Program Synthesis from Execution Traces and Demonstrations

by

Kuat Yessenov

Submitted to the Department of Electrical Engineering and Computer Science on March 1, 2016, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Abstract

In this thesis, we introduce an architecture for programming productivity tools that relies on a database of execution traces. Our database enables a novel user interaction model for a programmer assistant based on short demonstrations of framework usages in applications. By matching the demonstration traces against the complete traces in the database, our system infers the code snippets for the demonstrated feature including the missing set-up steps. We develop techniques for an interactive trace matching process, and evaluate them on a sample of Swing applications. We show that our system synthesizes code for several features of the Eclipse platform from traces of existing Eclipse plug-ins, and that the generated code is comparable in quality to the tutorial code.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor
Acknowledgments

I would like to thank my advisor, Armando, for the support and encouragement over the years. No matter how often I get stuck pursuing my research, he always finds a way to help me figure out a way to get back on track. Thank you, Armando, for believing in me, letting me explore my own ideas, and guiding me to the completion of this journey.

I also would like to thank Daniel Jackson for introducing me to the research in programming languages. My interactions with Viktor Kuncak, K. Rustan M. Leino, and Adam Kalai over the summers have undoubtedly influenced my views on verification and usability of programming tools.

Computer Aided Programming group has been my welcoming home at MIT. My fellow graduate students and friends, Eunsuk Kang, Aleks Milicevic, Joe Near, Rishabh Singh, Jean Yang, Alvin Cheung, Ivan Kuraj, Nadia Polikarpova, Shachar Itzhaky, and many others, provided me with a warm atmosphere. It was a joy to share their company.

And, last but not least, I am grateful to my parents and to Aizana for their love and patience.

This work has been generously supported by NSF CCF-1116362 and DARPA FA8750-14-2-0242.
## Contents

1 Introduction .................................................. 11
   1.1 Illustrative examples .................................... 15
      1.1.1 MATCHMAKER scenario ................................ 15
      1.1.2 DEMO MATCH scenario ................................ 17
   1.2 Overview .................................................. 19

2 Trace collection framework ................................. 21
   2.1 Bytecode instrumentation ................................ 22
   2.2 Collector runtime ........................................... 23
   2.3 Collection overhead ....................................... 25

3 Trace data model ............................................. 27
   3.1 Events ..................................................... 27
   3.2 Declarative trace view ................................... 29
   3.3 Call graph view ............................................ 31
   3.4 Heap series view .......................................... 32
      3.4.1 Heap series construction ......................... 33
      3.4.2 Container abstraction ............................ 33
      3.4.3 Heap series representation ..................... 35
      3.4.4 Heap abstractions ............................... 35
   3.5 Programming interfaces ................................. 39
## 4 Trace matching

4.1 Framework feature analysis .................................. 42
4.2 Concept analysis .............................................. 44
4.3 Types of call queries .......................................... 46
  4.3.1 Ranking scheme ........................................ 47
  4.3.2 Domain traces .......................................... 48
  4.3.3 Query refinement ...................................... 49
  4.3.4 Binary attributes ...................................... 50
4.4 Summary ...................................................... 51

## 5 Synthesis from traces ........................................ 52

5.1 Dynamic slicing ............................................. 53
5.2 Code generation ............................................. 56
  5.2.1 Projection to user events ............................ 56
  5.2.2 Generating statements from events ............... 57
  5.2.3 Code simplification .................................. 58
  5.2.4 Code combination .................................... 61
5.3 Programming interfaces .................................... 62

## 6 Experimental evaluation .................................... 65

6.1 Trace collector performance ................................ 67
6.2 Swing feature extraction ................................... 68
  6.2.1 Swing feature trove .................................. 69
  6.2.2 Movies application ................................... 70
  6.2.3 Passwordstore application .......................... 77
6.2.4 Stocks application .................................... 83
  6.2.5 Concluding remarks ................................ 89
6.3 Swing feature synthesis .................................... 90
  6.3.1 Text component example .............................. 90
  6.3.2 Field validation example ............................. 92
  6.3.3 Cut-copy-paste examples ............................. 92
6.3.4 Tooltip examples ............................................. 95
6.3.5 Table sorting example ..................................... 97
6.4 End-to-end examples from Eclipse RCP .................... 98
6.4.1 Eclipse auto-completion ................................... 98
6.4.2 Eclipse editor folding ..................................... 102
6.4.3 Eclipse auto-edit .......................................... 103
6.4.4 Eclipse outline navigation ................................. 106
6.4.5 Concluding remarks ....................................... 110

