analysis avoids the problem of infeasible paths [139, 162, 14], pointer analysis [171], etc. which are still under active investigation. Another benefit is that dynamic analysis methods are portable across various programming languages. One of the drawbacks of dynamic analysis is that the quality of the analysis depends on the quality of the input trace set. Also, for most cases, it is not feasible to generate traces for all possible paths in a program. It is best if static and dynamic analysis work hand in hand. Later in this dissertation, we show how specifications mined using dynamic analysis can be used together with static analysis tools.

This thesis contribution and novelty is in providing a comprehensive array of specification mining tools in mining various forms of specifications: automata [112, 113], frequent software behavioral patterns [118], temporal rules expressible in Linear Temporal Logic (LTL) [111, 119, 120] and Live Sequence Charts (LSCs) [122, 123]. Work related to this dissertation has been previously presented in [112, 113, 111, 119, 120, 118, 122, 123, 115, 116].

Our approaches in mining automata, frequent patterns, temporal rules and live sequence charts can be generalized into the following framework composed of four parts:

Part 1 Program instrumentation

Part 2 Trace generation and abstraction to sequences of symbols

Part 3 Execution of the mining algorithm

Part 4 Presentation of mined rules, post-processing, and downstream applications

Our mining framework is outlined diagrammatically in Figure 2.1. At the start of a mining task, three inputs are provided: a program (in source code, byte code or binary) to analyze, a test suite and a set of thresholds. These inputs will be fed to various parts of the mining framework resulting in a specification (in automata mining) or a set of (sub-) specifications (in patterns, rules and LSCs mining) ready for presentation to the user to aid program understanding and for inputs to downstream applications, e.g., model checking, run-time monitoring, etc.

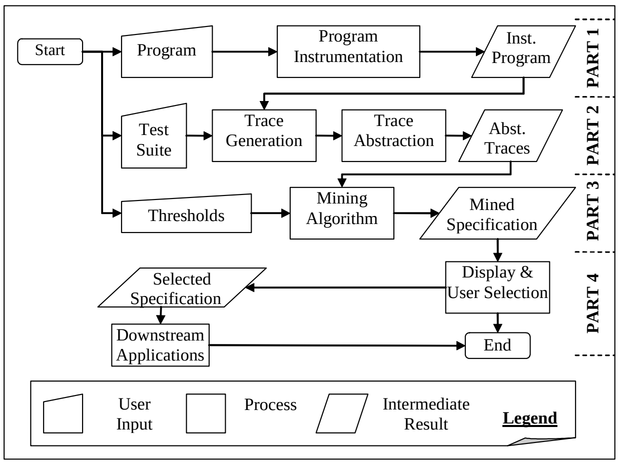


Figure 2.1: Mining Framework

An input program needs to be instrumented. Instrumentation inserts ‘print’ statements at the entry points of various function definitions. When the instrumented program is run, a trace will be generated. A trace is a series of bits of information (*e.g.*, method signature, caller, callee, etc.) of the methods being invoked.

Much open-source and industrial software comes with a test-suite. Running the instrumented program over a test-suite will produce a set of traces. Each trace corresponds to a series of method invocations. The set of raw traces can then be abstracted to a set of sequences of symbols by mapping a method invocation to a unique symbol. This set of sequences of symbols is the input of our mining algorithm. Alternatively, one can run the program on typical inputs or run the program as it is usually run (e.g., communicating with another party via instant messaging application under analysis, interacting with a game application under analysis, etc.).

At the end of the mining process, a specification or a set of (sub-) specifications are obtained. They are then presented to the user for program comprehension. These specifications can then be subjected to further manual refinements. Selected specifications can then be input to downstream applications like verification, run-time monitoring, etc.

The mining process can be applied periodically along with the various maintenance tasks performed during the software lifespan. The mined specifications can help developers understand how a software behaves, update obsolete specifications, verify a program, detect bugs, etc.

Software is varied and so does its specification. Different modeling formalisms have been proposed ranging from automata, design patterns, temporal logics to sequence diagrams. Each of these different formalisms has its own strengths and weaknesses. Often a well-documented software employs different models to capture different perspectives of interest of the software.

This dissertation describes an array of specification mining tools to mine specifications in the formats of automata, frequent patterns, temporal rules and live sequence charts, from program execution traces. *Mined automaton* captures global detailed behavior of a system. *Mined frequent patterns* capture common software behaviors. *Mined temporal rules* capture constraints or implicit rules that a piece of software under analysis should adhere to. *Mined Live Sequence Charts* capture inter-object behaviors that a system obeys represented as a formal variant of UML sequence diagrams.

We follow a multi-disciplinary approach. We first surveyed existing works and techniques on various computer science domains: software engineering, programming language, data mining, machine learning and bioinformatics. Opportunities to adapt, improve and merge existing techniques or possibly create a new one are then identified and pursued to help tackle the specification problem. A multi-disciplinary approach enables one to use the best tool for the job and even overcome difficult obstacles in a particular domain.

The high-level itemized contributions of this dissertation are as follows:

1. Extends the boundary of automaton-based specification mining by:
   - Proposing a novel objective evaluation measure and framework for automaton-based specification miners [112].
   - Improving accuracy, scalability and robustness of existing automaton-based specification framework through a plug-able architecture performing trace clustering and filtering [113].
2. Extends the boundary of pattern-based specification mining by mining frequent iterative or repetitive software behaviors from program execution traces [118]. The algorithm is statistically sound and complete as all patterns mined are frequent and all frequent patterns are mined.
3. Extends the boundary of rule-based specification mining by:
   - Proposing a novel notion of statistical soundness and completeness applied to rule-based specification mining [111].
   - Extending the expressiveness of mined rules and scalability of mining temporal rules from program execution traces [119, 120]. The algorithm is statistically sound and complete as all rules mined are significant and all significant rules are mined.
4. Mines statistically significant Live Sequence Charts from program execution traces [122, 123]. The algorithm is statistically sound and complete as all LSCs mined are significant and all significant LSCs are mined.

5. Creates a new bridge between the two areas of data mining and software engineering [115, 116].

The next four sections provide a summary of our approaches in mining automata, frequent patterns, temporal rules, and LSCs. Since each specification formats are different there is a need for a specialized algorithm to mine each of them.

2.3 Automaton-based Specification Mining

In the field of automaton-based specification mining, our contributions are a novel objective evaluation framework and a new pluggable architecture to respectively assess and improve the quality of existing automaton-based specification miners.

2.3.1 Objective Evaluation Framework

The value of having objective evaluation measures to compare related research work is recently emphasized [169, 133, 42]. At times, several papers addressing the same research issue reported their benefits based on different experiments or case studies. This raises two questions:

1. How to ensure that a technique works well across many, if not most, experiments?
2. How to evaluate relative performance (in terms of various dimensions of measurement) of two or more related techniques?

A good benchmark or evaluation framework can go a long way in answering these two questions. Clear research goals and evaluation criteria provided by such a framework can also help in directing the advancement of research in the respective area.

Several areas of research, such as bug localization, frequent itemset mining and frequent sequential pattern mining, have benefited from good evaluation frameworks (*e.g.*, [78, 152, 3, 5]). Both real test sets and simulated test generators have been employed in evaluating and comparing results of various proposed techniques. Of interest is the presence of a simulated test generator. A simulated test generator adds a degree of confidence that the technique runs well on a variety of test experiments and not only on a particular experiment under consideration. It also adds flexibility on evaluating the effect of varying a variable of interest while fixing other variables constant.

Despite the proliferation of specification mining research, there are not many studies on issues pertaining to the quality of automaton-based specification miners. Specifically, we note that issues such as scalability and robustness of miners and level of user intervention required during mining have not been comprehensively addressed. As an illustration, in [7], it was reported that "in order to learn the rule [*i.e.*, automaton], we need to remove the buggy traces from the training set." This indicates the problem with the limitation of choosing good training sets. In another work [8], it was noted that in order to debug a specification generated by a particular specification miner, it might be necessary to exhaustively inspect each of the traces, which can be hundreds or thousands in number.

Hence, there is a demand for a generic framework that can assess the quality of specification miners. Such a framework must address the issue of limited training sets as well as provide objective measures on the quality of specification miners. Addressing this, we propose QUARK (QUality Assurance framewoRK), a framework to enable assessments of the quality of automaton-based specification miners in the three dimensions of: miners' scalability, robustness and accuracy.

Scalability refers to a specification miner's ability to infer large specification. *Robustness* refers to its sensitivity to error present in the input data. *Accuracy* refers to the extent the miner is able to produce an inferred specification which is representative of the actual specification.

QUARK operates as follows: Given a specification miner, a simulator automaton and a percentage of expected error, QUARK generates a multiset of traces from the automaton with the specified percentage of erroneous traces. Running the specification miner against these traces produces a mined automaton. By comparing the similarities and differences of the mined automaton with that of the original automaton, QUARK can assess the accuracy of mining as performed by the given miner.

Furthermore, by varying the percentage of expected error and the size of the original automaton, QUARK enables the respective assessments of robustness and scalability of the miners.

We have built a prototype of QUARK, and used it to assess some existing specification miners. These experiments include mining of several real-world API-interaction specifications obtained from (1) programs using XLib and XToolkit intrinsic libraries for X11 windowing system [7], (2) IBM® WebSphere® Commerce [180], and (3) a Concurent Versioning System application built on top of Jakarta Commons Net [10].

2.3.2 Accurate, Robust and Scalable Mining

There is a need for new specification mining techniques that address the issue of *accuracy*, *robustness* and *scalability.* A more *automated* specification mining process that realizes the above three goals will be ideal.

To address the above need, we propose SMArTIC (Specification Mining Architecture with Trace fIltering and Clustering). SMArTIC is a specification mining architecture designed to improve the accuracy, robustness and scalability of automaton-based specification miners in learning a behavioral model from traces. This architecture is constructed based on two hypotheses:

1. Erroneous traces should be pruned from the input traces to a miner.
2. Clustering related traces will localize inaccuracies and reduce over-generalization in learning a behavioral model.

SMArTIC is comprised of four components or building blocks: an erroneous-trace filtering block, a similar-trace clustering block, a learner, and a merger. SMArTIC is a pluggable architecture where an existing specification miner can be put into the learner block. SMArTIC works in a multi-phase pipeline process where:

1. 'Potential' errors are automatically detected and removed. 'Potential' errors are traces that disobey any strong program temporal properties. These properties are automatically extracted from traces. This step addresses the issue of robustness.
2. Traces exhibiting similar behaviors are grouped together resulting in clusters of similar traces.
3. Each cluster of similar traces is then separately learned to form a sub-specification.
4. The resultant sub-specifications are then merged into an integrated specification. Steps 2 to 4 perform a divide and conquer approach to address the issue of scalability in learning a specification.

We show through experiments on real-life examples and simulated models that the quality of specification mining processes can be significantly improved using SMArTIC.

2.4 Pattern-based Specification Mining

It is of interest to obtain patterns of frequent repetitive software behaviors. These patterns form common software behavior and can serve as specifications. Some examples of common patterns of software behavior are as follows:

1. Resource Locking Protocol : $\langle lock, unlock\rangle$
2. Telecommunication Protocol (*c.f.*, [81]): ⟨*off_hook*, *dial_tone_on*, *dial_tone_off*, *seizure_int*,*ring_tone*, *answer*,*connection_on*⟩
3. Java Authentication and Authorization Service (JAAS) Authorization Enforcer Strategy Pattern (*c.f.*, [161]): $\langle Subject.getPrincipal$, $PrivilegedAction.create$, $Subject.doAsPrivileged$, $JAAS\_Module.invoke$, $Policy.getPermission$, $Subject.getPublicCredential$, $PrivilegedAction.run\rangle$
4. Java Transaction Architecture (JTA) Protocol (*c.f.*, [163]): $\langle TxManager.begin$, $TxManager.commit\rangle$, $\langle TxManager.begin$, $TxManager.rollback\rangle$, *etc.*

In this work, we extended two major trends in mining patterns from a set of sequences of events, namely sequential pattern mining [5] and episode mining [128]. Sequential pattern mining mines frequent patterns across multiple sequences. Episode mining mines frequent patterns whose events appear close together and is repeated frequently within one sequence. *Iterative pattern merges* the two and mines for frequent patterns that are repeated both across multiple sequences and within a single sequence. The semantics of iterative patterns follows that of standard software specifications namely Message Sequence Chart (MSC) [81] and Live Sequence Chart (LSC) [37].

Pattern mining in general is an NP-hard problem. For it to be practical, efficient search space pruning strategies need to be employed. One of the most important properties to help in ensuring scalability is the monotonicity or apriori property. We identify an appropriate apriori property to prune the search space containing insignificant patterns.

In consideration of possibly combinatorial number of frequent subsequences of a long pattern, it is practical to mine only a closed set of patterns (*c.f.*, [174] & [167]). *Closed pattern* mining discovers patterns without any super-sequence having a corresponding set of instances. An instance of a pattern corresponds to an instance of its sub-sequence pattern if the later is contained in or subsumed by the earlier. The set of closed patterns is likely to be *more compact* than the set of all frequent patterns and yet is *still complete* as the set of closed patterns represents the set of all frequent patterns. Closed pattern mining can also lead to a more efficient pattern mining strategy. *Early identification and pruning* of non-closed patterns employed by a closed pattern mining algorithm can significantly improve efficiency. In this work, we develop a novel closed pattern mining strategy for mining closed iterative patterns.

Our algorithm CLIPER (CLosed Iterative Pattern minER) mines a *closed* set of iterative patterns. A search space pruning strategy employed by *early identification and pruning* of non-closed patterns is used to mine a closed set of iterative patterns efficiently. Our performance study on synthetic and real-world datasets shows the major success of our pruning strategy: it runs with over an order of magnitude speedup especially on low support thresholds or when the frequent patterns are long. The algorithm proposed is sound and complete as all mined patterns are frequent and all frequent patterns are mined.

As a case study, we experimented with traces collected from the transaction sub-component of JBoss Application Server. Our mined patterns highlight important design patterns shedding light on program behavior.

2.5 Rule-based Specification Mining

There is a recent interest to mine specifications in the form of rules expressing temporal constraints from a program [176, 172]. While a mined automaton expresses a global picture of a software specification, mined rules break this into smaller parts each expressing a program property which is easily understood.

However, existing work on mining rules only mines two-event rules (*e.g.*, $\langle lock \rangle \rightarrow \langle unlock \rangle$), which are limited in their ability to express more complex temporal properties.

The focus of this study is to automatically discover rules *of arbitrary lengths* having the following form from program execution traces:

> "Whenever a series of precedent events ES_{pre} occurs, eventually another series of consequent events ES_{post} occurs."

The above multi-event rule can be expressed in temporal logic (i.e., Linear Temporal Logic (LTL)), and belongs to two of the most frequently used families of temporal logic expressions for verification (*i.e.*, response and chain-response) according to a survey in [41]. Notation-wise, we denote the above rule as $ES_{pre} \rightarrow ES_{post}$. Examples of such rules include:

1. Resource Locking: Whenever a lock is acquired, eventually it is released.
2. Initialization-Termination: Whenever a series of initialization events is performed, eventually a series of termination events is also performed.
3. Internet Banking: Whenever a connection to a bank server is made, an authentication is completed, and money transfer command is issued, eventually money is transferred and a receipt is displayed.

From traces, many rules can be inferred, but not all are important. Statistics of support and confidence employed in data mining [61] are therefore used to identify important rules. Rules satisfying user-specified thresholds of minimum support and confidence are referred to as being *statistically significant.*

Our algorithm TERMINAL (TEmporal Rule MINing ALgorithm) performs effective search space pruning strategies to efficiently mine multi-event rules from traces. To prevent an explosion in the number of mined rules, we define a *redundancy relation* among rules, and propose to generate only a minimal subset of rules containing non-redundant ones. With respect to input traces and given statistical significance thresholds, our algorithm is *statistically sound* as all mined rules are statistically significant (*i.e.*, they meet the thresholds). It is also *complete* as all statistically significant rules of the form $ES_{pre} \rightarrow ES_{post}$ are mined or represented.

This work is different from our work on mining iterative pattern whose semantics is based on MSC and LSC. In this work, we base our rules on the semantics of temporal rules, expressible in LTL, that are commonly used for verification. LTL, together with Computational Tree Logic (CTL), are the most widely-used formalisms for verification via model checking [29]. The semantics of MSC/LSC and our temporal rules are different. Hence, both the search space pruning strategies and the mining algorithm are very different from our work in iterative pattern mining. Also, mining specifications in the form of patterns and rules have their own application of interest (compare [176, 172, 41] with [107, 27, 19]). When frequent repetitive behaviors are desired iterative pattern mining is suitable to be employed; on the other hand, when constraints are needed temporal rule mining is suitable to be employed.

We carried out a performance study on several standard benchmark datasets to demonstrate the effectiveness of our search space pruning strategies. We performed a case study on JBoss Application Server – the most widely used J2EE server – to illustrate the usefulness of our technique in recovering the specifications that a software system obeys. We also performed a case study on a buggy Concurrent Versions System (CVS) application. It shows the usefulness of our technique in mining bug-revealing properties and aids program verification tools in finding bugs.

2.6 Live Sequence Chart-based Specification Mining

Damm and Harel's Live Sequence Chart (LSC) [37, 66] is a formal version of sequence diagram with pre- and post-charts. The semantics is an extension of Message Sequence Chart (MSC), which is a standard of International Telecommunication Union (ITU) [81]. An LSC adds modality to MSC and states that whenever a pre-chart is satisfied by a trace, the post-chart should also be satisfied by that trace. An example of an LSC is illustrated in Figure 2.2.

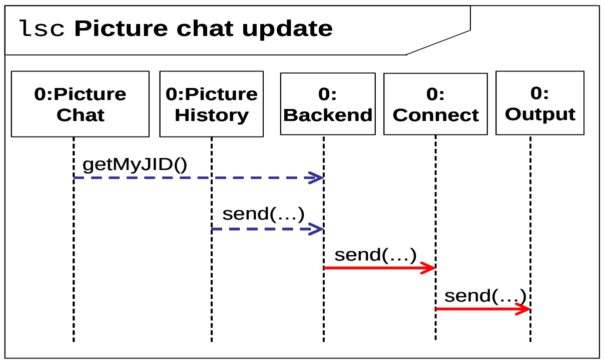


Figure 2.2: An Example of a Live Sequence Chart

An LSC includes a set of instance lifelines (shown as the vertical lines), representing the system's objects which are interacting. An LSC is divided into two parts, the *pre-chart* ('cold' fragment), shown by the dashed blue arrows, and the *main-chart* ('hot' fragment), shown by the solid red arrows. Each pre- and main-chart specifies an ordered set of method calls between the objects represented by the instance lifelines. We are interested with LSCs that specify *universal liveness requirement*: for all runs of the system, and for every point during such a run, whenever the sequence of events defined by the pre-chart occurs (in the specified order), eventually the sequence of events defined by the main-chart must occur (in the specified order). Events not explicitly mentioned in the diagram are not restricted in any way to appear or not to appear during the run (including between the events that are mentioned in the diagram).

Figure 2.2 shows an example LSC. This LSC specifies that "whenever *PictureChat* calls the *Backend* method *getMyJID()*, and sometime in the future the *PictureHistory* calls the *Backend* method *send()*, eventually the latter must call the *send()* method of *Connect* and *Connect* must call the *send()* method of *Output*". Note that if the pre-chart begins but never completes (or the order of events violates it), the main-chart does not have to occur and there is no other restriction on the order of the events appearing in it. In the example, if the first method *getMyJID()* is never called by the *PictureChat*, or if it is called but the next method *send()* never occurs, there is no constraint on the occurrence of the subsequent hot methods.

We propose an algorithm LIVE (LIVE sequence chart mining algorithm) to capture significant inter-object interactions in the form of LSCs. An LSC is significant if it satisfies given support and confidence thresholds. A search space pruning strategy is employed to remove the search space containing insignificant LSCs.

Iterative pattern mentioned earlier is based on MSC and LSC but is not an LSC. LSCs, simply put, can be considered as the rule version of iterative patterns. However, we need to adapt the semantics to address reactive systems (i.e., non-terminating programs that react to user inputs, e.g., servers, etc.) for whom LSC is originally designed. Also, different from the previous three mining approaches (mining automata, patterns and temporal rules) where an event corresponds to a method signature, in the LSC setting, an event is now composed of a triple: caller object identifier, callee object identifier, and method signature. An object is uniquely identified by its hash key (by calling System.identityHashCode() in Java) and the class it is instantiated from. In addition, we also propose mining class-level LSC which groups several object level LSCs belonging to the same class together. This symbolic version of the LSC can reduce the size of the mined specifications (c.f., [67, 153]).

Different from our work in mining temporal rules mentioned earlier, the pre-condition and post-condition of an LSC are in the form of a sequence diagram. Also, the semantics of temporal rules and LSCs are different and hence require different pruning strategies and algorithms. Temporal rules expressed in LTL are more commonly used for verification [41] while LSCs are more commonly used to specify software inter-object interaction requirements for reactive systems [67].

To demonstrate and evaluate our approach, we present the results of a case study we have conducted using traces from various components of Jeti [89], an open-source Java-based full featured instant messaging application. The results demonstrate the effectiveness of our mining technique in recovering non-trivial and highly expressive underlying interactions.

2.7 Structure of this Thesis

Chapter 3 outlines some essential terminologies, some details on program instrumentation strategies, and summaries of fundamental work on automata learning and data mining. Chapter 4 outlines QUARK, a quality assurance framework for automaton-based specification miners in more detail. Chapter 5 presents SMArTIC, a new mining architecture to improve accuracy, robustness and scalability of automaton-based specification miners. Chapter 6 outlines CLIPER, our algorithm mining closed iterative patterns. Chapter 7 presents our work in mining temporal rules via our algorithm TERMINAL. Chapter 8 describes LIVE, our algorithm mining Live Sequence Charts. Chapter 9 describes related work. Finally, Chapter 10 discusses future work and direction before this dissertation is concluded in Chapter 11.

CHAPTER III

PRELIMINARIES

In this chapter, first some essential definitions and terms are described. Next, we describe some program instrumentation strategies. Also, since the thesis builds upon existing work on automata learning and data mining, before proceeding to the details of the thesis' contribution presented in the subsequent chapters, this chapter summarizes some fundamental work in the two areas.

3.1 Terminologies

An execution trace can be viewed as a series of events. An event in turn corresponds to a behavior of interests. In this thesis, we consider an event of interest to be a method invocation. For most of the work described in the thesis, an event corresponds to the signature of the method being invoked. In the mining of Live Sequence Charts (LSC) described in detail in Chapter 8, since we need to mine sequence diagrams, the caller and callee information is also required. Hence, in LSC mining setting, an event corresponds to a triple (caller object, callee object, method signature). The set of traces under consideration can be considered as a sequence database where a trace corresponds to a sequence in the database.

Let I be the alphabet corresponding to a set of distinct events under consideration. Let a *sequence* S be an ordered list of events. We denote S as $\langle e_1, e_2, \ldots, e_{end} \rangle$ where each e_i is an event from I. We refer to the ith event in the sequence S as $S[i]$. The sequence database under consideration is denoted by $SeqDB$. We refer to the ith sequence in the database $SeqDB$ as $SeqDB[i]$.

In this thesis, the terms 'event' and 'symbol' are used interchangeably and both refer to a member of the alphabet under consideration. Similarly, the terms 'sequence,' 'string,' 'sentence,' and 'trace' are also used interchangeably to refer to a series of events (or symbols). We find that it is more natural to describe some concepts using one term rather than its synonym.

In this thesis, a pattern is defined syntactically as a series of events. A pattern can be mapped to its instances within a sequence database, where an instance is a part or segment of the sequence database that obey the pattern. The semantic meaning of a pattern (i.e., the definition of pattern instance) differs according to the pattern under consideration (*e.g.*, sequential pattern, iterative pattern, etc.). A pattern P_1 ($\langle e_1, e_2, \ldots, e_n\rangle$) is considered a *subsequence* of another pattern P_2 ($\langle f_1, f_2, \ldots, f_m\rangle$) if there exist integers $1 \leq i_1 < i_2 < i_3 < i_4 \ldots < i_n \leq m$ where $e_1 = f_{i_1}$, $e_2 = f_{i_2}$, $\cdots$, $e_n = f_{i_n}$. Notation-wise, we write this relation as $P_1 \sqsubseteq P_2$. We also say that P_2 is a *super-sequence* of P_1. We use the notations *first*(P) and *last*(P) to denote the first event and the last event of P, respectively. Reference to the database is omitted if it refers to the input sequence database $SeqDB$.

A pattern can also be concatenated or subtracted from another. These operations are defined in Definition 3.1.

Definition 3.1 (Concatenation and Truncation) *Concatenation of two patterns P_1 ($\langle a_1, \ldots, a_n\rangle$) and P_2 ($\langle b_1, \ldots, b_m\rangle$) results in a longer pattern P_3 ($\langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$). Truncation operation is only applicable between a pattern and its suffix. Truncation of a pattern P_3 ($\langle a_1, \ldots, a_n, b_1, \ldots, b_m\rangle$) and its suffix P_2($\langle b_1, \ldots, b_m\rangle$) results in the pattern P_1($\langle a_1, \ldots, a_n\rangle$). Pattern concatenation is denoted by $++$, while pattern truncation is denoted by $--$.*

3.2 Program Instrumentation Strategies

To generate traces, a program or system under study needs to be instrumented. Simply put, instrumentation puts 'print' statements at the entry points of various function definitions. Running instrumented program produces a trace, which is a series of bits of information of the methods being invoked when the program is run. Various instrumentation strategies can be employed including Aspect Oriented Programming (AOP) [44, 86], byte code manipulation [91] and binary editing [102].

Some recent tools (*e.g.*, [125, 137]) can even be attached to a *running* program and produce a file containing execution traces, hence addressing dynamically generated code. These strategies are listed in Table 3.2.

In our studies, we employed JBoss-AOP (for CLIPER and TERMINAL), AspectJ

Input	Strategy
Java source code	Compile-time weaving using AspectJ [44] or JBoss-AOP [86]
Java byte code	Load-time weaving using aspects or Java Runtime Analysis Toolkit [91] (*c.f.*, [176, 113])
.Net	Aspect.Net [154]
Binary	Executable Editing Library [102] (*c.f*, [7]), Pin [125] and Valgrind [137]

Table 3.1: Instrumentation Strategies

(for LIVE) and byte code manipulation (for SMArTIC) to instrument target programs under analyses.

To instrument using AOP, we wrote an *advice* corresponding to an instrumentation script and *weaved* it to the various *join points* corresponding to a set of methods of interest. When a method of interest is invoked, the instrumentation script which has been weaved in will run and write an entry to a trace file.

To instrument Java byte code we extended Java Runtime Analysis Toolkit (JRat) [91]. By default, JRat logs execution traces by associating them with a localized context. This context is simply a list of method calls in the runtime stack (*i.e.*, `main () -> FTPClient.<init> -> TelnetClient.<init>`). Information having the same context is grouped together. Given a class file to instrument, JRat will add instrumentation code to all methods except the constructor.

We modified JRat core classes and added a plug-in to it. The following features were added:

Capturing order of method calls along with context. We would like to capture the temporal order of method calls together with the context. However, JRat may destroy the order of method calls in the context. Information on calls to a method at two different times under the same context but with different temporal ordering should not be grouped together.

Instrumentation of class constructor. In order to capture the hierarchy of method calls well, we need to instrument the class constructor as well. The class constructor might call other methods. We would like to capture the information that their context are the same but different from method calls called at constructor context.

Thread slicing and Scalability. We sliced traces into threads and generated a separate trace file for each of them. For scalability, no large trace related information is stored in memory. They are always outputted incrementally.

The result of JRat instrumentation is an injection of tracing byte code into Java class files.

3.3 Automata Learning

An automaton is a transition system with start and end nodes. It represents a language corresponding to a set of sentences. A sentence or a sequence of symbols is formed by traversing the automaton from start to end nodes. A sentence is said to be a positive example if it is part of the language described by the automata.

Perfect learning of an automaton from a finite set of its positive examples has been shown to be undecidable [57, 7]. However, there are heuristic solutions to learn a reasonably good automaton from a finite set of positive examples. The first solution is by Biermann and Feldman [15], which is called k-tails. Their work was later extended by Raman and Patrick [149] to incorporate probabilities into the learning process in a new algorithm called sk-strings.

K-tails algorithm is a well-known *heuristic* algorithm proposed by Biermann and Feldman [15] to learn an automaton from a set of positive examples. It has been adapted/modified by various researchers to perform specification mining tasks [33, 150, 132]. From a training set of positive samples, the algorithm first builds a prefix tree acceptor. Informally, a prefix tree acceptor (PTA) is an automaton in the form of a tree where there is one node for every common prefix and each leaf is a final state. A PTA can be built by simply laying out the examples in the input set, using a node for a unique prefix. Given a PTA, a node q, a set of alphabet Σ, a set of final states (the leaves of PTA) F_c, and an extended transition function δ^*, the set of k-tails associated with the node q is given by $\{s|s \in \Sigma^*, |s| \leq k \wedge \delta^*(q, s) \cap F_c \neq \emptyset\}$. Two nodes of the PTA are then merged if their respective k-tails are indistinguishable.

Consider the following set of sentences in Table 3.2. The PTA built from the sentences is shown in Figure 3.1.

Consider the nodes in the PTA shown in Figure 3.1. For each of them, the corresponding set of k-tails (for k=2) is shown in Table 3.3. The resultant model generated by k-tails algorithm after the merging of nodes is performed is shown in Figure 3.2.

ID	Sentence
S1	$\langle A, B, C, D, E\rangle$
S2	$\langle A, B, C, X, Y\rangle$
S3	$\langle A, E, B, D, E\rangle$

Table 3.2: Example Sentences for PTA Building

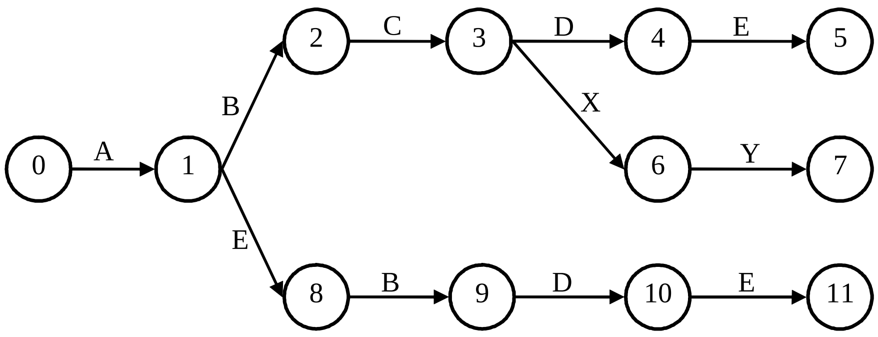


Figure 3.1: Prefix Tree Acceptor (PTA)

Node Id	K-tail Set	Node Id	K-tail Set
0	$\{\}$	6	$\{\langle Y\rangle\}$
1	$\{\}$	7	$\{\langle\rangle\}$
2	$\{\}$	8	$\{\langle\rangle\}$
3	$\{\langle D, E\rangle, \langle X, Y\rangle\}$	9	$\{\langle D, E\rangle\}$
4	$\{\langle E\rangle\}$	10	$\{\langle E\rangle\}$
5	$\{\langle\rangle\}$	11	$\{\langle\rangle\}$

Table 3.3: Sample k-tails for k=2

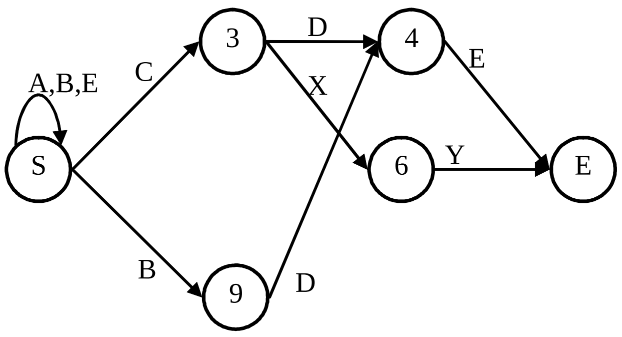


Figure 3.2: Automata Model - K-tails

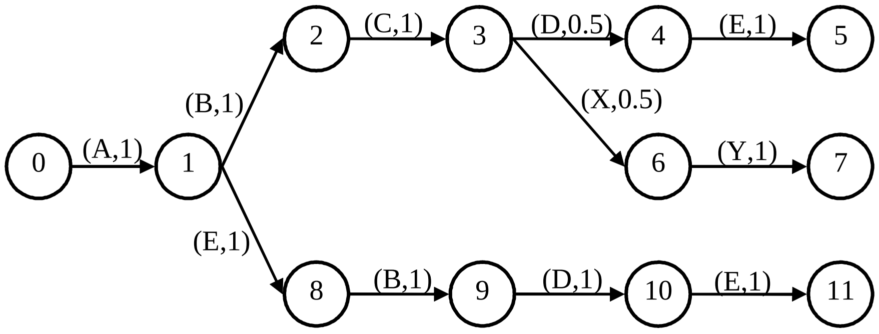


Figure 3.3: Prefix Tree Acceptor (PTA) With Probabilities

Node Id	Sk-string Set	Node Id	Sk-string Set
0	$\{\langle A, B\rangle, \langle A, E\rangle\}$	6	$\{\langle Y\rangle\}$
1	$\{\langle B, C\rangle, \langle E, B\rangle\}$	7	$\{\langle\rangle\}$
2	$\{\langle C, D\rangle, \langle C, X\rangle\}$	8	$\{\langle B, D\rangle\}$
3	$\{\langle D, E\rangle\}$	9	$\{\langle D, E\rangle\}$
4	$\{\langle E\rangle\}$	10	$\{\langle E\rangle\}$
5	$\{\langle\rangle\}$	11	$\{\langle\rangle\}$

Table 3.4: Sample sk-strings for k=2 and s=100%

Sk-strings algorithm is an extension of k-tails heuristic for stochastic automata [149]. It has been used by Ammons *et al.* in [7]. Similar to k-tails, sk-strings algorithm also builds a prefix tree acceptor from traces. The difference lies in the criteria for merging of nodes and the incorporation of probabilities. Given a PTA, a node q, a set of alphabet Σ, a set of final states (the leaves of PTA) F_c, and an extended transition function δ^*, the set of k-strings associated with the node q is given by $\{s | s \in \Sigma^*, |s| \leq k \wedge \delta^*(q, s) \neq \emptyset\}$. Two nodes are merged if they are indistinguishable with respect to the *top s% most probable k-strings* (instead of *k-tails*).

Based on the input sentences, probabilities can be attached to the PTA. The resultant PTA with probabilities attached is shown in Figure 3.3.

Consider the nodes of the PTA shown in Figure 3.3. For each of them, the corresponding set of sk-strings (for k=2 and s=100%) is shown in Table 3.4.

The resultant model generated by sk-strings algorithm after the merging of nodes is shown in Figure 3.4.

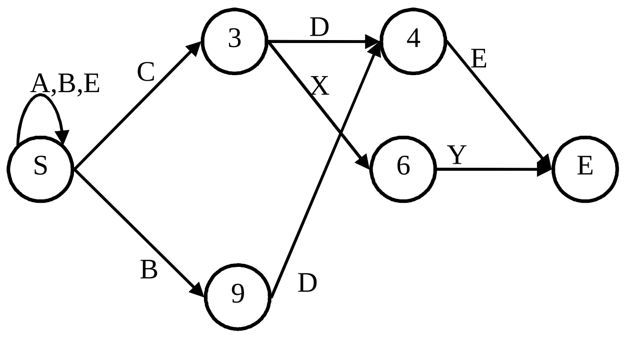


Figure 3.4: Automata Model - Sk-strings

3.4 Data Mining

There are two data mining techniques particularly relevant to this thesis, namely clustering and frequent itemset/pattern mining.

Clustering is an unsupervised learning process where data samples are divided into groups. Similar samples are grouped into the same group and different samples are separated to different groups. Frequent itemsets/pattern mining is a process where itemsets or patterns appearing frequently in a dataset are extracted. Usually, effective search space pruning strategy needs to be employed to render pattern mining algorithms feasible to handle non-trivial datasets.

3.4.1 Clustering

There are many clustering algorithms. A good summary and categorization are available in [61]. Clustering algorithms can be categorized broadly into hierarchical or partitional. In the hierarchical approaches, the clusters are build step by step by merging or dividing previously formed clusters. In the partitional approaches, all the clusters are build at once, further refinement are then performed by shifting data samples from one cluster to another in the successive clustering steps. Two of the most well-known and classic clustering algorithms are k-means [71] and k-medoids [92], which belong to the partitional family of clustering algorithms.

In k-means, the algorithm divides the dataset into k groups where the distance of each data items to the mean of each group is minimized (hence the name k-means). First, the algorithm forms an initial set of k groups randomly. The mean of each group is then computed. Next, each data point is assigned to the group where the distance between the data point and the group's mean is minimized. The procedure is repeated until a fix point is reached, namely no more data items move into a different group/cluster.

K-means algorithm is susceptible to outliers, or data items that are far different from the rest of the items. As an extension, k-medoids algorithm is proposed. In k-medoids algorithm again k groups are formed. However, a group is represented not by the mean of the constituent data items, rather by a data item in the group which is the "median" having minimal distance to all other data points in the same group (hence the name k-medoids). The algorithm starts by choosing k arbitrary data points and treating them as medoids. The other data points are then split based on which medoid they are nearer to. An iterative process is then performed to swap each medoid with a non-medoid data point. The algorithm stops when a fix point is reached and the set of k-medoids doesn't change any longer.

Aside from the algorithms, another central tenet in clustering is the assignment of distance between one point to another. Several distance metrics have been proposed. Some of them includes: Manhattan distance, Euclidean distance, and many more. In Manhattan distance the distance between two data points in x-y space is defined as $|x1 - x2| + |y1 - y2|$. In Euclidean distance the distance between two data points in x-y space is defined as the square root of $((x1 - x2)^2 + (y1 - y2)^2)$. Both the algorithms and distance metrics used affect the accuracy of a clustering algorithm for a particular task at hand.

3.4.2 Frequent Itemset/Pattern Mining

Frequent itemset mining was first proposed for market basket data analysis by Agrawal and Srikant [4]. The problem is given a set of transactions, where a transaction corresponds to a set of items bought, find items that are frequently purchased together. A set of items (or an itemset) is *frequent* if it is supported by a significant number of transactions based on a user given threshold of minimum support. An itemset is supported by a transaction if the itemset is a sub-set of the latter. These frequent itemsets like {Milk, Bread} can be used for a marketing campaign purpose, or to design good shelving arrangement for commonly purchased items.

To illustrate how frequent itemset mining works, consider the following dataset shown

ID	Transaction
T1	$\{B, D, M\}$
T2	$\{B, F, M\}$
T3	$\{B, F, M, V\}$

Table 3.5: Example Dataset for Frequent Itemset Mining

ID	Sequence
S1	$\langle D, B, M\rangle$
S2	$\langle D, B, M, M, M, K, B, S\rangle$
S3	$\langle D, Y, Y, M, Y, M, Y, M\rangle$

Table 3.6: Example Dataset for Frequent Sequential Pattern Mining

in Table 3.5.

The set of frequent itemsets mined using minimum support threshold set at 3 transactions is the set: {{B}:3,{M}:3,{B,M}:3} – the number after the colon denotes the support of the itemset. The algorithm works by traversing the search space of frequent patterns. The following apriori or anti-monotonicity property: "if a set is not frequent, so does its super-sets", is used to prune the search sub-spaces containing infrequent patterns. This apriori property enables effective pruning of search space and allows the mining algorithm to run efficiently on substantially large input dataset. The algorithm first tries all itemsets of size one. Based on frequent itemsets of size 1, it then produces a set of possible candidates of frequent itemsets of size 2. These candidates are then checked whether they are really frequent. The process continues until there are no more candidates that are generated.

Agrawal and Srikant later proposed sequential pattern mining [5]. In sequential pattern mining, sequences or series of events rather than sets of items are considered. The order of occurrences of events is taken into consideration. An event can also be repeated multiple times in a sequence. The problem is given a set of sequences, find all series of events (or sequential patterns) that are frequent. A pattern is frequent if it is supported by a *significant* number of sequences following a user-defined threshold of minimum support. A pattern is supported by a sequence if the pattern is a sub-sequence of the latter.

To illustrate how frequent sequential pattern mining works, consider the following sequence database shown in Table 3.6.

The set of frequent sequential patterns mined using minimum support threshold set at 3 sequences is the set: $\{\langle D\rangle$:3, $\langle M\rangle$:3, $\langle D, M\rangle$:3$\}$. The algorithm works by traversing the search space of frequent sequential patterns. The following apriori or anti-monotonicity property: "if a sequential pattern is not frequent, so does its super-sequences", is used to prune the search sub-spaces containing infrequent patterns. This apriori property enables effective pruning of search space and allows the mining algorithm to run efficiently on substantially large input dataset. The algorithm first tries all sub-sequences of length one. Based on frequent patterns of size 1, it then produces a set of possible candidates of frequent patterns of length 2. These candidates are then checked whether they are really frequent. The process continues until there are no more candidates that are generated.

There are various extensions and variants of the above two pattern mining algorithms to further improve its efficiency or address other types of data (*c.f.*, [61]).

CHAPTER IV

ASSESSING QUALITY OF AUTOMATON-BASED MINERS

Despite the proliferation of specification-mining research, there is not much report on issues pertaining to the quality of specification miners. Specifically, we note that issues such as scalability and robustness of miners, level of user intervention required during mining have not been comprehensively addressed. As an illustration, in [7], it was reported that "in order to learn the rule [*i.e.*, automaton], we need to remove the buggy traces from the training set." This indicates the practical problem in choosing good training sets. In another work [8], it was noted that in order to debug specification generated by specification miner, it might be necessary to exhaustively inspect each of the traces, which can be hundreds or thousands in number.

Hence, there is a demand for a generic framework that can assess the quality of specification miners. Such a framework must address the issue of limited training sets as well as provide objective measures to the performance of specification miners. Performance should be measured in multiple dimensions: miners' scalability, robustness and accuracy.

Scalability determines a specification miner's ability to infer large specification. *Robustness* refers to its sensitivity to error present in the input data. *Accuracy* refers to the extent of an inferred specification being representative of the actual specification.

These measurements extend from the existing set of measurements found in literature on software specification validation and program analysis. During our assessment, we generate program traces from a chosen specification, use the traces to mine a specification, and then compare the mined specification against the original specification. A good specification miner should infer a specification that matches the original specification as accurately as possible, if the set of traces generated is a good representation (sample) of the (possibly infinite number of) traces generated by the original specification. Our measurement of *accuracy* is adapted from the measurements of *recall* and *precision* of

Nimmer *et al.* in evaluating Daikon. Nimmer *et al.* also relate these measurements to the concept of soundness and completeness used in program analysis community [140].

