Mining Software Specifications
Methodologies and Applications
AIMS AND SCOPE
This series aims to capture new developments and applications in data mining and knowledge discovery, while summarizing the computational tools and techniques useful in data analysis. This series encourages the integration of mathematical, statistical, and computational methods and techniques through the publication of a broad range of textbooks, reference works, and handbooks. The inclusion of concrete examples and applications is highly encouraged. The scope of the series includes, but is not limited to, titles in the areas of data mining and knowledge discovery methods and applications, modeling, algorithms, theory and foundations, data and knowledge visualization, data mining systems and tools, and privacy and security issues.

PUBLISHED TITLES
UNDERSTANDING COMPLEX DATASETS: DATA MINING WITH MATRIX DECOMPOSITIONS
David Skillicorn

COMPUTATIONAL METHODS OF FEATURE SELECTION
Huan Liu and Hiroshi Motoda

CONSTRAINED CLUSTERING: ADVANCES IN ALGORITHMS, THEORY, AND APPLICATIONS
Sugato Basu, Ian Davidson, and Kiri L. Wagstaff

KNOWLEDGE DISCOVERY FOR COUNTERTERRORISM AND LAW ENFORCEMENT
David Skillicorn

MULTIMEDIA DATA MINING: A SYSTEMATIC INTRODUCTION TO CONCEPTS AND THEORY
Zhongfei Zhang and Ruofei Zhang

NEXT GENERATION OF DATA MINING
Hillol Kargupta, Jiawei Han, Philip S. Yu, Rajeev Motwani, and Vipin Kumar

DATA MINING FOR DESIGN AND MARKETING
Yukio Ohsawa and Katsutoshi Yada

THE TOP TEN ALGORITHMS IN DATA MINING
Xindong Wu and Vipin Kumar

GEOGRAPHIC DATA MINING AND KNOWLEDGE DISCOVERY, SECOND EDITION
Harvey J. Miller and Jiawei Han

TEXT MINING: CLASSIFICATION, CLUSTERING, AND APPLICATIONS
Ashok N. Srivastava and Mehran Sahami

BIOLOGICAL DATA MINING
Jake Y. Chen and Stefano Lonardi

INFORMATION DISCOVERY ON ELECTRONIC HEALTH RECORDS
Vagelis Hristidis

TEMPORAL DATA MINING
Theophano Mitsa

RELATIONAL DATA CLUSTERING: MODELS, ALGORITHMS, AND APPLICATIONS
Bo Long, Zhongfei Zhang, and Philip S. Yu

KNOWLEDGE DISCOVERY FROM DATA STREAMS
João Gama

STATISTICAL DATA MINING USING SAS APPLICATIONS, SECOND EDITION
George Fernandez

INTRODUCTION TO PRIVACY-PRESERVING DATA PUBLISHING: CONCEPTS AND TECHNIQUES
Benjamin C. M. Fung, Ke Wang, Ada Wai-Chee Fu, and Philip S. Yu

HANDBOOK OF EDUCATIONAL DATA MINING
Cristóbal Romero, Sebastian Ventura, Mykola Pechenizkiy, and Ryan S.J.d. Baker

DATA MINING WITH R: LEARNING WITH CASE STUDIES
Luís Torgo

MINING SOFTWARE SPECIFICATIONS: METHODOLOGIES AND APPLICATIONS
David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu

DATA CLUSTERING IN C++: AN OBJECT-ORIENTED APPROACH
Guojun Gan
## Contents

List of Figures vii
Preface xiii
Contributors xv

1 Specification Mining: A Concise Introduction 1
   David Lo, Siau-Cheng Khoo, Chao Liu, and Jiawei Han

2 Mining Finite-State Automata with Annotations 29
   Leonardo Mariani, Fabrizio Pastore, Mauro Pezzè, and Mauro Santoro

3 Adapting Grammar Inference Techniques to Mine State Machines 59
   Neil Walkinshaw and Kirill Bogdanov

4 Mining API Usage Protocols from Large Method Traces 85
   Michael Pradel and Thomas R. Gross

5 Static API Specification Mining: Exploiting Source Code Model Checking 113
   Mithun Acharya and Tao Xie

6 Static Specification Mining Using Automata-Based Abstractions 159
   Eran Yahav, Sharon Shoham, Stephen Fink, and Marco Pistoia

7 DynaMine: Finding Usage Patterns and Their Violations by Mining Software Repositories 201
   Benjamin Livshits and Thomas Zimmermann

8 Automatic Inference and Effective Application of Temporal Specifications 241
   Jinlin Yang and David Evans
Contents

9 Path-Aware Static Program Analyses for Specification Mining 309
Muralikrishna Ramanathan, Ananth Grama, and Suresh Jagannathan

10 Mining API Usage Specifications via Searching Source Code from the Web 357
Suresh Thummalapenta, Tao Xie, and Madhuri R. Marri

11 Merlin: Specification Inference for Explicit Information Flow Problems 377
Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee

12 Lightweight Mining of Object Usage 411
Andrzej Wasylkowski and Andreas Zeller

Index 433
# List of Figures

1.1 Finite state machine. File access library. ........................................... 5
1.2 Value-based invariants. A program that computes the square of an integer. ................................................... 5
1.3 Frequent pattern of usage. Context menu using JFace. ................... 6
1.4 Temporal rules: Windows (top) and Linux (bottom). .................. 6
1.5 Sequence diagram. Jeti instant messaging application. ............. 7

2.1 The ADABU model of a Vector object. ................................. 31
2.2 Traces of the execution of method `bookFlight`. ................. 35
2.3 The set of merged traces obtained from the traces in Figure 2.2. .................. 38
2.4 An example predicate generated by Daikon. .......................... 39
2.5 The initial EFSA obtained from the traces in Figure 2.3. ......... 39
2.6 The 2-future of the initial state of the EFSA in Figure 2.5. .. 40
2.7 Two 2-equivalent states. ......................................................... 41
2.8 State 4 weakly subsumes state 20. ........................................... 41
2.9 State 4 strongly subsumes state 20. ........................................... 41
2.10 The EFSA produced by gkTail from the interaction traces in Figure 2.2 by using weak subsumption and $k = 2$ as merging criteria. ........................................... 42
2.11 An example FSA extended with a new trace. ......................... 50
2.12 A KLFA model of the behavior of method `bookFlight`. ...... 51

3.1 Model of a simple text editor. .................................................. 61
3.2 Conventional reverse-engineering process. ............................. 63
3.3 Prefix Tree Acceptor for traces from the text editor example, and the merging of states b and f when $k = 2$. .................. 64
3.4 Prefix Tree Acceptor with added negative trace. .................... 69
3.5 Example of score computation for pair $(b, f)$ in the APTA in Figure 3.4. .................................................. 70