7 Related work ....................................................... 111

8 Conclusion ......................................................... 116
List of Figures

1-2 Eclipse JDT auto-completion pop-up ........................................ 17
1-3 User code required for the editor code completion functionality. DEMO-
MATCH automatically synthesizes the text in black. ....................... 18
1-4 SEMERU architecture overview .............................................. 20
2-1 Trace collection framework architecture ................................. 22
2-2 Full and demonstration trace configurations ............................ 26
3-1 Formal model of SEMERU .................................................. 28
3-2 Trace schemata .................................................................. 30
3-4 Effect of the time abstraction: Y-axis is the number of edges with N
or less contiguous ranges in their time intervals. ......................... 38
3-5 Core definitions from Scala implementation of the trace model ...... 39
4-1 Relationship between the user code and the framework features ... 43
4-2 SEMERU call queries as a formal grammar ............................. 47
5-1 Local slicing rule heuristics in presence of aliasing .................. 55
5-2 Simplification pass removes local containers from the method body . 60
5-3 DEMOMATCH and MATCHMAKER extract code snippets from SEMERU
traces by providing initial seeds to the slicer ............................... 63
6-3 Synthesized input validation examples from the focus subsystem tutorial 93
6-4 Synthesized buffer import examples from 7 Swing tutorials .......... 94
6-5 Synthesized code for the table tooltip call query ..................... 96
6-6 Synthesized code for the table sorting feature .......... 97
6-7 Eclipse auto-complete demo traces call query lattice .......... 99
6-8 Synthesized code from the Eclipse auto-completion query .......... 101
6-9 Synthesized code from the Eclipse auto-completion query for WikiText
plugin ...................................................... 102
6-10 Synthesized code from the Eclipse editor folding query .......... 104
6-11 Demonstrations of the Eclipse auto-edit feature .......... 105
6-12 Synthesized code for the Eclipse auto-edit feature .......... 106
6-13 Demonstrations of the Eclipse outline view navigation feature .... 107
6-14 Synthesized code for the content outline feature in Eclipse .......... 109
List of Tables

2.1 Instrumentation level domains ............................................. 23
2.2 Parameters to the collector runtime from the instrumented bytecode . 24

3.1 Atomic predicates in DeclView query language. For Argument(v), the corresponding predicate in SQL is ‘counter in (select counter from PARAMS where id = ??)’. ............................................. 30
3.2 Container abstractions .......................................................... 34
3.3 Abstract semantics of java.util.List methods ................................. 34

5.1 SEMERU slicing rules ............................................................. 53
5.2 Used and produced object values by event types ............................... 54
5.3 SEMERU Java-like syntax for statements. ..................................... 58

6.1 Movies demonstrations .......................................................... 70
6.2 DEMOMATCH feature extraction results for Movies ........................... 75
6.3 Passwordstore demonstrations ................................................. 77
6.4 Queries for cut and paste in Passwordstore ..................................... 79
6.5 Queries for menu action in Passwordstore ..................................... 79
6.6 Queries for table update in Passwordstore .................................... 80
6.7 Queries for list update in Passwordstore ....................................... 80
6.8 Queries for table sort in Passwordstore ....................................... 81
6.9 Queries for password generation in Passwordstore ............................ 81
6.10 Queries for undo in Passwordstore ............................................ 82
6.11 Summary of feature extraction for Passwordstore ............................. 82
6.12 Stocks demonstrations ........................................... 83
6.13 Call queries for the tooltip feature in Stocks ................. 84
6.14 Queries for toolbar D&D in Stocks ............................ 84
6.15 Queries for dialog display in Stocks ........................... 85
6.16 Queries for field validation in Stocks .......................... 85
6.17 Queries for keyboard shortcut in Stocks ........................ 86
6.18 Queries for keyboard shortcut in Stocks (re-run) ............... 86
6.19 Queries for toolbar action in Stocks ............................ 86
6.20 Queries for toolbar action in Stocks (re-run) ..................... 86
6.21 Queries for table sort in Stocks ................................. 87
6.22 Queries for table update in Stocks .............................. 87
6.23 Queries for web service connection in Stocks ..................... 88
6.24 Queries for tree selection and tree collapse actions in Stocks 88
6.25 Summary of feature extraction for Stocks ........................ 89
6.26 Call queries for the tooltip functionality ......................... 95
6.27 Demonstration traces for the Eclipse auto-completion task .... 99
6.28 Performance of the synthesis algorithm on the Eclipse auto-completion task for full traces of Eclipse plugins ................. 100
6.29 Demonstration traces for the Eclipse folding task ............... 102
6.30 Evaluation of the synthesis results for the Eclipse editor folding demonstration across full traces of the plugins ............... 104
Chapter 1