An additional advantage of having these objective assessments of specification miner is that they not only define the quality of specification miners in different dimensions, but also highlight areas for improvement, and aid the design and development of new specification miners.

In this work, we propose a generic framework for assessing the quality of automaton-based specification miners. Our framework (QUARK) requires any specification miner under assessment to exhibit the following input-output behavior:

> *Let a program execution trace be a sequence of method calls to an API interface. Given a (multi-)set of program execution traces T, a minority of which might be erroneous, the specification miner infers sequencing/temporal constraints among the method calls in the form of a finite-state automaton.*

We do not constrain automaton-based specifications to be deterministic; in fact, *a miner is expected to perform its task in the presence of non-deterministic specification.*

The original automaton can be either probabilistic or not (PFSA/ FSA). In fact, an FSA is a special form of a PFSA with probabilistic information dropped. Representing specifications as *Probabilistic FSAs* (PFSAs) instead of FSAs, however, has the following benefit: *Probabilities attached to a protocol specification enable more control over the trace-generation process so that the collection of traces generated mimics certain characteristics of the traces that can be collected from actual API interactions.* For example, sub-protocols within a protocol specification may appear more frequently than others in the actual interaction with API interface – analogous to the idea of hotspot found in program execution [77]. Such behavior can be made to exhibit in a set of generated traces through supply of appropriate probabilities at various transitions of a specification automaton.

In addition, it has also been proven that perfect learning of an FSA from positive examples is not decidable [7, 57], whereas perfect learning of a PFSA from examples is decidable (*cf.* [39, 9]) though inefficient (*cf.* [94]). This theoretical finding has prompted Ammons *et al.* to use PFSA as an intermediate step to the learning of an FSA [7].

QUARK enables any specification miner with the required input-output behavior to be assessed under a simulated environment. It operates as follows: Given a specification miner, a simulator automaton and a percentage of expected error, QUARK generates a multiset of traces from the automaton with the specified percentage of erroneous traces. Running the specification miner against these traces produces a mined automaton. By comparing the behavior of the mined automaton with that of the original automaton, QUARK can assess the accuracy of mining as performed by the given miner.

Furthermore, by varying the percentage of expected error and the size of the original automaton, QUARK enables the respective assessments of robustness and scalability of the miners.

We have built a prototype of QUARK, and used it to assess some existing specification miners. Later in this chapter, we describe our comprehensive experiments on three specification miners. These experiments include mining of several real-world API-interaction specifications obtained from (1) programs using XLib and XToolkit intrinsic libraries for X11 windowing system [7], (2) IBM® WebSphere® Commerce code [180], and (3) a simple Concurrent Version System (CVS) protocol built on top of Jakarta Commons Net [10].

The contribution of this work is as follows:

1. We propose a simulation based framework, to ensure more objective evaluation criteria and more repeatable results when applying mining process in different applications. Simulation also provides more control to the experiment and can help to localize the effect of a variable of interest (e.g. percentage of error traces, size of underlying specification, etc) to the accuracy of the mining process.

2. We propose and compute three measures of accuracy: precision, recall and probability similarity applied to automata-based specification mining.

3. We provide experimental quantitative comparison of different specification miners in three dimensions of quality assurance: scalability, robustness and accuracy.

4. We identify hidden weakness in specification mining process, namely precision of mined models. The situation is further aggravated when erroneous traces are present and when the underlying model is large.

The outline of the chapter is as follows: In section 4.1, a typical specification mining process is presented and the structure of QUARK is outlined. Sections 4.2 and 4.3 describe our solutions to two major issues related to quality assurance measurement: model-and-trace generation and the metrics and techniques for quality assurance. Section 4.4 briefly describes specification miners used in our experiments. Sections 4.5 and 4.6 describe our experiments and results. We discuss related work and conclude in Section 4.7.

4.1 Framework Structure

A specification miner's input is a set of execution traces. The miner then learns from these traces to produce a specification. The specification can be expressed in various forms. Human judgment rather than an objective measure is often employed at this stage to assess the performance of the miner.

In some systems, such as Daikon, the researchers assess the quality of the system by measuring their accuracy in recalling correct information (invariants) and in reducing the generation of incorrect information [140]. However, they fall short in providing systematic support for assessment of scalability and robustness of miners. It is clear that scalability and robustness are also important determinants for the usability of miners; the former determines the limit of a miner in handling complex systems, and the latter determines the usefulness of a miner in handling mildly corrupted input.

QUARK aims to address all the above quality assurance concerns pertaining to the assessment of automaton-based specification miners. It accepts specification models of varying complexity, and generates sets of simulated traces that reflect the characteristics of those protocol specifications, including the presence of error. It then evaluates a miner's performance in recovering the original model from three dimensions: its accuracy, robustness and scalability.

The structure of QUARK is shown in Figure 4.1. Its *trace generator* component generates traces based on a specification model in PFSA format. These simulated traces are then used to train the specification miner, culminating with a mined PFSA model. The original model and the mined model are then used by the *specification miner quality assurance sub-system* to generate various quality assurance metrics.

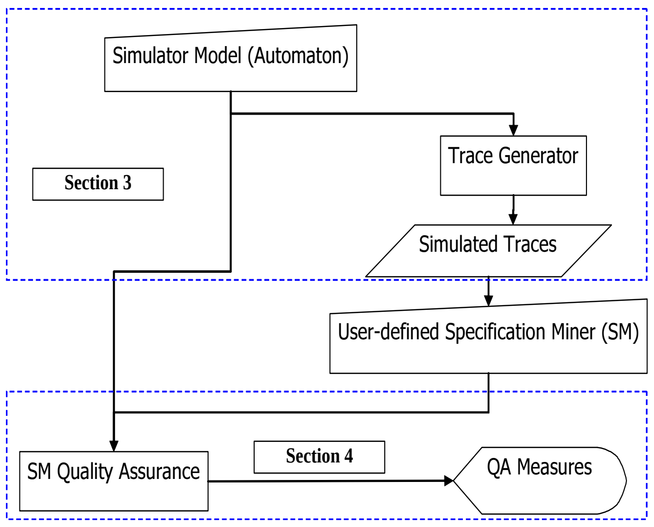


Figure 4.1: Framework Structure

There are two major issues in QUARK that need to be addressed: **(1)** model and trace generation, and **(2)** quality assurance metrics and their techniques. These will be discussed in sections 4.2 and 4.3 respectively.

4.2 Simulator Model & Trace Generation

QUARK admits two closely-related simulator models: FSA and PFSA. In both cases, it accepts both deterministic and non-deterministic models. Since PFSA is technically more complex to handle than FSA, we focus our discussion on PFSA and its associated trace-generation method. At the end of this section, we show how our method can be adapted to handle FSA.

4.2.1 Probabilistic Model

Figure 4.2 depicts an example of error-injected simulator model. Ignoring the dotted nodes and dashed edges, the remaining model is in the form of a probabilistic finite state automaton (PFSA). Each node in a PFSA represents an abstract program state. There are 3 types of nodes: start, end and normal nodes. Each transition in the automaton denotes an abstract representation of a viable API method call from that state. Every transition is attached with a probability, indicating how likely the associated method call will be invoked from that source state. It is an invariant of any PFSA under consideration that all transitions emitting from a source (excluding the transitions leading to error nodes) must have their probabilities summed up to 1.0.

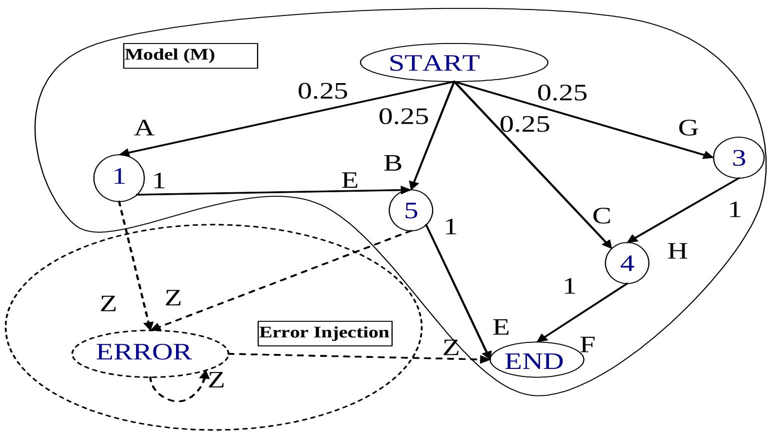


Figure 4.2: Sample Simulator Model

Error Injection A PFSA model can be *injected with error* by including error nodes and error transitions, shown as dotted nodes and dashed edges respectively in Figure 4.2. This inclusion enables generation of erroneous traces, and aids the evaluation of miner's ability to learn in the presence of error (*i.e.* robustness). Allocations of error nodes and transitions characterize the kind of errors allowed. Probabilities are not assigned to error transitions, as we do not intend to micro-manage the generation of erroneous traces. We will describe generation of erroneous traces in Section 4.2.2.

Model Size and Model Generation In addition to subjecting miners to tests with real-world specifications, we also devise ways to generate synthetic models. This allows us to perform controlled experiments on miners' quality.

To test a miner's scalability, we control the size of a simulator model by varying the number of nodes it has and the maximum number of transitions a node can emit. We automatically generate distinct models having n nodes and a maximum of m transitions per state with a common start and end nodes. Transition labels are chosen randomly, with repetition, from a pool of fixed number of labels. We first build a tree from a pre-determined number of nodes. Next, to mimic the behaviour of typical API-interaction, we introduce loops into the tree based on an idea similar to the principle of 'locality of reference'. This well-known principle states that a program tends to reuse data and instructions it has used recently [155]. Adapting from this principle, a method will more likely be invoked again if it has just been called before. Hence, loops between child and parent/ancestor nodes, including self-loop, are introduced with higher probability than those connecting to distant sibling nodes. Although we have not rigorously verified the applicability of this principle, the three real-world specifications shown in Section 4.5 are found to adhere to this principle.

Lastly, probabilities are assigned equally to transitions from the same source node.

4.2.2 Trace Generation

A program trace can be mapped to a string of alphabets, as shown by Ammons *et al.*, through a 'standardization' process, in which an alphabet (corresponds to a transition label in the simulator model) represents a particular method call [7]. Two types of traces are generated: normal and erroneous traces. A *normal trace* is defined as a sequence of transition names that forms a path leading from the start node to the end node of a PFSA. An *erroneous trace* is one that includes an error transition.

Since normal traces are generated from a PFSA, we can determine the probability of a trace by multiplying together the probability of its constituents.

Given an input model, the algorithm for trace generation is described below. Basically, it performs a *stratified random walk* over the input model, guided by the probability of the PFSA's transitions. Consequently, it ensures that highly probable traces (sentences) accepted by the PFSA model will statistically be more likely to appear in the multiset of generated traces. (We use the term "sentence" and "trace" interchangeably.)

This algorithm, called **TraceGen**, is akin to the "code and branch coverage" criterion used in generating program test cases [16]. Given a PFSA M, a cover (i.e., coverage multiplier) N, and a maximum trace number Max , **TraceGen** generates a multiset of traces T possessing the following asymptotic property:

Property 4.1 *For any $N > 0$, and for a sufficiently large number Max, every transitions in the PFSA M occurs at least N times in the trace multiset T of size at most Max.*

This property ensures that all transitions in M have the opportunity to be used for trace generation. This is the *coverage criterion* used in our experiments. The algorithm detail is depicted in Figure 4.3.

```
Procedure TraceGen
Inputs:
$M$,$EI$,$N$ :   Automaton model, error injection and cover, respectively.
$I$,$MaxPopE$:   Maximum number of iterations and maximum error trace
                 population, respectively
$Max$,$GE$ :     Maximum trace number and global error injection probability,
                 respectively
Outputs:
A multiset of traces
Method:
1:  let $M_{EI} = M \cup EI$
2:  let $E$ = list of all transitions in model $M$, each of which is identified by the
      transition name, its source and sink nodes
3:  let $Errlist$ = list of all possible error traces from $M_{EI}$ bounded by $MaxPopE$
      where for each, transitions in $M_{EI}$ are traversed at most $I$ times
4:  let $Ctr$ = a map from $e$ in $E$ to a number – initialized to 0
5:  do {
6:    Let $rand$ = a random number between 0 to 1
7:    if ($rand < GE$) {
8:       Let $TE$ = a trace selected randomly from $Errlist$
9:       Output $TE$
10:   }
11:   else {
12:      Let $T$ = a trace generated from $M$                           (see text) (*)
13:      Output $T$
14:      Let $E'$ = all $M$'s transitions traversed to produce $T$
15:      For each $e' \in E'$ increase $Ctr[e']$ by 1
16:   }
17: } while ($\exists e \in E$: $ctr[e] < N$ & number of traces $\leq Max$)
```

Figure 4.3: Trace Generation Algorithm

In Figure 4.3, at program point (*), a trace is generated by starting from start node of the model and independently "throwing a dice" at each node for decision on which transition to take *according to the probability of the transitions* until an end node is reached. Every trace generated will then reflect the probabilities of the transitions in the simulator model (*i.e.*, distribution of generated traces is governed by the model). Also in Figure 4.3, M_{EI} is the PFSA M with error EI injected.

Traces will continue to be generated until all transitions have been covered at least N times or MAX number of traces have been generated. We use N here rather than 1 to accommodate slower learner that requires more than 1 sentence in the language to infer the automaton model.

Erroneous Traces Generation. The percentage of erroneous traces generated are controlled at a global level by a probability GE. Before a trace is generated to join the trace set, the algorithm checks if erroneous trace needs to be generated. If so, such a trace is produced by choosing one randomly from previously generated pool of erroneous traces (*Errlist* - see Figure 4.3).

Non-Probabilistic Model. An FSA model can be easily obtained from a PFSA simulator model by dropping the probability associated with each transition in the model. The *major technical difference* between using FSA and PFSA simulator models is trace generation. In FSA, a *standard random walk* is performed, rather than *stratified random walk*. For the algorithm in Figure 4.3, this difference occurs at the program point (*): When FSA is used, a normal trace is generated by starting from start node of the model and randomly choosing an outgoing transition to reach the next node, until the end node is reached. Here, all outgoing transitions from a node have *equal chance* to be chosen.

4.3 Specification Miner Quality Assurance

The quality of a specification miner is measured along three dimensions: accuracy, robustness and scalability.

We define *robustness* of a specification miner as its ability in remaining accurate in recovering simulator models from simulated traces, in the *presence of error*. Erroneous traces usually constitute a small proportion of the entire collection of traces, and a robust miner should be able to filter out erroneous traces in building mined models.

We define *scalability* of a specification miner as its ability in remaining accurate in recovering *simulator models of varying sizes*.

As these measurements are orthogonal, we can conveniently compose them, and objectively discuss about the robustness of a scalable miner, or the scalability of a robust miner. Central to our assessments is a thorough treatment of accuracy. In the rest of the section, we shall provide a detailed account of metrics and techniques employed in measuring accuracy.

4.3.1 Trace Similarity

The *accuracy* of a specification miner is determined by its ability in recovering simulator models by learning the simulated traces, in the *absence of error*. For clarity sake, we denote a simulator model by X and a mined model by Y. We use the term "sentence" and "trace" interchangeably.

In assessing accuracy, we adopt two metrics to measure the similarity between X and Y in terms of their *generated traces* (or sentences). First, the percentage of sentences *generated by* X that are *accepted by* Y represents the amount of correct information that can be recollected by the mined model. This measurement is known as *recall* in information retrieval literature (*cf.* [58]). Second, the percentage of sentences *generated by* Y that are *accepted by* X represents the amount of correct information that can be produced by the mined model. This is known as *precision* (*cf.* [58]).

The notions of *recall* and *precision* are also used by Nimmer *et al* to evaluate Daikon. Nimmer *et al.* further relate them to measures of completeness and soundness, respectively [140].

To perform trace similarity measurement, we employ an *automaton language search technique*. This basically generates two sets of samples of traces from X and Y, respectively, and calculates the percentage of traces generated by X that are accepted by Y, and vice versa. The trace sample generated from X will be different from the set of traces used in training the miner. Separating the training set from the test set enables the detection of any "overfitting" done by the miner; *ie.*, the miner learns the training set so closely that it does not generalize well to original model [63].

This technique is effective in measuring the quality of Y(respectively X) provided the set of traces generated are representative of X(respectively Y). To this end, we use the **TraceGen** procedure in Figure 4.3 to help in trace generation.

4.3.2 Probability Similarity

For models that are represented by PFSAs, it is not sufficient to measure their similarity by simply examining their recall and precision. It is equally important to determine if both the simulator and the mined models generate *the same traces at similar frequencies*, and thus place emphasis on similar sub-protocols. Thus, our third metric measures the similarity in terms of probabilities assigned to common traces generated by both X and Y: A trace might possibly be generated by both X and Y; however, its probability might differ greatly.

Co-emission has been used in measuring probability similarity between two Hidden Markov Models [127]. Let $L(M)$ represent the language recognized by the automaton M, the co-emission is defined by the following formula:

$$P_{CE}(X,Y) \quad = \quad \Sigma_{s \in L(X \cap Y)}(P_X(s)P_Y(s)).$$

Here, $P_{CE}(X,Y)$ determines the probability that a sentence s is generated by both X and Y independently. $P_X(s)$ and $P_Y(s)$ denote the probability of generating sentence s by X and by Y, respectively.

The *probability similarity* between X and Y, denoted by PS, can then be defined as follows [127]:

$$\text{PS}(X,Y) \quad = \quad \frac{2 * P_{CE}(X,Y)}{(P_{CE}(X,X)+P_{CE}(Y,Y))}$$

This provides an unbiased and normalized probability similarity measurement of the two models. In practice, this computation is realized by a *HMM-HMM comparison-based technique*. This technique has been adapted from the work of Lyngsø *et al.* [127]. Figure 4.4 outlines the algorithm for calculating P_{CE} between two automatons. Probability similarity (PS) can be calculated from P_{CE} as defined above.

Note that at each iteration of probability table update, we extend (*in the worst case*) the co-emission probability computation to another pair of nodes (each from X and Y respectively) which is *only one* distance further away from the start nodes. We approximate co-emission by ending the probability computation only after each of the possible loops is executed at least twice. This is ensured by repeating the probability table update for $2 \times min(|X.Edges|, |Y.Edges|)$ times. We refer readers to [126] for further interesting discussion about the behavior of the algorithm that our algorithm shown in Figure 4.4 adapts.

4.4 Specification Miners Used

Three specification miners are used in our experiment. They are: (1) *k-tails FSA learner*, (2) *sk-strings PFSA learner*, (3) *sk-strings with coring.*

```
Procedure Compute Co-emission
Inputs: $X$: First automaton, $Y$: Second automaton
Outputs:
$P_{CE}$ (X,Y): Sum of co-emission probabilities of common sentences of $X$ and $Y$
Method:
1:  Let $uX$, $uY$ = Enumeration of all nodes in $X$ and $Y$ respectively
2:  Let $n$, $m$ = Number of nodes in $X$ and $Y$ respectively
3:  Let $dProb[][]$ = Create a table of size $n \times m$
4:  Let $xStart[]$, $yStart[]$ = Indices of start nodes in $uX$ and $uY$ respectively
5:  Let $xStop[]$, $yStop[]$ = Indices of end nodes in $uX$ and $uY$ respectively
6:  Let $iterNo = 2*min(|X.\text{Edges}|, |Y.\text{Edges}|)$
7:  For each ($0 \le i < |xStart|$) and ($0 \le j < |yStart|$)
8:     Initialize $dProb[xStart[i]][yStart[j]]$ to 1
9:  For rep=0 to $iterNo$
10:    For each ($0 \le i < n$) and ($0 \le j < m$)
11:       Let $dSum = 0$
12:       For each ($k$,$h$) where $uX[k]$ has a transition $t_k$ to $uX[i]$
13:       and $uY[h]$ has a transition $t_h$ to $uY[j]$
14:          If $t_h$ and $t_k$ have the same label, then
15:             $dSum$ += $dProb[k][h] \times$ probability of $t_h \times$ probability of $t_k$
16:       $dProb[i][j] = dSum$
17: $P_{CE}(X,Y) = \Sigma_{i,j}\ dProb[xStop[i]][yStop[j]]$
18: Output $P_{CE}$ (X,Y)
```

Figure 4.4: Co-emission Computation

k-tails algorithm is a well-known *heuristic* algorithm proposed by Biermann and Feldman [15] to learn automata from positive samples. It has been adapted/modified by various researchers to perform specification mining tasks [33, 150, 132]. sk-strings algorithm is an extension of k-tails heuristic for stochastic automata proposed by Raman and Patrick [149]. It has been used by Ammons *et al.* in [7]. The description of the two algorithms are provided in Chapter 3.

The default parameters of k-tails and sk-strings [149] as implemented by Raman *et al.* are used. For k-tails, the default value of the parameter k is 1. The default values of the parameters s and k are 50% and 1 respectively. Unless otherwise stated, these defaults are used in the experiments (k=3 is also used in some of our experiments).

In [7], Ammons *et al.* discussed *coring* method as a post-processing step to remove erroneous transitions from the mined automaton. Briefly, identification of erroneous transitions is determined by a notion of *heat*. The heat between a source node and a sink node is the probability that the sink is reached from the source in any amount of steps. A low *heat* transition is likely to be erroneous and will be pruned. In this chapter, we refer to sk-strings with *coring* as sk-coring. sk-coring is meant to remove errors and we evaluate its quality in our robustness experiment.

4.5 Experiments

Three sets of experiments were conducted to show the usefulness of QUARK in evaluating the performance of the three specification miners described earlier. These experiments aims to measure the accuracy of these miners in discovering various real-world specifications.

Material Simulator models used in these experiments are specifications from (1) programs using XLib and XToolkit intrinsic libraries for X11 windowing system [7], (2) IBM® WebSphere® Business Integration processes from WebSphere® Commerce [180] (3) Simple CVS (Concurrent Versions System) protocol built on top of Jakarta Commons Net [10]. These simulator models are shown in Figure 4.5, 4.6 and 4.7, and are referred to as *x11*, *ws* and *cvs* models respectively. Probabilities are *distributed equally* to transitions from the same source node (not shown in figures).

For each model, 100 experiments were run for each learner with the k parameter set to 1 and then to 3. A total of 1800 experiments were performed. For each experiment, a multi-set of traces was generated from the model using **TraceGen** (Figure 4.3) with parameters N, I and Max set to 10, 10 and 10,000 respectively. No error was introduced to the models.

In analyzing the results, any two results differing in absolute value by less than 1%(0.01) are considered equivalent, as the difference is deemed insignificant.

X11 Windowing Toolkit In [7], Ammons *et al.* described the mining of a specification, shown in Figure 4.5, from several programs using XLib and XToolkit intrinsic libraries for X11 windowing system.

In our experiments, the mining results obtained by the three learners are shown in the table below. k-len corresponds to the k parameter of sk-strings and k-tails algorithms. A default value of 50% for s was used. The columns Recall, Precs. and PS are the QA metrics defined in Section 4.3.

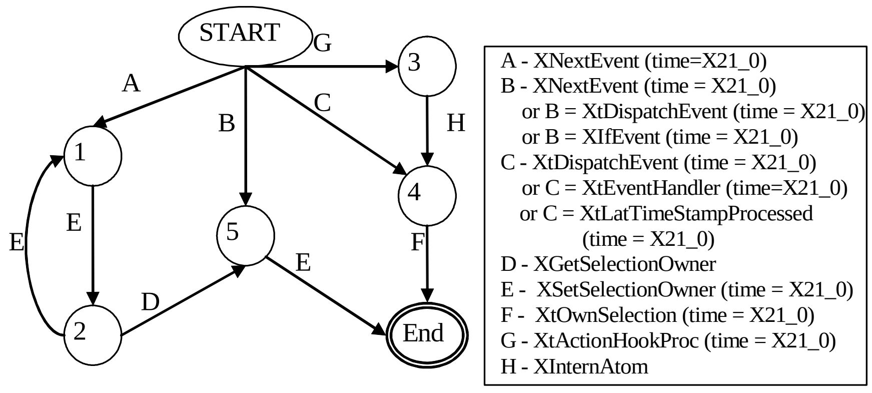

START
A
B
C
G
H
E
E
D
E
F
1
2
3
4
5
End
A - XNextEvent (time=X21_0)
B - XNextEvent (time = X21_0)
or B = XtDispatchEvent (time = X21_0)
or B = XIfEvent (time = X21_0)
C - XtDispatchEvent (time = X21_0)
or C = XtEventHandler (time=X21_0)
or C = XtLatTimeStampProcessed (time = X21_0)
D - XGetSelectionOwner
E - XSetSelectionOwner (time = X21_0)
F - XtOwnSelection (time = X21_0)
G - XtActionHookProc (time = X21_0)
H - XInternAtom

Figure 4.5: X11 Windowing Toolkit

Learner	**k-len = 1**			**k-len = 3**		
	Recall	Precs.	PS	Recall	Precs.	PS
k-tails	1.000	0.000	N/A	0.998	0.313	N/A
sk-strings	1.000	0.654	0.692	0.998	0.883	0.758

<u>Analysis</u> The results show that: (1) k-tails did not learn well at k-len = 1, while sk-strings learnt reasonably well. (2) With bigger k-len, all miners produced more precise automata. (3) sk-strings produced more precise automaton than k-tails.

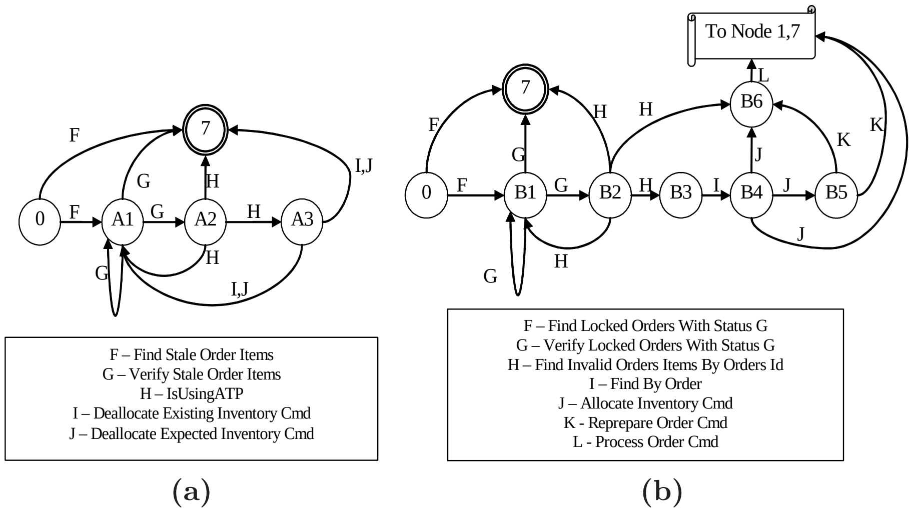

0
A1
A2
A3
7
F
F
G
G
G
H
H
H
I,J
I,J
F – Find Stale Order Items
G – Verify Stale Order Items
H – IsUsingATP
I – Deallocate Existing Inventory Cmd
J – Deallocate Expected Inventory Cmd
(a)
To Node 1,7
0
B1
B2
B3
B4
B5
B6
7
F
F
G
G
G
H
H
H
H
I
J
J
J
K
K
L
F – Find Locked Orders With Status G
G – Verify Locked Orders With Status G
H – Find Invalid Orders Items By Orders Id
I – Find By Order
J – Allocate Inventory Cmd
K - Reprepare Order Cmd
L - Process Order Cmd
(b)

Figure 4.6: WebSphere® Commerce Processes

WebSphere® Commerce. In [180], Zou *et al.* statically extracted workflows describing IBM® WebSphere® Business Integration business process from the IBM® WebSphere® Commerce code. They presented two workflows, in the form of automata, which correspond respectively to (1) the release of expired allocations and (2) the processing of backorders. These are shown in Figures 4.6(a) and (b), respectively. We combine the two automatons into a simulator model by joining their start and end

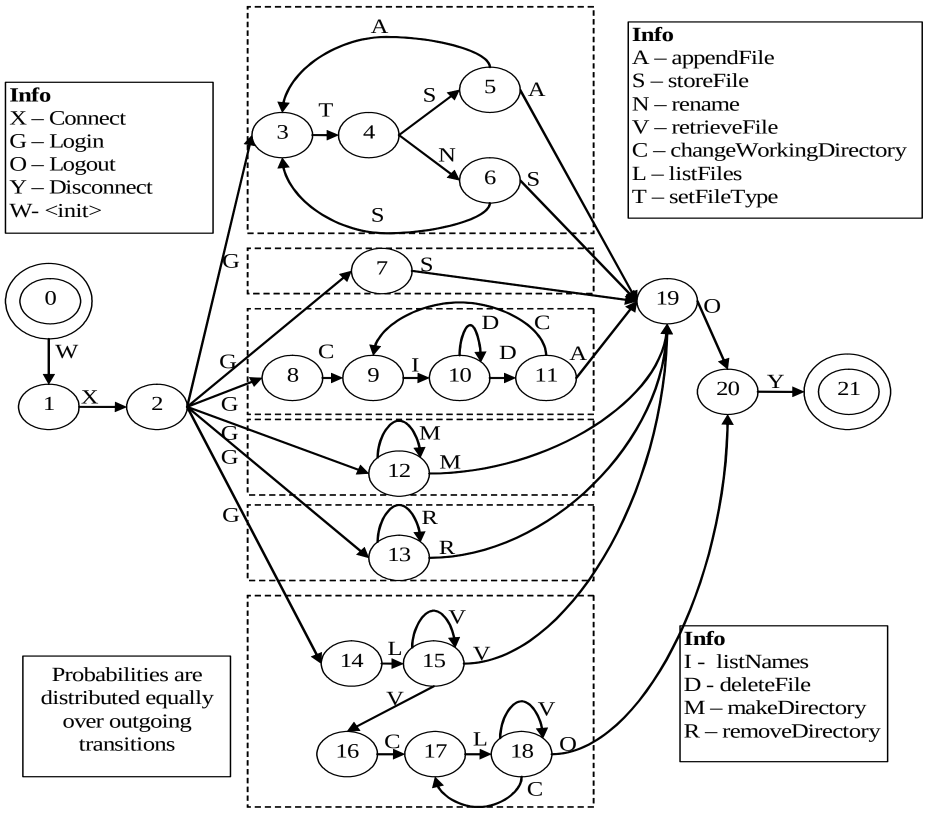


Figure 4.7: CVS Protocol

nodes. Note that, different from *x11* and *cvs* models, this model has more transitions per nodes and more loops (*ie.* it is more "bushy"). The experiment results are tabulated in the following table .

Learner	**k-len = 1**			**k-len = 3**		
	Recall	Precs.	PS	Recall	Precs.	PS
k-tails	1.000	0.000	N/A	0.998	0.597	N/A
sk-strings	1.000	0.536	0.785	1.000	0.538	0.785

Analysis The results show that: (1) k-tails did not learn well at k-len = 1 as compared with sk-strings. (2) Increasing the value of k-len did not improve the performance of sk-strings. (3) sk-strings performed worse than k-tails for k-len=3.

CVS on Jakarta Commons Net Jakarta Commons Net [10] is a set of reusable open source Java code implementing the clients of many commonly used network protocols. We built a simple CVS (Concurrent Versions System) client on top of the FTP library provided by Jakarta Commons Net.

There are six common FTP interaction scenarios in our CVS implementation: Initialization, multiple-file upload, download, and deletion, multiple-directory creation and deletion. All scenarios begin by connecting and logging-in to the FTP server. They end by logging-off and disconnecting from the FTP server. The client side only maintains a record of files backed-up in the FTP server.

All these scenarios are depicted in the automata shown in Figure 4.7. The dashed boxes, from top to bottom, represent upload files, initialization, delete files, make directories, remove directories and download files scenario, respectively.

Compared with *x11* and *ws* models, this model has the most number of nodes, but it remains to be less 'bushy'. The experiments results are tabulated below.

Learner	**k-len = 1**			**k-len = 3**		
	Recall	Precs.	PS	Recall	Precs.	PS
k-tails	1.000	0.000	N/A	1.000	0.000	N/A
sk-strings	1.000	0.226	0.509	0.999	0.017	0.030

Analysis The results show that: (1) k-tails did not learn well at k-len = 1 *and* k-len = 3. (2) At k-len = 3, the performance of sk-strings was degraded. (3) sk-strings performed better than k-tails.

4.6 Robustness and Scalability

Two sets of experiments were conducted to evaluate the robustness and scalability of the three miners. In total, 2400 robustness experiments were run to cover three error-injection levels, four learners and two k-len values. Also, 2400 scalability experiments were run to cover eight different pairs of node-numbers and maximum number of transitions per node, three learners and two k-len values. In sum, 800 different models were used in the scalability experiments (*ie.* experiments with the same settings but for different learners shared the same model and trace multi-set).

Material In the first set of experiments, we evaluated the learners' robustness. We used similar model of X11 Windowing Toolkit (shown in Figure 4.5). However, the model was modified so that it was *without any non-determinism nor repeated use of alphabet assigned to transitions*. This is meant to produce a base model that can be learned (almost) perfectly by all miners. Error nodes and transitions were then injected to the automaton to conduct the robustness tests. The model used with injection of errors (transitions labelled as Z) is shown in the Figure 4.8.

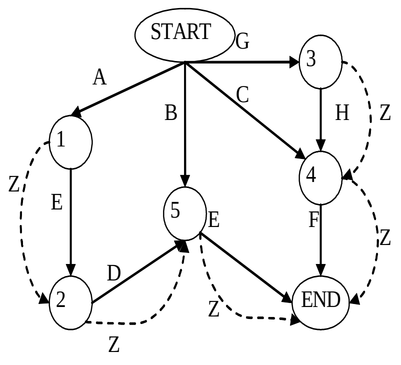


Figure 4.8: Robustness Simulator Model

We expect specification miner to be able to filter error. We compared the inferred automaton with the simulator model shown in Figure 4.8 without error nodes and transitions and recorded their similarity metrics. We generated traces using **TraceGen** (Figure 4.3) with parameters N, I and Max set to 10, 10 and 10,000 respectively. Error was injected at four, eight and ten percentages to the set of generated traces. In each case, we ran 100 experiments and recorded the average performance.

In addition to testing the two learners, we also tested sk-coring (combining sk-strings and coring method).

In the second set of experiments, we evaluated the learners' scalability. Two sub-experiments were conducted, each with a different independent variable. In the first sub-experiment, we varied the number of nodes (by 15, 20, 25, and 30) in the model and maintained the number of outgoing transitions per node to *at most* four (we refer to it as *nodes* experiment). In the second sub-experiment, we varied the maximum number of outgoing transitions per node (by 3,5,7,9) and maintained the number of nodes at 10 (we refer to it as *trans* experiment). For each case, we performed 50 experiments and recorded their average performance.

We generated traces using **TraceGen** with parameters N, I and Max set to 10, 10 and 10,000 respectively. No error was injected to the system. Since we imposed a cap of Max traces, there might be a concern that training trace-set does not satisfy the coverage criterion by *merely* generating up to Max traces. Fortunately, this did not happen that often, as only 18 out of 2400 experiments reached the cap; for all other experiments, the coverage criterion was met without the need to generate Max traces.

Robustness Experiment Results These are tabulated in the following table . Column

E% indicates the percentiles of erroneous traces.

Info		**k-len = 1**			**k-len = 3**		
E%	Learner	Recall	Precs.	PS	Recall	Precs.	PS
4%	k-tails	1.000	0.000	N/A	1.000	0.763	N/A
	sk-strings	1.000	0.944	0.947	1.000	0.949	0.948
	sk-coring	0.756	0.956	0.831	0.764	0.965	0.838
8%	k-tails	1.000	0.000	N/A	1.000	0.645	N/A
	sk-strings	1.000	0.892	0.944	1.000	0.899	0.944
	sk-coring	0.781	0.903	0.828	0.795	0.916	0.835
10%	k-tails	1.000	0.000	N/A	1.000	0.621	N/A
	sk-strings	1.000	0.864	0.935	1.000	0.872	0.933
	sk-coring	0.754	0.873	0.800	0.761	0.900	0.803

Analysis The presence of error affected miners' precision. We rank the learners' precisions in decreasing order wrt the degrees of their susceptibility to errors as follows: k-tails, sk-strings, and sk-coring. Also, increasing k-len value *did not* significantly reduce the susceptibility to error.

For sk-coring and sk-strings, losses in precision were about the same as the percentages of error injected. For k-tails however, the losses of precision were much larger. Although sk-coring removed error and improved precision, the ability to recall was adversely affected. Errors are removed along with correct behaviors.

Scalability Experiment Results The results of our two sub-experiments are shown below. Column "N/TN" corresponds to the number of nodes and the maximum number of transitions per node in the simulator models.

Info		**k-len = 1**			**k-len = 3**		
N/TN	Learner	Recall	Precs.	PS	Recall	Precs.	PS
15/4	k-tails	1.000	0.002	N/A	0.999	0.195	N/A
	sk-strings	1.000	0.094	0.152	0.997	0.296	0.344
20/4	k-tails	1.000	0.004	N/A	0.997	0.138	N/A
	sk-strings	1.000	0.025	0.059	0.997	0.338	0.371
25/4	k-tails	1.000	0.007	N/A	0.998	0.089	N/A
	sk-strings	1.000	0.008	0.029	0.997	0.123	0.197
30/4	k-tails	1.000	0.008	N/A	1.000	0.064	N/A
	sk-strings	1.000	0.007	0.031	0.999	0.079	0.105

Info		**k-len = 1**			**k-len = 3**		
N/TN	Learner	Recall	Precs.	PS	Recall	Precs.	PS
	k-tails	1.000	0.002	N/A	0.998	0.201	N/A
10/3	sk-strings	1.000	0.165	0.283	0.992	0.928	0.913
	k-tails	1.000	0.004	N/A	0.980	0.494	N/A
10/5	sk-strings	0.997	0.294	0.375	0.976	0.626	0.614
	k-tails	1.000	0.007	N/A	0.963	0.446	N/A
10/7	sk-strings	0.999	0.142	0.203	0.960	0.420	0.173
	k-tails	0.997	0.008	N/A	0.934	0.467	N/A
10/9	sk-strings	0.999	0.082	0.141	0.979	0.338	0.339

Analysis For all learners, their recalls were always greater than 90%. The average recalls for k-len = 1 and 3 were 99.6% and 96.6% respectively. In each experiment setting, recalls of different learners only differs by less than 5%. However, the precision results were less glossy. Even for k-len = 3, there were cases where precisions were less than 10% (*see* k-len=3;N=30;TN=4). The average precision for k-len = 1 and 3 are 14.2% and 45.3% respectively.

sk-strings' precision is almost always equivalent to or better than k-tails', except for very "bushy" automaton (*see* k-len=3;N/TN=10/9). Similar results were reported in the *ws* experiment described in Section 4.5. k-tails did not perform well with k-len=1 (precision < 1%). Increasing the "bushiness" of models – by increasing TN from 1 to 9 for 10-node automatons – improved the relative performance of k-tails over sk-strings.

4.7 Conclusion

In this work, QUARK, a framework to empirically assess quality of automaton-based specification miner is proposed. Our assessment of specification miners is guided by the conviction that: A good miner should have good recall, good precision and be able to retain probability distribution of the original specification (for PFSA learner). In addition, it should remain robust in the presence of error, and scalable in learning from traces generated from large automata.

To demonstrate the effectiveness of QUARK in assessing specification miners, we use it to assess three types of automaton-based specification miners: (1) *k-tails FSA learner* (2) *sk-strings PFSA learner* and (3) and (*sk-coring*) an extension of sk-strings by Ammons *et al.*.

Experiments using real-world specifications from X11 Windowing Toolkit (*x11*), IBM WebSphere® Commerce (*ws*) and Concurrent Versions System (*cvs*) were performed. Results show that for *x11* and *cvs* models, sk-strings performed better than k-tails. For *cvs* model, k-tails did not learn well even when k is set to 3. However, for *ws* model, k-tails performed slightly better than sk-strings. *ws* model is more "bushy" (i.e., more transitions per node) than the other two models.

Simulated experiments measuring robustness and scalability of the miners were also performed. The results indicate that specification miners typically have good recall but poor precision in the presence of error, resulting in inaccurate inferred specification. In the scalability experiments, increasing the number of nodes in simulator models can reduce recall; increasing the number of transitions per node in simulator models leads to narrowing in the performance gap between k-tails and sk-strings. For very "bushy" simulator models, k-tails perform better than sk-strings.

In summary, QUARK is specially designed to assess automaton-based *specification* miners rather than generic automaton miners, since: **(1)** Generated traces are viewed as abstract representation of actual program traces; **(2)** Trace generation conforms to 'code and branch coverage'-based criterion; **(3)** Various models extracted from real software have been used; **(4)** Synthetic models are generated following the principle of locality of reference; and **(5)** Metrics proposed are directly related to software engineering concerns.

The framework and metrics developed here do not only provide us a means for quality assurance measurement. They also provide hints for development of better specification miners to meet the stringent quality assurance requirements. While we acknowledge the usefulness of producing imperfect learned specification in meeting certain software engineering tasks, we also believe that improvement in specification miners' quality will greatly enhance their usefulness.

CHAPTER V

IMPROVING QUALITY OF AUTOMATON-BASED MINERS

In [7], Ammons *et al.* employ a machine-learning approach to discover program specifications in the form of automata by analyzing program execution traces. In this work, we leverage the work by Ammons *et al.* by proposing a novel specification mining architecture. Specifically, this chapter describes our proposed architecture that achieves specification mining through pipelining of four functional components: Error-trace filtering, clustering, learning, and automaton merging. We refer to our specification mining architecture as SMArTIC (Specification Mining Architecture with Trace fIltering and Clustering). The purpose of SMArTIC is to improve the accuracy, robustness and scalability of existing specification miners.

Contrary to other works done in automaton-based specification mining, we choose *probabilistic FSA* (PFSA) instead of (non-probabilistic) FSA as our learning target. PFSA is more expressive than FSA, since it provides details on the probabilities of state transitions. This enables detection of frequently-used interaction patterns (*e.g.*, "*open (read)* close*" pattern in a resource-access protocol) or sub-protocols within a specification, analogous to the idea of hotspots found in program execution [77].

We conducted experiments to show the benefit of our proposed mining architecture. In our experiments, we try to mine the API interaction protocol for a client of the Jakarta Commons Net open-source library [10]. We performed objective measurements on the quality of our mining architecture through the evaluation of *accuracy*, *robustness* and *scalability* of the architecture.