4.1 An inferred API usage protocol that shows correct iterator usage in Java. The weights describe how often each transition was observed. ........................................... 87
4.2 Overview of the protocol mining approach. ............................ 89
4.3 Two examples of iterator usage. .............................................. 90
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Method trace from executing ( \text{callM()} ) in Figure 4.3(a) (return events and other details omitted).</td>
<td>90</td>
</tr>
<tr>
<td>4.5</td>
<td>Several calls to objects from ( \text{java.io} ). Dataflow analysis extracts the highlighted calls, which are relevant for showing a typical use of ( \text{FileInputStream} ).</td>
<td>94</td>
</tr>
<tr>
<td>4.6</td>
<td>Skeleton of the aspect used to instrument programs so that they log method calls and returns.</td>
<td>98</td>
</tr>
<tr>
<td>4.7</td>
<td>Inferred protocol that shows a typical usage of a reader.</td>
<td>100</td>
</tr>
<tr>
<td>4.8</td>
<td>Complex protocol showing how to read entries of a ZIP file to store them in a collection.</td>
<td>100</td>
</tr>
<tr>
<td>4.9</td>
<td>Distribution of number of states and number of transitions of the inferred protocols.</td>
<td>101</td>
</tr>
<tr>
<td>4.10</td>
<td>Runtime of the analysis by number of events in the analyzed trace files.</td>
<td>103</td>
</tr>
<tr>
<td>5.1</td>
<td>A simple example illustrating our framework for mining partial orders from static traces.</td>
<td>118</td>
</tr>
<tr>
<td>5.2</td>
<td>(a) A partial order and (b) its transitive reduction.</td>
<td>120</td>
</tr>
<tr>
<td>5.3</td>
<td>Four orders ( R_1 \supset R_2 \supset R_3 \supset R_4 ).</td>
<td>121</td>
</tr>
<tr>
<td>5.4</td>
<td>Our framework for mining usage scenarios and specifications.</td>
<td>124</td>
</tr>
<tr>
<td>5.5</td>
<td>A property FSM where end1 and end2 are final states.</td>
<td>125</td>
</tr>
<tr>
<td>5.6</td>
<td>Trigger FSM that accepts the regular language ( e(a_1 + a_2 + \ldots + a_k)^* ).</td>
<td>125</td>
</tr>
<tr>
<td>5.7</td>
<td>Scenario extraction example.</td>
<td>127</td>
</tr>
<tr>
<td>5.8</td>
<td>The \textbf{PC-Chain} algorithm for scenario extraction.</td>
<td>128</td>
</tr>
<tr>
<td>5.9</td>
<td>X11 client programs used in our evaluation.</td>
<td>129</td>
</tr>
<tr>
<td>5.10</td>
<td>Call frequencies for the selected X11 APIs.</td>
<td>130</td>
</tr>
<tr>
<td>5.11</td>
<td>Trigger used to generate function calls, APIs, and expressions related to ( \text{XCreateGC} ).</td>
<td>131</td>
</tr>
<tr>
<td>5.12</td>
<td>Statistics for the specifications mined by our framework.</td>
<td>131</td>
</tr>
<tr>
<td>5.13</td>
<td>Statistics for the specifications mined around ( \text{XOpenDisplay} ).</td>
<td>131</td>
</tr>
<tr>
<td>5.14</td>
<td>A usage scenario around ( \text{XOpenDisplay} ) API as a partial order. Lower support values produce larger partial orders. Higher support values produce smaller partial orders (specifications). Specifications are shown with dotted lines.</td>
<td>133</td>
</tr>
<tr>
<td>5.15</td>
<td>Definitions and acronyms.</td>
<td>136</td>
</tr>
<tr>
<td>5.16</td>
<td>Code snippet from \textit{routed-0.17-14}.</td>
<td>137</td>
</tr>
<tr>
<td>5.17</td>
<td>Code snippet from \textit{x11perf-R6.9.0}.</td>
<td>138</td>
</tr>
<tr>
<td>5.18</td>
<td>Adapting our static trace generation framework for mining API error-handling specifications.</td>
<td>141</td>
</tr>
<tr>
<td>5.19</td>
<td>The algorithm for mining API error-handling specifications.</td>
<td>142</td>
</tr>
<tr>
<td>5.20</td>
<td>Evaluation results.</td>
<td>146</td>
</tr>
<tr>
<td>5.21</td>
<td>Multiple-API specifications mined for the clean up API ( \text{XFree} ).</td>
<td>150</td>
</tr>
<tr>
<td>6.1</td>
<td>A simple program using APIs of interest.</td>
<td>163</td>
</tr>
</tbody>
</table>
List of Figures

6.2 Partial specification for `SocketChannel`. .................. 164
6.3 Example of concrete histories for an object of type
`SocketChannel` in the example program. .................. 169
6.4 Example of concrete and abstract states: (a) concrete state
with three `SocketChannel` objects with the same instance key
`A1`; (b) abstract state with a single abstract object represent-
ing multiple concrete objects allocated at `A1`. .............. 170
6.5 Example of an abstract state with access path information:
(a) shown pictorially; (b) shown as a set of tuples. ........ 172
6.6 Abstract interpretation with past abstraction (exterior
merge). ........................................... 175
6.7 Abstract interpretation with future abstraction (exterior
merge). ........................................... 176
6.8 (a) Past abstraction step; and (b) future abstraction step,
before quotient construction. ............................. 178
6.9 Past abstraction versus future abstraction. ................. 178
6.10 Summarization: clustered weighted union versus weighted
union. ........................................... 181
6.11 (a) 1-Past abstraction; and (b) 2-Past abstraction. ....... 185
6.12 Mined specification for `java.security.Signature`, obtained with
APFocus/Past/Exterior. ................................ 189
6.13 `Session` API with Past/Exterior merge and (a) Base aliasing;
and (b) APFocus aliasing. ............................... 190
6.14 `KeyAgreement` API with Past/Exterior merge and (a) Base
aliasing; and (b) APFocus aliasing. ......................... 191
6.15 `PrintWriter` API with APFocus aliasing, and past (a) total
merge; and (b) exterior merge. .............................. 192

7.1 Architecture of DynaMine. ................................. 208
7.2 DynaMine pattern selection view. .......................... 214
7.3 DynaMine dynamic results view. ............................ 217
7.4 Example of workspace operations and locking discipline usage
in class `InternalWorkspaceJob` in Eclipse. ............... 225
7.5 Cross-file check-in containing two lines spread across two dif-
ferent files. ........................................ 229
7.6 A `try`–`finally` block involving a matching method pair. ... 230
7.7 Template creation for `(acquireRead, exitRead)` pattern. ... 231

8.1 Overview of our approach. .................................. 245
8.2 A Java implementation of the simplified Producer-Consumer
problem. ............................................ 247
List of Figures

8.3 A trace of running the Producer-Consumer program. Each line corresponds to a single event that represents the entrance or exit of a method. We omit the method’s signature, argument values, and return values to simplify presentation. For example, Enter:Producer.main():[main] indicates that the main thread enters the main method in the Producer class. 248

8.4 Partial order of property templates. Each box shows a property template and its regular expression representation. A pattern A is stricter than another pattern B if L(A) ⊂ L(B), where L(A) means all the strings accepted by A. The eight patterns form a partial order in terms of their strictness. For example, Alternating is the strictest pattern among them. In addition, these patterns have an internal logical relationship as illustrated by the logical ∧ operators among them. For example, a string satisfies the MultiEffect pattern if and only if it satisfies the OneCause and CauseFirst patterns. 251

8.5 Representing the Alternating template in different forms. 252

8.6 The inference algorithm for the Alternating pattern. 253

8.7 Inferring Alternating properties from a hypothetical trace ABCACBDC. 254

8.8 Alternating properties inferred from the trace in Figure 8.3. 255

8.9 Context handling techniques. Context-neutral does not differentiate between lock1.acq/rel and lock2.acq/rel and does not infer any Alternating property. Context-sensitive differentiates between the methods of lock1 and lock2 and infers six Alternating properties. However, only two of the six properties shown in boldface correspond to the property that acquiring a lock should alternate with releasing a lock. Context-slicing slices the original trace by the identity of the lock and produces two substraces. Hence, context-slicing infers lock.acq → lock.rel. 256