Introduction

Software libraries and frameworks are ubiquitous in modern software engineering practice. A well designed software framework delivers rich functionality for programmers to build featureful applications. However, the expressive power of the frameworks comes at a cost of the difficult initial learning curve. Mastering a framework requires understanding of the concepts comprising the framework and their interconnected programming interfaces, as well as the ability to select the right combination of its components relevant to the task at hand. The sheer size of some of these frameworks contributes to the programming challenge. For example, the rich client platform used by Eclipse has over 60 million lines of code spanning over 250 distinct open source projects [2].

In this thesis, we propose a novel method of assisting programmers in learning how to use software frameworks to solve concrete programming tasks. The foundation of our approach is the trace database called SEMERU that collects, aggregates, and analyzes execution traces of applications utilizing the framework. The database consists of millions of low-level execution event details obtained by instrumenting real world applications and recording the internal state updates and method calls. To leverage the knowledge stored in the database of program traces, we introduce several high-level query interfaces that allow a user of our tool to communicate programming intent and obtain code examples.

The research contribution of this thesis is twofold. First, we develop an architecture
for productivity tools using a database of detailed execution traces and a domain
specific query language that supports generation of code snippets. Second, building
upon the database, we introduce an informal user interaction model to search for
specific framework functionality. Together, the database and the interaction model,
hold promise in achieving the goal of reducing the programming burden.

**Query interfaces**  Given a vast amount of data in SEMERU, one big challenge that
we have to address is finding a mechanism to navigate it that is both precise and does
not require expert knowledge of the framework. The traditional solution relies on the
keyword-based search. A programmer provides a descriptive word that is matched
against code identifiers. One problem we see with this approach is that it is not
sufficiently specific when dealing with hundreds of methods in the documentation.
For example, there are over a hundred methods with “menu” in their name that are
exercised by the Swing tutorials. Previous work [32, 41, 23] recognized the importance
of ranking the result set for a keyword query as the key to usability of the code search
tools.

The more significant limitation of keyword search is that it operates purely on
syntactic terms completely missing important domain-specific knowledge and frame-
work conventions. Data and control dependencies, usage protocols, and configuration
steps are not apparent from the individual trace events matching the keywords. If the
functionality requires collaboration of several framework components, then locating
these components by the name of the feature alone is challenging. For example, to
implement keyboard shortcuts for menu actions in Swing, we use several classes for
the menu, menu items, actions, and action maps, that operate together to provide
the desired functionality. None of these classes and methods have “shortcut” in their
name.

We introduce two novel user interaction models, MATCHMAKER and DEMOMATCH,
that utilize the semantic knowledge in SEMERU traces and help programmers to
communicate their intent and to discover the relevant framework components and
keywords. Additionally, both tools are able to generate the code snippets explaining
the complex user code and framework interaction patterns that we refer to as the *glue code*.

The first query interface of SEMERU is MATCHMAKER [56]. The query consists of two framework types A and B. MATCHMAKER searches for interactions between these two types across executions. By assuming that the interactions require heap connections, SEMERU can identify the code that establishes chains of object references between instances of A and B. For example, a language plug-in for Eclipse would use both an editor and a scanner. Starting from the names of the two abstract types, MATCHMAKER identifies the code (interfaces to extend, methods to override, and statements within those methods) that bootstraps a user plug-in with a custom editor and a scanner.

For the second query interface, we rely on *live demonstration traces*, short traces of framework usage by existing applications. To obtain a demo trace, a programmer identifies the target framework functionality from existing applications. For example, a programmer might be interested in how an application utilizes Swing tables and would like to copy this functionality to their own application. Using DEMOMATCH, he/she records a short demonstration trace by manually triggering this feature of interest while SEMERU observes the application. Given the recorded trace, DEMOMATCH identifies similar functionality by matching the demo trace against all the traces in the database. Once matching executions are identified, SEMERU extracts the glue code from them that enables the feature, which can be subsequently transplanted to the user application.