As we recall from Chapter 4, *accuracy* refers to the extent of an inferred specification being representative of the actual specification. *Robustness* refers to its sensitivity to errors present in the input data. *Scalability* determines a specification miner's ability to infer large specifications.

The contributions of this study are as follows:

1. We propose an automated method to remove erroneous traces based on the inference of strong program properties in the form of rules is proposed.

2. We present a novel method to cluster traces exhibiting similar characteristics.

3. We employ a novel method to merge sub-specifications in the form of automata.

4. We propose a novel architecture to improve the quality of specification miners via a pipeline comprising of trace filtering, trace clustering, automata learning and automata merging blocks.

The outline of the chapter is as follows: Section 5.1 lays down the hypotheses which drive our construction of SMArTIC, and discusses the detail components in SMArTIC. Section 5.2 describes our experiments on the Jakarta Commons Net [10] open source library. Section 5.3 describes more comprehensive experiments and results using various simulated models. We conclude in Section 5.4.

5.1 Mining Architecture

SMArTIC aims to increase a miner's precision, robustness and scalability by employing several novel techniques in specification mining. It leverages on the lessons learnt and experience accumulated from the past work done in this and related areas (*eg.*, [7], [52], [95], *etc.*). The success of SMArTIC hinges on the affirmation of the following two hypotheses:

Hypothesis 1 *Mined specifications will be more accurate when erroneous behavior is removed before learning than when they are removed after learning.*

Hypothesis 2 *Mined specifications will be more accurate when they are obtained by merging the specifications learned from clusters of related traces than when they are obtained from learning the entire traces.*

Hypothesis 1 is made from observing the system built by Ammons *et al.* [7]. In their work, a *coring* method is employed to remove erroneous transitions from the mined automaton. As this is performed on the output automaton, erroneous transitions are

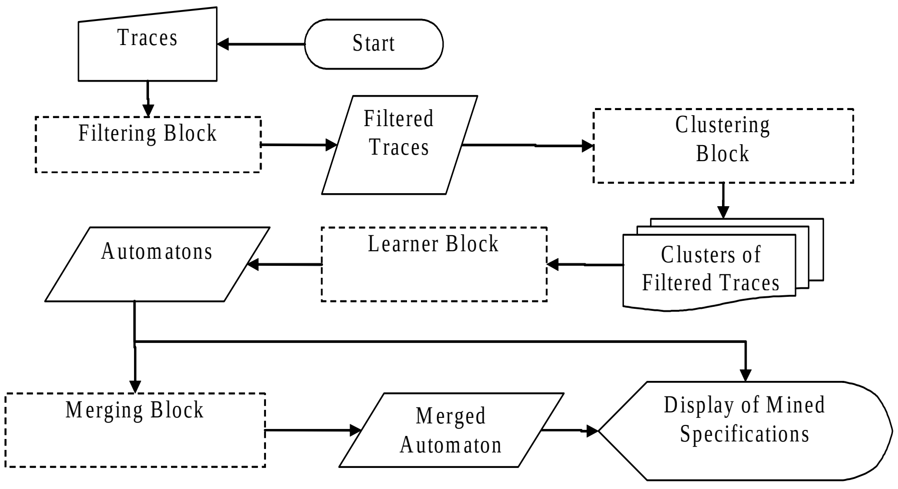


Figure 5.1: SMArTIC Structure

included during mining. Consequently, the performance of learning may be degraded. Moreover, pruning of transitions in an automaton may cause damage to the automaton, such as breaking an automaton into parts. This may then require substantial repairing of the automaton, and negate the effect of learning.

We believe that pruning of erroneous transitions should be done *before* learning. Consequently, we include a *filtering* process before the learning process in SMArTIC, as we shall describe in Section 5.1.1.

Hypothesis 2 is derived from the observation that the existence of unrelated traces may negate the effect of learning via generalization; *i.e.*, they can lead to over-generalization. Therefore, by clustering related traces and performing learning on each cluster, the effect of inaccuracies in learning can be localized to within a cluster. We believe this will result in a more accurate mined specification. Consequently, we include a clustering process in SMArTIC, as we shall describe in Section 5.1.2.

The overall structure of SMArTIC is shown in Figure 5.1. It comprises four major blocks, namely filtering, clustering, learning and merging blocks. Each block is in turn composed of several major elements. The filtering block filters erroneous traces to address the robustness issue. The clustering block divides traces into groups of 'similar' traces to address the scalability issue. The learning block generates specifications in the form of automata. The merging block merges the automatons generated from each cluster into a unified one.

5.1.1 Filtering Block

The filtering block aims to filter out erroneous traces based on common behavior found in a multi-set of program traces. To filter well, we need a representation of common behavior which is intuitive enough to be used for filtering. Since a trace is a temporal or sequential ordering of events, representing common behavior by "statistically significant" temporal rules will be appropriate. Temporal rules based on full set of temporal logics might be a good candidate, but it is desirable to have a more light-weight solution.

Given a set of traces, we would like to generate, through mining, rules of the form *pre* $\rightarrow$ *post*, where both *pre* and *post* are sequences of alphabets occurring in traces. Semantically, such a rule has the following temporal interpretation: Given $pre = \langle a_1, \ldots, a_m \rangle$ and $post = \langle b_1, \ldots, b_n \rangle$, the temporal interpretation of $pre \rightarrow post$ is expressed in Linear Temporal Logic (LTL) notation [79] as

$$G(XG(a_1 \rightarrow \ldots \rightarrow XG(a_m \rightarrow XF(b_1 \wedge \ldots \wedge XF\ b_n))))$$

The symbols 'G,' 'F' and 'X' above refer to LTL operators. The operator 'G' specifies that *globally* at every point in time a certain property holds. The operator 'F' specifies that a property holds either at that point in time or *finally (eventually)* it holds. The operator 'X' specifies that a property holds at the *next* point in time. A trace can be viewed as a series of alphabets each occurring at a particular point in time. Time increases or advances from the start to the end of the trace.

As an example, a rule $\langle a \rangle \rightarrow \langle b, c \rangle$ asserts that at any point in the trace when a occurs, b must eventually occur after a, and c must also eventually occur after b.

There are two commonly used measures of "statistical significance" in the field of data mining, namely, *support* and *confidence* (*c.f* [61]). Support of a rule *pre*$\rightarrow$*post* is the number of *trace points* exhibiting the property $pre \rightarrow post$. Confidence of the rule is the ratio of the number of *trace points* exhibiting the property $pre \rightarrow post$ to those exhibiting the property *pre*.

Rules having high confidence and reasonable support can be considered as "statistical" invariants. They thus characterize some general behaviors of a subgroup of traces. To detect outliers or anomalous traces in the input trace set, only rules with high but less than 100% confidence will be useful. Rules of 100% confidence will not be useful in detecting outliers as no trace in the input trace set violates the behaviors captured by the rules. We call rules of *pre→post* format and exhibiting the above properties as *outlier detection rules*.

Mined outlier detection rules will be used to filter out likely errors or unlikely behaviors. Any trace t_x of the following format $\langle a_1, \ldots, a_i, \ldots, a_{end}\rangle$ will be filtered out (as an outlier) by a rule-set RS iff the following holds:

$$
\begin{aligned}
&\exists pre \rightarrow post \in RS. \\
&\quad (\exists a_i, a_j.\ (1 \leq i \leq j) \wedge \neg(\langle a_i, \ldots, a_{j-1}\rangle \text{ satisfies } pre) \\
&\qquad \wedge(\langle a_i, \ldots, a_j\rangle \text{ satisfies } pre) \\
&\qquad \wedge\neg(\langle a_{j+1}, \ldots, a_{end}\rangle \text{ satisfies } post))
\end{aligned}
$$

A trace segment satisfies the *pre* or *post* of a rule if it is a super-sequence of the *pre* or *post*, respectively. The algorithm to filter out those traces that deviate from the general behaviors is depicted in Figure 5.1.1.

```
Procedure Filter
Inputs:
Traces: A set of traces
Rules: Outlier detection rules
Outputs:
Filtered: A set of filtered traces
Err: A set of traces deviant from rules
Method:
1:  $Filtered, Err = \emptyset$
2:  For each trace $t$ with format $\langle a_1, \ldots, a_i, \ldots, a_{end}\rangle$ in $Traces$
3:      Let $Satisfy = \emptyset$
4:      For each rule <pre,post> in Rules
5:          For each $\langle a_i, \ldots, a_j\rangle$ substring of $t$ satisfying pre
6:              If $a_{j+1} \ldots a_{end}$ does not satisfy post
7:                  $Err = Err \cup \{t\}$
8:                  Break to the first for-loop
9:      $Filtered = Filtered \cup \{t\}$
10: Output $Filtered, Err$
```

Figure 5.2: Filtering Traces using Mined Rules

Implementation-wise, the structure of the filtering block is as shown in Figure 5.3. Outlier detection rules can be extracted efficiently by adding pre and post processing steps to a closed sequential pattern miner, BIDE [167]. In this study, we will only approximate the confidence and support values of mined rules. For algorithms extracting rules with exact support and confidence values, please refer to Chapter 7. The end result

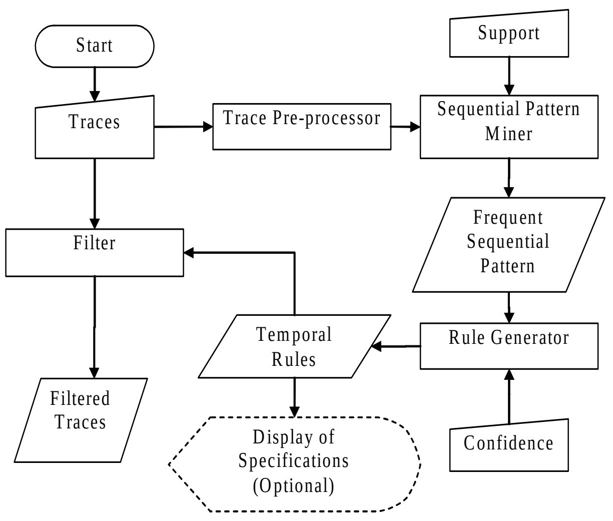


Figure 5.3: Filtering Block Structure

of the filtering block is a multi-set of filtered traces.

Implementation Details. Sequential pattern mining takes as input SA (a set of sequences) and min_sup (minimum support level). It then reports subsequences (or pattern) contained by at least min_sup number of sequences in SA. The number of such supporting sequences in SA is called 'support'. Subsequences having support more than min_sup are called 'frequent'. Sequential pattern miner will return both 'frequent' subsequences and its 'support'. Given a subsequence s, the support of s is denoted as sup(s).

Collected program traces, each of which is a sequence of method calls, can be considered as a set of sequences SA. Inputting collected program traces to a sequential pattern miner will generate a set of subsequences (or patterns) of method calls that is supported by many traces.

A program often contain loops. Subsequence (or pattern) of method calls can be repeated more than once in a trace. Rather than counting the number of traces, we should count the number of locations within traces (*i.e. temporal points*) where a pattern of method calls appear. 'Frequent' pattern of method calls should be based on the number of supporting *temporal points* rather than the number of supporting traces.

To facilitate approximate counting of supporting *temporal points* rather than traces, we perform pre-processing on input traces. Our pre-processing stage converts original multiset of traces (T_{Orig}) to its superset (T_{Result}) to facilitate approximate counting of *trace points* without changing the behavior of underlying sequential pattern miner. For every t ($\langle t_0,\ldots,t_{end}\rangle$) in T_{Orig},

$$\{\langle t_i,\ldots,t_j\rangle | (\langle t_i,\ldots,t_j\rangle \text{ suffix\_of } t) \wedge (\exists k.(k < i) \wedge t_k = t_j\} \cup \{t\}$$

is added to multiset T_{Result}. Inputting T_{Result} to a sequential pattern miner will generate frequent patterns of method calls supported by a substantial number of *temporal points.*

Any subsequence of a frequent subsequence will also be frequent (*i.e.* apriori property – *c.f.* [61]). Hence, a long and frequent subsequence will generate a combinatorial number of frequent subsequences (*c.f.* [174]). Long frequent subsequences of method call might appear due to 'wrapper' effect of deep class hierarchy, might be an effect of decomposition of a complex methods to a series of simpler ones, or might correspond to initialization and termination of a protocol (*e.g. connect* -> *login* -> *logout* -> *disconnect* in FTP protocol).

To reduce the combinatorial number of frequent subsequences, closed sequential pattern miner has been proposed [174]. A *closed* sequential pattern, is a frequent pattern which is not a subsequence of another frequent pattern with the same support. A set of closed sequential pattern captures the same information as a full set of sequential pattern without being combinatorial in number. BIDE [167] is an optimized miner of closed sequential pattern. Experimental results have shown its run-time is linear w.r.t. input size (number of sequences $\times$ avg length of sequences). We run BIDE with preprocessed traces and support level *Sup* to generate a set of closed sequential patterns.

Frequent sequential patterns returned by BIDE can be post-processed into *pre*→*post* rules. Given two frequent sequential patterns <A> and <A,B,C> with support s1 and s2, a rule $A \rightarrow BC$ can be generated with 'confidence' s2/s1 (*c.f* [157, 61]). Given a subsequence f which is frequent, sup(f) = $Max_{(c\ \in\ Closed\ \wedge f\ is\ subsequence\ of\ c)}$. sup(c). Hence, support of a frequent subsequence can be inferred from *Closed.* The conversion algorithm to generate rules from closed sequential pattern is as shown in Figure 5.4.

The algorithm receives as input a set of closed sequential patterns (*Closed*) and a minimum confidence level (*Conf*). It first builds a prefix tree (or trie) of the closed sequential patterns. A prefix tree is a tree where there is one unique series of nodes for every common prefix. A closed sequential pattern can be considered a sequence of events. For each node q ∈ trie, q.event,and q.prefix denotes their corresponding event and sequence of events from trie's root to q. A node's owner, q.owner, is the closed pattern sharing prefix q.prefix that has the maximum count. The support of q.prefix (denoted as q.count) is simply the count of q.owner. A closed pattern c 'shared' a trie node q, if q.prefix is a prefix of c.

The trie is later traversed to locate 'interesting' nodes. A node q is interesting if, $\exists$ qd. ((qd child-of q) $\wedge$ ($Conf \leq$ (qd.count/q.count) < 1)). From such nodes, rules with confidence $\geq Conf$ but < 1 of the form q.prefix $\rightarrow$ post, where post starts with qd.event can be generated.

The algorithm will extract *pre→post* rules having confidence at least $Conf$ but less than 100%. All rules generated will also be frequent since they are generated from closed sequential patterns. Some rules that can be inferred from others were pruned since they are redundant.

```
Procedure Generate Rules
Inputs:
Closed:  A set of closed sequential patterns
Conf:  Minimum confidence
Outputs:
Rules:  Outlier detection rules
Method:
Step 1: Trie building and traversal
1:  Let Trie =  Build a prefix tree from Closed with each node representing
                an event
2:  For each node p in Trie do
3:     Set p.prefix = Seq of events from root to p (inclusive) in Trie
4:     Set p.owner =  f, where f $\in$ Closed $\wedge$ f shared p $\wedge$
                          $\forall_{g \in Closed \wedge g\ shared\ p}$ . (f.count $\geq$ g.count)
5:     Set p.postfix =  post, where p.owner = p.prefix concatenated with post
6:     Set p.count =  p.owner.count
7:  Let Interesting =  {q|q $\in$ Trie $\wedge$ $\exists$ qd.((qd child-of q) $\wedge$
                          ($Conf \leq$ q.count/qd.count $< 1$))}
Step 2: Rule generation
8:  For each qi $\in$ Interesting do
9:     Let qdesc =  {qd | (qd  child-of  q) $\wedge$
                        ($Conf \leq$ (q.count/qd.count) $< 1$)}
10:    Let RS =  Generate rules pre $\rightarrow$ post where, pre == qi.prefix $\wedge$
                    post == qdesc.postfix
11:    Add RS to Rules
12:  Output  Rules
```

Figure 5.4: Rule Generation

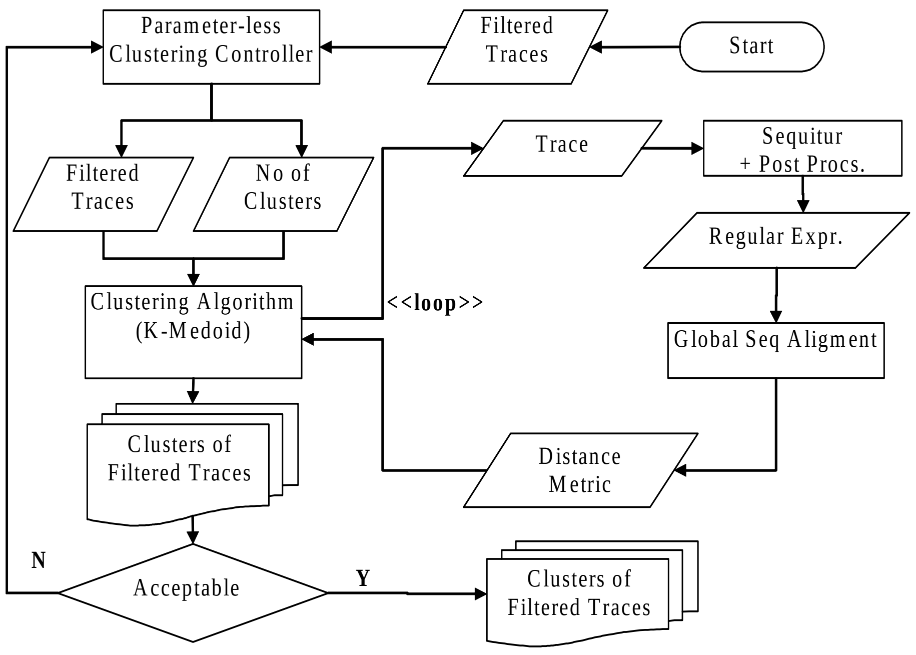


Figure 5.5: Clustering Block Structure

5.1.2 Clustering Block

Input traces might be "mixed up" from several unrelated scenarios, *e.g.* members of input trace set might represent various usage patterns of an API/component. Grouping unrelated traces together for a learner to learn might multiply the effect of inaccuracy in learning a scenario. Such inaccuracy can be further permeated into other scenarios through generalization.

The clustering block converts a set of traces into groups of related traces. Clustering is meant to *localize inaccuracy* in learning one sub-specification and prevent the inaccuracy from being permeated to other sub-specifications. Furthermore, by grouping related traces together, better generalization (*i.e.*, less over-generalization) can be achieved when learning from each cluster.

Two major issues pertaining to clustering are: the choice of clustering algorithm and an appropriate similarity metric; *ie.*, measurement of similarity between two traces. The performance of the clustering algorithm is affected by appropriate similarity/distance metric. Different clustering algorithms learn differently in terms of accuracy, efficiency and the level of user interaction required. (*c.f.* [61]) The general structure of the clustering block is as shown in Figure 5.5.

```
Procedure PLess–KMedoid
Inputs:
IPTraces :   Set of unique input traces to be split to clusters
Outputs:
Clusters:   Set of clusters taking into account similarity within each cluster and
            differences among different clusters
Method:
1:  Let FeatureScoreList = {}
2:  Let IsLocalMaxima = false
3:  Let Clusters = {}
4:  For (k=1;k<|IPTraces| && !IsLocalMaxima;k++)
5:     Set Clusters = Call k-medoid (IPTraces, k)
6:     Let FeatureScore = Call CalculateScore (Clusters)
7:     FeatureScoreList.Add (FeatureScore)
8:     Set IsLocalMaxima = Check whether a local maxima has been found in
                           FeatureScoreList by noting gradient change
9:  Output Clusters
```

Figure 5.6: PLess KMedoid Algorithm

5.1.3 Clustering Algorithm

We use a classical off-the-shelf clustering algorithm for our purpose, namely the k-medoid algorithm [93].[1] The k-medoid algorithm works by computing the distance between pairs of data items based on a similarity metric; this corresponds to computing the distance between pairs of traces. It then groups the traces with small distances apart into the same cluster. The *k* in k-medoid is the number of clusters to be created.

In our implementation, we adapt the Turn* algorithm presented by Foss *et al.* [54] into the k-medoid algorithm. The Turn* algorithm can automatically determine the number of clusters to be created by considering the similarities within each cluster and differences among clusters. Our algorithm will repetitively increase the number of clusters. For each repetition, it will divide datasets into clusters and evaluate a measure of similarities within each cluster and differences among different clusters. The algorithm will terminate once a local maximum is reached. The algorithm is shown in Figure 5.6, which makes call to the algorithm CalculateScore shown in Figure 5.7. In that figure, at each program point marked with *(*)* a similarity measure between two traces will be calculated. This similarity measure is described in sub-section 5.1.4.

[1] Another algorithm is K-means algorithm. It is not used here since we would be required to define the average/mean of a group of strings which might not be meaningful. Also, k-medoid algorithm has been found to be more accurate than k-means since it is less susceptible to outliers [61].

```
Procedure CalculateScore
Inputs:
IPClusters : Clusters of data items
Outputs:
Score: Feature score considering similarities within each cluster and differences
       among clusters
Method:
1:  Let MedoidCenters = {c.Medoid|c ∈ IPClusters}
2:  Let MedoidCentersCluster = Call k-medoid (MedoidCenters, 1)
3:  Let RepMedoid = MedoidCenterCluster.Medoid
4:  Let SimMedoids = 0.0
5:  Let SimWithinClusters = 0.0
Step 1: Calculate similarities between clusters medoids
6:  For each medoid in MedoidCenters not equals to RepMedoid do
7:      Set SimMedoids += Similarity of MedoidCenters[i] and RepMedoid
                                                     (see text) (*)
8:  Average SimMedoids by |MedoidCenters|
Step 2: Calculate similarities within each clusters
9:  For each cluster in IPClusters do
10:   Let AClusterSim = 0.0
11:   For each dataitem in cluster != cluster.Medoid
12:       Set AClusterSim += Similarity of dataitem and cluster.Medoid
                                                     (see text) (*)
13:   Average AClusterSim by |cluster|
14:   Set SimWithinClusters += AClusterSim
15: Average SimWithinClusters by |IPClusters|
16: Let Score = SimWithinClusters − SimMedoids
17: Output  Score
```

Figure 5.7: Calculate Score Algorithm

5.1.4 Similarity Metric

In many applications, comparisons between two data items are relatively clear – sometimes it only involves a simple subtraction of two numbers – *e.g.*, (Average profit of company x) - (Average profit of company y), *etc.* However, a comparison of two program traces is neither so clear cut nor easily obtained.

Our first idea is to use *global sequence alignment* [164] to measure the distance between two traces. This alignment is frequently used to obtain similarity metrics of two DNA sequences. Its main idea is to insert "dash" or spaces within strings to obtain the most accurate matching of two strings. Different from the Knuth-Morris-Prat (KMP) algorithm [36], the sequence alignment algorithm finds the best approximated alignment(s) rather than the occurrence of an exact match. Alongside the best alignment(s), an overall similarity score will also be reported. We use this score as the similarity metric between the two program traces.

This first idea does not work well in practice because, contrary to normal strings, program traces exhibit some characteristics which make it difficult to measure similarity by a simple alignment of two traces. Specifically, a trace might only be different than another due to different numbers of loop iterations during program execution. As an example, consider the following program segment:

```
function APICLIENT_ABCD (outer_iter, inner_iter[])
{
    for (int j=0;j<outer_iter;j++) {
        int k=0; Call API.A ();
        do{ k++;
            Call API.B();
            Call API.C();
        }while (k<inner_iter[j]);
        Call API.D ();
    }
}
```

Suppose that *APICLIENT_ABCD* is a client function of an API. It is conceivable that the API interaction patterns for various runs via the function call *APICLIENT_ABCD* with different input parameters should be grouped together. So, for a run with parameters *outer_iter = 2* and *inner_iter = [2,3]*, the generated trace is $\langle A, B, C, B, C, D, A, B, C,$

$B, C, B, C, D\rangle$. For another run with *outer_iter = 1* and *inner iter = [1]*, the generated trace is $\langle A, B, C, D\rangle$. Now, if we simply align these two strings, even in their best alignment their similarity score will be too low for them to be grouped into the same cluster.

Our solution to the above problem is to instead compare the regular expression representations (only parentheses and "+" quantifier are used) of the two traces rather than their actual sequence of alphabets. Converting to its regular expression, the first trace will be $\langle (A, (B, C)+, D)+\rangle$, which corresponds closely to $\langle A, B, C, D\rangle$.

We obtain the regular expression representation by converting a trace to its hierarchical grammar representation using Sequitur [138]. The output of sequitur will be post-processed to construct the regular expression representation and then be fed in as input to global sequence alignment. With these we obtain a method to find a reasonable metric for the similarity of program traces.

5.1.5 Learning Block

Although temporal rules have also been used ([176]) to capture certain information of a program specification, automata have been commonly used in capturing specifications, especially protocol specifications. The purpose of this learning block is to learn automatons from clusters of filtered traces.

This block is actually a placeholder in our architecture. Different PFSA specification miners can be placed into this block, as long as they meet the input-output specification of a learner. Once a learner is plugged in, it will be used to mine the traces obtained from each cluster. At the end, the learner produces one mined automaton for each cluster.

In the current experiment, we choose to use a PFSA specification miner that has been used for software specification mining earlier, *i.e.* sk-strings learner [149].

Sk-string learner is used by Ammons *et al.* to mine the specification of the X11 windowing library [7]. It is an extension of the k-tail heuristic algorithm of Biermann and Feldman [15] for learning stochastic automata. The description of k-tails and sk-strings algorithms has been provided in Chapter 4.

5.1.6 Merging Block

The merging process aims to merge multiple PFSAs produced by the learner into one such that there is no loss in precision, recall and likelihood before and after the merge. Equivalently, the merged PFSA accepts exactly the same set of sentences as the combined set (*i.e., union*) of sentences accepted by the multiple PFSAs.

The primary purpose of the merging process is to reduce the number of states residing in the output PFSA by collapsing those transitions behaving "equivalently" in two or more input PFSAs, thus improving scalability.

Merging of two PFSA's involves identification of *equivalent* transitions between the two PFSAs. The output PFSA contains all transitions available in the input PFSAs, with each set of equivalent transitions represented by a single transition. Two identically-labelled transitions from two different PFSAs are considered *equivalent* if one the following conditions holds: (1) Both their source nodes share the same set of suffixes with corresponding probabilities, (2) Both their sink nodes share the same set of prefixes with corresponding probabilities.

Given a node n in a PFSA and a string accepted by PFSA that involves a transition, δ say, emitted from n, we define a suffix of n with respect to the string as the suffix of that string beginning with δ. Similarly, we define a prefix of n as the prefix of that string ended just before δ. For instance, given the transition $n_1 \xrightarrow{\delta} n_2$, and a string

$$\langle t_1, t_2, \cdots, t_{m-1}, \delta, t_{m+1}, \cdots, t_p \rangle$$

A suffix of n_1 is the string $\langle \delta, t_{m+1}, \cdots, t_p \rangle$, and a prefix of n_1 is the string $\langle t_1, t_2, \cdots, t_{m-1} \rangle$.

In the case where no transition emitting from n appears in the string, both the prefix and suffix of n with respect to that string are just null.

Extending from the definitions above, the set of prefixes/suffixes of n in a PFSA is the set of prefixes/suffixes of n with respect to all (possibly infinite) strings accepted by the PFSA.

The definition of equivalent transitions above admits closure property: If two transitions are equivalent as defined by sharing the same set of suffixes with corresponding probabilities, then each of the transitions in the suffix-set is also an equivalent transition. The same behavior can be observed from equivalent transitions sharing the same set of prefixes.

The merging process also ensures that the likelihood of traces generated by the output PFSA remains the same as that of the combined input PFSAs. More specifically, let $\mathcal{A}$ be the output PFSA and $\mathcal{A}_i$ $(i = 1..n)$ be the input PFSAs. Let s be a sentence in $\mathcal{A}$, then

$$p_{\mathcal{A}}(s) = \sum_{i=1}^{n} w_i \cdot p_{\mathcal{A}_i}(s)$$

where $p_{\mathcal{A}}(s)$ and $p_{\mathcal{A}_i}(s)$ represent the probabilities of the sentence s generated by the PFSA $\mathcal{A}$ and $\mathcal{A}_i$ respectively. $p_{\mathcal{A}_i}(s)$ is assigned to 0 if s is not a sentence in $\mathcal{A}_i$. w_i is the weightage given to each cluster hosting their own input PFSA; it is the ratio of the number of traces in the cluster to the number of total traces in the entire system.

Implementation-wise, the closure property of equivalent transitions enables an incremental detection of such equivalent transitions, starting from either the start node (for finding prefixes) or the end node (for finding suffixes) of a PFSA.

Implementation Details The merging process is performed iteratively by merging 2 PFSAs at a time. Let's call them X and Y for ease of reference. The algorithm for automaton merger is as shown in Figure 5.8.

There are four steps in automaton merge algorithm. They are (1). Handling of exceptional cases, (2). Creation of a set of pairs of nodes that can be unified or merged together, (3). Computation of partially merged automata, and (4). Joining up partially merged automata to a single merged automaton.

We consider it an exceptional case if there is a transition sinking in start node. This is such since the algorithm assumes that the start nodes of the two automatons are unifiable/mergable. However, if one automaton has start node as sink node of a transition and not the other, the two start nodes are not unifiable. In step 1, if this exceptional case happens, a new start node is added with ϵ-transition to the original start node. Also, we assume that each of the automata mined by the learning block will have a unique end node.

Two nodes of X and Y are considered *unifiable* if they shared a common set of prefixes. In step 2, we first create a list of pairings between x $\in$ X.Nodes and y $\in$ Y.Nodes where x and y are unifiable (*i.e.* creation of unifiable list). Also, two nodes of X and Y are considered *mergable* if they shared a common set of suffixes. Next, we create the list of pairs of nodes that are mergable (*i.e.* creation of mergable list).

The two lists are created by generating and solving constraints. A unifiable list is created by marking pairs of equivalent nodes of X and Y starting from the start nodes top-down (*i.e.*, towards the end node). A mergable list is created by marking pairs of equivalent nodes of X and Y starting from the end node bottom-up (*i.e.*, towards the start node). To check whether two nodes $xNode$ and $yNode$ are unifiable, only transitions having $xNode$ or $yNode$ as their sources (*i.e.* $x.Next$ and $y.Next$) need to be considered. Similarly, for $xNode$ and $yNode$ to be mergable, only transitions having $xNode$ or $yNode$ as sink (*i.e.* $x.Prev$ and $y.Prev$) need to be considered.

In step 3, we create two partially merged automata. The first one *Top* is created by traversing from X's and Y's root nodes top-down concurrently according to the unifiable list created in step 2. Next, we create another partially merged automata *Bottom* by traversing from X's and Y's end nodes bottom-up concurrently according to the mergable list created in step 2.

In step 4, we join together these two partially merged automata by adding in nodes and transitions in X and Y that have not been included in the merged automaton. The weightage of X and Y is used to assign appropriate probabilities to the transitions.

5.2 Case Study: Jakarta Commons Net

Jakarta Commons Net [10] is a set of reusable open source Java code implementing the client side of many commonly used network protocols. We built a simple CVS (Concurrent Versions System) functionality on top of an FTP library provided by Jakarta Commons Net.

5.2.1 Protocol for CVS-FTP API Interaction

This CVS functionality can be considered a client of Jakarta Commons Net with a certain protocol pattern. Our CVS class and Commons Net library can be instrumented to generate traces which were then inputted to SMArTIC and sk-strings. The resultant models are then compared with the original CVS specification to evaluate the feasibility of SMArTIC in improving the accuracy of the results over the sk-strings learner.

There are six common FTP interaction scenarios in our CVS implementation: Initialization, multiple-file upload, download, and deletion, multiple-directory creation and

```
Procedure AutomatonMerge
Inputs:
$X$ : First automaton to merge
$Y$ : Second automaton to merge
$TrainSetX$ : Set of traces to train X
$TrainSetY$ : Set of traces to train Y
Outputs:
$Merged$: Merged automaton with L($Merged$) = L($X$) U L($Y$)
Method:
1: Let $WghtX =$ |TrainSetX|/(|TrainSetX|+|TrainSetY|)
2: Let $WghtY =$ |TrainSetY|/(|TrainSetX|+|TrainSetY|)
Step 1: Handle Exceptional Cases
3: HandleExceptionalCase(X)
4: HandleExceptionalCase(Y)
Step 2: Create Unifiable and Mergable Node Pairs
5: Let $Uni$ = Compute_Unifiable_Node_Pairs(X,Y)
6: Let $Merge$ = Compute_Mergable_Node_Pairs(X,Y)
Step 3: Create Partially Merged Automata
7: Let $Top$ = Create_Top_Merged_Automata(Uni)
8: Let $Bot$ = Create_Bottom_Merged_Automata(Merge)
Step 4: Create Final Merged Automata
9: Let $Merged$ = Merge_Top_And_Bottom ($Top$, $Bot$, $X$, $Y$, $WghtX$, $WghtY$)
10: Output Automaton $Merged$
```

Figure 5.8: Automaton Merger (see text for details)

deletion. All scenarios begin by connecting and logging-in to the FTP server. They end by logging-off and disconnecting from the FTP server. The client side only maintains a record of files backed-up in the FTP server.

All these scenarios are depicted in the automata shown in Figure 5.9. The dashed boxes, from top to bottom, represent upload files, initialization, delete files, make directories, remove directories and download files scenario, respectively.

5.2.2 Instrumentation, Trace Collection and Processing

We instrument Jakarta Commons Net with JRat, the extension of the Java runtime analysis toolkit [91] described in Chapter 3. Running the instrumented code will produce a tree of method calls capturing their order and context represented as an XML document. Methods called earlier will print earlier in the XML document, and each method will have its context as its ancestors in the XML tree.

We construct a wrapper class that takes in the automata shown in Figure 5.9 and

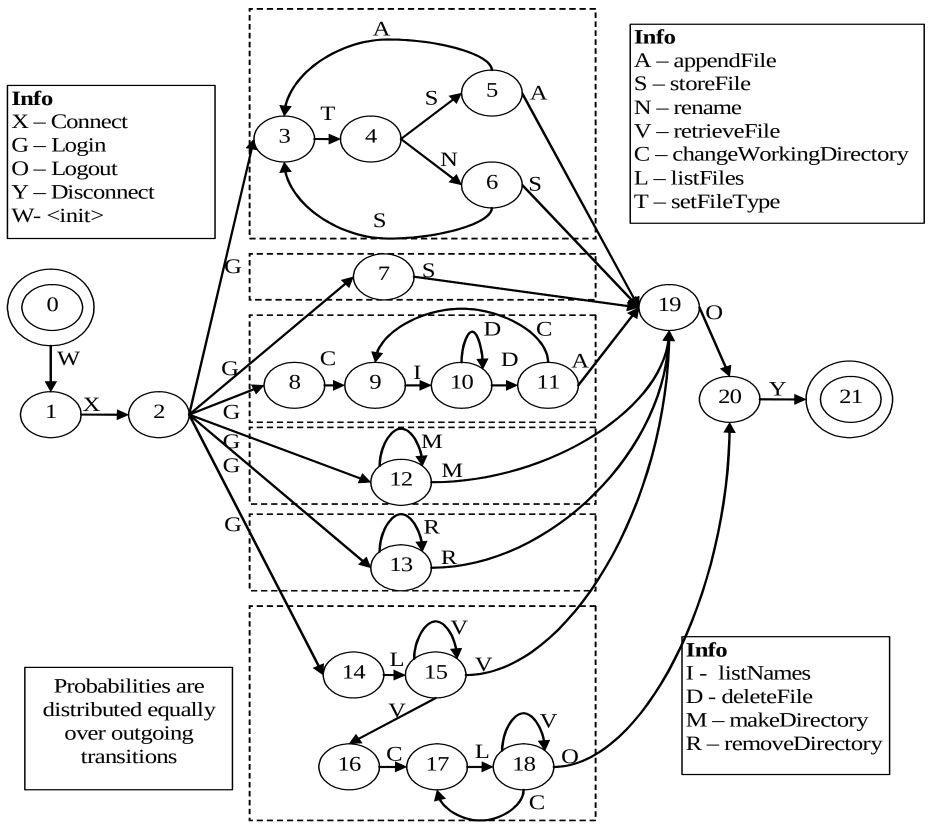


Figure 5.9: CVS Protocol

invokes *org.apache.commons.net.ftp. FTPClient* accordingly. The wrapper class will generate a list of sequences of method invocations by traversing the automata from the start to the end node multiple times until coverage criteria is met. The result is a simulation of a regression testing of CVS-FTP API interaction.

Each invocation of a method of *FTPClient* may generate exceptions, especially *FTPConnectionClosedException* and *IOException.* Hence the code accessing the *FTPClient* methods need to be enclosed in a *try..catch..finally* block. Every time such an exception happens we simply logout and disconnect from the FTP server. This is simulated by adding error transitions shown in Figure 5.10. Ten percent error is assumed and erroneous trace will be injected to 10% of the generated list of sequences of method invocations.

The trace file generated is likely to be huge because of the wrapper effect and long class hierarchies. On the other hand, what we really need are the traces capturing interaction between our own CVS classes and *FTPClient.* To get that, we process the trace file as follows : (1) Traverse the XML trace file depth-first, and locate all the first

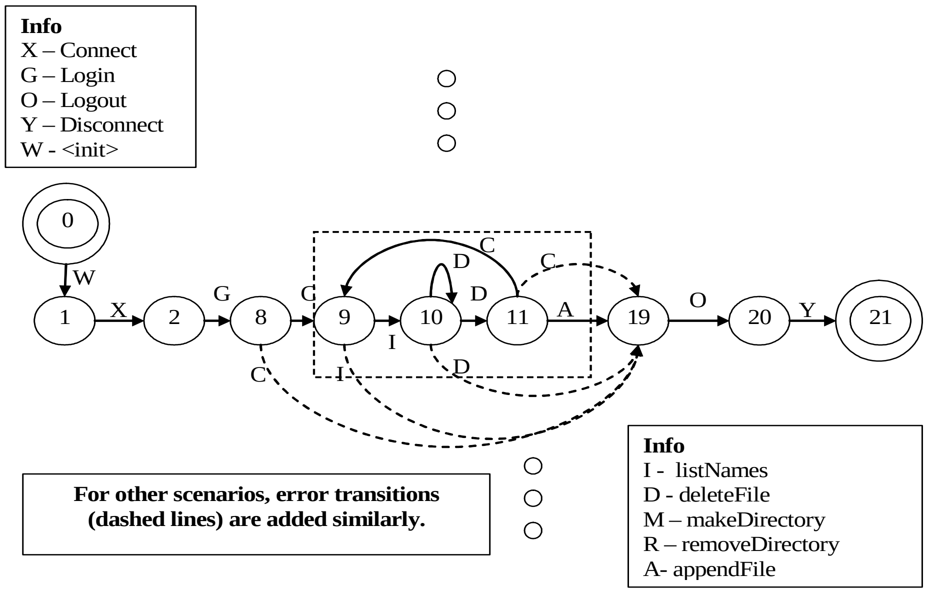


Figure 5.10: CVS Protocol With Error

invocations of the client method calls. Each such location will correspond to a scenario trace sequence. (2) From the above locations, traverse depth-first, and locate all the first invocations of the API method calls. The API method calls might not be directly below the client method calls in the trace file XML hierarchy.

5.2.3 Protocol Specification Generation and Results

The collected traces are inputted to different miners: SMArTIC (with sk-strings in the learner block), SMArTIC without filtering, SMArTIC without clustering, sk-strings with coring and standalone sk-strings. The coring threshold is set at 0.2 level. SMArTIC filtering confidence and support is set at 0.8 and 0.1, respectively. Default parameter settings are used for sk-strings both when standalone and within SMArTIC.

A protocol specification is then produced and compared against the original one (as shown in Figure 5.9) in terms of precision and recall. We repeat the above experiments 100 times using different lists of scenario trace sequences.

The following table shows the results of our experiment. The columns Precs, Recall and PS correspond to precision, recall and unbiased, normalized co-emission, respectively (as defined in Chapter 4).

	Precs	**Recall**	**PS**
SMArTIC	0.484	0.981	0.653
SMArTIC w/o filtering	0.426	1	0.616
SMArTIC w/o clustering	0.263	0.984	0.532
sk-strings(coring)	0.289	0.581	0.447
sk-strings	0.225	1.000	0.533

As shown in the table, SMArTIC improves the precision and co-emission while maintaining good recall of CVS protocol inference. The precision of SMArTIC are more than double the precision of sk-strings.

Both filtering and clustering help in increasing precision while maintaining good recall and equivalent or even better co-emission.

The drawback of coring is shown in the results where recall drops by almost half. Although precision is increased, there is a heavy penalty in recall: Pruning erroneous behaviors unavoidably removes a significant proportion of correct behaviors.

It is also of interest to know the number of erroneous traces our filtering algorithm filters out. On the average it filters out 43% of erroneous traces while only 4% of valid ones.

We have conducted thorough experiments using this application to verify both our hypotheses. The results show that significant improvement (with at least 95% confidence level) in SMArTIC over the Sk-strings.

5.3 Further Experiments

The experiment with CVS specification in Section 5.2 provides positive evidence that SMArTIC is a feasible architecture for improving mining accuracy; it also provides strong evidence to support our hypotheses stated in Section 5.1.

In this section, we perform further experiments on SMArTIC, not just on its accuracy, but also on its robustness and scalability. To this end, we have conducted almost 2000 experiments to support the superiority of SMArTIC.

Our experiments use the same set of miners, with sk-strings learner being employed either in standalone mode or in co-operation with other processes, especially as the learner block of SMArTIC.

5.3.1 Material

In the first set of experiments, two sets of sub-experiments using different types of error injection were performed to evaluate the two learners' performance in terms of robustness. These experiments are performed on sk-strings (standalone), sk-strings (with coring) and SMArTIC. For SMArTIC case, we disable the clustering sub-system to measure the effect of the filtering block.

We simulated the automaton generated by Ammons *et al.* in their analysis of the X11 windowing library (*cf.* [7]) – as shown in Figure 5.11. However, we modified the model slightly so that it was *without any non-determinism and repeated use of alphabet assigned to transitions* and we added probabilities. Probabilities are *distributed equally* to transitions from the same source node (not shown in the diagram). This is meant to produce a base model that can be learned perfectly. Error nodes and transitions were then injected to the automaton to conduct robustness tests. The models with different injections of errors (nodes and transitions labelled as Z and shown with dashed lines) are shown in Figure 5.12(a) & (b). Each of the two types of injections of errors shown in Figures 5.12(a) and (b) respectively corresponds to a separate set of sub-experiments.