8.10 A hypothetical program that forgets to release a lock. 257

8.11 The approximate inference algorithm. 258

8.12 Two scenarios of static call graph. 260

8.13 Alternating chains. 261

8.14 Algorithm for transforming an undirected graph to a DAG. 264

8.15 The chaining algorithm. 266

8.16 Alternating chains for the Producer-Consumer program. 267

8.17 Daisy’s System Architecture. Daisy emulates the hard drive through a Java RandomAccessFile object. 271

8.18 Inferred properties versus the acceptance threshold for pAL. 274

8.19 Alternating chain for the JBoss AS TM module. Grayed events are public methods. 276

8.20 Alternating chain for the public APIs of the JBoss AS TM module. 276
List of Figures

8.21 Use inferred properties in program verification. 280
8.22 The Java code for monitoring Alternating properties. 281
8.23 The Mutex class in Daisy. 284
8.24 The NTFS bug in Windows Vista. 285
8.25 Sample output of Bus Simulator with $n = 2$, $C = 1$, and $T = 2$. 286
8.26 A synchronization bug in one bus implementation. 288
8.27 SSL handshake protocol states. 289
8.28 A server event trace of normal handshake process. 290
8.29 Inferred alternating chains for correct OpenSSL clients. 292
8.30 Inferred alternating chains for non-error faulty OpenSSL clients. 292
8.31 Traces generated with a faulty OpenSSL client. 293
9.1 An example illustrating under- and over-approximation of predicates. 313
9.2 Extract from RI_FKey_check in postGreSQL-8.1.3. 316
9.3 Code fragments of two different call-sites to bind in openssh-4.2p1. 318
9.4 Specification inference via flow analysis. 322
9.5 A program fragment illustrating the need for structural matching of predicates. Syntactic sugar is used to simplify the examples. 324
9.6 A subset of the preconditions that hold prior to the call to procedure $f$ in procedure $g_1$ (a) and procedure $g_2$ (b). 325
9.7 A code fragment taken from openssh-4.2p1. 327
9.8 Illustrative example. 328
9.9 Software architecture. 331
9.10 Example showing the need for FPA evaluation. 332
9.11 Example showing the need for FPB evaluation. 333
9.12 Algorithm for building predicates. 333
9.13 Mining preconditions. 334
9.14 Predicate distributions. 335
9.15 Reduction in number of predicates in apache with increase in confidence threshold. 337
9.16 Distribution of call-sites. 338
9.17 Illustrative example. Rectangles indicate predicates, circles indicate call-sites. Empty rectangles/circles indicate arbitrary predicates/procedure calls. 342
9.18 Algorithm for building predicates. 344
9.19 Generate preconditions. 345
9.20 Comparison with non-profile based inference. 346
9.21 Correctness of different inference schemes. 348
10.1 Snapshots of Google code search and Krugle. 359
List of Figures

10.2 Phases in the life cycle of approaches developed based on searching and mining. .......................... 360
10.3 An example scenario where database contents are modified using the `executeUpdate` method call. ............... 362
10.4 An example scenario where database contents are retrieved using the `executeQuery` method call. ............... 366
10.5 An example exception-flow graph and a generated pattern candidate. .................................................. 367
10.6 Illustrative examples of general algorithm. ....................................................................................... 371
11.1 Simple cross-site scripting example. .................................................................................. 380
11.2 `MERLIN` system architecture. .......................................................................................... 380
11.3 Constraint formulation. Probabilities in italics are parameters of the constraints. ................................. 385
11.4 Factor graph for (11.3). ........................................................................................................... 389
11.5 Generating a factor graph from a propagation graph. ................................................................ 393
11.6 Computing $W(n)$ and $s(n)$. ..................................................................................................... 392
11.7 Set definitions for algorithm in Figure 11.5. ................................................................................ 394
11.8 Factor graph for the propagation graph in Example 2. ................................................................. 395
11.9 Algorithm `Path`. ..................................................................................................................... 395
11.10 Algorithm `Triple`. ................................................................................................................ 395
11.11 Benchmark application sizes. ..................................................................................................... 399
11.12 Statistics for the out-of-the box specification that comes with `CAT.NET`. ................................. 399
11.13 Size statistics for the propagation graph $G$ and factor graph $F$ used by `MERLIN`. .................... 400
11.14 New specifications discovered with `MERLIN`. ......................................................................... 401
11.15 Vulnerabilities before and after `MERLIN`. ............................................................................... 402
11.16 Function `CreateQueryLink` for Example 5. ............................................................................... 403
11.17 Program for Example 6. ........................................................................................................... 404
11.18 Specification inferred for Example 6. ......................................................................................... 404
11.19 Static analysis and specification inference running time, in seconds. .............................................. 405
12.1 Mining object usage patterns .................................................. 412
12.2 Sample function written in Java. ............................................................................................ 414
12.3 Function model ........................................................................................................................ 415
12.4 Code violating “normal” stack usage. ......................................................................................... 415
12.5 A generic abstract function representation ............................................................................... 417
12.6 Sample matrix used as an input to formal concept analysis . 420
12.7 Checkmycode.org .................................................................................................................. 428
Data mining and machine learning techniques have been used to learn models of software behavior. These models appear in various forms following different formalisms of software specifications, each capturing different aspects of a software system. The mined models can later be tuned, adapted, and used for various purposes.

Mined models help in understanding existing systems and hence reduce the cost incurred during software maintenance (i.e., when features are added, bugs are fixed, etc.). This is particularly advantageous in the cases of maintaining legacy systems and keeping pace with changes in evolving systems, where significant maintenance cost can be saved through better understanding of such systems with the help of mined models.

The mined models/specifications can also be used to aid program testing efforts and help program verification tools to find bugs and ensure correctness of systems. As reported by US National Institute of Standards and Technology (NIST) in 2002, bugs have been causing US economy to lose 59.5 billion dollars annually. Also, the first test flight of Ariane 5 ended up in an explosion due to a software bug. Automated methods to find bugs and ensure correctness of systems are certainly valuable.

There has been a proliferation of research in the area of specification mining in both academia and industry. Research and development in specification mining have been performed in various institutions across the globe with results presented in various venues such as conferences on software engineering, programming languages, data mining, and databases. Due to its appearance in a wide range of research domains, it is often hard to find and relate different works on mining specification in the literature. There has not been a single reference to describe and categorize these studies in a unified setting. This book aims to serve as the first reference to the wealth of knowledge in this new emerging field of mining software specifications.

There are diverse forms of target formalism considered in various research work. Two of the most common ones are finite state machines and rules/patterns of behavior. A finite state machine is typically larger and captures more complex behavior. Many works on mining finite state machine extract models that capture the overall behavior of a system. A set of rules or patterns tends to be smaller and decompose complex behavior into simpler parts. Much work on mining rules/patterns extracts strongly observed sub-behaviors either in terms of their frequency of appearance or some other statistical measures. In

Preface
this book, we present in detail a number of past studies on mining finite state machines (Chapters 2-6) and also those on mining rules/patterns (Chapters 7-12).

There are also differences in terms of how the raw data used as input for the mining and learning tasks are obtained. Some works analyze program code (i.e., static analysis) while others analyze execution traces (i.e., dynamic analysis). There are also works that analyze both code and execution traces. Either static or dynamic analysis has its own advantages and disadvantages. Static analysis could potentially consider all possible behaviors as all the behaviors of the system are in the code. Dynamic analysis on the other hand only analyzes a sample of the behaviors of a system exhibited on a set of runs of a system. However, dynamic analysis can be more precise as exact events that happen during runtime are collected rather than inferred. Its performance is not affected by the difficulties arising from dealing with infeasible paths in a software system or pointer analysis. This book presents both approaches and even some synergies of the two: Some chapters present works that employ dynamic analysis, some others present works that employ static analysis, and yet a few present works that employ a combination of both static and dynamic analyses.

We would like to thank the chapter authors (in alphabetical order): Mithun Acharya, Anindya Banerjee, Kirill Bogdanov, David Evans, Stephen Fink, Ananth Grama, Thomas R. Gross, Suresh Jagannathan, Benjamin Livshits, Leonardo Mariani, Madhuri R. Marri, Aditya V. Nori, Fabrizio Pa-store, Mauro Pezzé, Marco Pistoia, Michael Pradel, Sivaram K. Rajamani, Muralikrishna Ramanathan, Mauro Santoro, Sharon Shoham, Suresh Thummalapenta, Neil Walkinshaw, Andrzej Wasylikowski, Tao Xie, Eran Yahav, Jinlin Yang, Thomas Zimmermann, and Andreas Zeller, for their valuable contributions without which this book would not be possible. We would also like to thank the various reviewers (in alphabetical order): Mithun Acharya, Suresh Jagannathan, Lingxiao Jiang, and Venkatesh Prasad Ranganath, who help to review submitted chapters. Last but not least, we also thank various members of the SpecMine group, National University of Singapore, and Software Mining and Analysis Group, Singapore Management University, for performing some final checks during the compilation of the book.

We sincerely hope that this book can help in raising interest, growth, and collaboration in the area of specification mining. We also hope to see more industry adoption of specification mining techniques and more incorporation of these techniques to standard IDEs in the near future.