DEMO MATCH was inspired by the work on programming by demonstration [25, 51], and borrows its core idea of tracing a sequence of actions and generalizing it to a program. Unlike the traditional programming by demonstration approaches, DEMOMATCH is not specialized to a particular domain and relies on the framework to provide meaning to the demonstrated user actions. DEMOMATCH observes the internals of the framework interactions, and uses the demonstration as a database query into a larger set of traces to infer the user code that imitates the same framework behavior. In particular, the instructions to set-up and bind the components together,
that are missing from the demonstration, are present in the larger traces. We develop
techniques to analyze and compare demonstrations consisting of millions of method
calls against the traces in the database.

These two applications showcase the type of programmer assistants that SEMERU
enables. SEMERU traces contain full details of the covered execution paths and permit
generation of code statements from ambiguous, high-level programmer insight. The
analysis is fully automated, requiring only manual trace collection, and scales to the
real world applications, as well as multiple versions of the language and the library
with minimal effort.

**Challenges** The main technical challenges that we solved in this thesis are con-
struction of an end-to-end system for collecting and searching a large number of trace
records from real world applications. This system supports user interaction models
for potentially imprecise and ambiguous high-level DEMOMATCH and MATCHMAKER
queries. We showed viability of this approach by implementing a prototype system
and applying it to several large frameworks.

We developed programming abstractions for analyzing program traces via a declara-
tive query language for heap connections and call sequences. We showed that SEMERU
enables construction of synthesis tools that can search and generate code from hun-
dreds of millions of trace events at interactive speeds, by implementing MATCHMAKER
and DEMOMATCH as applications utilizing these abstractions.

**Limitations** The core assumption behind our synthesis approach is that the solutions
to many programming tasks already exist but are difficult to find, extract, and adapt
in a different context. We limit the scope of our work to the style of programming that
relies on software frameworks to deliver its functionality. We believe the assumption
applies for a large class of problems in this style of programming.

The second limitation stems from our reliance on dynamic execution traces. On
the one hand, SEMERU database contains a small subset of the possible exponential
number of framework execution paths. This implies that unless a framework method
is exercised in some trace stored in the database, SEMERU cannot reason about it. But on the other hand, the collected paths are obtained from the observed and likely intended usage of the framework. Therefore, all the results computed by SEMERU are backed by concrete evidence of execution of some user code utilizing the framework.

1.1 Illustrative examples

In this section, we walk through example interactions of the programmer with our tools. In each scenario, we propose a programming task and formulate it as a query to our tool. Then, we evaluate the output and discuss its derivation and sources of data used to compute it.

Consider the problem of developing a new language editor for Eclipse. In a modular system like Eclipse, editors are plugins built on the Eclipse rich client platform. In the two scenarios that follow, we extend a blank editor plugin with two core features, syntax highlighting and auto-completion, using MATCHMAKER and DEMOMATCH, respectively.

1.1.1 MatchMaker scenario

Assume the programming task is to add syntax highlighting to a custom user editor. First, we identify the abstract type for a token scanner, which happens to be RuleBasedScanner. Then, since we already have the blank editor, we pose the MATCHMAKER query as follows:

How do I get my editor (AbstractTextEditor) and a scanner (RuleBasedScanner) work together?

MATCHMAKER automatically synthesizes the code in fig. 1-1a from this query. According to the Eclipse documentation, the solution to this development task provides a subclass of RuleBasedScanner with the code to identify and color tokens in a file. However, the editor does not use the scanner directly. Instead, the interaction is mediated by five additional classes. First, the editor interacts with a component
class UConfiguration extends SourceViewerConfiguration {
    IPresentationReconciler getPresentationReconciler() {
        PresentationReconciler reconciler =
                new PresentationReconciler();
        RuleBasedScanner userScanner =
                new UScanner();
        DefaultDamagerRepairer dr =
                new DefaultDamagerRepairer(userScanner);
        reconciler.setRepairer(dr,
                                   DEFAULTCONTENTTYPE);
        reconciler.setDamager(dr,
                                DEFAULTCONTENTTYPE);
        return reconciler;
    }
}
class UEditor extends AbstractTextEditor {
    UEditor() {
        userConfiguration = new UConfiguration();
        setSourceViewerConfiguration(userConfiguration);
    }
}
class UScanner extends RuleBasedScanner {...}

(a) User code for syntax highlighting. MATCHMAKER automatically synthesizes the text in black.

class AbstractTextEditor {
    SourceViewerConfiguration fConfiguration;
    ISourceViewer fSourceViewer;
    createPartControl() {
        fSourceViewer = createSourceViewer();
        fSourceViewer.configure(fConfiguration);
    }
    setSourceViewerConfiguration(config) {
        fConfiguration = config;
    }
}
class SourceViewer {
    IPresentationReconciler fPresentationReconciler;
    configure(SourceViewerConfiguration config) {
        fPresentationReconciler =
                config.getPresentationReconciler();
    }
}