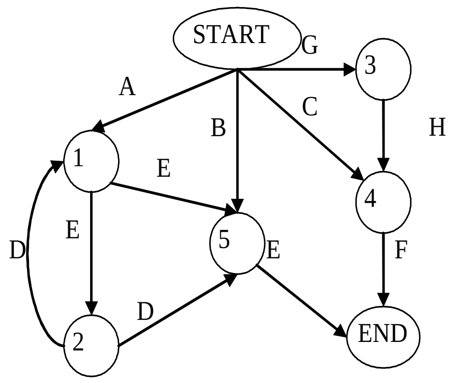


Figure 5.11: X11 Windowing Library Model

We expect the specification miners to be able to filter out errors. We compared the inferred automaton with the simulator model shown in Figures 5.12(a) and (b) without error nodes and transitions and recorded the similarity and difference metrics. We generated traces using the trace generation algorithm described in Chapter 4 and capped the maximum number of traces generated to 10,000. Four, eight and ten percent levels of error were injected to the system (*i.e.*, 4, 8 and 10 percent of generated traces,

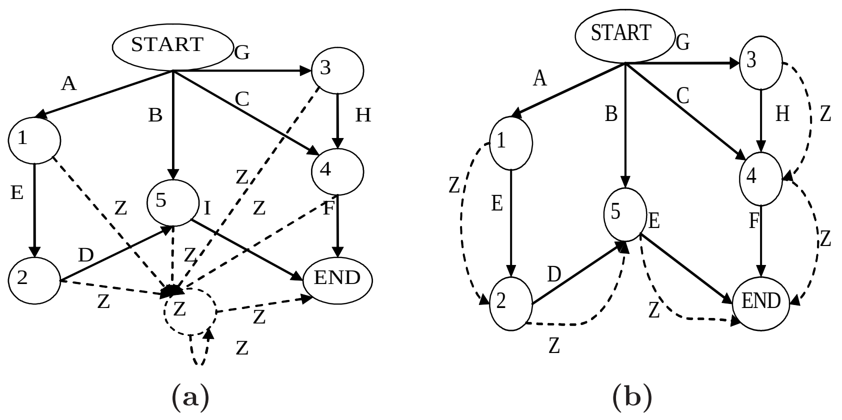


Figure 5.12: Models of Specification with Error

respectively, will be erroneous). We assume the error level is unknown to the learner except that it is low. Hence, the threshold used for coring was set to 0.2. SMArTIC's filter confidence is also set at an equivalent level of 0.8 while its support is set at 0.1. In each case, we ran one hundred experiments and recorded the average performance.

In the second experiment, we evaluated the scalability of the learners by generating distinct models of various sizes. Two sets of sub-experiments were conducted, each with a different independent variable. In the first set, we varied the number of nodes (by 10, 20, 30 and 40) in the specification model and maintained the number of outgoing transitions per node to at most four. In the second set, we varied the number of outgoing transitions per node (by 3, 5, 7 and 9) and maintained the number of nodes at 10. For each case, we performed 10 experiments and recorded their average performance.

We automatically generated distinct models having n nodes and a maximum of m transitions per state with common start and end nodes. Transition labels were chosen from a pool of a fixed number of labels randomly. Loops were introduced based on the principle of locality where loops between child and parent/ancestor nodes (including self-loop) occur with higher probability than those connecting to distant sibling nodes. The above properties are meant to generate *reasonably* complex models that are more likely to mimic reasonable protocols even in a large system (*e.g.*, business logic of an enterprise application).

These experiments were performed on sk-strings (standalone) and SMArTIC. In the SMArTIC case, we measured the effect of the clustering block by disabling the filtering sub-system.

We generated traces and capped the maximum number of traces at 10,000. No error was injected to the system. Since we imposed a cap of 10,000 traces, there might be a concern that training traces might not satisfy the coverage criterion for a model of large size. This was not the case in our experiments, as only once was the cap reached; for the other 159 experiments, the coverage criterion was met first.

5.3.2 Experiment 1 Findings

Here, we evaluated the robustness of sk-strings, sk-strings (coring) and SMArTIC with two different injections of errors. The models with different injection of errors are shown in Figures 5.12(a) and (b).

Results. The experiment results for ErrModel1 and ErrModel2 are captured in Table 5.1. Column E% corresponds to the percentage of erroneous traces. Columns 'Precs', 'Recall' and 'PS' correspond to precision, recall and unbiased, normalized co-emission, respectively (as defined in Chapter 4) .

Error Model 1							
sk-strings				**sk-strings(coring)**			
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.946	1.000	0.946	4	0.999	0.823	0.864
8	0.908	1.000	0.948	8	0.998	0.828	0.867
10	0.883	1.000	0.950	10	0.990	0.845	0.875
SMArTIC							
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.999	1.000	0.946	10	0.981	1.000	0.948
8	0.993	1.000	0.946				
Error Model 2							
sk-strings				**sk-strings(coring)**			
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.947	1.000	0.947	4	0.829	0.962	0.863
8	0.898	1.000	0.947	8	0.812	0.909	0.848
10	0.875	1.000	0.948	10	0.816	0.886	0.849
SMArTIC							
E%	Precs	Recall	PS	E%	Precs	Recall	PS
4	0.994	1.000	0.946	10	0.974	1.000	0.949
8	0.986	1.000	0.946				

Table 5.1: Robustness Experiment Results

Analysis. For sk-strings, all traces generated by the simulator model X were accepted by the inferred model Y. On the other hand, we noted a drop in the acceptance of traces generated by Y. This drop is slightly larger to the noise injected ((5.4%,5.3%) vs. 4%, (9.2%,10.2%) vs. 8% and (11.7%,12.5%) vs. 10%); learner precision degrades in the presence of erroneous traces. We conclude that the sk-strings learner is not robust.

The SMArTIC result is similar to sk-strings in that all traces generated by the simulator model X were accepted by the inferred model Y. Different from sk-strings, we noted only a slight drop in the acceptance of traces generated by Y. This drop is far less than the noise injected ((0.1%,0.6%) *vs.* 4%, (0.7%,1.4%) *vs.* 8% and (1.9%,2.6%) *vs.* 10%). These indicates that filtering of erroneous traces is effective in preventing loss of precision.

The most important observation here is that: Having coring as post-processing to sk-strings removes not just erroneous transitions but also quite a fair number of correct transitions. Consequently, the accuracy of the mined specification degraded. This result strongly supports our first hypothesis.

Another limitation of coring is due to the fact that transition labels are being ignored during the coring operation. The coring method only searches for the pair of nodes (i, j) where there is a low "heat transmission" from node i to node j [7]; it ignores the detail of how the node j is reached (which can be a single transition, a set of transitions, a single path or a set of paths). In the second sub-experiment, erroneous transitions *go to valid nodes* instead of the special error node. This results in *little/no filtering* when coring is used.

5.3.3 Experiment 2 Findings

We performed two sets of scalability sub-experiments. In the first set of sub-experiments, we generated distinct models by varying the number of nodes while keeping the maximum transitions per node at 4. In the second set of sub-experiments, we varied the maximum number of transitions while keeping the total number of nodes constant at 10.

Results. The experiment results for sk-strings and SMArTICare shown in Table 5.2. Columns X.N and X.TN correspond to the number of nodes and maximum number of transitions per node in the specification model.

Varying Number of Nodes						
	sk-strings			**SMArTIC**		
X.N/X.TN	Precs	Recall	PS	Precs	Recall	PS
10/4	0.437	1.000	0.560	0.584	0.988	0.690
20/4	0.003	1.000	0.015	0.185	0.998	0.227
30/4	0.0004	1.000	0.005	0.059	0.999	0.090
40/4	0.0004	1.000	0.004	0.014	1.000	0.007
Varying Maximum Number of Transitions						
	sk-strings			**SMArTIC**		
X.N/X.TN	Precs	Recall	PS	Precs	Recall	PS
10/3	0.113	1.000	0.218	0.453	0.984	0.504
10/5	0.187	1.000	0.284	0.578	0.993	0.424
10/7	0.084	1.000	0.213	0.500	0.984	0.508
10/9	0.073	0.997	0.087	0.514	0.990	0.329

Table 5.2: Scalability Experiment Results

Analysis The above results shows that sk-strings and SMArTIC were affected when we scaled up the model size. Comparing the two set of experiments, we observe that the precision is adversely affected in all cases when we increase the number of nodes, whereas the impact is less severe when we increase the number of transitions.

SMArTIC is generally better in terms of precision up to a factor of over 147.5 (*i.e.*, 30-node case). In the second set of experiments (*i.e.*, when we increase the maximum number of transitions), SMArTIC maintains its precision while sk-strings loses it as the maximum number of transitions increased.

5.4 Conclusion

In this chapter, we began with two hypotheses about how specification miners should be organized to alleviate the impact of erroneous transitions and to localize and minimize over-generalization. We then presented our novel Specification Mining Architecture with Trace fIltering and Clustering, (SMArTIC) to support our hypotheses. SMArTIC comprises four major blocks – clustering, filtering, learning and merging. Filtering and clustering is meant to address the issue of robustness and scalability, respectively.

Traces violating common trace population rules are removed. The resultant filtered traces are then separated into multiple clusters. By clustering common traces together, it is expected that the learner is able to learn better and over-generalization of a subset of traces is not propagated to other clusters. These clusters of filtered traces are then inputted to a specification miner. The sk-strings learner is used for learning, and each cluster is considered an independent (sub-)protocol. Lastly, a merger sub-system produces a merged automaton without sacrificing accuracy.

Along with the architecture, we have also proposed a novel trace clustering technique based on grammatical similarity, a novel outlier detection rule mining technique and a novel automaton merging method. Besides having automaton as specification, the outlier-detection rules produced by the filtering block can also be viewed as sets of simple specifications based on strong properties of significant trace groups useful for filtering. They can effectively capture those property patterns proposed in [41], which are interesting for program traces and useful for identifying potential bugs.

We experimented with the Jakarta Commons Net open-source library. Our experiments aim at deriving API interaction protocols for client applications of Jakarta Commons Net open-source library [10]. From one hundred experiments performed, the following are noted: (1) SMArTIC improves precision (more than double) and co-emission while maintaining good recall, (2) both clustering and filtering help in improving precision while maintaining good recall and equivalent co-emission, (3) coring removes erroneous behavior together with a significant proportion of valid behavior – recall is reduced by more than 40%, (4) outlier detection rules are able to filter on average 43% of erroneous traces while only wrongly filter 4% of valid ones.

Further experiments using simulation measuring precision and recall in the two dimensions of increasing percentage of error (*i.e.*, robustness) and increasing model size (*i.e.*, scalability) of sk-strings and SMArTIC were performed. A total of 1,800 tests on three percentage of error levels and 160 tests on different configurations of the number of nodes and the maximum number of transitions of the specification model were performed.

From the robustness experiments, the precision of sk-strings is reduced proportionally to the error induced. On the other hand, only a slight reduction of precision is observed for SMArTIC. Our experiments also show the limitation of the coring method in removing errors. From the scalability experiments, both sk-strings and SMArTIC are adversely affected by the increase in model scale (number of nodes). However, SMArTIC is able to retain better precision as compared to sk-strings up to factor of over 100.

Our experiments have strongly supported our belief that SMArTIC can produce *more precise results with good recall and equivalent, or even better, co-emission* in the presence of errors and increasingly large model.

CHAPTER VI

MINING FREQUENT SOFTWARE BEHAVIORAL PATTERNS

In this chapter, we propose mining frequent patterns of software behavior. Different from the previous chapter, instead of mining a model capturing the entire behavior of a system, we would like to extract common behaviors that appear often in execution traces. These patterns are intuitive and commonly found in software documentations, such as:

1. Resource Locking Protocol: $\langle lock, unlock \rangle$
2. Telecommunication Protocol (*c.f.*, [81]): ⟨*off_hook*, *dial_tone_on*, *dial_tone_off*, *seizure_int*, *ring_tone*, *answer*, *connection_on*⟩
3. Java Authentication and Authorization Service (JAAS) Authorization Enforcer Strategy Pattern (*c.f.*, [161]): $\langle Subject.getPrincipal$, $PrivilegedAction.create$, $Subject.doAsPrivileged$, $JAAS\_Module.invoke$, $Policy.getPermission$, $Subject.getPublicCredential$, $PrivilegedAction.run \rangle$
4. Java Transaction Architecture (JTA) Protocol (*c.f.*, [163]): $\langle TxManager.begin$, $TxManager.commit \rangle$, $\langle TxManager.begin$, $TxManager.rollback \rangle$, *etc.*

Each of these patterns reflecting interesting program behavior can be mined by analyzing a set of program traces – each being a sequence of method invocations. A pattern (*e.g.*, lock-unlock) can *appear a repeated number of times within a sequence.* Events in a pattern can be *separated by an arbitrary number of unrelated events* (*e.g.*, *lock* $\rightarrow$ resource use $\rightarrow \ldots \rightarrow$ *unlock*). Since a program behavior can be manifested in numerous ways, *analyzing multiple traces/sequences are often necessary.*

To mine software temporal patterns having the above characteristics from traces, we introduce iterative pattern mining. It mines series of events supported by a *significant number of instances repeated within and across sequences.* To mine such patterns, we propose a novel algorithm that leverages the techniques found in sequential pattern mining [5] and episode mining [128]. Unless otherwise stated, references to 'patterns' in this chapter refer to iterative patterns.

Sequential pattern mining first addressed by Agrawal and Srikant in [5] discovers temporal patterns that are supported by a *significant number of sequences.* A pattern is supported by a sequence if it is a sub-sequence of the later. On the other hand, Mannila *et al.* perform episode mining to discover *frequent episodes within a sequence of events* [128]. An episode is defined as a series of events occurring *relatively close* to one another (*e.g.*, they occur at the same window). An episode is supported by a window if it is a sub-sequence of the series of events appearing in the window. Episode mining focuses on mining from a single sequence of events.

Iterative pattern mining can be thought as a merge between sequential pattern mining and episode mining. Similar to sequential pattern mining, we consider a *database of sequences* rather than a single sequence. While sequential pattern mining ignores *repeated* occurrences of a pattern in a sequence, iterative pattern mining considers these repetitions.

These repetitions of patterns within a sequence are considered by work in episode mining. However, there are two notable differences between iterative pattern mining and episode mining.

First, program properties are often inferred from a set of traces instead of a single trace. Secondly, important patterns for verification, such as lock acquire and release or stream open and close (*c.f* [176, 28]), often have their events occur at some arbitrary distance away from each other in a program trace. Hence, there is a need to 'break' the 'window barrier' (as employed by episode mining) in order to capture these patterns of interest. Interestingly, these two notable differences are in turn observed by sequential pattern mining.

To support iterative pattern mining, we need a clear definition and semantics of iterative pattern different from episodes and sequential patterns. Our definition of iterative pattern is inspired by the common languages for specifying software behavioral requirements, namely Message Sequence Chart (MSC) [81] and Live Sequence Chart (LSC) [37].

MSC and LSC are variants of sequence diagram specifying how a system should behave. A sequence diagram is composed of a set of lifelines (*i.e.*, the vertical lines representing classes) and messages between these lines. An example of an MSC is a simplified telephone switching protocol shown in Figure 6 adapted from an example in [81].

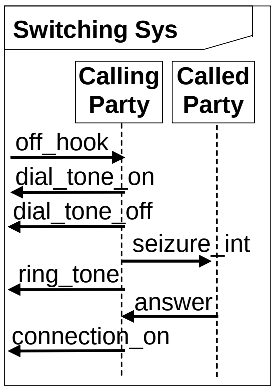


Figure 6.1: Example Message Sequence Chart

Abstracting caller and callee information, it can be represented as a pattern: ⟨*off_hook, dial_tone_on, dial_tone_off, seizure_int, ring_tone, answer, connection_on*⟩.

The full language of MSC/LSC is complicated and it is not our intention to mine the full notion of MSC/LSC. In this study, we ignore the partial order case of MSC/LSC. Iterative pattern mined abstracts away the caller and callee information but ensures total-ordering property and one-to-one correspondence between a pattern and its instance (*i.e.*, a segment of a trace) following the semantics described in [99]. (Please refer to Section 3.2 for details.)

Pattern mining in general is an NP-hard problem. For it to be practical, efficient search space pruning strategies need to be employed. One of the most important properties to help in ensuring scalability is the apriori or monotonicity property. There are several variants of it. Iterative pattern obeys the following apriori property utilized by depth-first search sequential pattern miners (*e.g.*, FreeSpan [62] and PrefixSpan [143]) which states:

If P is not frequent, then P++evs (where evs is a series of events) is also not *frequent.*

Apriori property holds for both sequential patterns and episodes. To ensure efficiency, it is desirable to maintain this property for iterative patterns. Fortunately, the formulation of iterative pattern guarantees this property as described in Section 6.1.

Furthermore, due to possibly combinatorial number of frequent subsequences of a long pattern, it's best to mine a closed set of patterns (*c.f.*, [174] & [167]). As discussed in the later part of sub-section 5.1.1 closed pattern mining discovers patterns without any super-sequence having a corresponding set of instances. The resultant pattern set is likely to be *more compact and yet still complete* (*i.e.*, every frequent pattern is represented by a closed pattern). Closed pattern mining can also lead to more efficient pattern mining strategy. *Early identification and pruning* of non-closed patterns can reduce the runtime significantly.

In this chapter, we mine a *closed* set of iterative patterns. A search space pruning strategy employed by *early identification and pruning* of non-closed patterns is used to mine a closed set of iterative patterns efficiently. Our performance study on synthetic and real-world datasets shows the major success of our pruning strategy: it runs with over an order of magnitude speedup especially on low support thresholds or when the frequent patterns are long.

As a case study, we experimented with traces collected from transaction sub-component of JBoss Application Server. Our mined patterns highlight important program behavioral patterns.

The contributions of this work are as follows:

1. We present a novel formulation of iterative pattern inspired by standards adopted for specifying software behavioral requirements (*i.e.*, MSC and LSC).
2. We propose an *efficient* algorithm to mine a *closed* set of software iterative patterns from program execution traces.
3. We extend episode mining by: (1) analyzing multiple sequences, (2) removing the 'window' barrier, and (3) extracting a closed set of patterns for software specification mining purpose.
4. We extend closed sequential pattern mining by considering repeated pattern occurrences within a sequence and across multiple sequences for software specification mining purpose.

The outline of this chapter is as follows: Section 6.1 provides an in-depth discussion on semantics of iterative pattern. Section 6.2 presents the principles behind the generation of *closed iterative patterns* and its associated pruning strategy. Section 6.3 describes our closed pattern mining algorithm. Section 6.4 presents the results of our performance study. Section 6.5 discusses a case study on mining program behavioral patterns from traces of JBoss Application Server. We conclude in Section 6.6.

6.1 Iterative Patterns

In this section, we define formally the iterative pattern, and provide the reasoning behind its semantics.

6.1.1 Basic Definitions

We refer to some definitions defined in Chapter 3. In addition to those definitions we defined an additional important operation termed as the erasure operation, as defined in Definition 6.1.

Definition 6.1 (Erasure Operator) *Given a pattern P ($\langle p_1, p_2, \ldots, p_n\rangle$) and a string S ($\langle s_1, s_2, \ldots, s_m\rangle$), the erasure of S wrt. P, denoted by $erasure(S, P)$, is defined as a new string S_{erased} formed from S where all events occurring in P are removed from S. Formally, S_{erased} is defined as ($\langle se_1, se_2, \ldots, se_k\rangle$) such that (1) $\forall i.se_i \notin P$ and (2) there exists a set of integers $\{i_1 \ldots i_k\}$ with $1 \leq i_1 < i_2 < i_3 < i_4 \ldots < i_k \leq m$ and $se_1 = s_{i_1}$, $se_2 = s_{i_2}$, $\cdots$, $se_k = s_{i_k}$ and $\forall j \notin \{i_1 \ldots i_k\}, s_j \in P$.*

As an example consider the pattern P ($\langle B, B\rangle$), and the string S ($\langle A, B, C, C, D\rangle$) the operation $erasure(S, P)$ will return the string $\langle A, C, C, D\rangle$.

6.1.2 Semantics of Iterative Patterns

Our definition of iterative pattern is inspired by the common languages for specifying software behavioral requirement: Message Sequence Chart (MSC) (a standard of International Telecommunication Union (ITU) [81]) and its extension, Live Sequence Chart (LSC) [37].

MSC and LSC are variants of the well known UML sequence diagram describing behavioral requirement of software. Not only do they specify system interaction through ordering of method invocation, but they also specify caller and callee information. An example of such charts is a simplified telephone switching protocol shown in Figure 6.

In verifying traces for conformance to an event sequence specified in MSC/LSC, the sub-trace manifesting the event sequence must satisfy the total-ordering[1] property: Given an event ev_i in an MSC/LSC, the occurrence of ev_i in the sub-trace occurs before the occurrence of every ev_j where $j > i$ and after ev_k where $k < i$ [81]. Kugler *et al.* strengthened the above requirement to include a one-to-one correspondence between events in a pattern and events in any sub-trace satisfying it [99]. Basically, this requirement ensures that, if an event appears in the pattern, then it appears as many times in the pattern as it appears in the sub-trace.

For the telephone switching example, the following traces are not in conformance to the protocol:

off_hook, *seizure_int*, *ring_tone*, *answer*,*ring_tone*, *connection_on*
off_hook, *seizure_int*, *ring_tone*, *answer*, *answer*, *answer*, *connection_on*

The first trace above doesn't satisfy the total-ordering requirement due to the out-of-order second occurrence of *ring-tone* event. The second doesn't satisfy the one-to-one correspondence requirement due to multiple occurrences of *answer* event.

Iterative pattern abstracts away the caller and callee information but retains the total ordering and one-to-one correspondence requirements of MSC/LSC.

[1]We ignore the partial order case.

The pattern instance definition capturing the total-ordering and one-to-one correspondence between events in the pattern and any of its instances (to be defined below) can be expressed unambiguously in the form of Quantified Regular Expression (QRE) [142]. Quantified regular expression is very similar to standard regular expression with ';' as concatenation operator, '[-]' as exclusion operator (*i.e.*, [-P,S] means any event except P and S) and * as the standard kleene-star.

Definition 6.2 (Iterative Pattern Instance - QRE) *Given a pattern P* $(p_1p_2\dots p_n)$*, a substring SB* $(sb_1sb_2\dots sb_m)$ *of a sequence S in SeqDB is an instance of P iff it is of the following QRE expression*

$$p_1;[-p_1,\dots,p_n]*;p_2;\dots;[-p_1,\dots,p_n]*;p_n.$$

Operationally, we use an equivalent definition of pattern instance described using the erasure operation:

Definition 6.3 (Iterative Pattern Instance) *Given a pattern P* $(p_1p_2\dots p_n)$*, a substring SB* $(sb_1sb_2\dots sb_m)$ *of a sequence S in SeqDB is an iterative pattern instance of P iff (1)* $first(P) = first(SB)$*, (2)* $last(P) = last(SB)$ *and (3) the following erasure constraint holds:*

$$erasure(SB, erasure(SB, P)) = P.$$

We use the term "pattern instance" and "iterative pattern instance" interchangeably in this chapter. The operation $erasure(SB, erasure(SB, P))$ basically removes all events that occur in SB but not in P. An iterative pattern is thus identified by a set of iterative pattern instances, which can occur repeatedly in a sequence as well as across sequences. We also use the term "pattern" and "iterative pattern" interchangeably.

An instance is denoted compactly by a triple $(s_{idx}, i_{start}, i_{end})$ where s_{idx} refers to the sequence index of a sequence S in the database while i_{start} and i_{end} refer to the starting point and ending point of a substring in S. By default, all indices start from 1. With the compact notation, an instance is both a string and a triple – the representations are used interchangeably. The set of all instances of a pattern P in a database DB is denoted as $Inst(P, DB)$. Reference to the database is omitted if it refers to the input sequence database.

As an example, consider a pattern P ($\langle A, B\rangle$) and a database consisting of two sequences:

Identifier	Sequence
$S1$	$\langle D, B, A, B, A, B, C, E\rangle$
$S2$	$\langle D, B, A, B, B, B, A, B\rangle$

The set $Inst(P)$ is the set of triples {(1,3,4),(1,5,6),(2,3,4), (2,7,8)}.

There is a one-to-one ordered correspondence between events in the pattern and events in its instance. This one-to-one correspondence can be captured by the concept of pattern instance landmarks defined below.

Definition 6.4 (Pattern Instance Landmarks) *Given a pattern P ($p_1p_2 \ldots p_n$), an instance I ($s_1s_2 \ldots s_m$) of pattern P has the following landmarks: $l_1, l_2, \ldots l_n$ where $1 \leq l_1 < l_2 < \ldots < l_n \leq m$ and $s_{l_1} = p_1, s_{l_2} = p_2, \ldots, s_{l_n} = p_n$. Due to erasure constraint, for each instance there is only one such set of landmarks. The landmarks of an instance I is denoted as $Lnd(I)$. The ith member of the set $Lnd(I)$ is called the ith landmark.*

The *support* of a pattern *wrt.* to a sequence database *SeqDB* is the number of its instances in *SeqDB*. A pattern P is considered *frequent* when its support, *sup*(P), exceeds a certain threshold (*min_sup*). For example, in the example database shown previously in this sub-section, the support of pattern P ($\langle A, B\rangle$) is equal to three since there are three instances of P in the database. If the *min_sup* threshold is more than or equal to three, P is frequent otherwise it is not frequent.

6.1.3 Apriori Property and Closed Pattern

Iterative patterns possess the following *'apriori' property* similar to the one used in PrefixSpan [143]:

Theorem 6.1 (Apriori Property - PrefixSpan) *If P is not frequent, then both patterns P++evs and evs++P (where evs is an arbitrary series of events) are also not frequent.*

Proof 6.1 *Part 1: If P is not frequent, then P++evs is not either.*

The above statement is the contrapositive of the following statement: if P++evs is frequent so does P. Let P is of length n. Due to the erasure constraint, there is a unique set of landmarks for every instance of P++evs. For each instance of P++evs, the subsequence marked from the first landmark to the n^{th} landmark is an instance of P. Hence, $sup(P) \geq sup(P$++$evs)$.

Part 2: If P is not frequent, then evs++P is not either.

The proof is similar to part 1.

In general, iterative patterns do not possess the apriori property used in GSP [5]: if a pattern is frequent so is its sub-sequences. However, considering patterns having corresponding instances as described in Definition 6.5 below, the GSP apriori property holds as stated in Theorem 6.2.

Definition 6.5 (Corresponding Pattern Instances) *Consider a pattern P and its super-sequence Q. We say that an instance I_P (seq_P, $start_P$, end_P) of P corresponds to an instance I_Q (seq_Q, $start_Q$, end_Q) of Q (and vice versa) iff $seq_P = seq_Q$ and $start_P \geq start_Q$ and $end_P \leq end_Q$.*

As examples of corresponding and non-corresponding instances of patterns consider the following database.

Identifier	Sequence
$S1$	$\langle A, B, B, A, C, D\rangle$
$S2$	$\langle A, B, A\rangle$

Consider two patterns P ($\langle A, B, A\rangle$) and its super-sequence Q ($\langle A, B, B, A\rangle$). The only instance of the pattern P is (2,1,3), while the only instance of pattern Q is (1,1,4). However, since their instances match *different segments* of the database, the instance of P doesn't correspond to the instance of Q. We say that P and Q have *non-corresponding instances.* On the other hand, the instance of $\langle C\rangle$ corresponds to the instance of $\langle C, D\rangle$.

Theorem 6.2 (Apriori Property - GSP-Like) *If a pattern Q is frequent and P is a sub-sequence of Q, then either P is frequent or every instance of Q does not correspond to any instance of P (and vice versa).*

Proof 6.2 *We would like to show for a pattern Q and its sub-sequence P, sup(P) $\geq$ sup(Q). Otherwise P and Q have non-corresponding instances.*

From Theorem 6.1, for a pattern P and its super-sequence Q, sup(P) $\geq$ sup (Q), if Q = P++evs or evs++P. In other words, Q is formed by adding extra prefixes or suffixes to P.

Hence to p, we only need to show that sup(P) is $\geq$ sup(Q) if Q is formed by adding extra infix events to P. Let the set of infixes be IFX. There are two cases:

Case 1: $\forall ev \in IFX.\ ev \notin P$

Consider an arbitrary instance I_Q of Q, erasure(I_Q,erasure(I_Q,Q)) = Q. However, since $\forall ev \in IFX.\ ev \notin P$, erasure($I_Q$,erasure($I_Q$, P)) = P. Hence every instance of Q is an instance of P as well. The support of P is greater or equal to the support of Q.

Case 2: $\exists ev \in IFX.\ ev \in P$

Consider an arbitrary instance I_Q of Q, erasure(I_Q,erasure (I_Q,Q)) = Q. However, since $\exists ev \in IFX.\ ev \in P$, erasure($I_Q$,erasure($I_Q$, P)) $\neq$ P. Every instance of Q is not an instance of P. Similarly, every instance of P is not an instance of Q.

For all cases we have shown that either apriori property holds for the two patterns P and its super-sequence Q or they have non-corresponding instances.

Definition 6.6 (Closed Pattern) *A frequent pattern P is said to be* closed *if there exists no super-sequence Q s.t.:*

1. *P and Q have the same support*
2. *Every instance of P corresponds to a unique instance of Q.*

Notation-wise, we denote the full set of closed iterative patterns mined from *SeqDB* by *Closed*. In this work, we address the following task: *Given a sequence database, find a closed set of iterative patterns.*

6.2 Generation of Iterative Patterns

Iterative pattern can be mined using a depth-first pattern growth-and-prune strategy (*c.f.*, FreeSpan [62] and PrefixSpan [143]). However, rather than using the usual database pseudo-projection operation that extracts sequential patterns, we perform a different type of pseudo-projection outlined below. Unless otherwise stated, in this chapter, references to projected database and projected database operation corresponds to projected-iter database and projected-iter operation defined in Definition 6.7.

Definition 6.7 (Projected-iter) *A database SeqDB* projected-iter *on a pattern P results in a* projected-iter *database $SeqDB_P^{itr}$ which corresponds to a set of triples. It is defined recursively as follows:*

Base Case: if P is a single event ev, $\{(sid, ev, sx) \mid \exists sid.\ s = SeqDB[sid] \text{ and } ev{+\!+}sx \text{ is a suffix of } s\}$
Inductive Case: if P is a multi-event pattern, $\{(sid, ox{+\!+}px{+\!+}last(P), sx) \mid$ $\exists(sid, ox, (px{+\!+}last(P){+\!+}sx)) \in SeqDB_{P--last(P)}^{itr}.$ $((last(P) \notin erasure(ox, P--last(P))) \wedge$ $(\forall ev \in P, ev \notin px))\}$

From the definition of projected-iter database, the projected-iter database of a pattern p captures all instances of p that occur, possibly repeatedly within a sequence and/or across multiple sequences. The first element of the pairings corresponds to pattern instances in string format. The second element corresponds to the remaining part of the sequences providing the contexts from which the pattern can be further extended. Support of a pattern P is equal to the number of instances supporting P – let us denote this as $|Inst(P, SeqDB)|$. In turn, $|Inst(P, SeqDB)|$ is equal to the size of the projected database $|SeqDB_P^{itr}|$.

Instances of a length-1 pattern $\langle e_1\rangle$ are simply the occurrences of event e_1 throughout the sequences in $SeqDB$. Instances of a length-k pattern $\langle e_1, \ldots, e_k\rangle$ can be found from instances of the length-(k-1) pattern $\langle e_1, \ldots, e_{k-1}\rangle$.

Instances of a length-2 pattern $\langle e_1, e_2\rangle$ can be formed by extending instance triples of $\langle e_1\rangle$, (sid, e_1, ss) in $SeqDB_{\langle e_1\rangle}^{itr}$, on the condition: $\exists i.ss[i] = e_2 \wedge \forall j < i, ss[j] \notin \{e_1, e_2\}$. This condition corresponds to the second conjunctive clause of the inductive case of Definition 6.7. The first conjunctive clause in the definition is trivially satisfied since the erasure of a length-1 pattern instance is an empty string.

Similarly, instances of a length-3 pattern $\langle e_1, e_2, e_3\rangle$ can be formed by extending instance triples of $\langle e_1, e_2\rangle$, (sid, ox, ss) in $SeqDB_{\langle e_1,e_2\rangle}^{itr}$, on the conditions:(1) $e_3 \notin erasure$ $(ox, \langle e_1, e_2\rangle)$ and (2) $\exists i.ss[i] = e_3 \wedge \forall j < i, ss[j] \notin \{e_1, e_2, e_3\}$. The first and second conditions correspond respectively to the two conjunctive clauses of the inductive case. The first condition is necessary, since a substring instance ox of a length-2 pattern $\langle e_1, e_2 \rangle$ only obeys the erasure constraint of the length-2 pattern – ox might contain e_3.

Generalizing from the above, instances of a length-k pattern can be formed from instances of a length-(k-1) pattern, by following the inductive case of Definition 6.7.

A simple depth-first algorithm to generate a full-set of iterative patterns is as follows. First, generate a set of length-1 patterns where the support of each is greater than the *min_sup* threshold. A projected-iter database can then be created from the set of frequent length-1 patterns according to the base case of Definition 6.7. Instances of a length-2 pattern can then be obtained by performing the inductive step of Definition 6.7 to the corresponding length-1 pattern's projected database. Patterns not satisfying *min_sup* will be pruned. Since patterns obey *apriori* property, we can stop extending pruned patterns. Thus, all length-(i+1) patterns can be obtained from length-i patterns accordingly.

For the ease of explanation of the algorithm in Section 6.3, in Definition 6.8, we define *projected-first* operator and another related operator *Seq*. *Projected-first* operator corresponds to the inductive step of Definition 6.7.

Definition 6.8 (Projected-first & Seq) *A projected database $SeqDB_P^{itr}$ can be* projected-first *on an event e resulting in a set of triples and denoted as –* $(SeqDB_P^{itr})_e^{fst}$. *The set is defined as follows:*

$$\{(sid, ox{+\!+}px{+\!+}e, sx) \mid \exists(sid, ox, (px{+\!+}e{+\!+}sx)) \in SeqDB_P^{itr}.$$
$$(e \notin erasure(ox, P)) \wedge (\forall ev \in (P{+\!+}e), ev \notin px))\}$$

We denote the size of $(SeqDB_P^{itr})_e^{fst}$ as $Seq(e, SeqDB_P^{itr})$.

Projected-first operator locates the *first* instance of an event e in the sequence ss for each (sid,ox,ss) in the projected database – hence the name *projected-first*. The corresponding operator *Seq* computes the *number of sequences* in the projected database supporting the event e. Note that constraints corresponding to the inductive step of Definition 6.7 are also checked to ensure $(SeqDB_P^{itr})_e^{fst} = SeqDB_{P{+\!+}e}^{itr}$.

We also define the following two operations: one defines the equivalence of two projected databases, and the other defines the inclusion of an event in a projected database.

Definition 6.9 (Operations on Projected DB) *Projected databases DB_1 and DB_2 are equivalent (denoted as $DB_1 = DB_2$) iff $|DB_1| = |DB_2|$ and $\forall$ (sid,p_1, ss_1)$\in DB_1$. $\exists$ (sid,p_2, ss_2) $\in DB_2$. $ss_1 = ss_2$. Also, an event e is in a projected database DB (denoted as $e \in DB$) iff $\exists(sid, p, ss) \in DB$. e is an event in ss.*

Consider the following running example. Let us have the following sequence database $SeqDB$ shown in Table 6.1.

Identifier	Sequence
$S1$	$\langle A, B, A, B, A, B, C, D, E\rangle$
$S2$	$\langle A, B, B, B, B\rangle$
$S3$	$\langle A, B, C, A, D, E, B, C\rangle$
$S4$	$\langle A, B, C, C, A, B\rangle$

Table 6.1: Sample $SeqDB$

Support of pattern $\langle A, B, C\rangle$ can be found by first constructing the projected database of $\langle A\rangle$. This is shown below in Table 6.2.

Instance	Remainder of Sequence
$(1, 1, 1)$	$\langle B, A, B, A, B, C, D, E\rangle$
$(1, 3, 3)$	$\langle B, A, B, C, D, E\rangle$
$(1, 5, 5)$	$\langle B, C, D, E\rangle$
$(2, 1, 1)$	$\langle B, B, B, B\rangle$
$(3, 1, 1)$	$\langle B, C, A, D, E, B, C\rangle$
$(3, 4, 4)$	$\langle D, E, B, C\rangle$
$(4, 1, 1)$	$\langle B, C, C, A, B\rangle$
$(4, 5, 5)$	$\langle B\rangle$

Table 6.2: Sample $SeqDB^{itr}_{\langle A\rangle}$

The projected database $SeqDB^{itr}_{\langle A,B\rangle}$ can then be constructed from $SeqDB^{itr}_{\langle A\rangle}$ using the inductive step of Definition 6.7. Equivalently, we apply the projected-first operation to the $SeqDB^{itr}_{\langle A\rangle}$ with respect to event B. The result is shown below in Table 6.3.

Instance	Remainder of Sequence
$(1, 1, 2)$	$\langle A, B, A, B, C, D, E\rangle$
$(1, 3, 4)$	$\langle A, B, C, D, E\rangle$
$(1, 5, 6)$	$\langle C, D, E\rangle$
$(2, 1, 2)$	$\langle B, B, B\rangle$
$(3, 1, 2)$	$\langle C, A, D, E, B, C\rangle$
$(3, 4, 7)$	$\langle C\rangle$
$(4, 1, 2)$	$\langle C, C, A, B\rangle$
$(4, 5, 6)$	$\langle\rangle$

Table 6.3: Sample $SeqDB^{itr}_{\langle A,B\rangle}$

Finally, performing the inductive step of Definition 6.7 to $SeqDB^{itr}_{\langle A,B\rangle}$ will result in $SeqDB^{itr}_{\langle A,B,C\rangle}$ from which support of $\langle A,B,C\rangle$ can be found. Equivalently, we apply the projected-first projection to $SeqDB^{itr}_{\langle A,B\rangle}$ with respect to event C. The projected database is as shown below in Table 6.4.

Instance	Remainder of Sequence
$(1,5,7)$	$\langle D,E\rangle$
$(3,1,3)$	$\langle A,D,E,B,C\rangle$
$(3,4,8)$	$\langle\rangle$
$(4,1,3)$	$\langle C,A,B\rangle$

Table 6.4: Sample $SeqDB^{itr}_{\langle A,B,C\rangle}$

The support of $\langle A,B,C\rangle$ is then given by the size of $SeqDB^{itr}_{\langle A,B,C\rangle}$ which is 4: one from $S1$, two from $S3$ and another one from $S4$ in $SeqDB$.

Generating a full-set of frequent iterative patterns results in many "redundancies". As all sub-sequences of a frequent iterative pattern P having corresponding instances are frequent, the number of frequent patterns is potentially exponential to the maximum length of the iterative patterns. Mining for closed patterns is an effective solution to circumvent this pattern explosion issue. Besides reducing the final number of patterns, closed pattern mining can usually reduce run-time by pruning the search space.

Definition 6.10 (Prefix Extension Events) *For a pattern P, its set of prefix extension events is defined as the set of length-1 items e where $sup(e{+\!+}P) = sup(P)$.*

Definition 6.11 (Infix Extension Events) *An event e is an infix extension of a pattern P iff $\exists$ a super-sequence Q where: (1) $SeqDB^{itr}_{P} = SeqDB^{itr}_{Q}$, (2) first(P) = first(Q), (3) $\forall$ event $ev1 \in erasure(Q,P)$. $ev1 = e$ (e can appear many times), (4) sup(P) = sup(Q), and (5) Every instance of P corresponds to a unique instance of Q.*

Definition 6.12 (Suffix Extension Events) *For a pattern P, its set of suffix extension events is defined as the set of length-1 items e where $sup(P{+\!+}e) = sup(P)$.*

Prefix/ suffix extension events define events that can be added as prefix/ suffix (of length 1) to a pattern and results in another pattern having the same support[2]. Infix

[2]Patterns $e{+\!+}P$ and $P{+\!+}e$ will have corresponding instances as P iff $sup(e{+\!+}P) = sup(P)$ and $sup(P{+\!+}e) = sup(P)$, respectively – see proof of Theorem 6.3

extension events define events that can be added as infix to a pattern and results in another pattern having the same support and corresponding instances.

As an example, consider the sample database in Table 6.1. For pattern $\langle D \rangle$, its set of prefix extension events is $\{\langle A,\ B,\ C \rangle\}$. For pattern $\langle A, C \rangle$, its set of infix extension events is $\{\langle B \rangle\}$. For pattern $\langle A \rangle$, its set of suffix extension events is $\{\langle B \rangle\}$.

The above definitions are used in the next two theorems, which are then used for incremental and early detection of closed patterns and early pruning of search space.

Theorem 6.3 (Extension Closure Checks) *If there exists no prefix, infix and suffix extension event w.r.t. a pattern P, P must be a closed pattern; otherwise P must be non-closed.*

Proof 6.3 *Part 1: If there exists a prefix, infix or suffix extension event, then P must be non-closed.*

Consider a pattern P (where $|P| = n$). If there exists a suffix extension event e, there exists another pattern Q (P++ e) having the same support and a corresponding set of instances as P.

Patterns P and Q have a corresponding set of instances due to the following. The sub-string from the 1^{st} to the n^{th} landmark (inclusive) of an instance of Q is an instance of P. Hence, every instance of Q matches an instance of P. Also, since sup(P) = sup(P++e), we have every instance of P matches an instance of Q as well. They have corresponding instances.

Similarly, if there exists a prefix extension event e, there exists another pattern Q (e++P) having the same support and a corresponding set of instances as P. Hence, if there exists a prefix or suffix extension event for P, we can create a super-sequence of P having the same support and a corresponding set of instances (i.e., P is not closed).

Consider a pattern P. If there exists an infix extension event e, we can create another pattern Q super-sequence of P having the same support and corresponding instances. Hence, P is not closed.

Part 2: If there exists no prefix, suffix and infix extension event P must be closed.

We can only extend a pattern by adding prefix, infix and suffix to it. Hence, if *we cannot find a prefix, infix and suffix extension event of a pattern P resulting in its super-sequence having the same support, P must be closed.*

It is enough to consider a single event extension since apriori *property holds for patterns having corresponding instances.*

As an example, consider the sample database in Table 6.1. For pattern $\langle A, B, C\rangle$, its sets of prefix, suffix and infix extension events are empty. We can conclude that the pattern $\langle A, B, C\rangle$ is closed. On the other hand, for pattern $\langle A\rangle$, its set of suffix extension events is not empty. Hence it is not closed since there exists a pattern $\langle A, B\rangle$ which is a super-sequence of $\langle A\rangle$ with the same support.