David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu
Contributors

Mithun Acharya
ABB Corporate Research
North Carolina, US

Anindya Banerjee
IMDEA Software
Madrid, Spain

Kirill Bogdanov
The University of Sheffield
Yorkshire, UK

David Evans
University of Virginia
Virginia, US

Stephen Fink
IBM T.J. Watson Research Center
New York, US

Ananth Grama
Purdue University
Indiana, US

Thomas R. Gross
ETH Zurich
Zurich, Switzerland

Jiawei Han
University of Illinois at Urbana-Champaign
Illinois, US

Suresh Jagannathan
Purdue University
Indiana, US

Siau-Cheng Khoo
National University of Singapore
Singapore

Chao Liu
Microsoft Research–Redmond
Washington, US

Benjamin Livshits
Microsoft Research–Redmond
Washington, US

David Lo
Singapore Management University
Singapore

Leonardo Mariani
University of Milano Bicocca
Milan, Italy

Madhuri R. Marri
North Carolina State University
North Carolina, US

Aditya V. Nori
Microsoft Research–Redmond
Washington, US

Fabrizio Pastore
University of Milano Bicocca
Milan, Italy
Mauro Pezzè  
University of Lugano and University of Milano Bicocca  
Lugano, Switzerland and Milan, Italy

Marco Pistoia  
IBM T.J. Watson Research Center  
New York, US

Michael Pradel  
ETH Zurich  
Zurich, Switzerland

Sriram K. Rajamani  
Microsoft Research–Bangalore  
Bangalore, India

Muralikrishna Ramanathan  
Coverity, Inc.  
California, US

Mauro Santoro  
University of Milano Bicocca  
Milan, Italy

Sharon Shoham  
Technion–Israel Institute of Technology  
Haifa, Israel

Suresh Thummalapenta  
North Carolina State University  
North Carolina, US

Neil Walkinshaw  
The University of Leicester  
Yorkshire, UK

Andrzej Wasylkowski  
Saarland University  
Saarbrücken, Germany

Tao Xie  
North Carolina State University  
North Carolina, US

Eran Yahav  
IBM T.J. Watson Research Center and Technion–Israel Institute of Technology  
New York, US and Haifa, Israel

Jinlin Yang  
Microsoft–Redmond  
Washington, US

Andreas Zeller  
Saarland University  
Saarbrücken, Germany

Thomas Zimmermann  
Microsoft Research–Redmond  
Washington, US
Chapter 1

Specification Mining: A Concise Introduction

David Lo  
School of Information Systems, Singapore Management University

Siau-Cheng Khoo  
School of Computing, National University of Singapore

Chao Liu  
Microsoft Research–Redmond

Jiawei Han  
Department of Computer Science, University of Illinois at Urbana-Champaign

1.1 Introduction

Many software systems are poorly documented. Developers tend to spend most of their time in developing functionalities rather than documenting them. This causes an issue as after some time it is hard to understand the existing system. A software maintenance task could be harder to perform as there is no documentations of existing functionalities. Indeed, past studies have shown that the cost of software maintenance could be up to 90% of the total software cost [22]. Another study reported that up to 50% of the maintenance cost could be attributed to the difficulty in understanding legacy/existing systems [12,27,67].
From another perspective, software bugs are prevalent. Bugs not only make it more expensive in developing software systems due to the high cost involved in debugging, but also may cause various security vulnerabilities. For many mission and safety critical system faults and bugs could mean the loss of lives or billions of dollars. The absence of specifications has made it harder to locate bugs. Many existing bug finding tools, e.g., model checking [13], require the availability of specifications in order to locate bugs which are defined as anomalies or violations of these specifications.

To address the above challenges (i.e., to improve program understanding and to find bugs), specification mining has been proposed. The term specification mining is first coined by Ammons et al. in [6]. Specification mining is a process of inferring models or properties that hold for a system. Many techniques have been proposed to mine or extract these specifications. Techniques used range from data mining, grammar inference, static analysis, etc.

Specification mining starts with a program under analysis and/or a set of test cases. Techniques employing dynamic analysis require the running of the test cases to produce a set of traces which is later analyzed. Often traces could also be generated statically by “walking through” the code. Some techniques employ symbolic execution to mitigate the effect of infeasible paths. Various learning or mining techniques can be employed on these sets of dynamically or generated traces to infer models.

Various formalisms have been proposed to specify a software system. Similarly, the models mined by these specification mining engines are also varied. They range from temporal properties, finite state machines, etc. In Section 1.2, we discuss in more detail the type of specifications that many past studies have considered.

Models mined in turn could be used to aid various activities. They could be used to guide novice developers in using an existing/legacy software system or a library. Many systems and libraries are hard to understand. Poor understanding of a system in turn could lead to the introduction of bugs. Mined specification describes the constraints that a system would need to obey and thus could help in either reducing the time needed to develop and maintain a system or reduce the number of bugs introduced.

One could also use the mined model as input to a model checking or other lightweight verification process to detect violations. The central tenet of many specification mining-based anomaly detection techniques is stated in [21], namely: If a model is observed “999 out of 1000 times, then it is probably a valid belief and the sole deviation a probable error.” Many specification mining studies have been successful in locating bugs from software systems. Employing model-based testing, mined models could also be used to generate regression tests or reduce the number of test cases and thus improves the quality of software systems while still ensure that the testing process could be performed in a limited amount of time.

In this chapter, we start the book by presenting a concise summary of many existing works on specification mining. This family of studies on specification
mining has been ongoing for over a decade and more than 50 papers have been published on this topic. We provide a brief summary starting from one of the first papers on this topic published in 1995 by Cook and Wolf [14], to many recent papers. The remaining chapters of this book describe 11 different techniques appearing in recent literature in detail.

Section 1.2 describes a general scheme to categorize existing works into several groups. Section 1.3 describes studies on the extraction of finite state machines. Section 1.4 describes works that mine for value-based invariants. Section 1.5 describes approaches to mine rules and patterns. Section 1.6 presents yet another family of studies extracting sequence diagram-like specifications. We conclude our discussion in Section 1.7.

1.2 Categorization

There are various ways to organize the many studies on specification mining. We choose to categorize them in terms of the specifications forms produced by the miners. These include finite state machine or automata, value-based invariants, patterns/rules, and sequence diagram-like representations. Some examples of studies mining the various specification formalisms are shown in Table 1.1.1

A finite state machine consists of states and transitions. The states or the transitions could be labeled – depending on whether a Mealy or Moore machine is mined. These labels could correspond to various pieces of information depending on the level of granularity considered. Typically, the labels correspond to method calls of interest. Most studies on the extraction of finite state machines listed in Table 1.1 propose the extraction of models governing the order in which methods of a particular library could be invoked. An example of a model specifying how a file access library should be used is shown in Figure 1.1. The finite state machine specifies that a call to open needs to be made before any call to read or write. At the end of an interaction, with the library a call to close is made.

A value-based invariant captures a constraint among various global or local variables at a particular point in a program. Various constraints could be considered ranging from a simple equality constraint involving two variables/ -values to more complex inequalities or even conditional constraints involving multiple variables. Value-based invariants thus capture the constraints among the various variables characterizing a system’s state. An example of a value-based invariant is shown in Figure 1.2. It illustrates that at the end of a simple program that computes a square, the computation result is greater than or equal to zero.