(b) Eclipse code in AbstractTextEditor (2950 LOC) and SourceViewer (537 LOC) relevant to the interaction with a scanner.

called SourceViewer (see fig. 1-1b), which manages its add-ons. SourceViewer in turn uses PresentationReconciler to maintain a representation of the document in the presence of changes. PresentationReconciler uses an IPresentationDamager to identify changes to a document, and an IPresentationRepairer to incrementally scan those changes, and it is these two classes that interact directly with the scanner. To glue these classes together, the programmer must extend SourceViewerConfiguration and override the getPresentationReconciler method to return an instance of the PresentationReconciler class. This instance must have its IPresentationDamager and IPresentationRepairer reference the new scanner. Finally, the new SourceViewerConfiguration must be registered with the editor by calling setSourceViewerConfiguration in the constructor of the editor.

The synthesized code includes all the necessary definitions and misses just one statement from the ideal solution. To complete the glue code, the programmer can consult the documentation for the methods already present in the snippet for the description of the constant arguments and the method call sequence protocols.

To derive the glue code in fig. 1-1a, MATCHMAKER uses a complete execution trace of Eclipse in SEMERU. In order for MATCHMAKER to locate the necessary user code to facilitate the interaction, the core problem that has to be addressed is
to give semantic meaning to the query; i.e. what does it mean for two objects to “interact with each other”? We give the query a semantic interpretation by exploiting a hypothesis about the design of object-oriented frameworks.

**Hypothesis 1** *In order for two objects to interact with each other, there must be a chain of object references linking them together. Therefore, the set of actions that led to the creation of the chain is the set of actions that need to take place to enable the interaction.*

The MATCHMAKER hypothesis does not always hold; sometimes, for example, two objects can interact with each other by modifying the state of some globally shared object, without having necessarily a chain of references that connects them. Nevertheless, we have experimental evidence to suggest validity of this assumption in many cases, and our user study showed significant impact on programmer productivity [56].

Given the two base types as input, MATCHMAKER finds a chain of references in the trace database by posing a SEMERU heap connectivity query. In this case, SEMERU finds a solution in the Eclipse ANT plugin:

![Diagram of software components]

Starting from the trace events establishing each link in the chain via field assignments, SEMERU computes an execution slice from the trace and generalizes it to symbolic code shown in fig. 1-1a.

### 1.1.2 DemoMatch scenario

Continuing with the Eclipse editor example, consider the task of adding code completion functionality to our editor. For example, in Eclipse Java editor, pressing a keyboard shortcut triggers a pop-up (fig. 1-2) that lists possible ways to complete the statement.

![Figure 1-2: Eclipse JDT auto-completion pop-up]
If we attempt to solve this problem with MATCHMAKER, we have to provide the internal name for the auto-completion class (ContentAssistant). But in this case, the solution does not require a user extension of the class. Instead, the user provides an extension of IContentAssistProcessor and configures ContentAssistant to use it. DEMOMATCH helps us discover these framework types by simply demonstrating this functionality.

To use DEMOMATCH, we identify an existing implementation of the auto-complete in Eclipse Java editor and record a short demonstration trace of showing the pop-up using a start/stop toggle button. There are over 150 000 method calls inside this trace that are triggered by the pop-up. In addition to drawing the widgets, Eclipse compiles the code in the background during the demonstration. Given the demo trace, DEMOMATCH presents it to us as a ranked list of call queries, signatures of methods or sequences of methods that characterize the demonstrated functionality. We can help DEMOMATCH to isolate the key implementation method by supplying a keyword or another demonstration of the functionality in a different Eclipse editor. Given two demonstrations, DEMOMATCH correctly identifies computeCompletionProposals method call query from which it automatically generates the glue code in fig. 1-3.

To implement the auto-completion functionality in an Eclipse editor, one starts by
extending the SourceViewerConfiguration class to return a ContentAssistant. The assistant provides a IContentAssistProcessor for a source fragment (identified by the second argument to setContentAssistProcessor). The processor supplies the implementation of the method to compute an array of ICompletionProposals. Finally, the source viewer configuration is set inside the TextEditor doSetInput method.

DEMO MATCH automatically discovers all the key classes and methods that need to be extended by the user code as well as the required method calls inside the bodies. The only inputs that the user of DEMOMATCH provides are two black box demo traces. For us to be able to identify the intended framework feature and connect the demo trace and the traces in the database exercising the same feature, we make another assumption about the design of object-oriented frameworks:

**Hypothesis 2** There exists a set of methods or sequences of methods that uniquely characterize a framework feature inside its demonstration trace.