Theorem 6.4 (InfixScan Search Space Pruning) *Given a pattern P, if there exists an infix extension event e w.r.t. a pattern P and arbitrary sid and ss where (sid,e,ss) $\notin$ $SeqDB_P^{itr}$, then all patterns of the form P++evs (where evs is an arbitrary series of events) are not closed. Hence, we can stop extending pattern P.*

Proof 6.4 *From Definition 6.11, if a pattern P has an infix extension event e, there exists a super-sequence pattern Q where: (1) $SeqDB_P^{itr} = SeqDB_Q^{itr}$, (2) $\forall$ event $ev1 \in erasure(Q, P)$. $ev1 = e$, (3) $sup(P) = sup(Q)$, and (4) every instance of P corresponds to a unique instance of Q.*

Since $SeqDB_P^{itr} = SeqDB_Q^{itr}$, if we can extend an instance s_x in $Inst(P)$ (and also in $Inst(Q)$) with a substring s_{ext} where $erasure(s_x$++s_{ext}, $erasure(s_x$++s_{ext}, P++s_{ext})) = P++s_{ext}, $erasure(s_x$++s_{ext},$erasure(s_x$++s_{ext}, Q++s_{ext})) will also be equal to Q++s_{ext}.

Since e is not in $SeqDB_P^{itr}$, whenever P++s_{ext} violate erasure constraint so does Q++s_{ext}.

Thus, given an arbitrary series of events s_{ext}, if P++s_{ext} is frequent, there exists another pattern Q++s_{ext} having the same support and corresponding instances. Hence, any pattern having P as prefix will not be closed. We can stop extending pattern P.

As an example, consider the sample database in Table 6.1. For pattern $\langle A, C\rangle$, its set of infix extension events is $\{B\}$. There is no point extending pattern $\langle A, C\rangle$ further. Consider, for example, pattern $\langle A, C, D\rangle$ of support 1. It is not closed since, there exists pattern $\langle A, B, C, D\rangle$, which is a super-sequence and has the same support and corresponding instances as the pattern $\langle A, C, D\rangle$.

The next section outlines our algorithm utilizing the above closure checks and InfixScan search space pruning for efficient memory and time utilization and for pruning of redundant search space.

6.3 Algorithm

Our CLIPER (CLosed Iterative Pattern minER) algorithm is shown in Figure 6.2. The main procedure to compute the closed set of iterative patterns: **MinePatterns**, is shown at the top of the figure. It calls a recursive procedure **MineRecurse** shown at the bottom of the figure.

```
Procedure MinePatterns
Inputs:
$SeqDB$ : Sequence Database
$min\_sup$: Minimum Support Threshold
Methods:
1:  Let $Freq = \{p|(|p|==1) \wedge (|Inst(p, ProjDB)| \geq min\_sup)\}$
2:  For every $f\_ev$ in $Freq$
3:    Call MineRecurse  ($f\_ev$,$SeqDB^{itr}_{f\_ev}$, $min\_sup$,$Freq$)
4:  End For

Procedure MineRecurse
Inputs:
$Pat$ : Pattern so far
$SeqDB^{itr}_{Pat}$ : Projected Database
$min\_sup$: Minimum Support Threshold
$EV$: Set of Frequent Events
Methods:
5:  Let $Freq = \{e|e \in EV \wedge (Seq(e, SeqDB^{itr}_{Pat}) \geq min\_sup)\}$
6:  If $(PreExt(Pat) == \{\} \wedge InfixExt(Pat) == \{\} \wedge SufExt(Pat) == \{\})$
7:    Output $Pat$
8:  End If
9:  For every $f\_ev$ in $Freq$
10:     Let $NxtPat = Pat{+\!+}f\_ev$
11:     Let $ProjDB = (SeqDB^{itr}_{Pat})^{fst}_{f\_ev}$
12:     If $(\nexists\, e.\ (e \in InfixExt(NxtPat) \wedge e \notin ProjDB))$
13:       Call MineRecurse ($NxtPat$,$ProjDB$,$min\_sup$,$EV$)
14:     End If
15:  End For
```

Figure 6.2: CLIPER Algorithm

Procedure **MinePatterns** first finds patterns of length one whose number of instances is more than or equal to *min_sup* threshold. For all frequent length-1 patterns, it then calls the procedure **MineRecurse** to recursively extends each of the patterns.

The recursive algorithm **MineRecurse** has as inputs the pattern prefix computed so far (Pat), the projected-iter sequence database ($SeqDB^{itr}_{Pat}$), the support threshold, and the set of frequent events.

The algorithm first finds length-1 event e such that $Pat{+\!+}e$ is frequent. Given the input pattern Pat and an event e, the number of instances of $Pat{+\!+}e$ equals to the number of triples (sid, px, sx) in $SeqDB^{itr}_{Pat}$ where we can extend px to an instance of $Pat{+\!+}e$. This number corresponds to $Seq(e, SeqDB^{itr}_{Pat})$.

A set of prefix extension events of Pat is the set of events e such that $sup(e{+\!+}Pat) = sup(Pat)$. A set of suffix extension events of Pat is the set of events e such that $sup(Pat{+\!+}e) = sup(Pat)$. Infix extension events define events that can be inserted (one or more times) as infix to a pattern and result in another pattern having the same support and corresponding instances. If there are no prefix, suffix and infix extensions of Pat, by Theorem 6.3, we can output Pat as a closed pattern.

Next, for any frequent pattern $Pat{+\!+}e$, following Theorem 6.4, we check for its infix extension events. If there is an infix extension event which does not appear in $SeqDB^{itr}_{Pat{+\!+}e}$, we do not need to extend the pattern $Pat{+\!+}e$ anymore.

Extension of patterns is performed recursively. At each step, given an extension event e, the *projected-iter* database $SeqDB^{itr}_{Pat{+\!+}e}$ needs to be computed. It can be computed incrementally by taking the projected-first database of $SeqDB^{itr}_{Pat}$ (*i.e.*, $(SeqDB^{itr}_{Pat})^{fst}_{e}$).

The algorithm can be adapted easily to mine a full set of frequent iterative patterns. This is performed as a point of reference for investigating the benefit of the closure check and InfixScan search space pruning strategies. To do this we simply skip the closure checks (at line 6) and the InfixScan pruning strategy (at line 12).

6.4 Performance Study

Experiments had been performed on both synthetic and real datasets to evaluate the scalability of our mining algorithm and the effectiveness of our pruning strategy. Similar to work in closed sequential pattern mining [174, 167], low support thresholds are utilized to test for scalability.

Datasets. We use three datasets in our experiments: a synthetic and two real datasets. A synthetic data generator provided by IBM was used with modification to ensure generation of sequences of events. The generators accept a set of parameters. The parameters D, C, N and S, correspond, respectively to the number of sequences (in 1000s), the average number of events per sequence, the number of different events (in 1000s) and the average number of events in the maximal sequences. We experimented with the dataset D5C20N10S20.

We also experimented on click stream dataset (*i.e.*, Gazelle dataset) from KDD Cup 2000 [97], which was also used to evaluate CloSpan [174] and BIDE [167]. It contains 29369 sequences with an average length of 3 and a maximum length of 651.

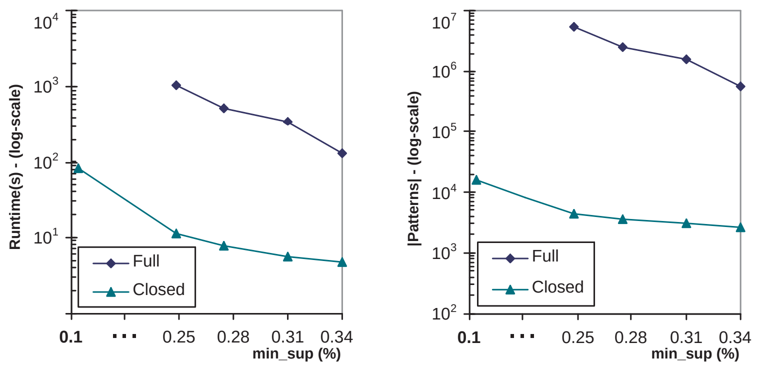
To evaluate our algorithm performance on mining from program traces, we generate traces from a simple Traffic alert and Collision Avoidance System (TCAS) from the Siemens Test Suite [78], which has been used as one of the benchmarks for research in error localization (*e.g.*, [110]). The test suite comes with 1578 correct test cases. We run these test cases to obtain 1578 traces.

To test for scalability, instead of tracing method invocations, we trace executions of basic blocks of TCAS's control flow graph. A basic block is a maximal block of sequential statements. Each trace of basic block ids is treated as a sequence. The sequences are of average length of 36 and maximum length of 70. It contains 75 different events – the events are the basic block ids of the control flow graph of TCAS. We call this dataset the TCAS dataset.

Environment and Pattern Miners. All experiments were performed on a Pentium 4 3.0GHz PC with 2GB main memory running Windows XP Professional. Algorithms were written using Visual C#.Net running under .Net Framework 2.0 with generics compiled with the release option using Visual Studio.Net 2005.

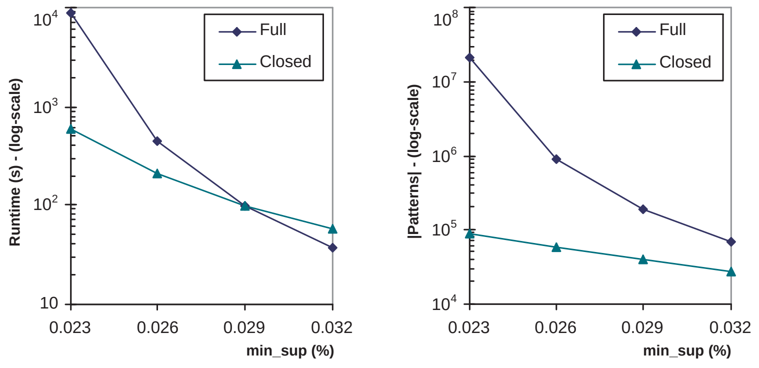
For the experiments we tested our pattern miner on two configurations to test the effectiveness of our pruning strategy. The first mines a closed set of iterative patterns while another mines a full set of iterative patterns. Let's refer to the earlier as *closed* iterative pattern miner and the latter as *full-set* iterative pattern miner.

Experiment Results and Analysis. The results of experiments performed on the D5C20N10S20, Gazelle and Siemens dataset using closed and full-set iterative pattern miners are shown in Figures 2, 3, and 4 respectively. The Y-axis (in log-scale) corresponds to the runtime taken or the number of generated patterns. The X-axis corresponds to the minimum support thresholds. The thresholds are reported relative to the number of sequences in the database. Note that due to repeated patterns within a sequence this number can exceed 1.



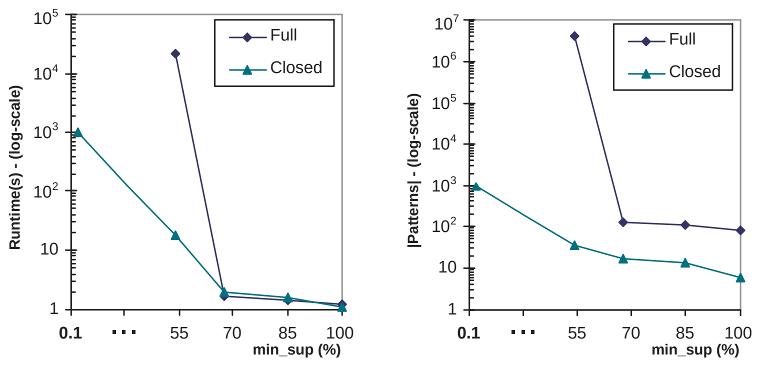
(a)*Runtime* (b)*No. of Patterns*

Figure 6.3: Performance Results for D5C20N10S20 Dataset



(a)*Runtime* (b)*No. of Patterns*

Figure 6.4: Performance Results for Gazelle Dataset



(a)*Runtime* (b)*No. of Patterns*

Figure 6.5: Performance Results for TCAS Dataset

From the plotted results, we note that the pruning strategy significantly reduces the runtime and the number of patterns mined, especially on low support threshold and when the reported patterns are long. Admittedly, when the numbers of closed and full-set of patterns differ by only a small factor, the overhead of mining closed patterns may result in longer runtime as compared to mining a full-set of patterns. However, when the length of the patterns is long, the number of closed patterns is likely to be much less than that of a full-set of patterns.

For all datasets, even at very low support, closed pattern miner is able to complete in less than 17 minutes. TCAS dataset especially highlights the performance benefit of our pruning strategy. Closed iterative pattern miner is able to complete in reasonable time even at the *lowest possible support threshold* (at 1 instance). On the other hand, full-set iterative pattern miner runs with excessive runtime ($>$ 6 hours) even at a relatively high support threshold of 867 instances.

The above shows that our miner can efficiently perform its task on various benchmark data. Comparison of performance results of closed and full-set pattern miner highlights the benefit and effectiveness of our pruning strategy.

6.5 Case Study: JBoss Application Server

A case study was performed on the transaction component of JBoss Application Server (JBoss AS) [87]. JBoss AS is the most commonly used J2EE application server. It contains over 100,000 lines of code and comments. The transaction component alone contains over 5,000 lines of code and comments. The purpose of this case study is to show the usefulness of the mined patterns by discovering iterative patterns describing the behavior of the transaction sub-component of JBoss AS.

Traces are obtained by running JBoss-AOP [86] over JUnit and Ant on a regression test of the JBoss AS transaction manager. We trace invocations of methods within the transaction component of JBoss AS (*i.e.*, org.jboss.tm package). This produces 28 traces of a total of 2551 events and an average of 91 events. The longest trace is of 125 events. There are 64 unique events. Using min_sup of 65%, the closed iterative pattern mining algorithm runs in less than a minute (29s). Full-set pattern mining doesn't terminate even after running for *more than 8 hours* and produces more than *5 GB of patterns.*

There are a total of 44 patterns resulting from the following post-processing step after iterative pattern mining:

1. Density. Only report patterns whose number of unique events is $> 80\%$ of its length.
2. Subsumption. Only report pattern P if none of its super-sequences is frequent.
3. Ranking. Order them according to length and support values.

We found at least five interesting software patterns of behavior resulting from mining the traces. These correspond to the patterns of longest length and highest support. Their abstracted representations are as follows:

1. ⟨Connection Set Up Evs, TxManager Set Up Evs, Transaction Set Up Evs, Transaction Commit Evs, Transaction Disposal Evs⟩
2. ⟨Connection Set Up Evs, TxManager Set Up Evs, Transaction Set Up Evs, Transaction Rollback Evs, Transaction Disposal Evs⟩
3. ⟨Resource Enlistment Evs, Transaction Execution Evs, Transaction Commit Evs, Transaction Disposal Evs⟩
4. ⟨Resource Enlistment Evs, Transaction Execution Evs, Transaction Rollback Evs, Transaction Disposal Evs⟩
5. ⟨Lock-Unlock Evs⟩

The first four patterns correspond to some of the longest patterns; the last pattern on lock and unlock events corresponds to the pattern with the highest support of 313. The actual mined pattern for the first pattern shown above, which is the longest pattern mined (of length 32), is shown in Figure 6.6.

The first pattern specifies a frequent behavior: a connection is first set up to the server, the transaction manager is set up, the transaction is set up, the transaction is committed and the transaction is finally disposed. The second pattern specifies a similar behavior except that the transaction is being roll-backed.

The third and fourth patterns specify the pattern observed when the actual work is being performed. A resource needs to be enlisted to the transaction and the transaction execution then takes place. At the end of the execution, the transaction can either be committed or roll-backed. Note that there can be one or more resource enlistments and transaction executions before a commit. Hence the pattern is not included in the body of the first two patterns.

Connection Set Up
TransactionManagerLocator.getInstance
TransactionManagerLocator.locate
TransactionManagerLocator.tryJNDI
TransactionManagerLocator.usePrivateAPI

Tx Manager Set Up
TxManager.begin
XidFactory.newXid
XidFactory.getNextId
XidImpl.getTrulyGlobalId

Transaction Set Up
TransactionImpl.associateCurrentThread
TransactionImpl.getLocalId
XidImpl.getLocalId

Transaction Set Up (Con't)
LocalId.hashCode
TransactionImpl.equals
TransactionImpl.getLocalIdValue
XidImpl.getLocalIdValue
TransactionImpl.getLocalIdValue
XidImpl.getLocalIdValue

Transaction Commit
TxManager.commit
TransactionImpl.commit
TransactionImpl.beforePrepare
TransactionImpl.checkIntegrity
TransactionImpl.checkBeforeStatus

Transaction Commit (Con't)
TransactionImpl.endResources
TransactionImpl.completeTransaction
TransactionImpl.cancelTimeout
TransactionImpl.doAfterCompletion
TransactionImpl.instanceDone

Transaction Dispose
TxManager.releaseTransactionImpl
TransactionImpl.getLocalId
XidImpl.getLocalId
LocalId.hashCode
LocalId.equals

Figure 6.6: Longest Iterative Pattern Mined from JBoss Transaction Component

The fifth pattern corresponds to a more fine grained iterative pattern occurring most often, namely lock and unlock.

6.6 Conclusion

In this chapter, we propose iterative pattern mining which extracts a set of frequently occurring series of events exhibited repeatedly within a sequence and across multiple sequences. We extend sequential pattern mining to consider repeated occurrences of pattern instances *within* sequences. We extend episode pattern mining by removing the constraint on *window size* and consider a *database of sequences* rather than a single sequence. To mine iterative pattern efficiently, we present CLosed Iterative Pattern MinER (CLIPER).

The motivation of our work comes from the emerging field of dynamic analysis where a set of program traces is analyzed to mine interesting software properties. Due to looping similar patterns occur within a sequence and across multiple sequences. Mining interesting patterns should take into account both multiple sequences, and multiple occurrences of patterns within a sequence. Also, since important patterns like lock-acquire followed-by lock-release and file-open followed-by file close (*c.f.*, [176, 28]) are often separated by a considerable number of events, we need to remove the window size constraint of frequent episode mining.

To reduce the number of reported patterns and improve efficiency, we mine for the set of closed iterative patterns. This reduces the run-time needed for mining patterns and aids a user in analyzing important patterns by sifting out patterns “absorbed” by another.

Our performance study shows the efficiency of our method in both real-world and synthetic datasets. The effectiveness of our pruning strategy to mine closed patterns is evident by comparing the runtime and the number of patterns generated before and after the pruning strategy is employed. The set of interesting patterns mined from JBoss Application Server transaction component confirms the usefulness of our method in discovering software specifications in iterative pattern form.

Besides mining software behavioral pattern, we believe iterative pattern mining can potentially be applied to other knowledge discovery domains.

CHAPTER VII

MINING SOFTWARE TEMPORAL RULES

Aside from mining frequent patterns of software behavior, most specification mining algorithms extract specification in the form of an automaton (*e.g.*, [113, 7, 150, 33]) or a set of two-event rules (*e.g.*, [176]). Different from frequent patterns, these two forms of specifications describe constraints on the ordering of events. While a mined automaton expresses a global picture of a software specification, mined rules break this into smaller parts each expressing a strongly expressed program property which is easily understood. On the other hand, existing work on mining rules only mines two-event rules (*e.g.*, $\langle lock \rangle \rightarrow \langle unlock \rangle$) which are limited in their ability to express complex temporal properties. A longer rule can correspond to an interesting temporal property that a software developer is more likely to miss. In general, the longer a rule is the harder it would be to mine the rule.

In this chapter, we describe a novel technique to automatically discover rules *of arbitrary lengths* having the following form from program execution traces:

> "Whenever a series of precedent events occurs, eventually another series of consequent events occurs"

A trace can be viewed as a series of events, with each event corresponding to a software behavior of interest. In the existing literature on specification mining [176, 113, 7] a trace usually corresponds to a series of signatures of methods which are invoked when a program is executed. A multi-event rule is denoted by $ES_{pre} \rightarrow ES_{post}$, where ES_{pre} and ES_{post} are the premise/pre-condition and the consequent/post-condition, respectively.

The above multi-event rule can be expressed in temporal logic, and belongs to two of the most frequently used families of temporal logic expressions for model checking (*i.e.*, response and chain-response) according to a survey in [41]. Examples of such rules include:

1. Resource Locking: Whenever a lock is acquired, eventually it is released.
2. Initialization-Termination: Whenever a series of initialization events is performed, eventually a series of termination events is also performed.
3. Internet Banking: Whenever a connection to a bank server is made, an authentication is completed, and money transfer command is issued, eventually money is transferred and a receipt is displayed.

From traces, many rules can be inferred, but not all are important. Statistics of support and confidence employed in data mining [61] is therefore used to identify important rules. Rules satisfying user-specified thresholds of minimum support and confidence are referred to as *statistically significant* rules.

Effective search space pruning strategies are utilized to efficiently mine multi-event rules from traces. To prevent an explosion in the number of mined rules, we define a *redundancy relation* among rules, and propose to generate only a minimal subset of rules containing non-redundant ones (see Section 7.2). With respect to input traces and given statistical significance thresholds, our algorithm is *statistically sound* as all mined rules are statistically significant (*i.e.*, they meet the thresholds). It is also *complete* as all statistically significant rules of the form $ES_{pre} \rightarrow ES_{post}$ are mined or represented.

We carried out a performance study on several standard benchmark datasets to demonstrate the effectiveness of our search space pruning strategies. We performed a case study on JBoss Application Server – the most widely used J2EE server – to illustrate the usefulness of our technique in recovering the specifications that a software system obeys. We also performed a case study on a buggy Concurrent Versions System (CVS) application. It shows the usefulness of our technique in mining bug-revealing properties, thus aids model checkers in finding bugs.

Our contributions are as follows:

1. We address the limitations of approaches extracting automata-based specification from traces, by discovering statistically sound & complete, and easily understood specifications in temporal logic format frequently used for model checking purpose.

2. We increase the power of existing algorithms mining two-event LTL rules/properties to mining properties of arbitrary lengths. Longer rules/properties can express more complex temporal constraints which are more likely to be missed by software developers.
3. We propose a mining-maintenance framework composed of: instrumentation, trace collection & abstraction, rule mining, post-processing, visualization and model checking.
4. We show the utility of our technique in recovering specifications of a large industrial program.
5. We demonstrate the usefulness of mined LTL rules/properties to reveal bugs from a buggy CVS application.

The outline of this chapter is as follows. Section 7.1 contains important background information on LTL formalizing our definition of temporal rules. Section 7.2 presents the principles behind mining temporal rules and the pruning strategies employed. Section 7.3 presents our algorithm. Section 7.4 describes our performance study on standard data mining benchmark datasets. Section 7.5 describes our case studies. Section 7.6 discusses some issues and their potential solutions, and Section 7.7 concludes this chapter.

7.1 Preliminaries

This section introduces preliminaries on LTL and its verification which dictate the semantics of temporal rules.

Linear-time Temporal Logic Our mined rules can be expressed in Linear Temporal Logic (LTL) [79]. LTL is a logic that works on possible program paths. A possible program path corresponds to a program trace. A path can be considered as a series of events, where an event is a method invocation. For example, (file_open, file_read, file_write, file_close), is a 4-event path.

There are a number of LTL operators, among which we are only interested in the operators 'G','F' and 'X'. The operator 'G' specifies that *globally* at every point in time a certain property holds. The operator 'F' specifies that a property holds either at that point in time or *finally (eventually)* it holds. The operator 'X' specifies that a property

$F(unlock)$ Meaning: Eventually *unlock* is called
$XF(unlock)$ Meaning: From the next event onwards, eventually *unlock* is called
$G(lock \rightarrow XF(unlock))$ Meaning: Globally whenever *lock* is called, then from the next event onwards, eventually *unlock* is called
$G(main \rightarrow XG(lock \rightarrow (\rightarrow XF(unlock \rightarrow XF(end)))))$ Meaning: Globally whenever *main* followed by *lock* are called, then from the next event onwards, eventually *unlock* followed by *end* are called

Table 7.1: LTL Expressions and their Meanings

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

Table 7.2: Rules and their LTL Equivalences

holds at the *next* point in time. Some examples are listed in Table 7.1.

Our mined rules state whenever a series of precedent events occurs eventually another series of consequent events also occurs. A mined rule denoted as *pre* $\rightarrow$ *post*, can be mapped to its corresponding LTL expression. Examples of such correspondences are shown in Table 7.2. Note that although the operator 'X' might seem redundant, it is needed to specify rules such as $\langle a\rangle\rightarrow\langle b, b\rangle$ where the 'b's refer to *different occurrences of 'b'*. The set of LTL expressions minable by our mining framework is represented in the Backus-Naur Form (BNF) as follows:

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

Checking/Verifying LTL Expressions. LTL expressions are mainly used for checking software systems expressed in the form of an automaton [75] (a transition system with start and end nodes). There are existing tools converting code to an automaton (*e.g.*, [35]). Given an automaton and an LTL property one can check if the automaton satisfies the LTL property through a well-known technique of model checking [29].

Consider the example in Figure 7.1, the pseudo-code on the left corresponds to the automaton on the right. Given the property $\langle main, lock\rangle \rightarrow \langle unlock, end\rangle$, a model

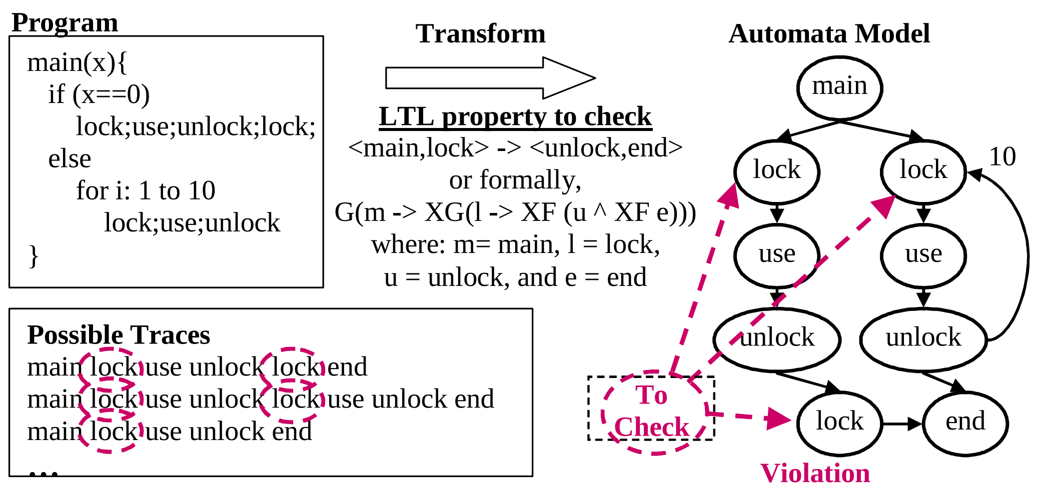


Figure 7.1: Checking Property: Code -> Model -> Verification

checking tool (*c.f*, [29]) will ensure that for all states in the model where *lock* preceded by a *main* occurs (marked by the (red) dashed arrows), eventually (whichever path is taken) *unlock and* then eventually *end* can be reached. For the above model in Figure 7.1, the property is violated. The *lock* immediately before *end* is not followed by an *unlock*. Note however, the property $\langle main, lock, use\rangle \rightarrow \langle unlock, end\rangle$ is satisfied. This is the case since the *lock* immediately before *end* is not followed by a *use*, i.e., the pre-condition of the rule is not satisfied and the rule vacuously holds.

An execution trace can be considered to be a finite path in the automaton and corresponds to a series of events. An event in turn corresponds to a behavior of interest, e.g. method call. A mined rule (or property) *pre* → *post* with a perfect confidence (*i.e.*, confidence=1) states that in the traces from all states where *pre* holds eventually *post* occurs. In the above example, for all points in the traces (*i.e.*, temporal points) where $\langle main, lock\rangle$ occurs (marked with dashed red circle), one needs to check whether eventually $\langle unlock, end\rangle$ occurs. Based on the definition of LTL properties and how they are verified, our technique analyzes traces and captures *strongly observed* LTL expressions i.e., obeying minimum support and confidence thresholds.

7.2 Generation of Temporal Rules

Each temporal rule of interest has the form $P_1 \rightarrow P_2$, where P_1 and P_2 are two series of events. P_1 is referred to as the *premise* or *pre-condition* of the rule, while P_2 is referred to as the *consequent* or *post-condition* of the rule. The rules correspond to temporal

Identifier	**Trace/Sequence**
S1	$\langle a, b, e, a, b, c\rangle$
S2	$\langle a, c, b, e, a, e, b, c\rangle$
S3	$\langle a, d\rangle$

Table 7.3: Example Database – $DBEX$

constraints expressible in LTL notations. Some examples are shown in Table 7.2.

In this chapter, since a trace is a series of events, where an event corresponds to a software behavior of interest, e.g., method call, *we formalize a trace as a sequence and a set of input traces as a sequence database.* We use the sample trace or sequence database in Table 7.3 as our running example to illustrate the concepts behind generation of temporal rules.

7.2.1 Concepts & Definitions

Mined rules are formalized as Linear Temporal Logic(LTL) expressions with the format: G(... → XF...). The semantics of LTL and its verification technique described in Section 7.1 will dictate the semantics of temporal rules described here. Noting the meaning of the temporal operators illustrated in Table 7.1, to be precise, a temporal rule expresses:

"Whenever a series of events *has just occurred at a point in time (i.e. a temporal point)*, eventually another series of events occurs"

From the above definition, to generate temporal rules, we need to "peek" at interesting temporal points and "see" what series of events are likely to occur next. We will first formalize the notion of temporal points and the related notion of occurrences.

Definition 7.1 (Temporal Points) *Consider a sequence S of the form $\langle a_1, a_2, \ldots, a_{end}\rangle$. All events in S are indexed by their positions in S, starting at 1 (e.g., a_j is indexed by j). These positions are called* temporal points *in S. For a temporal point j in S, the prefix $\langle a_1, \ldots, a_j\rangle$ is called the j-prefix of S.*

Definition 7.2 (Occurrences & Instances) *Given a pattern P and a sequence S, the* occurrences *of P in S are defined by a set of temporal points $\mathcal{T}$ in S such that for each $j \in \mathcal{T}$, the j-prefix of S is a super-sequence of P and $last(P)$ is indexed by j. The set of* instances *of pattern P in S is defined as the set of j-prefixes of S, for each $j \in \mathcal{T}$.*

Example. Consider a pattern P $\langle a, b\rangle$ and the sequence $S1$ in Table 7.3 (*i.e.*, $\langle a, b, e, a, b, c\rangle$). The *occurrences* of P in $S1$ form the set of temporal points {2,5}, and the corresponding set of *instances* are $\{\langle a, b\rangle, \langle a, b, e, a, b\rangle\}$.

We define database projection operations to capture events occurring after specified temporal points. The following are two different types of projections and their associated support notions.

Definition 7.3 (Projected & Sup) *A database* projected *on a pattern p is defined as:*

SeqDB$_P$ = {(j, sx) | *the* j^{th} *sequence in SeqDB is s, where* $s = px{+\!\!+}sx$, *and* px is the minimum prefix of s containing p}

Given a pattern P_X*, we define sup(*P_X*,SeqDB) to be the size of SeqDB*$_{P_X}$ *(equivalently, the number of sequences in SeqDB containing* P_X*). Reference to the database is omitted, i.e., we write it as sup(*P_X*), if the database is clear from the context, e.g., it refers to input sequence database SeqDB.*

Definition 7.4 (Projected-all & Sup-all) *A database* projected-all *on a pattern p is defined as: SeqDB*$_P^{all}$ = {(j, sx) | *the* j^{th} *sequence in SeqDB is s, where* $s = px{+\!\!+}sx$, *and* px is an instance of p in s *and last(px) = last(p)*}

Given a pattern P_X*, we define* $sup^{all}(P_X, SeqDB)$ *to be the size of* $SeqDB^{all}_{P_X}$*. Reference to the database is omitted if it is clear from the context.*

Definition 7.3 defines a standard database projection (*c.f.* [174, 167]) capturing events occurring after the *first temporal point.* Definition 7.4 is a new type of projection to capture events occurring after *each temporal point.*

Example. To illustrate the above concepts, we project and project-all the example database $DBEX$ with respect to the pattern $\langle a, b\rangle$. The results are shown in Tables 7.2.1(a) & (b) respectively.

The two projection methods' associated notions of *sup* and sup^{all} are different. Specifically, sup^{all} reflects the number of occurrences of P_X in *SeqDB* rather than the number of sequences in *SeqDB* supporting P_X.

Example. Consider the example database, $sup(\langle a, b\rangle, DBEX) = |DBEX_{\langle a,b\rangle}| = 2$. On the other hand, $sup^{all}(\langle a, b\rangle, DBEX) = |DBEX^{all}_{\langle a,b\rangle}| = 4$.

(a)

Identifier.	**Trace/Sequence**
S1	(1,$\langle e, a, b, c\rangle$)
S2	(2,$\langle e, a, e, b, c\rangle$)

(b)

Identifier	**Trace/Sequence**
$S1_1$	(1,$\langle e, a, b, c\rangle$)
$S1_2$	(1,$\langle c\rangle$)
$S2_1$	(2,$\langle e, a, e, b, c\rangle$)
$S2_2$	(2,$\langle c\rangle$)

Table 7.4: $(a); DBEX_{\langle a,b\rangle}$ & $(b); DBEX^{all}_{\langle a,b\rangle}$

From the above notions of temporal points, projected databases and pattern supports, we can define the support and confidence of temporal rules.

Definition 7.5 (Support & Confidence) *Consider a rule R_X ($pre_X \rightarrow post_X$). The* support *of R_X is defined as the number of sequences in SeqDB where pre_X occurs, which is equivalent to sup(pre_X, SeqDB). The confidence of R_X is defined as the likelihood of $post_X$ happening after pre_X. This is equivalent to the ratio of sup($post_X$,$SeqDB^{all}_{pre_X}$) to the size of $SeqDB^{all}_{pre_X}$.*

Example. Consider $DBEX$ and a temporal rule R_X, $\langle a, b\rangle \rightarrow \langle c\rangle$. From the database, the support of R_X is the number of sequences in $DBEX$ supporting (or is a super-sequence of) the rule's pre-condition – $\langle a, b\rangle$. There are 2 of them – see Table 7.2.1(a). Hence support of R_X is 2. The confidence of the rule R_X ($\langle a, b\rangle \rightarrow \langle c\rangle$) is the likelihood of $\langle c\rangle$ occurring after each *temporal point* of $\langle a, b\rangle$. Referring to Table 7.2.1(b), we see that there is a $\langle c\rangle$ occurring after each temporal point of $\langle a, b\rangle$. Hence, the confidence of R_X is 1.

Significant rules to be mined must have their supports greater than the *min_sup* threshold, *and* their confidences greater than the *min_conf* threshold.

In mining program properties, the confidence of a rule (or property), which is a measure of its certainty, matters the most (*c.f.*, [176]). Support values are considered to differentiate high confidence rules from one another according to the frequency of their occurrences in the traces. Rules with confidences less than 100% are also of interest due to the imperfect trace collection and the presence of bugs and anomalies [176]. Similar to the assumption made by work in statistical debugging (*e.g.*, [49]), simply put, if a program behaves in one way 99% of the time, and the opposite 1% of the time, the latter likely corresponds to a possible bug. Hence, a high confidence and highly supported rule is a good candidate for bug detection using program verifiers.

We added the notions of support and confidence to the temporal rules. The formal notation of temporal rules is defined below.

Definition 7.6 (Temporal Rules) *A temporal rule R_X is denoted by pre $\rightarrow$ post (sup,conf). The series of events pre and post represent the rule's pre- and post-condition and are denoted by R_X.Pre and R_X.Post respectively. The notions sup, and* conf *represent the support, and confidence of R_X respectively. They are denoted by sup(R_X) and conf(R_X) respectively.*

Example. Consider $DBEX$ and the rule R_X, $\langle a, b\rangle \rightarrow \langle c\rangle$ shown in the previous example. It has support of 2 and confidence of 1. It is denoted by $\langle a, b\rangle \rightarrow \langle c\rangle(2, 1)$.

7.2.2 Apriori properties and Non-Redundancy

Our algorithm is a new addition to the family of pattern mining algorithms, e.g. [4, 5, 174, 167]. Apriori properties have been widely used to ensure efficiency of many pattern mining techniques (*e.g.*, [4, 5]). One of the novelty in our new mining algorithm is the identification of new and suitable apriori properties that apply. Fortunately, temporal rules obey the following apriori properties:

Theorem 7.1 (Apriori Property – Support) *If a rule $evs_P \rightarrow evs_C$ does not satisfy the* min_sup *threshold, neither will all rules $evs_Q \rightarrow evs_C$ where evs_Q is a super-sequence of evs_P.*

Theorem 7.2 (Apriori Property – Confidence) *If a rule $evs_P \rightarrow evs_C$ does not satisfy the min_conf threshold, neither will all rules $evs_P \rightarrow evs_D$ where evs_D is a super-sequence of evs_C.*

To reduce the number of rules and improve efficiency, we define a notion of rule redundancy defined based on *super-sequence relationship* among rules having the same support and confidence values. This is similar to the notion of *closed* patterns applied to sequential patterns [174, 167].

Definition 7.7 (Rule Redundancy) *A rule R_X ($pre_X \rightarrow post_X$) is redundant if there is another rule R_Y ($pre_Y \rightarrow post_Y$) where:*

(1) R_X is a sub-sequence of R_Y (i.e., $pre_X{+\!\!+}post_X \sqsubseteq pre_Y{+\!\!+}post_Y$)

(2) Both rules have the same support and confidence values

Also, in the case that the concatenations are the same (i.e., $pre_X{+\!\!+}post_X = pre_Y{+\!\!+}post_Y$), to break the tie, we call the one with the longer premise as being redundant (i.e., we wish to retain the rule with a shorter premise and longer consequent).

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

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

If all of the above rules have the same support and confidence values, rules R2-R5 are redundant since they are represented by rule R1. To keep the number of mined rules manageable, we remove redundant rules. Noting the combinatorial nature of redundant rules, removing redundant rules can drastically reduces the number of reported rules.

A simple approach to reduce the number of rules is to first mine a full-set of rules and then remove redundant ones. However, this "late" removal of redundant rules is inefficient due to the exponential explosion of the number of intermediary rules that need to be checked for redundancy. To improve efficiency, it is therefore necessary to identify and prune a search space containing redundant rules "early" during the mining process. The following two theorems are used for 'early' pruning of redundant rules.

Theorem 7.3 (Pruning Redundant Pre-Conds) *Given two pre-conditions P_X and P_Y where $P_X \sqsubset P_Y$, if $SeqDB_{P_X} = SeqDB_{P_Y}$ then for all sequences of events post, rules $P_X \rightarrow post$ is rendered* redundant *by $P_Y \rightarrow post$ and can be pruned.*

Proof 7.1 *Since $P_X \sqsubset P_Y$, from Definition 7.7 of rule redundancy, we only need to prove that the rules R_X ($P_X \rightarrow post$) and R_Y ($P_Y \rightarrow post$) have the same values of support and confidence.*

Since $SeqDB_{P_X} = SeqDB_{P_Y}$ *, the followings are guaranteed: (1)* P_X *and* P_Y *must share the same suffix (at least* $last(P_X) = last(P_Y)$*) and (2)* $\forall s \in SeqDB$*. the first instance of* P_X *corresponds to the first instance of* P_Y*. From points (1) and (2) above, not only the first instance, but every instance of* P_X *in* $SeqDB$ *must also correspond to an instance of* P_Y *(and vice versa). In other words,* $SeqDB_{P_X} = SeqDB_{P_Y}$ *iff* $SeqDB^{all}_{P_X} = SeqDB^{all}_{P_Y}$*.*

Since $SeqDB^{all}_{P_X} = SeqDB^{all}_{P_Y}$*, and* R_X *and* R_Y *share the same post-condition,* R_X *and* R_Y *must have the same support and confidence values. Hence,* R_X *is rendered redundant by* R_Y *and can be pruned.* □

Theorem 7.4 (Pruning Redundant Post-Conds) *Given two rules* R_X *(*$pre \rightarrow P_X$*) and* R_Y *(*$pre \rightarrow P_Y$*) if* $P_X \sqsubset P_Y$ *and* $(SeqDB^{all}_{pre})_{P_X} = (SeqDB^{all}_{pre})_{P_Y}$ *then* R_X *is rendered* redundant *by* R_Y *and can be pruned.*

Proof 7.2 *Since* $P_X \sqsubset P_Y$*, from Definition 7.7 of rule redundancy, we only need to prove that the rule* R_X *(*$pre \rightarrow P_X$*) and* R_Y *(*$pre \rightarrow P_Y$*) have the same values of support and confidence. The equality of support values is guaranteed since the two rules have the same pre-condition.*

Since $(SeqDB^{all}_{pre})_{P_X} = (SeqDB^{all}_{pre})_{P_Y}$*, it implies* $sup(P_X, SeqDB^{all}_{pre}) = sup\ (P_Y, SeqDB^{all}_{pre})$*. Hence, the two rules will have the same confidence values.*

Hence, we have shown that R_X *is rendered redundant by* R_Y *and can be pruned.* □

Utilizing Theorems 7.3 & 7.4, many redundant rules can be pruned 'early'. However, the theorems only provide sufficient conditions for the identification of redundant rules – there are redundant rules which are not identified by them. To remove remaining redundant rules, we perform a post-mining filtering step based on Definition 7.7.

Our approach to mining a set of non-redundant rules satisfying the support and confidence thresholds is as follows:

Step 1 Leveraging Theorems 7.1 & 7.3, we generate a *pruned* set of pre-conditions satisfying *min_sup*.

Step 2 For each pre-condition *pre*, we create a *projected-all* database $SeqDB^{all}_{pre}$.

Step 3 Leveraging Theorems 7.2 & 7.4, for each $SeqDB^{all}_{pre}$, we generate a *pruned* set containing such post-condition *post*, such that the rule $pre \rightarrow post$ satisfies *min_conf.*

Step 4 Using Definition 7.7, we filter any remaining redundant rules.

In the next section, we describe our algorithm in detail.

7.3 Algorithm

In the previous section, the process of mining non-redundant rules has been divided into 4 steps. Steps 1 and 3 sketch how a pruned set of pre- and post- conditions are mined. The following paragraphs will elaborate them in more detail.

7.3.1 Mining Algorithm

Before proceeding, we first describe a set of patterns called *projected database closed* (or LS-Set) first mentioned in [174]. A pattern is in the set if *there does not exist any super-sequence pattern having the same projected database.* Patterns having the same projected database must have the same support, but not vice versa. *Projected database closed* patterns is of special interest to us, as explained in the following paragraphs.