1Our list might not be complete.
### TABLE 1.1: Categorization Based on Target Formalisms

<table>
<thead>
<tr>
<th>Formalisms</th>
<th>Past Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automata</td>
<td>Cook &amp; Wolf [14], Ammons et al. [6], Ammons et al. [7], Ahur et al. [5], Henzinger et al. [31], Mariani &amp; Pezzè [53], Acharya et al. [3], Dallmeier et al. [15], Lo &amp; Khoo [38], Acharya et al. [1], Mariani et al. [52], Shevertalov &amp; Mancoridis [65], Quante &amp; Koschke [60], Walkinshaw et al. [72], Gabel &amp; Su [30], Gabel &amp; Su [29], Lorenzoli et al. [49], Mariani &amp; Pastore [54], Walkinshaw &amp; Bogdanov [71], Lo et al. [46], Pradel &amp; Gross [58], Zhong et al. [77], and Gabel &amp; Su [28]</td>
</tr>
<tr>
<td>Value-Based Invariants</td>
<td>Ernst et al. [23–26], Perkins &amp; Ernst [57], Nimmer &amp; Ernst [56], Pytlik et al. [59], Brun &amp; Ernst [11], and Boshernitsan et al. [9]</td>
</tr>
<tr>
<td>Patterns/Rules</td>
<td>Engler et al. [21], El-Ramly et al. [20], Li &amp; Zhou [37], Livshits &amp; Zimmermann [37], Mandelin et al. [50], Weimer &amp; Necula [75], Kremenek et al. [34], Safyallah &amp; Sartipi [64], Yang et al. [76], Lo et al. [39], Ramanathan et al. [61], Ramanathan et al. [62], Thummalapenta &amp; Xie [68], Lo et al. [40], Ramanathan et al. [63], Zhong et al. [79], Zhong et al. [78], Acharya &amp; Xie [2], Heydarnoori et al. [32], Lo et al. [47], Nguyen et al. [55], Thummalapenta &amp; Xie [70], Thummalapenta &amp; Xie [69], Wasylikowski &amp; Zeller [73], Zhong et al. [77], and Lo et al. [41]</td>
</tr>
<tr>
<td>Sequence Diagrams</td>
<td>Briand et al. [10], Lo et al. [48], de Sousa et al. [17], Lo &amp; Maoz [43], Lo &amp; Maoz [42], Lo &amp; Maoz [44], and Doan et al. [19]</td>
</tr>
</tbody>
</table>
There are many different approaches that mine for frequent patterns. A pattern corresponds to a template that matches many concrete instances obeying the pattern in a trace or in a program. Data mining concepts such as itemset, sequential pattern, and graph pattern are often used. An itemset is a set of items. A sequential pattern is a series of items/events. A graph pattern captures relationships between labeled nodes and labeled edges. Each item/event/label could denote various pieces of information although typically they represent method calls. A pattern is frequent if it appears frequently in a dataset of interest. Various types of patterns have been mined including sequential patterns that appear often in execution traces to graph patterns that appear often in code bases. For example, a sequential pattern specifying a partial method call sequence to implement a context menu using JFace is shown in Figure 1.3 – c.f. [32].

There are other studies that mine for significant rules. Different from a pattern that captures things which appear frequently, a rule captures a constraint between two things; namely, the premise/pre-condition and the consequent/post-condition of the rule. A rule can capture a constraint between itemsets, sequential patterns, or graph patterns. There are many rules that programmers obey, for example, a “lock” must eventually be followed by an “unlock”, an “allocation” must eventually be followed by a “free”, a “use”
must be preceded by an “allocation”, etc. Some examples of these rules are shown in Figure 1.4.

Another form of specifications is sequence diagram which is composed of lifelines and messages. They are intuitive to capture the communication among the various objects through various method calls. There are also some extensions to standard UML sequence diagrams including Message Sequence Charts (MSC), which is the standard of International Telecommunication Union (ITU) [33] and Live Sequence Chart (LSC) [16] that extends MSC with modalities. LSC is able to express a rule-like property in the form of a modal sequence diagram. An example of a Live Sequence Chart is shown in Figure 1.5. It captures the constraint “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.”

From Table 1.1, we notice that there are many studies extracting finite state machines and rules/patterns. There are also a good number of studies mining other forms of specifications including value-based invariants and sequence diagrams.
1.3 Mining Finite State Machines

In this section, we describe past studies on mining finite state machines. In this and the following sections, we present the studies in a semi-chronological order.

One of the pioneer works on the inference of finite state machines from software systems is done by Cook and Wolf [14]. They investigate various approaches to infer a finite state model of a process from a set of event sequences. Various approaches ranging from neural network to grammar inference (i.e., k-tails [8]) are investigated.

The work by Cook and Wolf is later extended by Ammons et al., who coin the term “mining specification” [6]. Ammons et al. directly address software specifications rather than generic process models. They employ a program analysis technique to extract related trace segments from dynamic executions of a program. These trace segments are later fed into a probabilistic grammar inference engine. They also develop a process to prune potentially wrong edges from the resultant finite state machine. In their later work [7], they propose an approach to reduce the amount of user manual feedback needed in their system to debug and correct errors in mined specifications.

Whaley et al. propose a static and a dynamic analysis approach to extract specifications in the form of a finite state machine [74]. Rather than producing a single finite state machine, they produce multiple finite state machines. Each finite state machine models a sub-behavior 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 static analysis approach, for each
method \( m \) they find fields and predicates that guard exceptions. Next, they
detect other methods \( m' \) that set the values of the fields and the predicates that
would trigger some exceptions in \( m \). Calls to \( m' \) which are directly followed
by \( m \) are thus illegal. A model that excludes the illegal behaviors is finally
produced. For the dynamic analysis approach, their approach keeps track of
the history of the last state-modifying method that was called. Each method
corresponds to a unique state. Their algorithm then builds a graph connecting
two methods that are called one after another.

Alur et al. propose an approach to infer a finite state machine from Ap-
plication Programming Interface (API) code [5]. Given a library code and
certain assertions that must hold, the task is to describe the most general way
to invoke the methods in the API such that the assertions are not violated. A
sequence of method calls that does not lead to a violation of the assertions is
considered safe. Alur et al. provide a sound algorithm to solve this problem.
The work is extended by Henzinger et al. [31]. In the latter work, they could
assure that all safe sequences of method calls are represented in the mined
finite state machine. The resulting model learned is termed as “permissive
interface” as it does not wrongly forbid any client from using the API.

Mariani and Pezzé propose a new grammar inference engine specially
suited for program executions named k-behavior [53]. They employ k-behavior
along with a value-based invariant generation engine (see Section 1.4) to con-
cisely capture the various behaviors of a software system. These behaviors
could later be replayed to test the correctness of the software system.

Acharya et al. [3] propose a static analysis approach to mine finite state
machines from systems. Their tool leverages a model checker to statically gen-
erate traces of a system. Based on certain trigger automata, the model checker
could generate concrete labels corresponding to various condition checks and
method calls as transition labels of the automata. These concrete labels are
then further analyzed to generate a mapping between high level concepts, e.g.,
check, free, etc., to concrete statements. Users could then specify a generic au-
tomata to be automatically converted to a concrete automata that could in
turn be used to find violations using a model checker.

Dallmeier et al. propose a hybrid static and dynamic analysis approach to
infer a finite state machine [15]. First, a set of methods termed as mutators
(i.e., those that change a system state) and another set termed as inspectors
(i.e., those that read/inspect a system state) are determined statically. A
dynamic analysis approach is then employed to get possible states of a system
based on abstracted values of variables, e.g., size>0, size=0, etc. These states
form the nodes in finite state machine. Invocations of the mutator methods
that transform one state to another act as transitions in the resultant finite
state machine.

Lo and Khoo extend the work by Ammons et al. by proposing a metric of
precision and recall in evaluating the quality of a specification mining engine
producing finite state machines [38]. They also introduce trace clustering and
trace filtering to reduce the effect of bad traces and inaccuracies during the inference of mined specifications.

Acharya et al. extend their previous work [3] in [1]. Static traces are first extracted from program code similar to their previous work [3]. Relevant segments of traces are then recovered. These segments of traces fed to a frequent partial order miner. The resultant partial orders are then composed to generate a specification in the form of a finite state machine.

Mariani et al. extend their previous work [53] in [52]. They propose a technique to generate prioritized regression test cases for the integration of Commercial-off-the-Shelf (COTS) components. A model in the form of a finite state machine and a set of boolean expressions on a set of variables, operators, and values is learned based on an older version of a system. This model is used to generate test cases to verify whether there are problems in a future version of the system when a component is replaced.

Shevertalov and Mancoridis propose an approach to extract a finite state machine capturing a network protocol [65]. Their system first captures packets from a network traffic. These packets are then clustered using a hierarchical clustering approach with a distance metric defined based on the longest common subsequence of two packets when compared byte per byte. Each cluster is given a unique identifier. A set of traces of captured packets can then be converted to a set of sequences of identifiers. A set of states and transitions between states are then inferred. Various refinement operations to split and merge states are also proposed.

While many specification mining approaches extract a finite state machine from sequences or stream of events, Quante and Koschke develop an approach that extracts a finite state machine from graphs [60]. An object process graph, which is a projection of a control flow graph on a single object, is used. The approach takes in a set of object process graphs corresponding to the various usage scenarios of a particular application and generalizes them to form a finite state machine via some transformation operations.

Walkinshaw et al. employ a state-of-the-art grammar inference technique that incorporates active learning [72]. The inference engine asks a series of membership questions while performing the inference. A membership question asks whether a series of events is valid or not. Inputs from a user or new test cases could be used to answer these membership questions. With answers to membership questions serving as feedbacks, the quality of the mined specification could be improved.

Gabel and Su use a Binary Decision Diagram (BDD) based approach to mine small size automata efficiently [30]. They show that the problem of specification mining under a particular setting, i.e., given an automaton template A and a trace T, enumerate all possible concrete automata following A that are satisfied by T, is inherently NP-Complete. A BDD-based approach could be used to speed up the process. In the process they show that they are able to mine small size automata from traces of sizes up to millions of events. They
extend their approach by proposing a technique to merge multiple small size finite state machines to a larger one [29].

Lorenzoli et al. mine extended finite state machines (EFSMs) in [49]. An extended finite state machine enriches a standard finite state machine with value-based invariants. Value-based invariants make the finite state machine more expressive: It could specify that if a certain constraint is met a set of behaviors becomes possible, otherwise another set of behaviors becomes possible. Their approach consists of merging of equivalent traces, generating value-based invariants, creating an initial model, and merging the initial model’s states to result in a final model.

Mariani and Pastore mine finite state machines to identify failure causes from system logs [54]. System logs are first collected. Several pre-processing modules to detect for events in logs and to transform data to an appropriate format that abstracts away concrete values are first employed. Based on the transformed log a model is learned using the approach proposed in [53]. When a failure occurs, the corresponding log and the mined model are analyzed to produce suspicious statements as potential root causes.

Shoham et al. propose an approach to infer specifications via an interprocedural static analysis using abstract interpretation [66]. Their approach works in two steps. The first step is abstract-trace collection. In this step, abstract histories in automata form are learned. Next, a summarization step is performed. A statistical approach is employed in this step to consolidate information collected in the first step and remove noise.

In [71], Walkinshaw and Bogdanov further automate their active learning finite state machine inference approach [72]. The approach in [72] potentially asks a large number of questions to users. The work in [71] reduces the amount of questions by asking users to provide a set of constraints. This set of constraints is used to automatically answer many of the generated questions.

In [46], Lo et al. propose mining of short temporal rules (in future and past-time temporal logics) that are incorporated to finite state machine inference. Different from the approach in [71], these rules are mined rather than specified. This approach addresses a technical concern about undesired merges of states during an inference leading to an imprecision in the constructed finite state machine. Specifically, the mined rules are used to prevent these bad merges, resulting in the learning of a more accurate finite state machine. A sound but incomplete approach to efficiently check the mined rules on the intermediate finite state machines is also proposed.

In [58], Pradel and Gross develop an efficient solution to mine for specifications involving multiple objects from a large amount of traces. Their approach first splits the traces into multiple smaller traces consisting of calls to related objects and methods. Each set of splitted traces is analyzed separately. This divide and conquer approach scales their approach to process a large amount of events in traces collected from various applications; each can contain up to tens of millions of events.

In [80], different from previous approaches, Zhong et al. extract specifi-
cations from natural language documentations (specifically Javadocs). These Javadocs are analyzed to form action-resource pairs. These pairs are later combined into a finite state machine. They have shown that the generated finite state machines are accurate and are able to detect bugs.

Gabel and Su propose an online specification mining approach [28]. Rather than performing an offline mining on a set of traces, they devise a technique to perform mining at the same time as traces are generated. A trace is processed window by window. Specification is mined/updated based on information collected in each window. Mining and checking of mined specifications are performed along the way as new traces are collected.

1.4 Mining Value-Based Invariants

In this section, we describe past studies mining value-based invariants. The pioneer work in extracting value-based invariants is Daikon by Ernst et al. [23–26]. Daikon contains many value-based invariant templates. It monitors a program in execution and matches one or more invariant templates to particular program points of interest. It then reports the invariants that holds at the program point. Various filtering and template selection strategies are provided by Daikon. Daikon is also integrated with constraint solvers to reduce the redundancy in the mined invariants. Optimization strategies have also been proposed to speed up the extraction of value-based invariants, e.g., by incremental detection of invariants [57].

Many other studies leverage Daikon to perform other software engineering tasks. For example, Nimmer and Ernst integrate Daikon with ESC/Java to evaluate the effectiveness of the mined invariants in detecting bugs via static analysis [56]. Pytlik et al. leverage Daikon invariants for fault localization [59]. Given a set of execution traces that fail and those that are successful Pytlik et al. find invariants that discriminate failing and correct traces in an effort to localize or pinpoint the wrong statement which is the root cause of the failure. They encounter a negative result in their experiment as many of the invariants are not related to bugs. Their study is later extended by Brun and Ernst in [11]. The latter study is able to show that with machine learning, Daikon invariants could be used to localize bugs. Demsky et al. use Daikon to learn data structure consistency invariants for automatic data structure repair [18].

In several studies the work on mining value-based invariants is combined with mining other invariant types. For example, Ramanathan et al. mine both value-based invariants and precedence rules as pre-conditions of various methods using an inter-procedural path-sensitive static analysis [62]. Lorenzoli et al. integrate Daikon invariants into a finite state machine inference algorithm.
Mining Software Specifications: Methodologies and Applications

to mine specifications that capture both the temporal ordering constraints and invariants among values/variables [49]. Lo and Maoz also extend their algorithm with Daikon to enable the extraction of sequence diagrams in the form of Live Sequence Charts with value-based invariants from program executions [45].

1.5 Mining Patterns and Rules

In this section, we describe past studies mining rules and patterns.

Engler et al. are among the first pioneers in proposing statistical approaches to find bugs [21]. Their tool is built upon the tenet: “If two beliefs contradict, we know that one is an error without knowing what the correct belief is.” Their system works on a set of templates that associates a few variables together, e.g., “Does lock (L) protect a resource (R)?” “Must IS_ERR be used to check the return value of function F?” etc. Instances of these templates with sufficient statistical values are then identified. Violations of these instances are reported as potential bugs in a ranked list based on statistical likelihood of them being errors.

El-Ramly et al. extract frequent patterns of usage termed as interaction patterns from a set of sequences of events extracted from runtime program executions [20]. They formulate a new frequent pattern mining semantics and an efficient algorithm to mine the patterns. They applied their solution in a software maintenance task. In particular, they show the viability of the mined patterns in helping the migration of a legacy system to a new web-based solution.

Li and Zhou propose an approach that mines for rules from program code [37]. In their approach, every function is mapped into a transaction. The database of transactions corresponding to functions in a program under analysis is later mined. Association rule mining is employed to find rules that satisfy minimum support and confidence thresholds. The mined rules are used for the detection of anomalies.

Livshits and Zimmermann propose an approach that combines repository mining with the analysis of program execution traces to infer specifications [35]. First, transactions are formed from sets of method calls that are added together in a revision in a software repository. These transactions are later subject to association rule mining to mine for rules satisfying minimum support and confidence thresholds. Mined rules are later filtered and ranked. For the remaining set of rules, a user can choose some of them. These selected rules would be verified using a dynamic analysis approach. If there are many violations of a rule in execution traces, the rule is less likely to be valid. On
the other hand, if the traces conform to the rule, the rule is more likely to be valid. Mined rules are used for the detection of anomalies.

Mandelin et al. propose a concept referred to as *jungloid mining* [50]. The process accepts a pair of input and output types of a desired code fragment. The system then produces a piece of code fragment that would transform the input type to the output type (i.e., jungloid). The resultant code fragment refer to the variables in the program under investigation and thus can be inserted with ease. The system behaves like a programmer’s search engine and produces a list of possible jungloids based on a user query.

Weimer and Necula extract temporal rules statically by analyzing program code [75]. A temporal rule specifies an ordering constraint among events. In particular, their work focuses on an approach to mine two event rules, e.g., a lock is eventually followed by an unlock, etc. First, a set of static traces is extracted from program code. These traces are categorized into error and normal traces. An error trace passes through an error handling code. Next, based on the intuition that programmers tend to make more bugs in handling exceptional conditions, they devise a set of heuristics to automatically extract temporal rules that can result in finding a set of violations, particularly those that go through error handling code.