At step 1, a pruned set of pre-conditions is generated from the input database *SeqDB*. From Theorem 7.3, a pattern is in the pruned pre-condition set if *there does not exist any super-sequence pattern having the same projected database.* Comparing with the definition of *projected database closed* patterns in the previous paragraph, we note that this pruned set of pre-conditions corresponds to the *projected database closed set* (or LS-Set) mined from *SeqDB*.

At step 3, starting with a projected-all database $SeqDB^{all}_{pre}$, we generate a pruned set of post-conditions. From Theorem 7.4, a pattern is in the pruned post-condition set if *there does not exist any super-sequence pattern having the same projected database.* Again, this set of pruned post-conditions corresponds to the *projected database closed set* (or LS-Set) mined from $SeqDB^{all}_{pre}$.

Our mining algorithm, TERMINAL (TEmporal Rule MIning ALgorithm) is shown

```
Procedure Mine Non-Redundant Temporal Rules
Inputs:
$SeqDB$ : Set of Input Traces Represented As a Sequence Database
$min\_sup$ : Minimum Support Threshold
$min\_conf$ : Minimum Confidence Threshold
Output:
$Rules$: Non Redundant Set of Temporal Rules
Method:
1:  Let $PreCond$ =  Generate an LS-Set from $SeqDB$ with the
                     threshold set at $min\_sup$
2:  For every $pre$ in $PreCond$
3:     Let $SeqDB^{all}_{pre}$ = $SeqDB$ projected-all by pattern $pre$
4:     Let $bthd = min\_conf \times |SeqDB^{all}_{pre}|$
5:     Let $PostCond$ =  Generate an LS-Set from $SeqDB^{all}_{pre}$ with the
                         threshold set at $bthd$
6:     For every $post$ in $PostCond$
7:        Add ($pre \rightarrow post$) to $Rules$
8:  For every $rx$ in $Rules$
9:     If ($rx$ is redundant according to Def. 7.7)
10:       Remove $rx$ from $Rules$
11: Output $Rules$
```

Figure 7.2: TERMINAL Pseudocode

in Figure 7.2. First, a pruned set of pre-conditions satisfying the minimum support threshold (*i.e.*, *min_sup*) is mined using an LS-Set miner modified from BIDE [167], the state-of-the-art closed sequential pattern miner. BIDE in effect prunes all search sub-spaces containing patterns not in LS-Set. To mine LS-Set using BIDE, we keep the search space pruning strategy but remove the closure check. The details are available in sub-section 7.3.2. Next, for each pre-condition mined, a database projected-all on it is formed. Consequently, another LS-Set Generator is run on each projected-all database to mine the set of post-conditions of the corresponding candidate rules having enough confidence values. Finally, a filtering step to remove any remaining redundant rules based on Definition 7.7 is performed. To perform the final filtering step scalably, each remaining rule is first hashed based on its support and confidence values. Only rules falling into the same hash bucket need to be checked for super-sequence relationship.

The algorithm can be adapted easily to generate a full set of temporal rules satisfying *min_sup* and *min_conf* thresholds. This is performed to serve as a point of reference for investigating the benefit of early identification and pruning of redundant rules. To generate the full set we can simply: (1) Generate a full set of pre- and post- conditions of rules satisfying the support and confidence thresholds at lines 1 and 5 of the algorithm respectively (we use PrefixSpan [143] for this purpose), and (2) Skip the final redundancy filtering step (*i.e.*, lines 8-10 of the algorithm in Figure 7.2).

7.3.2 Mining LS-Closed Patterns

To see how BIDE [167], the state-of-the-art closed sequential pattern miner, can be modified to mine LS-Closed patterns, we first describe some terminology mentioned in BIDE's paper [167]. We next present some theorems and relate these to how BIDE's algorithm can be modified.

Definition 7.8 (First instance of a pattern) *Given a sequence S which contains a single event pattern $\langle e_1 \rangle$, the prefix of S to the first appearance of the event e_1 in S is called the first instance of pattern $\langle e_1 \rangle$ in S. Recursively, we can define the first instance of a (i + 1)-event pattern $\langle e_1, e_2, \ldots, e_i, e_{i+1} \rangle$ from the first instance of the i-event pattern $\langle e_1, e_2, \ldots, e_i \rangle$ (where $i \geq 1$) as the prefix of S to the first appearance of event e_{i+1} which also occurs after the first instance of the i-event pattern $\langle e_1, e_2, \ldots, e_i \rangle$. For example, the first instance of the prefix sequence $\langle A, B \rangle$ in sequence $\langle C, A, A, B, C \rangle$ is $\langle C, A, A, B \rangle$.*

Definition 7.9 (I-th last-in-first appearance) *For an input sequence S containing a pattern $P = \langle e_1, e_2, \ldots, e_n \rangle$, the i-th last-in-first appearance w.r.t. the pattern P in S is denoted as LF_i and defined recursively as: (1) if $i = n$, it is the last appearance of e_i in the first instance of pattern P in S; (2) if $1 \leq i < n$, it is the last appearance of e_i in the first instance of the pattern P in S while LF_i must appear before LF_{i+1}. For example, if $S = \langle C, A, A, B, C \rangle$ and $Sp = \langle C, A \rangle$, the 2nd last-in-first appearance w.r.t. pattern P in S is the first A in S.*

Definition 7.10 (I-th semi-maximum period) *For an input sequence S containing a pattern $P = \langle e1, e2, \ldots, e_n \rangle$, the i-th semi-maximum period of the pattern P in S is defined as: (1) if $1 < i \leq n$, it is the piece of sequence between the end of the first instance of pattern $\langle e_1, e_2, \ldots, e_{i-1} \rangle$ in S (exclusive) and the i-th last-in-first appearance w.r.t. pattern P (exclusive); (2) if $i = 1$, it is the piece of sequence in S locating before the 1st last-in-first appearance w.r.t. pattern P. For example, if $S = \langle A, B, C, B \rangle$ and the* *pattern $P = \langle A, C \rangle$, the 2nd semi-maximum period of prefix $\langle A, C \rangle$ in S is $\langle B \rangle$, while the 1st semi-maximum period of pattern $\langle A, C \rangle$ in S is an empty string.*

Lemma 7.1 *A pattern P is in LS-Closed if and only if there does not exist an event e, where e appears in each of the i-th semi-maximum periods w.r.t. P for all $S \in SeqDB$.*

Proof 7.3 The left to right direction. *We first would like to show that if P is in LS-Closed, then there is no e, where e is in each of the i-th semi-maximum periods w.r.t. P for all $S \in SeqDB$. Taking the contrapositive of the above statement we have if there is an e which is in each of the i-th semi-maximum periods w.r.t. P for all $S \in SeqDB$, then P is not in LS-Closed.*

Suppose there is an e which is in each of the i-th semi-maximum period w.r.t. P for all $S \in SeqDB$, we can then form a longer pattern P' by inserting e between event (i-1) and (i) of P (if $i > 1$) or by pre-pending e before P (if $i = 1$) which will have the same support as P. From the definition of semi-maximum period, first instances of P' and P in SeqDB will be the same. Hence, P and P' have the same projected database. Since there exists a P' which is a super-sequence of P having the same projected database, P is not in LS-Closed. This is a contradiction. We have proven the left to right direction *of the lemma.*

The right to left direction. *We next need to show that if there is no e where e is in each of the i-th semi-maximum period w.r.t. P for all $S \in SeqDB$, P will be in LS-Closed. Again, taking the contrapositive, the above statement is equivalent to: if P is not in LS-Closed then there exists an event e, where e is in each of the i-th semi-maximum period w.r.t. P for all $S \in SeqDB$.*

Suppose P is not in LS-Closed, this means that there exists a longer pattern P', where P' is a super-sequence of P, the length of P' is one event longer than P and they have the same projected database. It must be the case then that there exists two shorter patterns X and Y, where:

$$P = X{+\!+}e2{+\!+}Y$$
$$P' = X{+\!+}e{+\!+}e2{+\!+}Y$$

Since P and P' have the same projected database, for all sequence S in SeqDB, *the first instance of P' in each sequence S which is a super-sequence of P' in $SeqDB$ will also be the first instance of P. Let i = the length of pattern X. From the above, e must occurs between the first instance of X (exclusive) and the $(i+1)$-th last-in-first appearance w.r.t. to P (exclusive) for all sequence S in $SeqDB$. From the definition of semi-maximum period, e must be in each of the i-th semi-maximum period w.r.t. P for all $S \in SeqDB$. We have proven the* right to left direction *of the lemma.*

Lemma 7.2 *If P and P' have the same projected database and P' is a super-sequence of P, then for an arbitrary series of events evs, P++evs will not be in LS-Closed.*

BIDE employs the search space pruning strategy called backscan pruning: Let *evs* be an arbitrary series of events, if a pattern P has an event e appearing in each of its i-th semi-maximum period for all sequence S in $SeqDB$ than P as well as P++ *evs* are not in CS-Closed. Lemma 7.1 guarantees that any pattern not pruned by the backscan pruning strategy must be in LS-Closed. Lemma 7.2 guarantees that there is no point in extending pattern P if it has been pruned by the backscan pruning strategy.

Using the above two lemmas, one can continue to cut the search space by using the backscan pruning of BIDE. BIDE employs an online check to see whether a pattern which is not pruned is in CS-Closed which is called the BIDE closure checking scheme. Removing this closure checks will modify BIDE to mine LS-Closed instead of CS-Closed pattern set.

7.4 Performance Evaluation

Experiments have been performed on both synthetic and real datasets to evaluate the *scalability of our mining framework* on standard data mining benchmark datasets. Low support threshold similar to the range considered in [174, 167, 118] is utilized to test for scalability. Our algorithms are the *first* algorithms mining multi-event temporal rules, hence we compare and contrast the runtime required when full and non-redundant sets of temporal rules are mined to evaluate the *effectiveness of our non-redundant rule pruning strategies* (*i.e.*, Theorems 7.3 & 7.4).

We use 2 datasets in our experiments: one synthetic and another real. Synthetic data generator provided by IBM (*c.f.* [5]) was used with modification to generate synthetic traces. We produce a synthetic dataset by running the IBM synthetic data generator

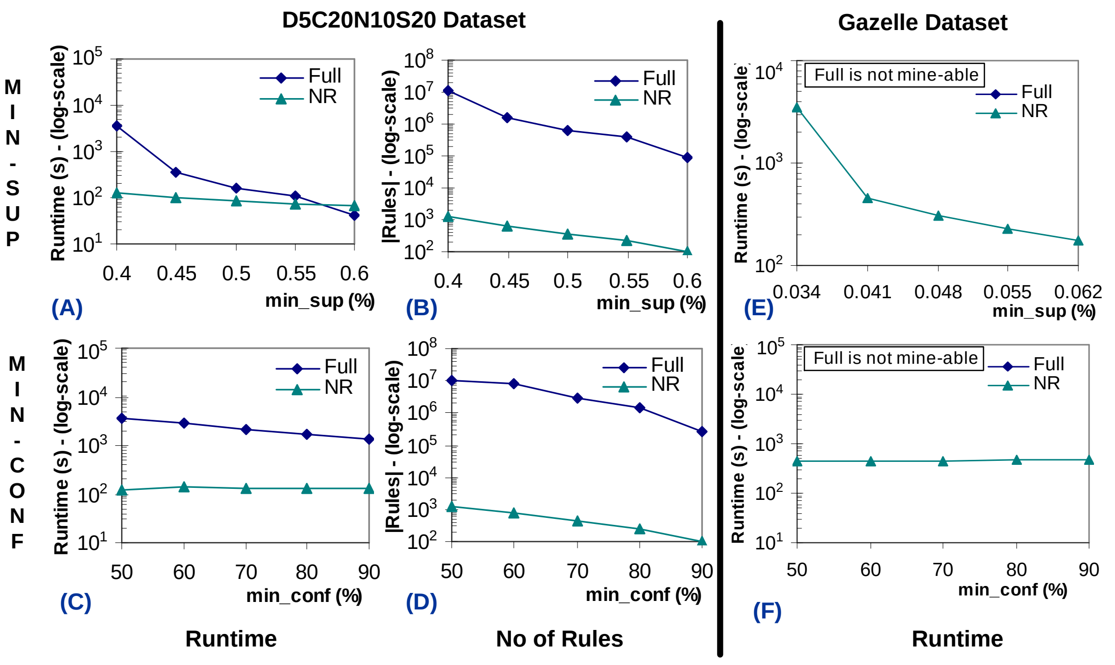


Figure 7.3: Varying *min_sup* at *min_conf*=50% (A,B) and *min_conf* at *min_sup*=0.4 % (C,D) for D5C20N10S20 dataset. Varying *min_sup* at *min_conf*=50% (E) & *min_conf* at *min_sup*=0.041% (F) for Gazelle dataset

with the following parameter setting: D (no of sequences - in 1000s) = 5, C (avg. sequence length) = 20, N (no of unique events - in 1000s) = 10 and S (avg. no of events in maximal sequences) = 20. We also experimented on a click stream dataset (*i.e.*, Gazelle dataset) from KDD Cup 2000 [97]. It contains 23639 sequences with an average length of 3 and a maximum length of 651. Both the synthetic data generator and Gazelle dataset have been standard benchmarks used in pattern mining research [5, 174, 167, 118].

All experiments were performed on a Pentium M 1.6GHz tablet PC with 1.5GB memory, running Windows XP Tablet PC Edition 2005. Algorithms were written using C#.Net.

Experiments were performed by varying *min_sup* & *min_conf* thresholds. The *min_sup* value is represented as a percentage ratio to the number of sequences in the database. The results are plotted as line graphs. 'Full' and 'NR' correspond to the full set and non-redundant set of rules respectively. The x-axis of each graph corresponds to one of the thresholds considered while the y-axis represents either the algorithm runtime or the number of mined rules.

The experiment results for mining rules from the synthetic dataset are shown in

Figure 7.3 (A,B,C,D). The experiment results for the Gazelle dataset are shown in Figure 7.3 (E,F).

From the two experiments we note that the runtime is significantly increased when the *min_sup* threshold is lowered. On the other hand, lowering the *min_conf* threshold has no much effect on the runtime. The results also show that we can efficiently mine temporal rules from standard data mining benchmark datasets even at a low *min_sup* thresholds. The lower the threshold the more difficult it is to mine the rules. We did not experiment with low *min_conf* thresholds as we believe the usefulness of low confidence rules (if any) is minimal.

Mining non-redundant rules rather than a full set of rules reduced the runtimes and the number of rules by up to more than *28 times less* and up to *8500 times less* respectively. Admittedly, the pruning strategies themselves require some computation cost, hence for cases where the benefit of the pruning strategies is less, mining a full set of rules might be slightly faster than the non-redundant set. However, the desired result is the non-redundant set of rules. The full set of rules contains too many redundant rules – up to more than ten million rules were produced! Also, for experiments with the real-life benchmark dataset Gazelle, the full set of rules is not mine-able even at the highest *min_sup* threshold shown in Figure 7.3 (E).

7.5 Case Studies

In this section we discuss our case studies on two different systems: JBoss Application Server and a buggy CVS application. The first study shows the utility of our method in recovering specifications of a large industrial system. The second study demonstrates the usefulness of mined LTL rules/properties to reveal bugs from a buggy application. A discussion on additional strategies to improve the scalability of our approach is also presented at the end of this section.

7.5.1 JBoss Application Server

JBoss AS is the most widely used J2EE application server. It contains over 100,000 lines of code and comments. The purpose of this study is to show the usefulness of the mined rules to describe the behavior of a real software system.

Case 1: JBoss AS Security Component. We instrumented the security component

Premise $\longrightarrow$	Consequent
XmlLoginCI.getConfEntry()	ClientLoginMod.initialize()
AuthenInfo.getName()	ClientLoginMod.login()
	ClientLoginMod.commit()
	SecAssocActs.setPrincipalInfo()
	SetPrincipalInfoAction.run()
	SecAssocActs.pushSubjectCtxt()
	SubjectThreadLocalStack.push()
	SimplePrincipal.toString()
	SecAssoc.getPrincipal()
	SecAssoc.getCredential()
	SecAssoc.getPrincipal()
	SecAssoc.getCredential()

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

Figure 7.4: A sample rule from JBoss-Security (top) and another from JBoss-Transaction (bottom). Each of the rules are read from top to bottom, left to right.

of JBoss-AS using JBoss-AOP and generated traces by running the test suite that comes with the JBoss-AS distribution. In particular, we ran the regression tests on Enterprise Java Bean (EJB) security implementation of JBoss-AS. Twenty-three traces of a total size of 4115 events, with 60 unique events, were generated. Running the algorithm with the minimum support and confidence thresholds set at 15 and 90% respectively, 6 non-redundant rules were mined. The algorithm completed within 3 seconds.

A sample of the mined rules is shown in Figure 7.4 (left). It describes authentication using Java Authentication and Authorization Service (JAAS) for EJB within JBoss-AS. When authentication scenario starts, first configuration information is checked to determine authentication service availability – this is described by the premise of the rule. This is followed by: invocations of actual authentication events, binding of principal information to the subject being authenticated, and utilizations of subject's principal and credential information in performing further actions – these are described by the consequent of the rule.

Case 2: JBoss AS Transaction Component. We instrumented the transaction component of JBoss-AS using JBoss-AOP and generated traces by running the test suite that comes with the JBoss-AS distribution. In particular, we ran a set of transaction manager regression tests of JBoss-AS. Each trace is abstracted as a sequence of events, where an event corresponds to a method invocation. Twenty-eight traces with a total size of 2551 events containing 64 unique events, were generated. Running the algorithm on the abstracted traces with the minimum support and confidence thresholds set at 25 traces and 90% respectively, 182 non-redundant rules were mined. The algorithm completed within 30 seconds.

In the presentation of mined rules, we display first rules in which their constituent events rarely repeat, and sort them according to their support and confidence values. These help to distinguish more interesting rules from the others.

A sample of the mined rules is shown in Figure 7.4 (right). The 19-event rule in Figure 7.4 (right) describes that the series of events ⟨connection to a server instance events, transaction manager and implementation set up event⟩ (event 1-10) at the start of a transaction is always followed by the series of events ⟨transaction completion events and resource release events⟩ (event 11-19) at the end of the transaction. The above rule describing the temporal relationship and constraint between the 19 events is hard to identify manually. The rule sheds light into the *implementation details* of JBoss AS which are implemented at various locations in (*i.e.*, crosscuts) the JBoss AS large code base.

7.5.2 Buggy CVS Application

This section describes a case study conducted on a buggy Concurrent Versions System (CVS) application adapted from the one studied in [112, 113]. This case study shows the usefulness of mined rules for model checking and bug detection.

CVS Scenarios. The CVS application is built on top of the FTP library provided by Jakarta Commons Net [10]. Jakarta Commons Net is a set of reusable open source Java code implementing the client side of many commonly used network protocols. The CVS application can be considered as a client of Jakarta Commons Net, and it follows a certain protocol described by a set of scenarios.

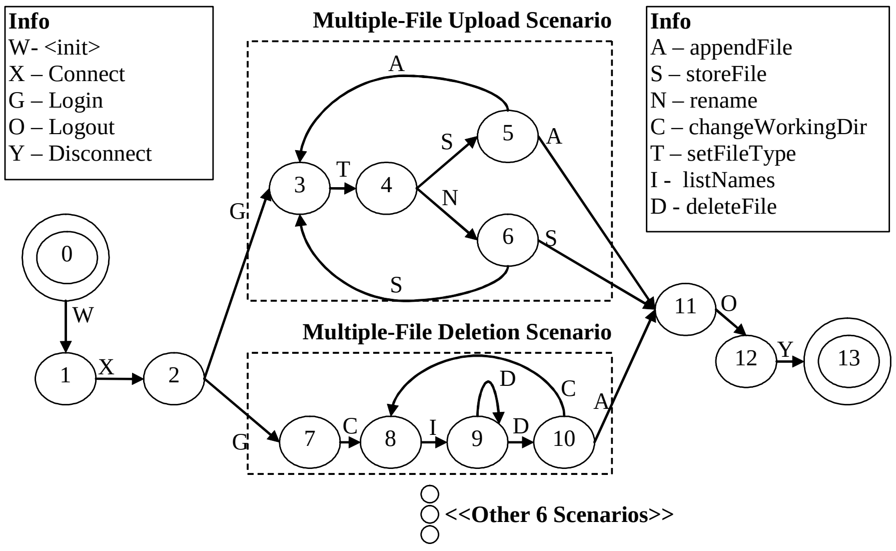


Figure 7.5: CVS Protocol

There are eight common FTP interaction scenarios in our CVS implementation: Initialization and re-initialization of a repository, multiple-file upload, download, rename and deletion, and multiple-directory creation and deletion. All scenarios begin by connecting and logging-in to the FTP server. They end by logging-off and disconnecting from the FTP server. The CVS interaction protocol can be represented as a 33-state automata partially drawn in Figure 7.5. We focus on the following two scenarios :

Multiple-File Upload Scenario. One can store one or more files. For each file, the following is performed. First, the type of the file to be transferred is set. If the file is new, store the file directly and append the new file information to the *CVS system file* in the server. Otherwise, rename the old file by adding to the filename the time it is replaced, and proceed to store the new file – the *CVS system file* need not be updated.

Multiple-File Deletion Scenario. One can delete one or more files. For each file, first go to its directory. List all files in the current working directory . Delete the file and all its previous versions. If the files are not located in the same directory, change the working directory accordingly. Finally, record file deletion information by appending it to the *CVS system file.*

Bug Description and Trace Generation Each invocation of a method of *FTPClient* may raise exceptions, especially *FTPConnectionClosedException* and *IOException.* Hence the code accessing *FTPClient* methods needs to be enclosed in a *try..catch..*

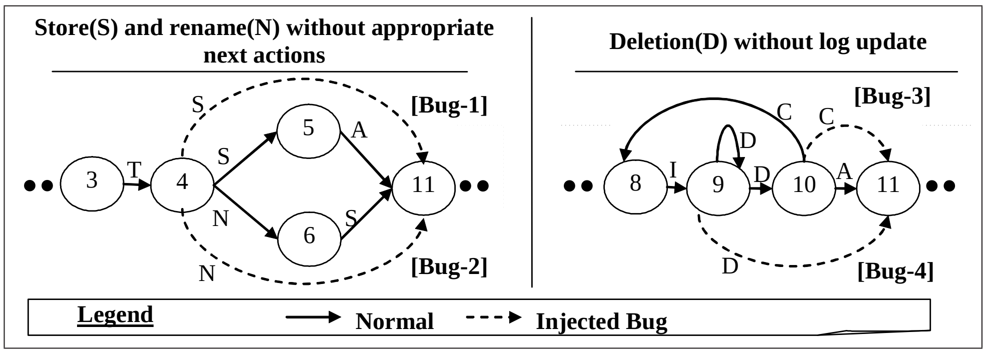

Store(S) and rename(N) without appropriate next actions
Deletion(D) without log update
[Bug-1]
[Bug-2]
[Bug-3]
[Bug-4]
3
4
5
6
11
8
9
10
11
T
S
N
A
I
D
C
Legend
Normal
Injected Bug

Figure 7.6: Injected Bug

finally block. Every time such an exception occurs the program simply logs out and disconnects from the FTP server. These are represented by adding *error transitions* (shown as dashed lines) to the automata as shown in Figure 7.6.

The above exceptions might cause a number of bugs, but we focus only on 4 bugs causing *the system log file to be in an inconsistent state.* The system file describes the state of the CVS repository and should be kept consistent with the stored files. Bugs of this type is illustrated by the error transitions shown in Figure 7.6. Due to the bugs, a file can be added or deleted without a proper log entry being made. An old version of a file can be renamed by appending a time-stamp without the new version being stored in the CVS. These bugs occur because the CVS scenarios are not executed atomically.

To generate traces, we followed the process discussed in [112]. First, we instrumented the CVS application byte code using an adapted version of Java Runtime Analysis Toolkit [91]. Next, we ran the instrumented CVS application over a set of test cases to generate traces. Via a trace post-processing step, we then filtered events in the traces not corresponding to the interactions between the CVS application and the Jakarta Commons Net FTP library. Thirty-six traces of a total size of 416 events were generated.

Mined Rules and Model Checking Results. We ran our mining algorithm on the generated traces. It completed within 1 second and mined 5 rules with minimum support and confidence thresholds set at fifteen traces and ninety percent respectively. Among the mined rules, the following two rules are the bug-revealing program properties:

1. Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, file type is set (T) and an old file is renamed(N), *then eventually* a new file is stored(S), followed by a logout (O) and a disconnection

from server (Y). This is *denoted as*: $\langle W, X, G, T, N \rangle \rightarrow \langle S, O, Y \rangle$.

2. Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, a working directory is set (C), *then eventually* the files in the directory is listed (I), and a file is deleted (D), a log entry is made (A), followed by a logout (O) and a disconnection from server (Y). This is *denoted as*: $\langle W, X, G, C \rangle \rightarrow \langle I, D, A, O, Y \rangle$.

We used a model checker outlined in [74]. We drafted an abstract model of the buggy CVS system from the code manually in the format accepted by the model checker and checked against the above two properties. Alternatively, one can try to use some tools that automate model generation from code, *e.g.*, Bandera [35]. Since the focus of this case study is on mining bug-revealing properties, we leave this for future work. The model checker reported violations of the above properties. These violations correspond to 3 out of the 4 bugs in the model. Bug-2 violates the first property. Bug-3 and Bug-4 violate the second property.

It is interesting to note that no two-event rules can be used in detecting these bugs, as invocations of FTP file delete(D), rename(R) and store(S) follow a different protocol in different scenarios (*e.g.*, 'D' must be followed by 'A' in file deletion but not in repository re-initialization scenario). Multi-event rules provide more precise contexts identifying the scenarios where specified events occur, thus enabling the detection of the bugs.

One of the bugs (Bug-1) cannot be revealed because the required bug-revealing property is outside the bound of the LTL expressions minable by our algorithm. This bug-revealing property is: Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, a file type is set (T), *and there is no rename (N) until a new file is stored (S)*, then eventually a log entry is made (A), followed by a logout (O) and a disconnection from server (Y). Note that the phrase in italics is not expressible by any of our mined rules whose format (in BNF) is described in Section 7.1. To mine such properties, we need to mine rules containing the LTL operators not($\neg$) and until(U) – this is a possible future work.

7.6 Discussion

Mining multi-event rules is more expensive than mining two-event ones. Search space pruning strategies help it scale for practical use. To further help in analyzing large systems, we can employ additional strategies during the trace generation step.

Traces can be generated by running the test suite that usually comes with the system to be analyzed. This test suite is usually composed of a set of regression tests, each focusing on a separate component of the system. Some of the regression tests are stress tests. From the above observations, the following trace-reduction strategies are employed:

1. Perform a divide-and-conquer strategy. Rather than analyzing traces generated by running the *entire* test suite, run each regression test testing a particular component *one at a time* and for each, analyze the generated traces *separately.*

2. Remove stress tests from the test suite. Running system stress tests usually contribute most to the size of the generated traces. With regard to mining various behaviors of a system (represented by a set of mined rules), stress tests are not very useful since each usually only tests a single behavior a large number of times. At times test cases corresponding to stress tests are identifiable from their name.

If the size of the traces is still too large, the following two strategies can be employed:

1. Mine n-event rules. Rather than mining all rules of arbitrary lengths, one can focus on mining rules composed of n or less events.
2. User-guided mining. Users can provide a set of premises to be considered for mining. The miner will then only mine for rules with the given premises.
3. Split long traces. A long trace can be split into a number of shorter trace segments. This is especially effective if a good separator event is known. In Section 7.4, we have shown that the algorithm scales well up to almost 30,000 traces (or sequences).

The above three strategies trade a degree of completeness for scalability. We find that in our case study, the trace-size reduction strategies are sufficient.

Note that the time taken for mining is *much improved* with search space pruning strategies. Without employing any search space pruning strategy (*i.e.*, if a similar approach to those in [176, 172] to mine two-event rules is employed), the mining process will require at least E^L check operations, where E is the number of unique events and L is the maximum length of the trace. For traces from JBoss AS considered, the mining process will require more than 50^{100} operations. Considering 1 picosecond per operation, it will only complete *in about* 2.501x10^{148} *centuries*! This highlights the power and importance of search space pruning strategies in improving the scalability of the mining process.

Mining more complex, general LTL expressions will be useful as they can capture more complex specifications which in turn can be used to detect more bugs. However this will enlarge the search space that a mining algorithm needs to traverse. Mining such LTL expressions scalably is challenging since the search space is much larger. To do this, there is a need first, to identify a way to systematically traverse the search space and second, to identify new effective pruning strategies to cut the search space containing insignificant or redundant LTL expressions. This is a potential future work.

7.7 Conclusion

In this chapter, a novel method to mine a *non-redundant* set of *statistically significant* rules *of arbitrary lengths* of the form: "Whenever a series of events ES_{Pre} occurs, eventually another series of events ES_{Post} also occurs" is presented. According to a survey in [41], these rules belong to two of the most frequently used families of temporal logic expressions for model checking. Our approach is *statistically sound and complete*, meaning all mined rules are statistically-significant and all statistically significant rules are mined or represented. The problems of a potentially exponential runtime cost and a huge number of reported rules have been effectively mitigated by employing search space pruning strategies and elimination of redundant rules. A case study on JBoss Application Server shows the utility of our technique in recovering specifications of a large industrial program. Another case study on a buggy CVS application demonstrates the usefulness of our approach in mining bug-revealing properties for bug detection using a model checker.

CHAPTER VIII

MINING LIVE SEQUENCE CHARTS

In this chapter, we focus on reactive systems [68]. A reactive system is a discrete event system which maintains ongoing interactions with its environment. Our goal is to mine such systems' behavioral specifications using inter-object scenarios. Scenarios, typically depicted using variants of sequence diagrams, are popular means to specify the inter-object behavior of systems (see, e.g., [69, 72, 98, 165]), are included in the UML standard [141], and are supported by many modeling tools.

In particular, we are interested in Damm and Harel's Live Sequence Charts (LSC) [37]. LSC can be viewed as a formal version of UML sequence diagram. It is composed of a *pre-chart* and a *main-chart*. It extends the partial order semantics of sequence charts with universal and existential modalities. Most relevant in our context is LSCs ability to express *universal liveness requirements*: whenever the pre-chart sequence of events occurs, the main-chart sequence must eventually be completed.

Moreover, based on the semantics of LSC Symbolic Instances [130], LSC allow the use of class-level lifelines. With the class level lifelines, user is able to specify scenarios not only at the level of concrete objects but also at the level of abstract classes. This takes advantage of object-oriented inheritance, and result in creating more expressive and succinct specifications.

The popularity and intuitive nature of sequence diagrams as a specification language in general, together with the additional unique features of LSC described above, motivate our choice for the target formalism of our miner. Moreover, the choice is supported by previous work on LSC (see, e.g., [70, 67, 96, 105, 129]), which in combination with existing standard tools can be practically used to visualize, analyze, manipulate, test, verify, and thus indeed use and evaluate the specifications we mine (see the examples in Section 8.4).

Different from temporal rules described in the previous chapter, an LSC is a sequence diagram that captures inter-object behavior. Hence, both caller-callee relationship and object identities need to be taken into consideration during LSC mining process. Moreover, we consider an abstraction mechanism to abstract object level LSCs to class and super-class level LSCs.

To mine for modal sequence diagrams, we start off with traces of events, where an event corresponds to the triple (caller object, callee object, method signature). In preceding chapters, an event only corresponds to a method signature.

Many possible modal scenarios can be inferred from any given trace – but not all are equally important. To reduce the potential information overload, as a measure of their importance, we utilize the concepts of *support* and *confidence*, adopted from the domain of data mining [61]. LSCs satisfying user-defined thresholds for minimum support and confidence are mined – these LSCs are referred to as being *statistically significant*. Introducing stochastic notions of support and confidence helps to recover the common behaviors, despite slight variations (which lead to so called "imperfect traces" [176]). The notions of support and confidence used are formally defined in Section 8.1.

LSC mining involves exploring the space of all LSCs for statistically significant ones. To allow the mining of multi-event LSCs to scale, we employ a search space pruning strategy, inspired by pattern mining methods [5]. Moreover, to further improve the effectiveness and usability of the mined LSCs, we introduce an array of extensions to the basic mining algorithm of object level LSCs. These include various filters and abstraction mechanisms as motivated and explained in Section 8.3. These can be utilized to reduce the number of mined LSCs, which not only improves efficiency and allows our approach to scale, but also, equally important, provides the user with only the most significant and informative LSCs mined.

To demonstrate and evaluate our approach, we present the results of a case study we have conducted using traces from various components of Jeti [89], an open-source Java based full featured instant messaging application (see Section 8.4). The results demonstrate the effectiveness of our mining technique in recovering non-trivial and highly expressive underlying interactions.

The paper is structured as follows. Section 8.1 discusses the semantics of Live Sequence Charts, the characteristics of the traces we use, and the formal definitions of our statistical metrics. An outline of the mining algorithm is given in Section 8.2. Section 8.3 describes the end-to-end mining framework, including extensions, filters, and abstraction methods. A comprehensive evaluation of our framework through a case study is presented in Section 8.4.Finally, Section 8.5 concludes.

8.1 Preliminaries

We briefly recall the syntax and semantics of modal sequence diagrams as they are used in our work and provide some preliminary notations and definitions.

8.1.1 Live Sequence Chart (LSC)

Live Sequence Charts (LSC) [37] is a visual formalism for inter-object scenario-based specifications which extends the partial order semantics of sequence diagrams in general with universal and existential modalities. LSC allows to specify not only 'may' scenarios but also 'must' and 'must not' (sub) scenarios. Its expressive power is comparable to that of various Temporal Logics [99]. LSC has been the subject of much previous work, e.g., in the contexts of scenario-based testing [105], synthesis [64], execution (play-out) [67], formal verification [96], specification and verification of hardware [22] and telecommunication systems [32], and compilation into aspects [70, 129].

We use here a restricted subset of the LSC language, adopted to our needs, i.e., we consider only universal diagrams that are built off a single pre-chart/main-chart cold/hot sub-chart pair, include no conditions (state-invariants), and induce a total rather than partial order. For a thorough definition of the LSC language we refer the reader to [37, 66, 67]. A possible extension of our work to a larger subset of the language is suggested in Section 8.5.

An LSC includes a set of instance lifelines, representing system's objects, and is divided into two parts, the *pre-chart* ('cold' fragment) and the *main-chart* ('hot' fragment), each specifying an ordered set of method calls between the objects represented by the instance lifelines. A universal diagram specifies a *universal liveness requirement*: for all runs of the system, and for every point during such a run, whenever the sequence of events defined by the pre-chart occurs (in the specified order), eventually the sequence of events defined by the main-chart must occur (in the specified order). Events not explicitly mentioned in the diagram are not restricted in any way to appear or not to appear during the run (including between the events that are mentioned in the diagram). If the pre-chart never occurs in a run, the universal LSC requirement still holds (in the context of mining, however, we are looking for universal LSCs with positive support (see sub-section 8.1.2)).

Syntactically, instance lifelines are drawn as vertical lines, pre-chart events are colored in blue (and drawn using a dashed line), and main-chart events are colored in red (and drawn using a solid line). LSCs can be edited and visualized within standard UML2 compliant modeling tools (e.g., IBM Rational Software Architect [80]) using the MSD profile [66].

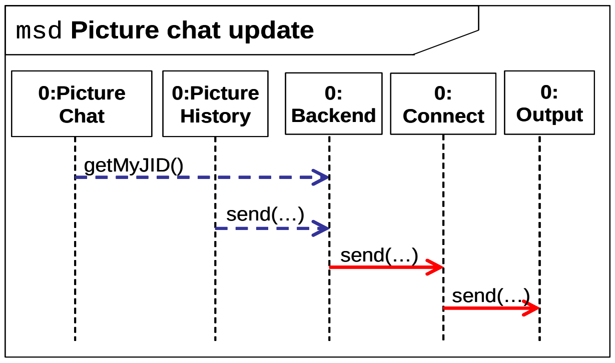


Figure 8.1: Mined LSC: *Picture chat update*

Figure 8.1 shows an example LSC (adopted from the case study described in Section 8.4). This LSC specifies that "whenever *PictureChat* calls the *Backend* method *getMyJID()*, and sometime in the future the *PictureHistory* calls the *Backend* method *send()*, eventually the latter must call the *send()* method of *Connect* and *Connect* must call the *send()* method of *Output*". Note that if the pre-chart begins but never completes (or the order of events violates it), the main-chart does not have to occur and there are no other restrictions on the order of the events appearing in it. In the example, if the first method *getMyJID()* is never called by the *PictureChat*, or if it is called but the next method *send()* never occurs, there is no constraint on the occurrence of the subsequent hot methods.

8.1.1.1 Symbolic Class-level LSCs

An additional important feature of LSC is its semantics of Symbolic Instances [130]. That is, rather then referring to concrete objects, instance lifelines may be labeled with a name of a class (or an interface) and defined as symbolic, i.e., formally representing any object of the referenced class. This allows a designer to take advantage of object-oriented inheritance and create more expressive and succinct specifications. Note that when an object 'binds' to a lifeline, it remains 'bound' for the entire scenario. Multiple occurrences of the same scenario, where lifelines are bound to different objects, may be ongoing simultaneously.

8.1.2 LSCs Over Finite Traces

As an input to the mining algorithm, we consider finite traces consisting of events, where each event corresponds to a triplet: caller object identifier, callee object identifier, and method signature.

In order to relate between LSCs and execution traces we use the following notation and definitions. To simplify the presentation, we abstract away object and method data from the notation below. We thus consider traces to be finite words over a finite alphabet of events $\Sigma = \{a, b, c...\}$. We use the symbol $+\!\!+$ to represent the concatenation operator between finite words. An LSC $M(pre, main)$ defines a word m built from the concatenation of its pre-chart and main-chart finite words, i.e., $m = pre+\!\!+main$. For two words w, u we denote the projection of w onto the alphabet of events appearing in u by w_u.

Next we define the notions of positive-witness, negative-witness, and strong-negative-witness for an LSC M and a trace T. A *positive witness* of a word w with respect to a trace T is defined as a *minimal* subword s of T such that $s_w = w$. A *positive witness* of an LSC $M(pre, main)$ with respect to a trace T is defined as a *minimal* subword s of T such that $s_m = m$ where $m = pre+\!\!+main$. The set of *positive witnesses* of a word w (an LSC M) with respect to a trace T is denoted $pos(w, T)$ ($pos(M, T)$). Note that T may include many positive-witnesses of M. For example, the trace $T_1 = eaeebabcedacbccdaaadabe$ includes 2 positive-witnesses of $M_1 = (ab, d)$, corresponding to the sub-strings *abced* and *acbccd* starting at the 6th and 11th positions of the trace, respectively.

A *negative-witness* of an LSC $M(pre, main)$ with regard to a trace T, is a positive-witness of the word *pre* that cannot be extended to a positive-witness of M. The set of *negative-witnesses* of an LSC M with respect to a trace T is denoted by $neg(M, T)$.

Using the example above, T_1 includes 2 negative-witnesses of M_1, corresponding to the sub-strings *aeeb* and *ab* starting at the 2nd and 21st positions of the trace, respectively.

The semantics of LSC (following that of LSC, and like most formal specification languages used for reactive systems, e.g. LTL [79]) is originally defined over infinite paths. The traces we consider, however, are, of course, finite, and we do not want the arbitrary truncation of the trace to affect the confidence of the universal liveness requirement specified by the mined LSC. We therefore need to adopt the semantics of LSC, and specifically, the definition of negative witnesses, to finite (so called 'truncated') paths using a notion of strong-negative-witness. Roughly, a strong-negative-witness is negative because it explicitly violates the order specified by the main part of the LSC and not because it reaches the end of the trace. Formally, a *strong-negative-witness* of an LSC $M(pre, main)$ with regard to a trace T, is a positive-witness of pre, p, such that for any word w, p cannot be extended to a positive-witness of M over $T{+}{+}w$.[1] The set of *strong-negative-witnesses* of an LSC M with respect to a trace T is denoted by $strong\_neg(M, T)$. Using the example above, note that the second negative-witness of M_1 in T_1 ends at the end of the trace and is not a strong-negative-witness.

We use the above notions of positive and strong-negative witnesses to define the statistical *support* and *confidence* metrics for LSC. Given a trace T, the *support* of an LSC $M(pre, main)$, denoted $sup(M)$, is simply defined as the number of positive witnesses of M found in T. The *confidence* of an LSC M, denoted $conf(M)$, measures the likelihood of a sequence satisfying pre to be followed by a sequence satisfying $main$. Formally:

$$\boxed{\begin{array}{l} sup(M,T) \equiv_{def} |pos(M,T)| \\ conf(M,T) \equiv_{def} \frac{|pos(M,T)|+(|neg(M,T)|-|strong\_neg(M,T)|)}{|pos(pre,T)|} \end{array}}$$

When T is understood from the context, it can be omitted. Using the example above we have $sup(M_1, T_1) = 2$, $conf(M_1, T_1) = (2 + 2 - 1)/4 = 0.75$.

8.2 Basic LSC Mining

We are now set to describe the basic LSC mining algorithm and sketch its soundness and completeness.

[1]This adaptation may be considered a special case of the notions suggested for LTL in [47], e.g., the distinction between strong and weak X (next) operator.

Many previous algorithms used for specification mining, e.g., [176], need to check *all possible* specifications obeying a certain template. The above approach does not scale for specifications of an arbitrary length since the number of possible specifications is *arbitrarily large*. Listing all of them out first, before checking their significance, simply would not scale. Instead, we use a user defined support threshold and the following monotonicity or apriori property to immediately prune search spaces containing non-significant LSCs.

Property 8.1 Monotonicity of Support *For a trace T, an LSC $M(pre, main)$, and a word w: $sup(pre{+\!+}main, T) \geq sup(pre{+\!+}main{+\!+}w, T)$.*

Intuitively, the above property means that if a certain LSC does not meet the minimum support threshold, all its extensions will not meet the minimum support threshold either.

Our algorithm LIVE (LIVE sequence chart mining algorithm) is outlined in Figures 8.2 & 8.3. Its input includes a trace and thresholds for support and confidence, and its output is a set of LSCs. The algorithm starts by mining a complete set of words meeting the support threshold, and then continues to compose these words into LSCs meeting the confidence threshold.

The main algorithm is given in procedure *MineLSC*, which calls the procedure *Mine-SupportedWords* to mine a complete set of words that meet the support threshold (line 1). *MineSupportedWords* calls *MineRecursive* to recursively add events to the current set of words in a depth first fashion. Once an extended word does not meet the support threshold (line 22), we know all its extensions will not meet the support threshold either, and thus we can stop recursing. After the set of words meeting the support threshold is mined, the algorithm continues to compose these words into LSCs meeting the confidence threshold (lines 3-11).