Kremenek et al. mine for annotations of methods in a code base [34]. As the first step, a set of annotations is defined, e.g., allocation and deallocation. Next, a model is built based on the annotations. Using the assumption that a program is generally correct they make use of a factor graph. A factor is a relationship that maps the deduction that a variable (i.e., the methods to be annotated) has a particular value (i.e., the annotation, e.g., allocation, etc.) to a positive real number. A factor graph is used to represent the relationship between various factors and variables. It is powerful enough to represent the fusion of various pieces of information and to propagate information learned on one method to other related methods.

Safyallah and Sartipi develop an approach to find frequent behavioral patterns from program traces [64]. They propose a frequent sub-string mining algorithm to realize the task. A sub-string mining algorithm is more efficient than a regular sub-sequence mining algorithm first proposed in [4] as it does not need to consider gaps between the events in a pattern. They also propose a two-step process in which the result of a mining operation is used as an input of another mining operation to detect higher level patterns.

Yang et al. develop a linear time solution to extract two-event rules from execution traces [76]. They particularly focus on expressing alternating behavior in which two events must occur in a strict alternating ordering. A concept of satisfaction rate is used to detect interesting rules. The linear time algorithm is realized by maintaining a matrix-like data structure of quadratic size to the number of unique events in the trace. As each event in the trace is processed a corresponding row and a corresponding column in the matrix are updated. They also propose a chaining process to link multiple two event rules that are mined.
Lo et al. mine iterative patterns to capture frequent software behaviors [39]. An iterative pattern captures pattern instances that appear within a sequence and across multiple sequences. It merges frequent subsequence mining [4] with frequent episode mining [51]. Since a program contains loops, recursions, etc., patterns of interest could appear many times in an execution of a program. A depth-first pattern mining solution is employed. A notion of closed patterns (i.e., the longest pattern without a super-sequence pattern having the same support) for iterative patterns is also proposed. Several pruning strategies to remove search space containing infrequent and redundant patterns (i.e., non-closed patterns) are detected early and those patterns are removed en masse.

Ramanathan et al. propose a static analysis technique to generate inter-procedural path-sensitive constraint repository from program code [61]. The constraint repository captures precedence relationships in the form of a sequence/chain of elements. From these trace-like chains of elements, a sequence mining operation is performed to find frequent patterns in the repository. These frequent patterns are then used to detect anomalies.

Ramanathan et al. next propose another inter-procedural path-sensitive static analysis technique to infer pre-conditions that must be observed before a method call is made [62]. The pre-conditions are in two forms: dataflow and control flow properties. The data flow property is in the form of value-based invariants, e.g., when a procedure X is called the integer parameter n must be less than or equal to zero. The control flow property captures ordering constraints among method calls, e.g., when “unlock” is called, “lock” must be called before.

Thummalapenta and Xie mine for frequent event sequences from the source code repositories on the web [68]. The approach first downloads code from source code repositories, e.g., Google code. The code is then analyzed to extract Method-Invocation Sequences (MIS) that could transform an object of a particular type to another. The extracted MISs are later clustered and ranked before being presented to the user. Two ranking heuristics based on frequency and length are used.

Lo et al. extends the algorithm in [76] to mine rules of longer length [40]. Recall that the approach in [76] mines for rules of length two and later composes them to rules of longer lengths. However some rules might be omitted and some other generated rules might not be valid (i.e., when the satisfaction rate is not perfect). Mined temporal rules are in the format: “Whenever a series of events occurs, another series of events must eventually occur.” Their algorithm follow a depth first approach to traverse the rule search space. Search spaces containing non-significant rules are identified and those rules are removed en masse. The resulting algorithm guarantees that 1) all mined rules satisfy the user defined threshold of support and confidence, 2) all interesting rules satisfying support and confidence thresholds are mined. A redundancy criterion is also defined and a process to identify and remove search space containing redundant rules is proposed. The early detection of
redundant rules could greatly reduce the runtime needed and the number of rules mined.

Ramanathan et al. provide a summarization mechanism to statically generate paths in programs [63]. The proposed technique summarizes static paths that reach a particular point in a program. The static paths are collected by propagating and merging predicate information in an inter-procedural path-sensitive analysis. For each node in a control flow graph (CFG) information on the number of paths that reach it, and the number of paths satisfying various predicates that go through it, is collected. The summary is termed as static path profile. The static path profile is used to mine pre-conditions of various methods.

Zhong et al. extract specifications from code in the form of a rule graph [79]. Nodes in the rule graph correspond to method calls and edges correspond to relationships between these methods. Their approach starts with a set of basic known rule graphs. These rule graphs are extended to form new rule graphs. Basic facts are extracted from program source code. These facts are used to infer other graphs based on the existing pool of known rule graphs in an iterative manner. The rule graphs could later be visualized to show the relationships among various methods.

Zhong et al. study the effect of trace quality on the quality of mined specifications [78]. They focus on mining specifications in the form of frequent sub-sequence patterns in a trace set. They introduce the notion of “polluting” events. They show that some events in the trace might lower the quality of the mined specifications. Several automated filtering strategies are proposed to remove these polluting events. They show that the quality of mined specifications is improved after the polluting events are filtered and mining is performed on the resultant traces.

Acharya and Xie mine for error-handling specifications from program code [2]. Their approach first extracts traces corresponding to error-handling and correct behaviors. Two types of specifications are mined, namely, error checking specification (i.e., a particular check is performed before a returned result is used) and multiple-API specification (i.e., right clean-up API methods are called). To mine for multiple-API specification, a frequent sub-sequence mining algorithm is employed. The sub-sequence mining algorithm is used to extract frequent sequences of method calls from correct executions.

Heydarnoori et al. propose an automated approach to extract concept implementation templates [32]. A concept implementation template is a sequence of method calls together with related information (e.g., a set of import statements in Java, etc.) to implement a particular generic functionality in a particular framework, e.g., implementing a context menu in JFace, etc. The approach starts with a collection of execution traces that is relevant to the context of interest. The extracted traces are later marked and sliced to get the relevant method calls. Events in the extracted sliced traces are then generalized. Several sets of facts, e.g., the dependency among events, etc., are extracted from the generalized traces. An intersection of these sets is taken.
Based on the generalized traces and common facts, a code corresponding to the template is finally generated.

Livshits et al. mine explicit information flow specifications [36]. They first extract a propagation graph which is a directed graph whose nodes are methods and transitions are explicit information flow between these methods. The task of explicit flow information is to label the nodes in the propagation graph by one of the following labels: sources, sinks, sanitizers, and regular. Given the labels and the graph, one could find information flow issues by running a reachability analysis. A probabilistic inference approach is used based on the intuition that most paths in the propagation graph are valid or secure.

Lo et al. propose an algorithm to mine for quantified temporal rules [47]. They propose a linear time algorithm to mine such quantified rules of length two. The approach could mine regular temporal rules, temporal rules with equality constraints, and quantified temporal rules with equality constraints. An example of quantified temporal rules is “for all object z, all calls to method m1(z), must eventually be followed by a call to method m2(z).” Only significant rules that satisfy minimum support and confidence thresholds are reported. A redundancy criterion is also proposed and redundant rules are filtered before being reported to the user.

Nguyen et al. capture common usage patterns of one or multiple objects from program code [55]. Their approach first constructs a graph-based object usage model (groum). The nodes in a groum correspond to various method calls or control structures. The edges in a groum correspond to temporal usage orders or data dependencies among various nodes. The groums are extracted from all the methods in a code base. The set of groums is input to a frequent graph mining algorithm. The resultant patterns are reported and are used to detect for anomalies in the code base.

Thummalapenta and Xie mine a set of exception handling rules to capture the behaviors that should occur when an exception is encountered [70]. Exception handling rules are expressed in a rule format: “A function F should be followed by a series of function G1, . . . , Gn if it is preceded by a series of function E1, . . . , En.” They also propose an extension to control flow graph termed exception flow graph that captures the flow of a program when exception happens. Static traces are extracted from the EFG, post-processed, and input to the mining algorithm. Rules are then produced and used to detect for anomalies.