Our algorithm for LSC mining is sound and complete; i.e., not only all the output LSCs meet the support and confidence thresholds set in the input, but also all the possible LSCs that meet these thresholds are indeed included in the output. Soundness follows immediately from the algorithm. Completeness follows from the monotonicity property. The formal proof is outside the scope of this chapter.

```
Procedure MineLSC
Inputs:
$TR$ : Input Trace
$min\_sup$: Minimum Support Threshold
$min\_conf$: Minimum Confidence Threshold
Output:
A set of statistically significant LSCs
1: Let $WSet$ = MineSupportedWords($TR$,$min\_sup$)
2: Let $LSCResult = \{\}$
3: For every word $w$ in $WSet$
4:    For every prefix $pfx$ of $w$
5:       Let $main = sfx$, where $pfx ++ sfx = w$
6:       Let $NewLSC$ = Create new LSC ($pfx$,$main$)
7:       If ($conf(NewLSC) \geq min\_conf$)
8:          Output $NewLSC$
```

Figure 8.2: LIVE's Pseudocode

```
Procedure MineSupportedWords
Inputs:
$TR$ : Input Trace
$min\_sup$: Minimum Support Threshold
Output:
A set of supported words
9: Let $EV$ = Single-events appearing $\geq min\_sup$ in $TR$
10: Let $WSet = \{\}$
11: For every $f\_ev$ in $EV$
12:    Call MineRecurse($TR$,$min\_sup$,$EV$,$f\_ev$,$WSet$)
Return $WSet$
```

```
Procedure MineRecurse
Inputs:
$TR$ : Input Trace
$min\_sup$: Minimum Support Threshold
$EV$: Frequent Events
$curW$: Current word considered
$WSet$: Current set of supported words
Output:
Updated supported words set ($WSet$)
13: Add $curW$ to $WSet$
14: For every $f\_ev$ in $EV$
15:    Let $nxtW = curW ++ f\_ev$
16:    If ($|pos(nxtW,TR)| \geq min\_sup$)
17:       Call MineRecurse($TR$,$min\_sup$,$EV$,$nxtW$,$WSet$)
```

Figure 8.3: Mine Supported Words

8.3 The Big Picture

In this section we discuss the use of object information, present a series of extensions to the basic mining algorithm, and outline the complete LSC mining framework.

8.3.1 Using object information

8.3.1.1 Mining class-level LSCs

Class-level LSCs, following LSC Symbolic Instances [130], allow to specify scenarios at the level of (abstract) classes, and thus enable expressive and succinct scenario-based specifications in an object-oriented context. Syntactically, this is done by labeling instance lifelines with class names rather than specific object names.

Roughly, a class-level LSC specifies a modal scenario that applies to all objects of the classes referenced on its lifelines. However, in a specific instance of the scenario (i.e., in a specific positive-witness), each lifeline binds to a single object throughout the scenario. The semantics of class-level LSCs (as LSCs with symbolic instances) was defined in [130], implemented in the Play-Engine tool [67], and was specifically adopted to support object-oriented inheritance and interface implementation in Java in [70, 129].

As the input of our specification mining algorithm is a concrete trace, the basic algorithm mines only concrete object-level LSCs. In order to find class-level LSCs, we employ a process of generalization and aggregation, using the caller and callee identifiers attached to each event on the trace.

Since any positive (negative) witness of a concrete object-level LSC is also a positive (negative) witness of the corresponding class-level LSC, the support and confidence metrics extend naturally to the class-level case as totals. In addition, we compute and present for each class-level LSC mined the maximum support and confidence values over its corresponding concrete LSCs, and also the number of unique concrete corresponding LSCs. We allow the user to set minimum thresholds for these metrics. The miner will filter out class-level LSCs not meeting the thresholds.

8.3.1.2 Connectivity criterion

An LSC may be drawn as a graph whose nodes represent the participating instances and whose directed edges represent method calls (as in a UML2 *Communication Diagram*). We say that an LSC is connected if the resulting graph representation is connected (i.e., if all nodes are reachable from the initial node). While unconnected LSCs may be useful in general, in the context of mining they are probably of little value to the user. We thus check LSC connectivity using a simple depth-first search algorithm, and allow the user to filter out unconnected LSCs, despite their statistical significance. Note that connected sub-graphs of such LSCs, if statistically significant, are not filtered out in the process.

The ability to mine class-level LSCs and use connectivity as a criterion, rely on the fact that object information for caller and callee is *not* abstracted away in our traces. We believe these two features are novel and have important impact on the quality of the results and the usefulness of our approach.

8.3.2 User-guided filters and abstractions

Although the LSCs mined by the basic algorithm are statistically significant, many of them may still not be of high value to the user. The following lists a series of extensions to the basic algorithm that we have defined and implemented. Some extensions allow the user to use apriori knowledge (or otherwise, knowledge obtained from previous mining sessions) in order to define various abstractions and filters aiming at improving the quality and usefulness of the LSCs mined. Others are motivated by various properties of sequence diagrams in general and modal sequence diagrams in particular. Some of the presented extensions, when used, may also speed up the mining process significantly.

Ignore intra-object method calls. Some of the traced events may refer to method calls from an object to itself. Since LSC typically focus on inter-object scenarios, we allow the user the option to filter out these method calls from the trace, pre-mining. Note that this will *not* remove method calls between two different objects of the same class.

Ignore set. Based on previous knowledge about the system, the user may consider some of the method calls as not interesting. We thus allow the user to specify a set of method signatures to be ignored. The mining algorithm will ignore the specified methods.

Consider equivalent sets. The user may have previous knowledge which hints that two or more methods correspond to the same abstract concept. We thus allow to specify these using (necessarily disjoint) sets of events. The miner will not distinguish between methods belonging to the same set.

Equivalent consecutive duplicates. In some cases, consecutive repetitions of the same event may not be interesting to the user. We thus allow to specify a list of methods whose consecutive repetitions, if found, should be abstracted away. For example, if method x is in the list, the miner will not distinguish between the words $x, xx, xxx \ldots$ etc.

Disjoint sets. In some cases, the user may know that two or more methods should not appear in the same LSC. We thus allow to provide this information to the miner and have extended the basic mining algorithm to support this requirement.

Predefined pre-charts. We allow the user to specify that one is only interested in mining LSCs whose pre-charts are included in a specific predefined set or use only a predefined subset of the events alphabet.

Removing logically redundant LSCs. Given any two LSCs $M_1(pre_1, main_1)$, M_2 (pre_2, $main_2$) such that pre_1++$main_1$ = pre_2++$main_2$, the one with shorter pre-chart logically entails the other. Therefore, given such LSCs with equal support and confidence values, we keep the one with the minimal pre-chart length and filter out the rest.

Removing sub LSCs. Given any two LSCs $M_1(pre_1, main_1)$, $M_2(pre_2, main_2)$, we say that M_1 is a sub LSC of M_2 iff pre_1++$main_1$ is a sub-word of pre_2++$main_2$. We allow the user to choose to filter out sub LSCs (given equal support and confidence) and keep the ones with the richer alphabet. Note that if M_1 is a sub LSC of M_2, it is not always true that M_2 logically entails M_1.

Min/max length thresholds. In general, long LSCs convey more information than short ones and are more difficult to identify manually. On the other hand, the longer the LSCs mined, the longer it takes for the mining algorithm to run. We thus allow the user to set minimum and maximum length thresholds. The miner will filter out LSCs which are shorter (longer) than the minimum (maximum) length threshold.

Density. Given a trace containing many repetitions of lock (l) and unlock (u) (abstracting object information in this example), the following LSCs may be found statistically significant: $(l, u), (lu, l), (lul, u)$ etc. Only the first, however, is probably of interest to the user. To distinguish the first from the rest we introduce a notion of density; i.e., the ratio between the number of unique events and the total number of events in the LSC; and allow the user to set a corresponding minimum threshold.

Main-pre ratio. In general, LSCs with shorter pre-chart and longer main-chart are more restrictive and thus more informative. We define the main-pre ratio as the ratio between the length of the main-chart and the length of the pre-chart, and allow the user to set a corresponding minimum threshold.

Finally, the user may choose to sort the output LSCs by their support, confidence, length, or number of participating objects.

8.4 Evaluation

8.4.1 Settings and methodology

We have implemented our ideas and complete framework including the above listed extensions using Visual C#.Net 2.0. In our setting, we used AspectJ to instrument Java programs and create trace files, where each element in a trace is a triplet (caller identifier, callee identifier, method signature). In general, each unique triplet is mapped to a unique event symbol, and the trace is converted to a sequence of symbols which is used as the input to our basic mining algorithm (in subsequent stages of the mining process, when object identifiers are required (e.g., in class-level aggregation), we reuse the pre-mapped triplets).

To demonstrate and evaluate our work in this chapter we use Jeti [89], a popular full featured open source instant messaging application based on the Jabber (XMPP) open standard for Instant Messaging and Presence technology (see [82]). Jeti has an open plug-in architecture and supports many chat features including file transfer, group chat, picture chat (whiteboard group drawing), buddy lists, dynamic presence indicators, etc. Its core contains more than 49K lines of code consisting of about 3400 methods, divided between 511 classes in 62 packages. [2] We recorded interactions between several Jeti clients into 6 separate trace files, each of which approximately 1K events long (about 120 unique methods, 600 unique events). In addition, we recorded a longer trace of approximately 10K events to check the effectiveness of some of the implemented extensions in reducing the required mining time (see next).

[2]Metrics computed using the Eclipse Metrics plug-in [45].

We used the following extensions by default: ignoring intra-object method calls, removing logically redundant LSCs, and removing sub LSCs. We further fine-tuned the results over several mining iterations using different thresholds for min and max length, density, and main-pre ratio.

8.4.2 Results

In general, mining time for a 1K long trace ranged between a few seconds and several minutes on a Pentium IV 3Ghz PC with 2GB memory. Many informative LSCs were mined. A small selection is highlighted below.

First, a mined LSC involving sending of messages when one client starts communicating with another is shown in Figure 8.4 (left). The scenario starts whenever a user uses the roster tree to select a party to communicate with. Then, the roster tree will initiate the chat and set up the chat window. After several resources and identifiers of communicating parties are obtained, eventually, an initial message is sent via the *Backend*/*Connect*/*Output* channel. Note that the more common behavior of messages sent via this channel (i.e., "whenever the *Backend* sends a packet to the *Connect*, eventually the latter sends a packet to the *Output*"), was found in another mined LSC (see Figure 8.4(right)). The miner discovered both LSCs, and as expected, the latter's support value was much higher than that of the former. This demonstrates that our method not only can extract recurring scenarios, but is also able to distinguish more frequently observed scenarios from relatively rare ones.

Second, a scenario involving flashing icons, which occurs whenever a message is received by a chat window that is not in focus, is shown in Figure 8.5. The scenario starts with messages being received and appended to the user interface (i.e., *ChatSplitPane* located within *ChatWindow*). A check is made to the identity of the incoming message and the icon starts flashing. Eventually, when the user clicks on the flashing window, flashing is stopped. We used this scenario to test the user-guided predefined pre-chart extension described earlier, by providing the miner the following series of events ending in flashing icons being started as additional input, and applying it to a long 10K events trace.

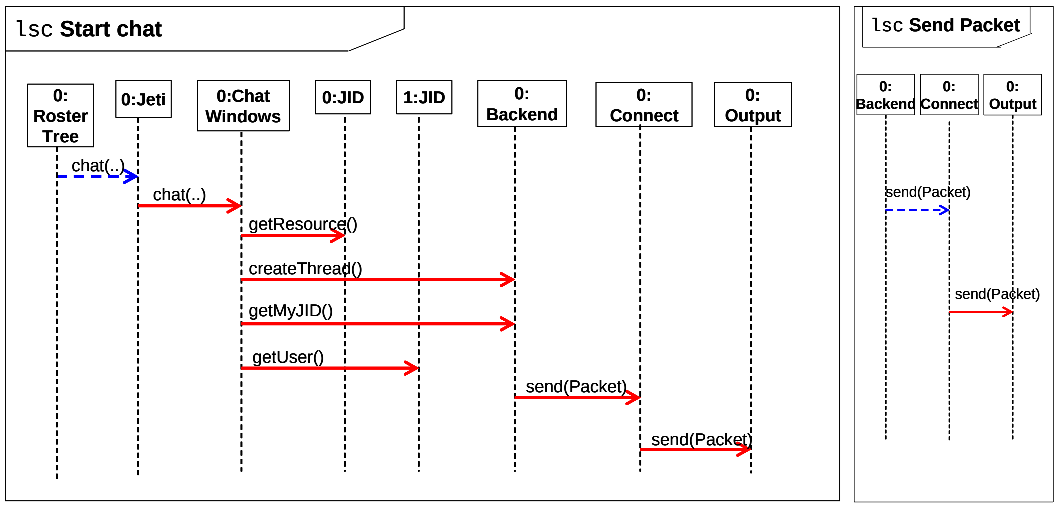


Figure 8.4: Mined LSCs: *Start chat* (left) and *Send packet* (right)

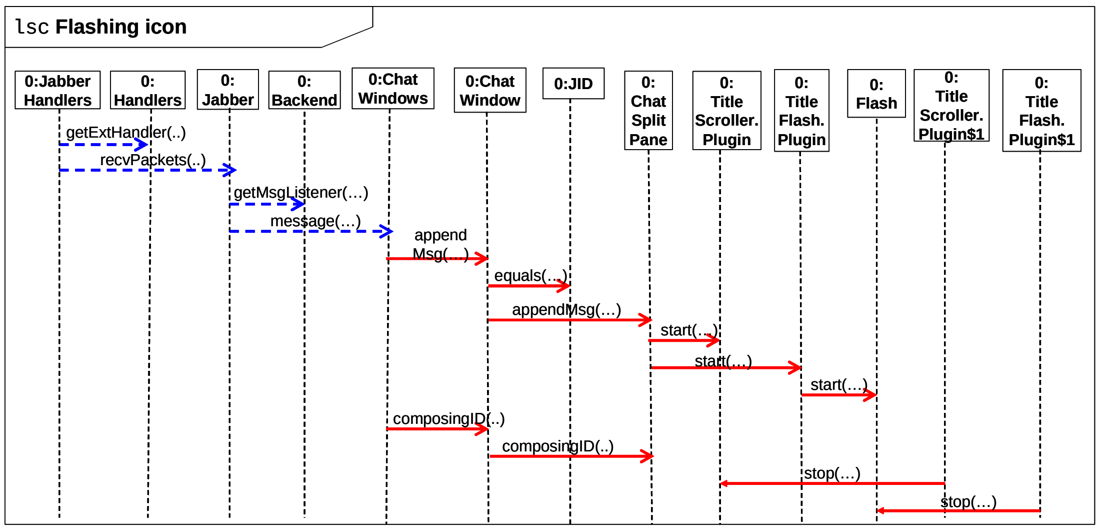


Figure 8.5: Mined LSC: *Flashing icon*

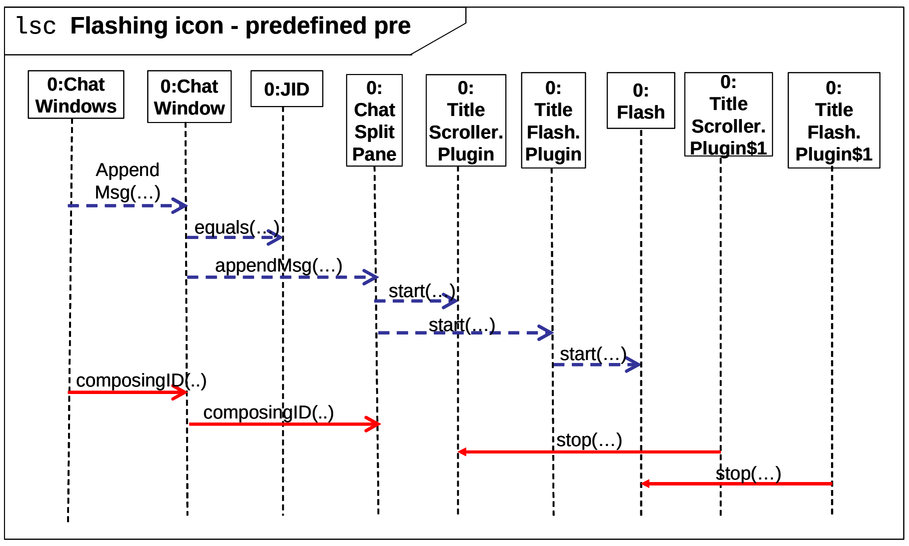


Figure 8.6: Mined LSC: *Flashing icon* (10K long trace)

(Caller, Callee, Method Signature)
1. (ChatWindows;ChatWindow;appendMessage)
2. (ChatWindow;JID;equals)
3. (ChatWindow;ChatSplitPane;appendMessage)
4. (ChatSplitPane;titlescroller.Plugin;start)
5. (ChatSplitPane;titleflash.Plugin;start)
6. (titleflash.Plugin;Flash;start)

While without the additional input, mining this rather long trace took a few hours, using the predefined pre-chart mining was completed within a few minutes. One of the resulting mined LSCs is shown in Figure 8.6. The above illustrates the usefulness of the user-guided predefined pre-chart extension and suggests a methodology: the user may mine relatively short traces to find candidate LSCs of particular interest, and then evaluate these LSCs or related ones on much longer traces at a very low computational cost. Finally, we note that we obtained the long trace by tracing the interaction between a number of communicating entities; while we show here only the class-level LSC, many of its corresponding object-level LSCs, bound to different *ChatWindow*s were 'active' simultaneously, i.e., one had started flashing before another one had stopped. This shows that our method is able to extract common class-level scenarios from complex traces where the different corresponding object-level positive-witnesses interleave and thus overlap.

Third, from traces involving the use of Jeti's group whiteboard, the miner has captured a scenario of drawing a line and sending it to the other chat users (see Figure 8.7). In Jeti, the different graphic elements (*LineMode*, *EllipseMode*, *RectangleMode*, etc.) are all sub-classes of the same abstract class *Mode*. Interestingly, the mining results included additional very similar LSCs corresponding to drawing of ellipses and rectangles. Indeed, the only difference between these LSCs was the participating classes of the first leftmost lifelines. We thus performed a super-class aggregation resulting in the LSC shown in Figure 8.8. Note the abstract class *Mode* referenced on the leftmost lifeline. This mined LSC takes advantage of the semantics of LSC symbolic instances in defining compact and expressive scenarios.

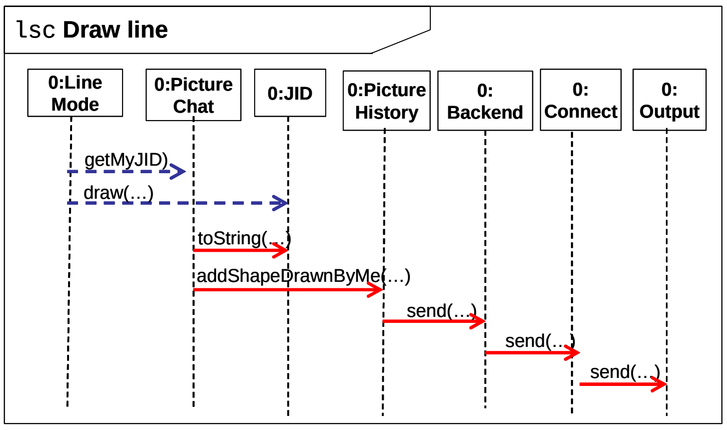


Figure 8.7: Mined LSC: *Draw line*

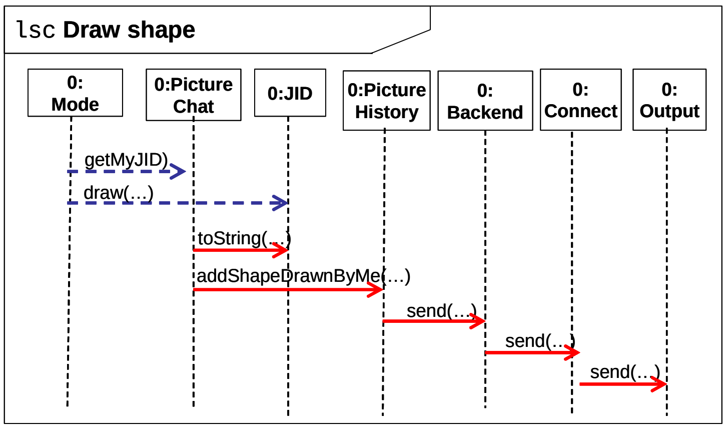


Figure 8.8: Mined LSC: Drawing a general shape (*Mode*)

8.4.3 Presentation and validation

We have implemented a programmatic translation of the mined LSCs (represented in simple textual format) into UML2 Sequence Diagrams, using the Eclipse UML2 APIs [46] and the LSC profile [65]. Thus, we viewed selected results from Jeti visually inside IBM Rational Software Architect (RSA) [80] (see Figure 8.9). In addition to the visual representation itself, which helped a lot in understanding the mined scenarios, we were able to use RSA to edit and manipulate the mined LSCs, group them into use cases, annotate them, print them, etc.

Finally, we used the S2A compiler [70], developed at the Weizmann Institute of Science, to programmatically compile selected LSCs into (monitoring) Scenario Aspects [129]. The generated scenario aspects traced the application while simulating the progress of each of the previously mined scenarios, and thus served as scenario-based

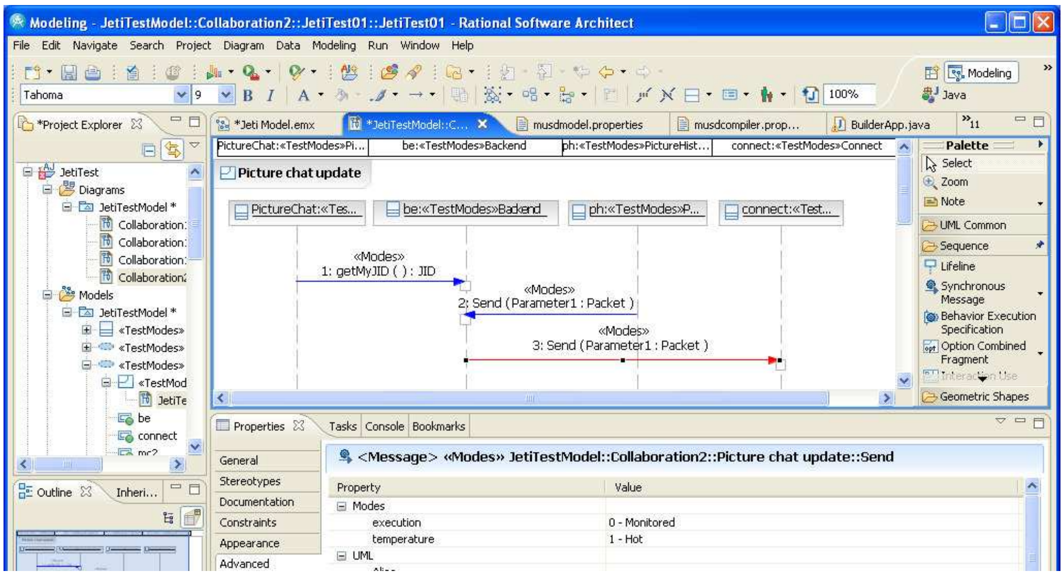

Modeling - JetiTestModel::Collaboration2::JetiTest01::JetiTest01 - Rational Software Architect
Picture chat update
PictureChat:«Tes...
be:«TestModes»Backend
ph:«TestModes»P...
connect:«Test...
«Modes»
1: getMyJID ( ) : JID
«Modes»
2: Send (Parameter1 : Packet)
«Modes»
3: Send (Parameter1 : Packet)
<Message> «Modes» JetiTestModel::Collaboration2::Picture chat update::Send
Property
Value
Modes
execution
0 - Monitored
temperature
1 - Hot
UML
Palette
Lifeline
Synchronous Message

Figure 8.9: Part of the *Picture chat update* mined LSC, shown inside IBM RSA. Note the use of the Modal profile.

```
E: 1180527437140 75: void nu.fw.jeti.jabber.Backend.send(Packet)
B: MUSDAspectJetiTest01[57] lifeline 1 <- nu.fw.jeti.jabber.Backend@2bee2bee
B: MUSDAspectJetiTest01[57] lifeline 0 <- nu.fw.jeti.plugins.drawing.shapes.PictureChat@2bdc2bdc
C: MUSDAspectJetiTest01[57] (1,1,0,0) Cold
E: 1180527437140 76: void nu.fw.jeti.jabber.Backend.send(Packet)
B: MUSDAspectJetiTest01[57] lifeline 2 <- nu.fw.jeti.plugins.drawing.shapes.PictureHistory@76687668
C: MUSDAspectJetiTest01[57] (1,2,1,0) Hot
F: MUSDAspectJetiTest01[57] Violation
```

Figure 8.10: Excerpt from the scenario-based monitor output generated by S2A. A violation detection in Jeti. Note the events that have occurred, the lifelines' bindings, and the 4-tuples representing cut state changes.

tests for Jeti. This allowed us to 'validate' selected mined LSCs during subsequent executions of Jeti. A log file generated by S2A includes a scenario-based trace where completions (occurrences of positive-witnesses) and violations (occurrences of negative-witnesses) are shown.

For example, we compiled the LSC shown in Figure 8.1 discussed previously into a scenario aspect, ran Jeti, and checked that the scenario-based trace produced includes no violations. We then found in Jeti's code the call that corresponds to the first hot method call in the mined LSC, and changed it so that it will not always occur. In other words, we embedded a bug into Jeti, so that although all updates to the chat picture will still be sent to the *Backend*, some of them will not be further sent to the *Connect* object, i.e., some will not be sent to the remote users on the chat. We were then able to view, in the scenario-based trace produced during subsequent executions (see Figure 8.10), the hot violations that have occurred when new packets were sent without sending earlier ones first through the connection.

8.5 Conclusion

In this chapter, we have proposed a novel method to mine a sound and complete set of statistically significant Live Sequence Charts (LSC) from program execution traces. The mining method utilizes a monotonicity property of LSCs positive-witnesses to cut down the search space and thus enable mining LSCs of arbitrarily length scalable and hence feasible for practical use. We have proposed and implemented novel extensions which aim at improving the quality and usefulness of the LSCs presented to the user. The case study on Jeti, a full-featured messaging package, demonstrates the utility of our approach in discovering universal modal scenarios.

LSC mining is a general framework suitable for discovering and presenting universal specifications of interactions between components in reactive systems automatically. That is, the framework is independent of the specific system's implementation and the trace extraction method, as long as the trace includes not only method signatures but also identifiers of caller and callee at the object instance level. In the case study described in this chapter we mine Java programs and use AspectJ for trace extraction. Given appropriate trace extraction mechanisms, LSC mining can be used effectively to mine the behavior of other reactive systems, such as distributed web applications or embedded systems.

The usefulness of LSC mining relies not only on the popularity of sequence diagrams in general in specifying inter-object scenario-based requirements, but also on the additional unique features of LSC with regard to expressive power: the universal/existential must/may modalities as reflected in the pre-chart/main-chart structure and the use of class-level symbolic instance lifelines.

CHAPTER IX

RELATED WORK

In recent years, we have seen a surge within the software engineering research community to adopt dynamic analysis, machine learning and statistical approaches to address the problem of missing, incomplete or outdated specification. In [55], Fox illuminates the use of machine learning to bridge the gap between high level abstractions expressing software engineering problems and low level program behaviors. He points out that some baseline models can be learned automatically to aid in the characterization and monitoring of systems.

These methods, generally termed *specification mining*, can be categorized based on the types of specifications they mined: automata, temporal rules, frequent patterns, sequence diagrams, algebraic expressions, etc. Also, they can be distinguished based on whether static or dynamic analysis is employed.

In this chapter, we first review related work on evaluation frameworks and benchmarks employed in related areas. Next, we review related work in mining various forms of specifications. Since our techniques are based on dynamic analysis, we focus on related work on dynamic analysis. A separate section is dedicated to review work employing static analysis for mining software specifications.

9.1 Evaluation Frameworks and Measures

Several areas such as bug localization, frequent itemset mining and frequent sequential pattern mining, have benefited from good evaluation frameworks (*e.g.*, [78, 152, 3, 5]). Bug localization [152, 30, 21, 110, 168] tries to pinpoint the root source of a program error (*i.e.*, line numbers within a source file) by comparing two groups of execution traces - one correct and another buggy - generated by executing a test suite. An integrated test suite first developed by researchers in Siemens, also known as the Siemens Test Suite [78], has been consistently used in evaluating the performance of error localization tools. Also, an objective performance measure based on the proximity of reported and actual errors has been developed to aid comparison of various techniques [152].

Frequent itemset mining [3] and frequent sequential pattern mining [5] are data mining techniques frequently used to help solve software engineering tasks (*c.f.*, [13, 173, 108, 109]). For these two families of techniques, both real test sets and simulated test generators have been employed in evaluating and comparing results of various proposed techniques. Of interest is the presence of a simulated test generator. A simulated test generator adds a degree of confidence that an assessed technique runs well on a variety of test experiments and not only on a particular experiment under consideration. This is the case as the generator can be easily used to produce a wide variety of test inputs. It also adds flexibility on evaluating the effect of varying a variable of interest while fixing other variables constant.

An evaluation framework is especially important if only a heuristic but not an optimal solution exists or can be feasibly computed for practical purposes. In automaton-based specification mining domain, the problem of finding an optimal solution is often NP-hard or even undecidable (*c.f.*, [7, 57]). For these cases only heuristic solutions exist. Hence, objective evaluation frameworks are needed. Good evaluation frameworks can also shed light to the problems at hand and discover blind-spots in research work so far. This is the motivation of our quality assurance framework QUARK used for assessing the quality of automaton-based specification miners [112].

In the literature of automaton-based specification mining (*e.g*, [34, 7, 150, 170], etc.), often only a qualitative comparison is made to evaluate the goodness of specifications mined. Also, usually only one or two cases are considered. With QUARK, several quantitative measures are given to help users assess and compare the relative degree of goodness of different specification miners objectively. Also, QUARK is a simulation framework; with it one can automatically create simulation models of varying sizes, inject errors of varying probabilities, generate representative set of traces satisfying a specification coverage criterion, and automatically evaluate the goodness of specification mined with respect to the input or the automatically generated model. Also with simulation, different sets of traces from the same model can be produced easily. This is significant as a miner might work well on one set of traces but not on another.

In QUARK, we use three metrics – precision, recall and probability similarity –

as measures for assessing the accuracy of automata generated by specification miners. These metrics have been defined to measure the degree of goodness of other learning tasks. Nimmer *et al.* provide precision- and recall-based quality measures for Daikon [140], which mines value-based invariants usually in the form of algebraic equations. Lyngsø *et al.* provide a similarity measure based on comparison of probability distributions of two Hidden Markov Models [127].

9.2 Mining Automata

Cook and Wolf use three methods to mine automata, namely k-tail, neural network, and a novel Markov-model-based learning [34]. K-tail [15] is a purely algorithmic approach, neural network [38] is a purely statistical approach, while the Markov-model-based learning is a hybrid of algorithmic and statistical approaches. K-tail and Markov model are easily tunable while this is not the case with neural network. A qualitative comparison is made with respect to the quality of the specification mined by the three miners. Their experiment results show that k-tails is the best among the three techniques.

Reiss *et al.* propose several techniques to compact and encode program execution traces [150]. The encoded program traces can be used to aid visualization and understanding of programs. The proposed methods are especially useful when the volume of raw traces generated is very large. Several encodings based on frequency of events, context free grammar and automata are proposed. Automata encoding is produced by employing a novel variant of k-tails algorithm.

Arts *et al.* present a dynamic analysis tool package for a specific programming language (i.e., Erlang) [11]. Their tool package includes trace generation, collection and analysis. Since Erlang programs are often implemented based on state-based design patterns, it might be relatively easier to extract models/specifications from Erlang programs. Arts *et al.* use the mined models for visualization and model checking.

Whaley *et al.* extract object-oriented component interface sequencing constraints in the form of automata [170]. They propose two approaches, static and dynamic, to mine automata. Rather than producing a single automaton, they produce multiple automata. Each automaton corresponds to a sub-behaviour corresponding to a group of method calls implementing the same Java interface or accessing a particular field. They distinguish method calls into two types: state-preserving (side effect free) and state-modifying (with side effect). For the dynamic approach, they employ a simple approach that simply keeps track of the history of the last state-modifying method that was called. Each method corresponds to a unique state. The algorithm will build a graph connecting two methods that are called in sequence. Since only the last method is considered, the method can not detect context information involving two or more methods. The algorithm is likely to produce incorrect specifications if a method follows different protocols for different contexts. Also, specification encompassing methods accessing different fields can not be obtained. The dynamic analysis approach requires performing static analysis on the source code to obtain information on state-preserving and state-modifying methods. For some applications, like analyzing third party Commercials-Off-The-Shelf components, the source code is often not available (*c.f.*, [131]).

Ammons *et al.* employ a machine learning approach called sk-strings [149], to discover automaton-based specifications by analyzing program execution traces[7]. Their technique focuses on mining of specifications which reflect temporal and data dependency relations of a program through traces of its client-Application Programming Interface (API) interaction. The specifications discovered model client-API interaction protocols, which are expressed initially as a probabilistic finite state automaton (PFSA). To reduce the effect of errors in training traces, transitions with a low likelihood of being traversed can later be pruned. After pruning, the probabilities are dropped and an FSA is obtained. Under the assumption that the program being mined must "reveal strong hints of correct protocols" during its execution, Ammons *et al.* demonstrate that correct specifications can be obtained by their technique.

In [8], Ammons *et al.* utilize hierarchical clustering via concept analysis to aid a specification miner user to detect and delete clusters of error traces *en masse*. In our approach, SMArTIC, discussed in Chapter 5, we also propose clustering similar traces and removal of errors. Our clustering technique is not meant for error detection rather, it is an approach for performing divide-and-conquer to a specification mining task. Our filtering technique performs error detection by generation of characteristic rules strongly observed in the traces. Different from their method which heavily depended on user input, our proposed method is automated.

Mariani and Pezze extract automata using their algorithm called k-behavior, which is an extension of k-tails [132]. K-behavior is an incremental algorithm where not all trace samples are present at the start of the algorithm. The algorithm also reduces over-generalization and over-restrictiveness of standard automata miners. The automata mined and a set of value-based invariants mined by Daikon [52] are then used in a behavior capture and test framework for testing the compatibility of components when re-used in a new setting.

Lorenzali, Mariani and Pezze extract an extended form of automata in which the transitions corresponding to method calls are marked with parameters. Their proposed algorithm is called gk-tail, which is an extension of k-tails algorithm [124]. Parameter information on a transition shows the range of values a parameter took in the traces. Daikon is used to infer these range of values for the parameters in the transitions.

Walkinshaw *et al* mine automata by interactive grammar inference. The algorithm is based on active learning where questions are posed back to the user [166]. User inputs are then used to guide the inference process to produce a better mined automaton. However, one concern is there is still a challenge to reduce the number of questions posed to the user. If the number of questions is too high, this might be a barrier to the usability of their approach.

Quante and Koschke mine automata by first obtaining object process graphs from program execution traces. An object process graph is an interprocedural control flow graph belonging to one object [148]. The extracted object process graphs can later be transformed to an automaton via several transformation strategies. With the extracted object process graphs, loops can be easily identified. However, the method requires the availability of the source code of the system under analysis for the construction of the object process graphs.

Our work SMArTIC is a plug-able architecture where existing specification miners can be improved by removing anomalous traces, grouping similar traces into several groups, learning each of the groups separately into an automaton representing a sub-specification, before merging the sub-specifications together into a unified representation. The learner block in the architecture can be replaced with any automaton-based specification miners. We believe that most of the miners mentioned above can benefit from our architecture with minimal changes.

9.3 Mining Frequent Patterns

Agrawal and Srikant propose sequential pattern mining [5] to find frequent patterns in a set of sequences. A pattern can be simply thought of as a series of events. A pattern P is supported by a sequence S if P is a sub-sequence of S. A pattern is frequent if it is supported by a substantial number of sequences. The problem is: given a set of sequences and a support threshold, find a set of frequent patterns.

To remove redundant patterns, closed sequential pattern mining was proposed by Yan *et al.* [174] and later improved by Wang and Han [167]. A pattern P is closed if there does not exist another pattern P', such that P' is a super-sequence of P and both patterns share the same support. The objective of mining closed sequential pattern is to cut the search space of those patterns which are not closed as early as possible during the mining process. This can greatly speed-up the mining process and also produce much fewer patterns.

Mannila *et al.* [128] propose frequent episode mining. An episode is a series of events happening close together (*i.e.*, in a window of pre-defined length). The problem statement is: given a single long sequence, a window-length threshold and a minimum support threshold, find the set of frequent episodes. Casas-Garriga later improves the algorithm by replacing the fixed-window size with a gap constraint between one event to the next in an episode [56].

In mining DNA sequences, Zhang *et al.* introduce the idea of "gap requirement" in mining periodic patterns from sequences [179]. They detect repeated occurrences of patterns within a sequence and across multiple sequences. The pattern definition proposed in [179] does not follow apriori property [61] and hence potentially degrades the efficiency of the mining process. The method only guarantees the mining of a complete set of patterns, all with length less than n, where n is a user defined parameter. Often, the appropriate value of this parameter n is not obvious to the user.

El-Ramly *et al.* mine user-usage scenarios of Graphical User Interface (GUI)-based programs composed of screens – these scenarios are termed as interaction patterns [48].

Given a set of series of screen ids, frequent patterns of user interactions are obtained. Similar to ours, interaction pattern mining takes as an input a set of sequences and discovers patterns occurring repeatedly within sequences. Interaction pattern mining appears to be incomplete, as some frequent patterns will not be generated. The algorithm mines a full-set of patterns before throwing away every pattern that is a sub-sequence of another. This might potentially cause a performance bottleneck as the number of intermediary patterns can be very large.

In Chapter 6, we mine iterative patterns, which is based on the semantics of Message Sequence Charts [81] and Live Sequence Charts, to discover frequent iterative software behavior in a set of traces [67]. The proposed approach is different from any of the above studies. First of all, the semantics of iterative pattern is different than any of the above patterns. Hence, different search space pruning strategies and mining algorithms are needed. Other differences with other related pattern mining studies described above are provided below.

Different from sequential pattern mining, iterative pattern mining captures multiple occurrences of pattern not only *across multiple sequences* but also includes those repeated *within each sequence*. Due to the presence of loops, a pattern of interest might be repeated multiple times in a trace (which is a sequence of events). Hence, it is necessary to consider repeated patterns within a trace as well as patterns that happen across multiple traces.

Different from episode mining, which mines patterns whose constituent events occur close to one another, expressed by either "window size" and gap constraint, iterative pattern mining does not have the notion of "episode". A pattern can have its events occurring far apart from one another. This difference is significant since important program behavioral patterns, for example, lock acquire and release or file open and close (*c.f* [176, 28]), often have their events occur at some arbitrary distance away from one another in a trace. In addition, episode mining only handles one single sequence while iterative pattern mining operates over a set of sequences.

Different from work by Zhang *et al.* [179], we mine a complete set of patterns not limited to patterns less than a particular length. Also, we don't have the notion of "gap requirement" since an important software pattern can be separated by an arbitrary number of unrelated events.

Different from work by El-Ramly *et al.* [48], we employ an early pruning strategy where search space containing non-closed patterns are removed early. Hence, we do not have the problem of having a very large set of intermediary patterns. Our algorithm guarantees generation of a complete set of patterns. Also, we don't have the notion of maximal number of 'insertions' as for many important software patterns, their constituent events can be separated by an arbitrary number of un-related events.

9.4 Mining Temporal Rules

Yang and Evans infer two event temporal rules commonly used in verification tasks (based on a survey by Dwyer *et al.* [41]) from program execution traces [176, 175]. In [175], only rules which are satisfied without any violations in the traces are mined. However, traces are often imperfect due to the presence of bugs or imperfection in the trace collection mechanism. As a result, mining for perfect rules might potentially miss some important rules including those useful for bug detection. In [176], this limitation is addressed by considering a statistical notion of satisfaction rate. Rules with statistics that are above a given threshold are mined. The rules are expressible in Linear Temporal Logic [79].

There is also work on mining episode rules [128] and sequential rules [157] by composing mined frequent episodes and sequential patterns described in the previous section. Given two patterns p and pq where pq is equal to p concatenated with q, a rule $p \rightarrow q$ can be formed.

In Chapter 7, we extend Yang *et al.*'s algorithm in [176] to mine temporal rules of arbitrary lengths. There, *all possible patterns* of length two are checked for statistical significance. Clearly, this algorithm is not scalable when extended to mining multi-event rules. Checking for *all possible patterns of arbitrary lengths* will involve an arbitrary large number of checks. Since the length of a mined pattern is capped by the length of the longest trace in the trace set, the runtime of a simple extension of Yang *et al*'s algorithm is exponential to this maximal trace length. For a trace of length 100, this algorithm will only complete after many centuries! We propose mining a non-redundant set of rules and employ novel search space pruning strategies to prune the search space of non-significant and redundant rules. Much performance benefit is gained and the algorithm runs on real traces of an industrial program.

The rules mined in Chapter 7 have a different semantics with episode and sequential rules and hence require a different mining strategy. A sequential rule $pre \rightarrow post$ states: "whenever a sequence is a super-sequence of *pre* it will also be a super-sequence of *pre* concatenated with *post*". An episode rule $pre \rightarrow post$ states: "whenever a window is a super-sequence of *pre* it will also be a super-sequence of *pre* concatenated with *post*". Mined multi-event temporal rules generalize sequential rules such that for each rule, *multiple* occurrences of the rule's premise and consequent both *within a sequence and across multiple sequences* are considered. Temporal rules generalize episode rules by allowing precedent and consequent events to be separated by an *arbitrary* number of events in a *sequence database*. Also, a set of sequences rather than a single sequence is considered during mining.

Also, we can not simply compose iterative patterns proposed in Chapter 6 to mine temporal rules as they have different semantics and are based on different formalisms. Iterative pattern is based on MSC and LSC while temporal rules is based on commonly used Linear Temporal Logic (LTL) expressions for verification purposes.

In Chapter 5, as a filtering phase of our automaton-based specification miner called $SMArTIC$, we also generate multi-event rules referred to as outlier detection rules. It focuses on a different problem of improving the quality of our automaton-based specification miner. In our past work, the confidence and support values of rules are only approximated – they might not be accurate. The generation method is neither sound nor complete. In this work, we guarantee soundness, completeness and non-redundancy of mined rules.

9.5 Mining Sequence Diagrams

Sequence diagram is part of UML and is commonly used to represent program specification in the industry. Many studies propose different variants of reverse engineering of objects' interactions from program traces and their visualization using sequence diagrams (see, e.g., [43, 20, 76, 88, 134, 156]). All the above study address only concrete, continuous, non-interleaving, and complete object-level interactions. They basically convert a trace into its sequence diagram representation. An extracted diagram represents an example behavior of a system rather than a temporal invariant. Also normal sequence diagram is often not formal enough to be used for verification.

In Chapter 8, we mine a set of statistically significant Live Sequence Charts (LSC). LSC is a formal version of sequence diagram. LSC is an extension of Message Sequence Chart (MSC), a standard of International Telecommunication Union (ITU) [81], by adding universal modality to the chart. Different from the related work mentioned in the previous paragraph, we used aggregations and statistical methods to look for higher level recurring scenario patterns. We look for universal (modal) sequence diagrams, which aim to abstract away the concrete traces and reveal statistically significant recurring scenario-based patterns, at the object-level as well as the class-level, ultimately suggesting some scenario-based system requirements.

9.6 Mining Other Forms of Specifications

Henkel and Diwan present an automatic extraction of high-level component interfaces in the form of algebraic equations [90]. The specifications are mined by repeatedly invoking the program under analysis. Different from previous approaches, Henkel and Diwan do not analyze a fixed set of run-time traces; rather, they dynamically create needed runtime traces. These traces are then used to infer algebraic equations. The mining process is composed of the following steps: extraction of algebraic signatures through reflection, generation of terms, generation of equations, generation of axioms and elimination of redundant axioms. A term corresponds to a series of method calls calling one after another without causing an exception to be raised. A term is generated by first starting with a constructor and then heuristically continuing to grow the sequence of method calls. Methods will be added to the term one by one as long as no exception is raised. An equation is formed by heuristically equating terms with equivalent behavior. Axioms are then formed by replacing sub-terms in equations with free variables. Elimination of redundant axioms is performed by term rewriting using existing axioms. Every axiom with the property that either its left or right side is longer and contains more free variables than the other is used as a rewriting rule. A term which can be unified with the longer side of a rewriting rule will be replaced by the term on the shorter side of the rule. Every time a term is successfully rewritten, existence of another identical axiom is checked. If an identical axiom exists, the repeated axiom will be removed.

Ernst *et al.* discover value-based program invariants occurring at a certain program point by analyzing program execution traces [52]. These value-based invariants are usually in the form of algebraic equations or boolean expressions (*e.g.*, $X > y$, $Z < Y$, etc.). The invariants are inferred using a set of templates. The satisfaction of a template at a certain program point is checked as a program is run. Templates that satisfy a given threshold are then reported as candidate invariants mined from the traces. Ernst *et al.*'s tool called Daikon can also be integrated with tools mining temporal specifications described in the preceding sections (*c.f.*, [132, 124]).

9.7 Static Analysis Approaches

Studies in [6, 73, 170] find specifications describing legal usages of an Application Programming Interface (API). The interface specifications are represented in the form of an automaton. Mined interfaces can be used to detect buggy clients that violate legal uses of an API. The requirement is that the code needs to be defensively written (e.g., with assert statements to indicate situations where illegal uses of API occur). Also, many APIs are of general purpose (*e.g.*, Jakarta Commons Net [10]) with little constraints, while the client is designed specifically for a particular purpose. For these cases, the produced interface will not be able to reveal client-family-specific bugs present in the client interactions with the API.

In [1], Acharya *et al.* instantiate a given generic property to its concrete representation by analyzing the source code of a program under analysis. The generic property is represented in the form of automata with transitions labelled with generic labels, *e.g.*, call, use, check, etc. These generic labels are later replaced by concrete statements which implement the operations specified by the generic label. An off-the-shelf static analyzer with a data flow extension is employed to do the task of retrieving concrete statements corresponding to the generic labels.

In [2], Acharya *et al.* generate traces statically by employing a model checker to generate inter-procedural control-flow-sensitive traces. After an abstraction step, the resultant traces are later fed to an off-the-shelf partial order mining algorithm to produce partial orders of method calls in the form of DAGs. The resultant DAGs are then merged to form an automaton.

Studies in [109, 6, 73] mine specifications from code, and present them in the form of rules. The study in [109] ignores ordering of events and hence mined rules do not describe temporal properties. Past studies on extracting rules expressing temporal properties from code [49, 172] are limited to extract two-event rules. All of the above studies [109, 6, 73] explore only intra-procedural specifications (*i.e.*, specifications involving events happening in the same procedure).

CHAPTER X

FUTURE WORK

This chapter describes possible future work and directions to further expand the thesis. One area of future work will be direct continuations of work done so far on automaton-based, pattern-based, rule-based and LSC-based specification mining. It is of interest to also investigate mining yet other forms of useful and expressive specifications. Merging different specification mining techniques and developing a general framework for different techniques and classes of specifications are also in the list of future work. Investigation on new strategies to clean input trace set is also another interesting future direction.

10.1 Automaton-based Specification Mining

There is still room for improvement to automaton-based specification mining. In particular, precision is not perfect even for SMArTIC (despite major improvements over other state-of-the-art specification miners).

Possible enhancements to existing automaton-based specification miners include:

1. Synergy of static and dynamic analysis (in the spirit of *e.g.*, [51, 59]) to mine specifications.

2. Utilization of both positive and negative examples (*c.f.*, [83]) during the specification mining process.

3. Improvements to the trace clustering process. In SMArTIC, we have implemented a process to group similar traces into clusters (*i.e.*, horizontal splits of the input trace set). Program traces are often long and involve several phases [151]. Recently, semantic clustering has been proposed to split the traces vertically into phases based on comments or annotations in the source code [100]. Merging the idea in SMArTIC and semantic clustering together, it is interesting to investigate splitting the traces not only horizontally (*i.e.*, splitting traces into separate clusters) but also vertically (*i.e.*, splitting traces into phases). This will likely further

improve the benefit of the divide and conquer approach in learning specifications.

4. Mining more expressive specifications. Perfect learning of context free grammar (CFG) from positive examples is non-decidable (*c.f.*, [57]). However, there are recent advances in heuristic learning of CFG (*c.f.*, [160]). Efficient and yet reasonably accurate learning of CFG provides the first step towards mining more expressive specifications.

10.2 Pattern-based Specification Mining

By taking as input sequences of tokens from program source code and performing iterative pattern mining, one can reveal repeated code segments (with possible in-between gaps). These patterns express sequential structural clones both across a source file and across multiple source files. This is an improvement over the clone mining strategy proposed in [108]. This improvement is significant especially when applied to legacy COBOL application where the program is often written as one large monolithic function. Furthermore, a significant proportion of legacy code for business and finance applications is written in COBOL.

It will also be of interests to further improve the efficiency of iterative pattern mining algorithm. Longer traces and a lower level of abstraction (*i.e.*, where events correspond to statement numbers rather than method names) can then be considered.

Another interesting direction is to utilize mined patterns for classification of software behavior. A pattern mined can be thought as a feature. If these features are mined from two sets of traces buggy and not, they can be used to classify future behaviors as buggy or not (*c.f.*, [19, 27]).

Another application is to use iterative pattern in other domains. One is in the domain of Bioinformatics where iterative pattern could potentially be a more generic way to detect different forms of repeat families in DNA sequences [136, 145]. Another is in the field of social network analysis, where iterative patterns can potentially be used to detect frequent phrases that one use in an online conversation or online focus group discussions [177].

10.3 Rule-based Specification Mining

To support this thesis, a technique to efficiently mine temporal rules has been proposed. Only non-redundant rules are mined. However, the process proposed only removes syntactic redundancies (*i.e.*, if one rule is a subsequence of another having the same support and confidence). It is also of interest to consider logical redundancy (*i.e.*, if one rule can be inferred by another).

Our study leads us to mine sequential generators. This work has been accepted for publication [117]. Sequential generators are frequent sequential patterns of minimal length. A set of sequential patterns supported by the same set of sequences is said to belong to an equivalence class. Minimal members of this equivalence class are referred to as generators. Composing generators with closed patterns (*i.e.*, maximal members of equivalence classes) potentially produces a set of logically non-redundant rules.

To investigate the link between patterns and non-redundant rules, we have also performed a study on the relationship of various sub-sets of frequent sequential patterns to logically non-redundant sets of sequential rules [121].

The approaches so far focus on data mining strategies employed to efficiently mine temporal rules. As a future work, additional case studies and further applications of mined rules to related software engineering tasks are planned.

Theoretically, it will be of interest to formalize a hierarchy of different rule reduction strategies based on expressiveness and compactness of the resultant reduced rule sets. Expressiveness refers to the degree the reduced set of rules is equivalent to the full-set of rules with respect to a particular application (*e.g.*, for understanding the semantics of data, outlier detection, classification, etc.). A more expressive rule set will tend to be less compact. The less compact a rule set is, the more inefficient its mining algorithm is likely to be. A theoretical study relating levels of expressiveness to compactness will be interesting. Design of algorithms to extract rule sets of different compactness and expressiveness levels will also be interesting.

So far only response rules mentioned in Dwyer *et al.*'s survey of useful temporal properties for verification [41] are mined. It will also be interesting to mine other families of rules in Dwyer *et al.*'s survey.

It will be of interest to further improve the efficiency of temporal rule mining algorithm. Longer traces and a lower level of abstraction (*i.e.*, where events correspond to statement numbers rather than method names) can then be considered.

To aid users in analyzing a set of mined rules, a good user interface is certainly a plus. An interactive process to aid mining of rules will be helpful. A process where user can manipulate and validate extracted rules over a set of traces will be useful. A query language (*c.f.*, [85]) can also be developed to evaluate or extract interesting subsets of mined rules.

10.4 LSC-based Specification Mining

We consider a number of directions for future work in LSC mining. First, we consider further improvements to the efficiency of the mining algorithm based on additional properties enabling better pruning of search space. In particular, extending research direction in mining a set of closed patterns and generators in data mining domain [106, 167] looks promising.

Second, one may combine LSC mining with Daikon [52] or a similar (dynamic or static) invariants detection tool, to enhance the LSCs mined with hot and cold (i.e., must and may) 'state invariants' (see the semantics of hot and cold conditions in LSC in [66, 67]). In addition, one may combine our dynamic analysis method with static analysis methods for trace generation (see, e.g., [1, 172]) or specification mining (see, e.g., [6, 73]). These would indeed result in more expressive and useful scenario-based specifications which can improve program comprehension further and, for example, can be ultimately translated into richer and more effective scenario-based tests.

Third, our current method is limited to mining of total order LSCs. We thus consider ways to support additional features of sequence diagrams in general, such as explicit partial order and various structural constructs (alternatives, loops, etc.), e.g., detecting a loop and presenting it in the mined LSC using an explicit loop construct.

Finally, we plan to package our prototype implementation into a user friendly tool and perform additional case studies to further evaluate our approach.

10.5 Trace Cleaning

As another possible future work, we plan to investigate advanced techniques to reduce the size of input traces while retaining the quality of the specification mined. One can do so by throwing away non-important or less important events.

In [178], Zaidman *et al.* identify important key classes using a webmining algorithm. In [60], Hamou-Lhadj *et al.* propose a technique to summarize the content of large traces. A similar approach to that in [178, 60] can potentially be employed to identify and remove less important events from the traces.

In [101], Kuhn and Greevy partition a trace into different phases. We are investigating on the possibility of separately mining specifications for each phase of a program. Additionally, we plan to investigate additional pruning strategies to further improve the performance of our algorithm.

CHAPTER XI

CONCLUSION

Documented software specifications are often lacking, imprecise or out-dated. This is inherent in many software development projects due to the short time-to-market requirement, software evolution and high turn-over rate of IT professionals. Automated processes to extract specifications from programs can help to solve or alleviate this specification problem. These automated processes are called specification mining. Mined specifications can be used to reduce maintenance cost by improving program comprehension, and improve reliability of systems by aiding verification tools in detecting bugs.

As a step forward in advancing the frontier of research in software specification mining, we propose the following thesis:

> Expressive software specifications in diversified formats can be extracted with more automation, accuracy and scalability from program execution traces.

To realize the thesis stated above, we have proposed four novel mining tools to improve current state-of-the-art automaton-based, pattern-based, rule-based and sequence-diagram-based specification miners. A novel framework to evaluate the quality of automaton-based specification miners has also been proposed. The work has been presented and/or published in various international conferences [112, 113, 111, 119, 120, 118, 122, 123, 115, 114, 116]. A book chapter summarizing all the above work has also been accepted for publication [116]. All the tools and framework have also been implemented and have been tested on various case studies.

As future research directions, further extensions of completed work on automaton-based, pattern-based, rule-based and sequence-diagram-based specification miners are planned. It is also interesting to further investigate other types of specifications useful for program understanding, verification and other software engineering tasks. During the thesis work, several productive collaborations have been made with researchers in the field of data mining and software modeling.

The author believes the thesis can further push the border of research frontiers in the domain of specification mining in particular and the domains of software engineering, programming languages and data mining in general. It has been a hard, but also a rewarding and interesting task to accomplish!

BIBLIOGRAPHY

[1] Acharya, M., Sharma, T., Xu, J., and Xie, T., "Effective generation of interface robustness properties for static analysis," in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*, 2006.

[2] Acharya, M., Xie, T., Pei, J., and Xu, J., "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proceedings of European Software Engineering Conference/ SIGSOFT Symposium on the Foundations of Software Engineering*, 2007.

[3] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. I., "Fast discovery of association rules," in *Advances in Knowledge Discovery and Data Mining* (U.M. Fayyad, G. Piatetsky-Shapiro, P. S. and Uthurusamy, R., eds.), pp. 307–328, MIT Press, 1996.

[4] Agrawal, R. and Srikant, R., "Fast algorithms for mining association rules," in *Proceedings of International Conference on Very Large Data Bases*, 1994.

[5] Agrawal, R. and Srikant, R., "Mining sequential patterns," in *Proceedings of IEEE International Conference on Data Engineering*, 1995.

[6] Alur, R., Cerny, P., Gupta, G., and Madhusudan, P., "Synthesis of interface specifications for java classes," in *Proceedings of SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[7] Ammons, G., Bodik, R., and Larus, J. R., "Mining specification," in *Proceedings of SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[8] Ammons, G., Mandelin, D., Bodik, R., and Larus, J., "Debugging temporal specifications with concept analysis," in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[9] Angluin, D., "Identifying languages from stochastic examples," *Yale University technical report, YALEU/DCS/RR-614, March 1988.*

[10] Apache Software Foundations, "Jakarta Commons Net." http://jakarta.apache.org/commons/net/.

[11] Arts, T. and Fredlund, L., "Trace analysis of Erlang program," in *Proceedings of Erlang Workshop*, 2002.

[12] Banks, J., Carson, J., Nelson, B. L., and Nicol, D. M., *Discrete-Event System Simulation.* Prentice Hall, 2001.

[13] Basit, H. and Jarzabek, S., "Detecting higher-level similarity patterns in programs," in *Proceedings of European Software Engineering Conference/ SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[14] Baskiotis, N., Sebag, M., Gaudel, M.-C., and Gouraud, S., "A machine learning approach for statistical software testing," in *Proceedings of International Joint Conferences on Artificial Intelligence*, 2007.

[15] Biermann, A. and Feldman, J., "On the synthesis of finite-state machines from samples of their behaviour," *IEEE Transactions on Computers*, vol. 21, pp. 591–597, 1972.

[16] Binder, R., *Testing Object-Oriented Systems Models, Patterns, and Tools.* Addison-Wesley, 2000.

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

[18] Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., and Selby, R., "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, pp. 57–94, December 1995.

[19] Bowring, J. F., Rehg, J. M., and Harrold, M., "Active learning for automatic classification of software behavior," in *Proceedings of International Symposium on Software Testing and Analysis*, 2004.

[20] Briand, L., Labiche, Y., and Leduc, J., "Toward the reverse engineering of uml sequence diagrams for distributed java software," *IEEE Transactions on Software Engineering*, vol. 32, pp. 642–663, 2006.

[21] Brun, Y. and Ernst, M., "Finding latent code errors via machine learning over program executions," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 2004.

[22] Bunker, A., Gopalakrishnan, G., and Slind, K., "Live Sequence Charts Applied to Hardware Requirements Specification and Verification: A VCI Bus Interface Model," *Software Tools for Technology Transfer*, vol. 7, no. 4, pp. 341–350, 2005.

[23] Butler, R., "What is formal methods ?." Online reference: http://shemesh.larc.nasa.gov/fm/fm-what.html [6 October 2008].

[24] Canfora, G. and Cimitile, A., "Software maintenance," in *Handbook of Software Engineering and Knowledge Engineering* (Chang, S., ed.), pp. 91–120, World Scientific, 2002.

[25] Capilla, R. and Dueñas, J., "Light-weight product-lines for evolution and maintenance of web sites," in *Proceedings of European Conference On Software Maintenance And Reengineering*, 2003.

[26] Chen, F. and Roşu, G., "MOP: An Efficient and Generic Runtime Verification Framework," in *Proceedings of SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2007.

[27] Cheng, H., Yan, X., Han, J., and Hsu, C.-W., "Discriminative frequent pattern analysis for effective classification," in *Proceedings of IEEE International Conference on Data Engineering*, 2007.

[28] Chin, W.-N., Khoo, S.-C., Qin, S., Popeea, C., and Nguyen, H., "Verifying safety policies with size properties and alias controls," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 2005.

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

[30] Cleve, H. and Zeller, A., "Fault localization: Locating causes of program failures," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 2005.

[31] "COLT: COmputational Learning Theory." Online reference: http://www.learningtheory.org [6 October 2008].

[32] Combes, P., Harel, D., and Kugler, H., "Modeling and Verification of a Telecommunication Application Using Live Sequence Charts and the Play-Engine Tool," in *Proceedings of International Symposium on Automated Technology for Verification and Analysis*, 2005.

[33] Cook, J. E. and Wolf, A. L., "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, pp. 215–249, July 1998.

[34] Cook, J. and Wolf, A., "Automating process discovery through event-data analysis," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 1995.

[35] Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C. S., Robby, and Zheng, H., "Bandera: Extracting finite-state models from java source code," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2000.

[36] Cormen, T., Leiserson, C., Rivest, R., and C.Stein, *Introduction to Algorithms.* MIT Press, 2001.

[37] Damm, W. and Harel, D., "LSCs: Breathing Life into Message Sequence Charts," *J. on Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001.

[38] Das, S. and Mozer, M., "A unified gradient-descent/clustering architecture for finite state machine induction," in *Proceedings of Annual Conference on Advances in Neural Information Processing Systems*, pp. 19–26, 1993.

[39] de la Higuera, C. and Thollard, F., "Identification in the limit with probability one of stochastic deterministic finite automata," in *Proceedings of International Colloquium of Grammatical Inference and Applications*, 2000.

[40] Deelstra, S., Sinnema, M., and Bosch, J., "Experiences in software product families: Problems and issues during product derivation," in *Proceedings of Software Product Line Conference*, 2004.

[41] Dwyer, M., Avrunin, G., and Corbett, J., "Patterns in property specifications for finite-state verification," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 1999.

[42] Dwyer, M., Person, S., and Elbaum, S., "Controlling factors in evaluating path-sensitive error detection techniques," in *Proceedings of SIGSOFT Symposium on the Foundations on Software Engineering*, 2006.

[43] Eclipse, "Eclipse Test and Performance Tools Platform." Available from: http://www.eclipse.org/tptp/ [6 October 2008].

[44] Eclipse, "The AspectJ Project." Available from: eclipse.org/aspectj [6 October 2008].

[45] "Eclipse Metrics plug-in.." Available from: http://metrics.sourceforge.net/ [6 October 2008].

[46] "Eclipse UML2." Available from: http://wiki.eclipse.org/index.php/MDT-UML2 [6 October 2008].

[47] Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., and Campenhout, D. V., "Reasoning with temporal logic on truncated paths," in *Proceedings of International Conference on Computer Aided Verification*, 2003.

[48] El-Ramly, M., Stroulia, E., and Sorenson, P., "Interaction-pattern mining: Extracting usage scenarios from run-time behavior traces," in *Proceedings of SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2002.

[49] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B., "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proceedings of Symposium on Operating Systems Principles*, 2001.

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

[51] Ernst, M. D., "Static and dynamic analysis: Synergy and duality," in *Proceedings of International Workshop on Dynamic Analysis*, 2003.

[52] Ernst, M., Cockrell, J., Griswold, W., and Notkin, D., "Dynamically discovering likely program invariants to support program evolution," *IEEE Transaction on Software Engineering*, vol. 27, pp. 99–123, February 2001.

[53] Fjeldstad, R. and Hamlen, W., "Application program maintenance-report to our respondents," in *Tutorial on Software Maintenance* (Parikh, G. and Zvegintzov, N., eds.), pp. 13–27, IEEE Computer Society Press, 1983.

[54] Foss, A. and Zaiane, O., "A parameterless method for efficiently discovering clusters of arbitrary shape in large datasets," in *Proceedings of IEEE International Conference on Data Mining*, 2002.

[55] Fox, A., "Addressing software dependability with statistical and machine learning techniques," in *Proceedings of ACM/IEEE International Conference of Software Engineering*, 2005. Invited Talk.

[56] Garriga, G., "Discovering unbounded episodes in sequential data," in *Proceedings of European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2003.

[57] Gold, E. M., "Language identification in the limit," in *Information and Control*, vol. 10, pp. 447–474, 1967.

[58] G.Salton, *Automatic Information Organization and Retrieval.* McGraw-Hill, 1968.

[59] Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., and Rajamani, S., "SYNERGY: A new algorithm for property checking," in *Proceedings of SIGSOFT Symposium on the Foundations on Software Engineering*, 2006.

[60] Hamou-Lhadj, A. and Lethbridge, T., "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *Proceedings of IEEE International Conference on Program Comprehension*, 2006.

[61] Han, J. and Kamber, M., *Data Mining Concepts and Techniques, 2nd Edition.* Morgan Kaufmann, 2006.

[62] Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., and Hsu, M.-C., "Freespan: Frequent pattern-projected sequential pattern mining," in *Proceedings of SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000.

[63] Hand, D., Mannila, H., and Smyth, P., *Principles of Data Mining.* MIT Press, 2001.

[64] Harel, D., Kugler, H., and Pnueli, A., *Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements*, pp. 309–324. Springer, 2005.

[65] Harel, D. and Maoz, S., "Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams," in *Proceedings of International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, 2006.

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

[67] Harel, D. and Marelly, R., *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Springer, 2003.

[68] Harel, D. and Pnueli, A., "On the development of reactive systems," in *Logics and Models of Concurrent Systems* (Apt, K. R., ed.), vol. F-13 of *NATO ASI Series*, (New York), pp. 477–498, 1985.

[69] Harel, D., "From play-in scenarios to code: An achievable dream.," *IEEE Computer*, vol. 34, no. 1, pp. 53–60, 2001.

[70] Harel, D., Kleinbort, A., and Maoz, S., "S2A: A compiler for multi-modal UML sequence diagrams," in *Proceedings of International Conference on Foundations of Software Engineering*, 2007.

[71] Hartigan, J. and Wong, M., "A K-Means clustering algorithm," *Applied Statistics*, vol. 28, no. 1, pp. 100–108, 1979.

[72] Haugen, Ø., Husa, K. E., Runde, R. K., and Stølen, K., "STAIRS towards Formal Design with Sequence Diagrams," *Software and System Modeling (SoSyM)*, vol. 4, no. 4, pp. 355–367, 2005.

[73] Henzinger, T., Jhala, R., and Majumdar, R., "Permissive interfaces," in *Proceedings of European Software Engineering Conference/ SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[74] Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D., "Prism: A tool for automatic verification of probabilistic systems," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[75] Hopcroft, J., Motwani, R., and Ullman, J., *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[76] Hosking, J. G., "Visualisation of object oriented program execution," in *Proceedings IEEE Symposium on Visual Languages*, 1996.

[77] "The Java hotspot performance engine architecture." Available from: http://java.sun.com/products/hotspot/whitepaper.html [6 October 2008].

[78] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 1994.

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

[80] IBM, "IBM Rational Software Architect." Available from: http://www-01.ibm.com/software/rational/ [6 October 2008].

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

[82] Jabber Software Foundation, "Jabber." Available from: http://www.jabber.org/ [6 October 2008].

[83] Jain, S. and Kimber, E. B., "On learning languages from positive data and a limited number of short counterexamples," in *Proceedings of Annnual Conference on Learning Theory*, 2006.

[84] Jain, S., Osherson, D., Royer, J., and Sharma, A., *Systems That Learn.* MIT Press, 1999.

[85] Jarzabek, S., "PQL: A language for specifying abstract program views," in *Proceedings of European Software Engineering Conference*, 1995.

[86] "JBoss-AOP." Available from: http://www.jboss.org/jbossaop [6 October 2008].

[87] "JBoss application server." Available from: http://www.jboss.org/jbossas/downloads/ [6 October 2008].

[88] Jerding, D. F., Stasko, J. T., and Ball, T., "Visualizing interactions in program executions," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 1997.

[89] "Jeti. Version 0.7.6." Available from: http://jeti.sourceforge.net/ [October 2006].

[90] J.Henkel and Diwan, A., "Discovering algebraic specifications from java classes," in *Proceedings of European Conference of Object Oriented Programming*, 2003.

[91] "JRat the Java Runtime Analysis Toolkit." Available from: http://jrat.sourceforge.net/ [6 October 2008].

[92] Kaufman, L. and Rousseeuw, P., *Clustering by means of medoids*, pp. 405–416. Elsevier, 1987.

[93] Kaufman, L. and Rousseeuw, P., *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley & Sons, 1990.

[94] Kearns, M., Mansour, Y., Ron, D., Rubinfeld, R., Schapire, R., and Sellie, L., "On the learnability of discrete distributions," in *Proceedings of ACM Symposium on Theory of Computing*, 1994.

[95] Keogh, E., Lonardi, S., and Ratanamahatana, C., "Towards parameter-free data mining," *Proceedings of SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004.

[96] Klose, J., Toben, T., Westphal, B., and Wittke, H., "Check it out: On the efficient formal verification of Live Sequence Charts," in *Proceedings of International Conference on Computer Aided Verification*, 2006.

[97] Kohavi, R., Brodley, C., Frasca, B., Mason, L., and Zheng, Z., "KDD-Cup 2000 organizers' report: Peeling the onion," *SIGKDD Explorations*, vol. 2, pp. 86–98, 2000.

[98] Krüger, I., "Capturing overlapping, triggered, and preemptive collaborations using mscs.," in *Proceedings of International Conference on Foundations of Software Engineering*, 2003.

[99] Kugler, H., Harel, D., Pnueli, A., Lu, Y., and Bontemps, Y., "Temporal logic for scenario-based specifications," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[100] Kuhn, A., Ducasse, S., and Girba, T., "Enriching reverse engineering with semantic clustering," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2005.

[101] Kuhn, A. and Greevy, O., "Exploiting analogy between traces and signal processing," in *Proceedings of IEEE International Conference on Software Maintenance*, 2006.

[102] Larus, J. and Schnarr, E., "EEL: Machine-independent executable editing," in *Proceedings of SIGPLAN Conference on Programming Language, Design and Implementation*, 1995.

[103] Law, A. M. and Kelton, W. D., *Simulation Modeling and Analysis.* McGraw-Hill, 2000.

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

[105] Lettrari, M. and Klose, J., "Scenario-Based Monitoring and Testing of Real-Time UML Models," in *Proceedings of International Conference on the Unified Modeling Language*, 2001.

[106] Li, J., Li, H., Wong, L., Pei, J., and Dong, G., "Minimum description length principle: Generators are preferable to closed patterns," in *AAAI Conference on Artificial Intelligence*, 2006.

[107] Li, Z., Lu, S., Myagmar, S., and Zhou, Y., "CP-miner: A tool for finding copy-paste and related bugs in operating system code," in *Proceedings of Symposium on Operating System Design and Implementation*, 2004.

[108] Li, Z., Lu, S., Myagmar, S., and Zhou, Y., "CP-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[109] Li, Z. and Zhou, Y., "PR–miner: Automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[110] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P., "SOBER: Statistical model-based bug localization," in *Proceedings of European Software Engineering Conference/ SIGSOFT Symposium on the Foundations on Software Engineering*, 2005.

[111] Lo, D., "A sound and complete specification miner," in *SIGPLAN Conference on Programming Language Design and Implementation Student Research Competition (2^{nd} position) – http://www.acm.org/src/winners.html*, 2007.

[112] Lo, D. and Khoo, S.-C., "QUARK: Empirical assessment of automaton-based specification miners," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2006.

[113] Lo, D. and Khoo, S.-C., "SMArTIC: Towards building an accurate, robust and scalable specification miner," in *Proceedings of SIGSOFT Symposium on the Foundations on Software Engineering*, 2006.

[114] Lo, D. and Khoo, S.-C., "Model checking in the absence of code, model and properties," in *Asian Symposium on Programming Languages and Systems (poster presentation)*, 2007.

[115] Lo, D. and Khoo, S.-C., "Software specification discovery: A new data mining approach," in *Proceedings of the National Science Foundation (NSF) Symposium on Next Generation Data Mining and Cyber-Enabled Discovery for Innovation (online)*, 2007.

[116] Lo, D. and Khoo, S.-C., "Mining software specifications," in *Encyclopedia of Data Warehousing and Mining, 2nd Edition (Volume 3)* (Wang, J., ed.), IGI, 2008.

[117] Lo, D., Khoo, S.-C., and Li, J., "Mining and ranking generators of sequential patterns," in *Proceedings of SIAM International Conference on Data Mining*, 2008.

[118] Lo, D., Khoo, S.-C., and Liu, C., "Efficient mining of iterative patterns for software specification discovery," *Proceedings of SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2007.

[119] Lo, D., Khoo, S.-C., and Liu, C., "Mining temporal rules from program execution traces," in *Proceedings of International Workshop on Program Comprehension through Dynamic Analysis*, 2007.

[120] Lo, D., Khoo, S.-C., and Liu, C., "Efficient mining of recurrent rules from a sequence database," in *Proceedings of International Conference on Database Systems for Advanced Applications*, 2008.

[121] Lo, D., Khoo, S.-C., and Wong, L., "Theory and algorithm for mining non-redundant sequential rules," in *Draft Paper*, 2008.

[122] Lo, D., Maoz, S., and Khoo, S.-C., "Mining modal scenario-based specifications from execution traces of reactive systems," in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*, 2007.

[123] Lo, D., Maoz, S., and Khoo, S.-C., "Mining modal scenarios from execution traces," *Companion to the Proceedings of SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2007.

[124] Lorenzoli, D., Mariani, L., and Pezzè, M., "Inferring state-based behavior models," in *Proceedings of International Workshop on Dynamic Analysis*, 2006.

[125] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, P. G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of SIGPLAN Conference on Programming Language, Design and Implementation*, 2005.

[126] Lyngsø, R., Pedersen, C., and Nielsen, H., "Measures of hidden Markov model," *BRICS Report Series*, 1999.

[127] Lyngsø, R., Pedersen, C., and Nielsen, H., "Metrics and similarity measures for hidden Markov models," in *Proceedings of International Conference Intelligent System for Molecular Biology*, 1999.

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

[129] Maoz, S. and Harel, D., "From multi-modal scenarios to code: Compiling LSCs into AspectJ," in *Proceedings of SIGSOFT Symposium on Foundations of Software Engineering*, 2006.

[130] Marelly, R., Harel, D., and Kugler, H., "Multiple Instances and Symbolic Variables in Executable Sequence Charts," in *Companion to the Proceedings of SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2002.

[131] Mariani, L., Papagiannakis, S., and Pezzè, M., "Compatibility and regression testing of COTS-component-based software," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 2007.

[132] Mariani, L. and Pezzè, M., "Behavior capture and test: Automated analysis for component integration," in *Proceedings of IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[133] Marin, M., Moonen, L., and Deursen, A., "A common framework for aspect mining based on crosscutting concern sorts," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2006.

[134] McGavin, M., Wright, T., and Marshall, S., "Visualisations of execution traces (VET): An interactive plugin-based visualisation tool," in *Proceedings 7th Australasian User Interface Conference*, pp. 153–160, Australian Computer Society, Inc., 2006.

[135] McManus, J., "A stakeholder perspective within software engineering projects," in *Proceedings of IEEE International Engineering Management Conference*, 2004.

[136] Meng, S.-W., Zhang, Z., and Li, J., "Twelve c2h2 zinc finger genes on human chromesone 19 can be each translated into the same type of protein after frameshifts," *Bioinformatics*, vol. 20, no. 1, pp. 1–4, 2004.

[137] Nethercote, N. and Seward, J., "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[138] Nevill-Manning, C., Witten, I., and Maulsby, D., "Compression by induction of hierarchical grammars," in *Proceedings of Data Compression Conference*, 1994.

[139] Ngo, M. and Tan, H., "Detecting large number of infeasible paths through recognizing their patterns," in *Proceedings of European Software Engineering Conference/ SIGSOFT Symposium on the Foundations of Software Engineering*, 2007.

[140] Nimmer, J. W. and Ernst, M. D., "Automatic generation of program specifications," in *Proceedings of International Symposium on Software Testing and Analysis*, 2002.

[141] Object Management Group, "The Unified Modeling Language." Available from: http://www.omg.org [6 October 2008].

[142] Olender, K. and Osterweil, L., "Cecil: A sequencing constraint language for automatic static analysis generation," *IEEE Transactions on Software Engineering*, vol. 16, pp. 268–280, 1990.

[143] Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C., "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *Proceedings of IEEE International Conference on Data Engineering*, 2001.

[144] Pozgaz, Z., Sertic, H., and Boban, M., "Strategies for successful software development project preparation," in *Proceedings of International Conference on Information Technology Interfaces*, 2004.

[145] Price, A., Jones, N., and Pevzner, P., "De novo identification of repeat families in large genomes," *Bioinformatics*, vol. 21, pp. 351–358, 2005.

[146] "Programming language." Online reference: http://itmanagement.webopedia.com/TERM/P/programming_language.html.

[147] "Programming language." Online reference: http://en.wikipedia.org/wiki/Programming_language [6 October 2008].

[148] Quante, J. and Koschke, R., "Dynamic protocol recovery," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2007.

[149] Raman, A. V. and Patrick, J. D., "The sk-strings method for inferring PFSA," in *Proceedings of Workshop on Automata Induction, Grammatical Inference and Language Acquisition*, 1997.

[150] Reiss, S. P. and Renieris, M., "Encoding program executions," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 2001.

[151] Reiss, S., "Dynamic detection and visualization of software phases," in *Proceedings of International Workshop on Dynamic Analysis*, 2005.

[152] Renieris, M. and Reiss, S. P., "Fault localization with nearest neighbor queries," in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*, 2003.

[153] Roychoudhury, A., Goel, A., and Sengupta, B., "Symbolic message sequence charts," in *Proceedings of European Software Engineering Conference/ SIGSOFT Symposium on the Foundations of Software Engineering*, 2007.

[154] Safonov, V., "Aspect.Net." Available from: http://www.academicresourcecenter.net/curriculum/pfv.aspx?ID=6334 [6 October 2008].

[155] Silberschatz, A., Galvin, P., and Gagne, G., *Operating System Concepts*. Wiley, 2003.

[156] Sousa, F., Mendonca, N., Uchitel, S., and Kramer, J., "Detecting implied scenarios from execution traces," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2007.

[157] Spiliopoulou, M., "Managing interesting rules in sequence mining," in *Proceedings of European Conference on Principles and Practice of Knowledge Discovery in Databases*, 1999.

[158] Standish, T., "An essay on software reuse," *IEEE Transactions on Software Engineering*, vol. 5, no. 10, pp. 494–497, 1984.

[159] Standish Group, "The CHAOS report," 1994.

[160] Starkie, B., Coste, F., and Zaanen, M.-V., "The omphalos context-free grammar learning competition," in *Proceedings of International Colloquium on Grammatical Inference*, 2004.

[161] Steel, C., Nagappan, R., and Lai, R., *Core Security Patterns.* Sun Microsystem, 2006.

[162] Suhendra, V., Mitra, T., Roychoudhury, A., and Chen, T., "Efficient detectiono and exploitation of infeasible paths for software timing analysis," in *Proceedings of Design Automation Conference*, 2006.

[163] Sun Microsystems, "Java Transaction API Specification." Online Reference: http://java.sun.com/products/jta/ [6 October 2008].

[164] Tompa, M., "Lecture notes on biological sequence analysis," *Technical Report 2000-06-01 University of Washington*, 2000.

[165] Uchitel, S., Kramer, J., and Magee, J., "Detecting implied scenarios in message sequence chart specifications.," in *Proceedings of SIGSOFT Symposium on Foundations of Software Engineering*, 2001.

[166] Walkinshaw, N., Bogdanov, K., Holcombe, M., and Salahuddin, S., "Reverse engineering state machines by interactive grammar inference," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2007.

[167] Wang, J. and Han, J., "BIDE: Efficient mining of frequent closed sequences," in *Proceedings of IEEE International Conference on Data Engineering*, 2004.

[168] Wang, T. and Roychoudhury, A., "Automated path generation for software fault localization," in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering*, 2005.

[169] "Theme: Empirically assessing reverse engineering techniques and tools," in *IEEE Working Conference on Reverse Engineering*, 2006.

[170] Whaley, J., Martin, M., and Lam, M., "Automatic extraction of object oriented component interfaces," in *Proceedings of International Symposium on Software Testing and Analysis*, 2002.

[171] Wilson, R. and Lam, M., "Efficient context-sensitive pointer analyis for c programs," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[172] W.Weimer and G.Necula, "Mining temporal specifications for error detection," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[173] Xie, T. and Pei, J., "MAPO: Mining API usages from open source repositories," in *Proceedings of International Workshop on Mining Software Repositories*, 2006.

[174] Yan, X., Han, J., and Afhar, R., "CloSpan: Mining closed sequential patterns in large datasets," in *Proceedings of SIAM International Conference on Data Mining*, 2003.

[175] Yang, J. and Evans, D., "Automatically inferring temporal properties for program evolution," in *Proceedings of International Symposium on Software Reliability Engineering*, 2004.

[176] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and M.Das, "Perracotta: Mining temporal API rules from imperfect traces," in *Proceedings of ACM/IEEE International Conference on Software Engineering*, 2006.

[177] Yasui, N., Llorà, X., Goldberg, D., Washida, Y., and Tamura, H., "Delineating topic and discussant transitions in online collaborative environments," in *Proceedings of International Conference on Enterprise Information Systems*, 2007.

[178] Zaidman, A., Calders, T., Demeyer, S., and Paredaens, J., "Applying webmining techniques to execution traces to support the program comprehension process," in *Proceedings of the European Conference on Software Maintenance and Reengineering*, 2005.

[179] Zhang, M., Kao, B., Cheung, D., and Yip, K., "Mining periodic patterns with gap requirement from sequences," in *Proceedings of SIGMOD International Conference on Management of Data*, 2005.

[180] Zou, Y., Lau, T., Kontogiannis, K., Tong, T., and McKegney, R., "Model-driven business process recovery," in *Proceedings of IEEE Working Conference on Reverse Engineering*, 2004.

APPENDIX: GLOSSARY

Automata Theory A study of properties, semantics and structure of abstract computing devices, or ”machines” [75].

Automaton A labeled transition system with start and end nodes describing a language. A path from the start to an end node corresponds to a sentence in the language.

Data Mining A study for automated process of extracting knowledge and information from large amount of data of various forms: web access data, biological data, gene sequence, sequel databases, temporal databases, etc. Its sub-domains include: pattern mining, classification, clustering, etc. For references see [61, 63].

Episode Mining A process of finding episodes (series of events occuring close to one-another) that are repeated a significant number of times in a single sequences. The first paper on episode mining was by Manilla *et al.* in [128]. Many other work on episode mining has been proposed since then e.g., [56].

Formal Methods A study of mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase ‘mathematically rigorous’ means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductive processes in that logic (i.e., each step follows from a rule of inference and hence can be checked by a mechanical process) [23].

Learning Theory A study of generalizations of past observed behavior to create formal models or hypotheses. This includes studies on methods, theoretical bounds and limits of learning an automata from samples of its behavior. The grand goal is to clarify human learning process where one learns or generalizes about one’s environment [84] or make predictions of the future [31].

Linear Temporal Logic Formalism commonly used to describe temporal requirements precisely. There are a few basic operations given with symbols G, X, F, U, W, R corresponding to English language terms ‘Globally’, ‘neXt’, ‘Finally’, ‘Until’, ‘Weak-until’ and ‘Release’.

Live Sequence Charts A formal version of UML sequence diagram. It is composed of a pre- and main- chart. The pre-chart describes a condition which if satisfied entails that the behavior described in the main-chart will occur.

Program Comprehension A process of understanding the behavior of a piece of software.

Program Instrumentation Simply put, it is a process of inserting ‘print’ statements to a program such that by running the instrumented program, a trace file reflecting the behavior of the program is produced.

Program Testing A process to detect bugs and provide a measure of assurance that a piece of software is correct by running a set of test cases.

Program Trace A series of events where each event can correspond to a statement that is being executed, a function that is being called, etc., depending on the abstraction level considered.

Program Verification A process to ensure that a piece of software is always correct no matter what input is given with respect to some properties, e.g., whenever a resource is locked for usage, it is eventually released.

Programming Languages A study of structures and semantics of languages (of vocabulary and grammatical rules) used to control the behavior of a machine, in particular, a computer [147, 146].

Sequential Pattern Mining A process of finding patterns (or series of events) that are supported by a significant number of sequences above a user defined minimum support threshold in a sequence database. A pattern is supported by a sequence if it is a subsequence of the latter. The first paper on sequential pattern mining was by Agrawal and Srikant in [5]. Many other work on sequential pattern mining has been proposed since then e.g., [174, 167].

Simulation and Modelling A study of using computer to imitate behavior of real-world systems, facilities or processes based on a set of assumptions on how they works. The goal is to gain insight or estimate behavior or true characteristics of a system under study. For references see [103, 12].

Software Engineering A study of better ways to engineer a software system which include better methods to design, construct, analyze and manage a software system.

Software Maintenance A process of incorporating changes to existing software, e.g., bug fixes, feature additions, etc., while ensuring the resultant software works well.

Software Specification A description on how a piece of software is supposed to behave. It can be described in various formats including class diagrams, sequence diagrams, automata, temporal logic expressions, etc. Some specifications are very precise while others are loosely defined. The earlier is referred to as formal specification.

Specification Mining A process of extracting knowledge and information from programs automatically or semi-automatically. Usually, it refers to the extraction of a program behavioral model from execution traces. However, it can also refer to the extraction of other models and information from either program code or traces.