Thummalapenta and Xie also mine for alternative patterns to detect for neglected conditions [69]. An alternative pattern captures a disjunction among patterns. To capture alternative patterns a frequent itemset mining algorithm is run multiple times. The first run would obtain the set of frequent patterns. For each frequent pattern, two sub-databases are formed: One corresponds to itemsets that do not contain all items in the frequent pattern, another corresponds to itemsets that support the frequent pattern. Alternating patterns are mined on the first sub-database by re-running the frequent itemset mining algorithm. The support of an alternating pattern is the support of the pattern
in the first sub-database minus that of the second sub-database. The mined patterns are then used to detect for anomalies corresponding to neglected condition checks.

Wasylkowski and Zeller mine temporal rules in the form of Computational Tree Logics (CTL) from program code [73]. Their approach, named Tikanga, takes as input a program code to be analyzed and a set of CTL templates (of one or two placeholder events). An object usage model capturing how an object is being used in a program is first created via an intra-procedural static analysis. Each of these models is later converted into a Kripke structure (i.e., a finite state machine representation normally used in model checking). Next, a model checking procedure is used to check the satisfiability of CTL formulas from the templates on the Kripke structures. For each formal parameter of a method, a set of common CTL formulas that holds for objects passed as the actual arguments is mined via concept analysis. The resultant CTL formulas are used to detect for a list of violations ranked from the most probable to the least probable.

Zhong et al. mine frequent usage patterns from code bases in the web, e.g., Google code [77]. API call sequences are first extracted from a code base. The call sequences are then clustered. Each of the clusters is then input to a frequent subsequence mining algorithm. The resultant patterns are indexed to code snippets that exhibit such patterns. The patterns are presented to the user and once the user selects a particular pattern, all corresponding code snippets are also displayed. The approach is shown to be better than using Google code alone as it could automatically extract and utilize the context of an API method call.

In [19], Doan et al. package the algorithm described in [48] into a tool. The tool supports the creation of mining projects, management of traces, and running of the mining algorithm. The resultant mined scenarios could also be saved and re-visited later. It also supports identification of segments of the traces that satisfy/violate the mined scenarios. A set of pre-processing and post-processing options is also available to help users in refining the result of the mining process.

In [41], Lo et al. extend the mining of closed iterative patterns [39]. They propose an approach to mine for generators of iterative patterns. A generator is a frequent pattern without any sub-sequence pattern of the same support. Whereas closed patterns are the maximal patterns, generators are the minimal patterns. Aside from closed patterns, generators are another alternative set of patterns to compactly represent a set of frequent patterns. The concept of representative rules that merge closed patterns and generators is also proposed. The resultant rules can capture interesting temporal constraints including specifying a set of events that must happen before, after, and in-between sequences of precursor events.
1.6 Mining Sequence Diagrams

In this section, we describe past studies mining sequence diagrams.

Briand et al. extract a UML sequence diagram from execution traces [10]. A UML sequence diagram contains lifelines corresponding to objects, and messages corresponding to method calls between these objects. They propose a framework to instrument a distributed system via an aspect-oriented language (i.e., AspectJ). Running an instrumented program would produce a trace. The collected traces are then analyzed to form a sequence diagram that could capture the caller, callee, and method signature information. In addition, branching points and loops in the trace would also be represented in the extracted sequence diagram.

Lo et al. extract modal sequence diagrams [48]. Such modal sequence diagram is represented as Live Sequence Charts (LSCs) [16]. An LSC is composed of a prechart and a mainchart. It could express a constraint: “Whenever the pre-chart is satisfied, the post-chart would eventually be satisfied.” Both the pre- and post-charts are sequence diagrams capturing caller, callee, and method signature information. Mined LSCs could be used as input to a runtime verification tool to find for anomalies.

Sousa et al. extract implied scenarios from execution traces [17]. Their approach starts with trace collection with each trace corresponding to an execution of a scenario. The collected traces are later filtered to distill the method calls that are relevant to the scenario and remove irrelevant ones. Based on this set of scenarios, their approach infers implied scenarios that are not present in the original set. The final set of scenarios is output and presented to the user.

In [42], Lo and Maoz extend the initial work on mining Live Sequence Charts published in [48]. The work introduces the concept of triggers and effects. The task is: Given a trigger, mine all significant effects. Similarly, given an effect, find all significant triggers. Thus users could help in directing the specification miner by providing a trigger or an effect. The resultant specification mining process could reduce both the runtime cost and the number of mined specifications. Only specifications related to the trigger or effect of interest would be mined.

In [43], Lo and Maoz further extend the work in [48]. They focus on the semantics of symbolic Live Sequence Charts. Symbolic LSCs have symbolic lifelines. A symbolic lifeline specifies a set of objects of a particular type or class. The work in [48] only approximates the symbolic LSCs by aggregating all frequent object-level/concrete LSCs in which all lifelines correspond to concrete objects. However it could be the case that a symbolic LSC is frequent while the corresponding object level LSCs are not. In this case, the work in [48] would miss the symbolic LSCs. This issue is addressed in [43].

In [44], Lo and Maoz introduce the concept of hierarchical specification
mining. Given a hierarchical structure, they mine a set of specifications at a certain abstraction level of interest. The resultant mined patterns could be zoomed-in or zoomed-out by expanding or collapsing lifelines of interest. It helps users by mining specifications at the right abstraction level. If more information is needed, the mined specification could be zoomed in. If there is too much information, the mined specification could be zoomed out. Thus, it provides an interactive mining experience to the user.

In [45], Lo and Maoz incorporate mining value-based invariants to mining Live Sequence Charts. They introduce the concept of scenario-based slicing. A set of scenarios is first mined. For a desired scenario of interest, slices of program traces that satisfy the scenario are extracted. These slices are fed to a value-based invariant miner, namely, Daikon [24], to produce a set of invariants. These invariants are then incorporated to the mined LSC of interest. Thus the work could capture both temporal ordering constraints and value-based invariants from execution traces.

1.7 Conclusion

Specification mining corresponds to the inference of models in various formats from programs either from code or execution traces or even natural language documents. Mined specifications are useful for various software engineering activities. These include program comprehension, software maintenance, bug finding, and test cases generation.

Categorizing based on the type of specification formalisms that were mined, the work on specification mining could be grouped into these mining: finite state machines, value-based invariants, patterns/rules, and sequence diagrams. More than 50 papers have been published on specification mining and we have surveyed many of them in this chapter.

In the following chapters, various works are discussed in more detail. We focus particularly on specification mining works that extract finite state machines and patterns/rules. The book covers approaches employing static analysis, dynamic analysis, and even the combination of the two.
Bibliography


Chapter 2

Mining Finite-State Automata with Annotations

Leonardo Mariani, Fabrizio Pastore, Mauro Pezzè, and Mauro Santoro
Department of Informatics, Systems and Communication
University of Milano Bicocca

2.1 Introduction ........................................................... 30
2.2 Modeling Software Systems with Annotated FSA .................. 31
2.3 Introducing a Running Example ..................................... 32
2.4 Inferring FSA Annotated with Constraints ......................... 36
  2.4.1 Merging Similar Traces ............................................ 36
  2.4.2 Generating Constraints ........................................... 37
  2.4.3 Initializing EFSA .................................................. 37
  2.4.4 Merging Equivalent States in EFSA ............................ 39
2.5 Inferring FSA Annotated with Data-Flow Information ............. 43
  2.5.1 Preprocessing Data ................................................ 43
    2.5.1.1 Identifying Data Clusters ................................. 43
    2.5.1.2 Global Ordering Rewriting Strategy ..................... 44
    2.5.1.3 Instantiation Rewriting Strategy ......................... 45
    2.5.1.4 Access Rewriting Strategy ............................... 47
    2.5.1.5 Choosing a Rewriting Strategy .......................... 48
  2.5.2 Generating Models ................................................. 50
2.6 Comparing the Generated Models .................................... 52
2.7 Related Work .......................................................... 53
  2.7.1 Mining Models That Represent Ordering of Events ............. 53
  2.7.2 Mining Models That Represent Program Variables ............. 54
2.8 Concluding Remarks .................................................... 54
Bibliography ............................................................... 55