In our experience, demonstrable reactive framework features generally satisfy this requirement. However, the assumption does not always hold. For example, if the feature is implemented at a very low level of abstraction (as say a sequence of drawing commands to a rendering interpreter), then there is no distinguishing method signatures to separate the feature from the rest of the framework.

### 1.2 Overview

DEMO MATCH and MATCH MAKER are complementary tools that reason about software frameworks from different perspectives. In the case of MATCH MAKER, the heap evolution determines the insight into its behavior. For DEMOMATCH, it is the call sequences that capture the programming intent. To support both tools, we have designed a programming model and a rich query language as part of the SEMERU trace database.

We structure our thesis around the system architecture of the trace database (see fig. 1-4). We start by explaining how we obtain the trace information from
executions in chapter 2. The traces are ingested and aggregated in SEMERU database, and exposed to the tools using the trace data model and a domain specific query language implemented in Scala, as described in chapter 3. In chapter 4, we discuss our approach to the comparative analysis of traces and extraction of the framework features from demonstration traces. Chapter 5 explains the code generation algorithms that produce code snippets from individual events in the traces.

Figure 1-4: SEMERU architecture overview

In our evaluation studies in chapter 6, we have analyzed multiple real-world applications built on Swing and Eclipse platforms. For Swing, we used the official Swing tutorial [4] to accumulate a broad collection of features in the trace database. For Eclipse, we have looked at five plugins available in the Eclipse distribution to collect full traces and record demonstrations of the platform features.

Chapter 7 compares SEMERU with the relevant work. We finish with a conclusion in chapter 8.
Chapter 2

Trace collection framework

SEMERU relies on dynamic bytecode instrumentation to collect traces from live executions of Java applications. A trace consists of records for each executed instruction in the application. The instrumentation is implemented using ASM bytecode instrumentation framework [7] and Java agent facility of the Java virtual machine (JVM). We describe the modifications to the byte code in section 2.1, the dynamic runtime in section 2.2, and discuss the inherent trade-offs between the collection overhead and the quality of the analysis in section 2.3.

SEMERU approach to trace collection is motivated by a specific trade-off between the precision of the recorded data and the performance overhead of the collection. We instrument interactive applications with hard time-outs which imposes constraints on the extent of SEMERU trace collection. On the other hand, the high degree of precision is necessary for the code generation algorithm to synthesize code that is close to executable. We have designed the collection framework with a flexibility to adjust the level of the observed execution details. We use the same framework to construct the database of full execution traces and to record demonstration traces.

Prior work on tracing executions. A number of tools have been proposed for collecting full details of executions. Flashback [46], BugNet [34], liblog [14], Leap [21], R2 [17], iTarget [52], and bbr [9] focus on the problem of deterministic replay debugging with the goal of minimizing the collection overhead while preserving the root causes
of bugs. Unlike this category of tools, SEMERU aggregates traces in a database to capture common usages of the framework rather than exceptional behaviors. iDNA [6] instruction-level tracing framework enables post-execution program analysis such as time-travel debugging. However, it operates at the level of binaries, imposes high overhead, and uses sophisticated predictive compression to minimize the size of the traces. SEMERU organizes the trace data for random access patterns and efficient search. There are also tools that rely on aggregating samples of program execution. Sampling is typically done by skipping program instructions. One such technique called statistical debugging has been used for bug isolation [27]. For code generation, it is important to track all data and control dependencies which makes it undesirable to skip program events.

2.1 Bytecode instrumentation

Figure 2-1 shows the overall architecture of the tracing framework with SEMERU components highlighted. SEMERU provides a Java agent to be loaded alongside the target Java application into the Java virtual machine. SEMERU modifies bytecode by inserting instructions into the method bodies that invoke static methods from the collector runtime. To capture exceptional exit values, method bodies are wrapped into try–catch blocks and static calls to the runtime are placed into the finally block.
The agent assigns every class to one of three instrumentation domains and applies distinct transformation rules to classes from each domain. These domains are application, library, and exclusion (see table 2.1). The library classes are instrumented at the top-most level: only calls to the library and from the library are recorded and all internal field, array, and method accesses are ignored. The underlying assumption behind the lightweight tracing is that library classes are encapsulated, that is their behavior is adequately captured by the input and the output of calls from the application. This is similar to the idea of the replay interface in [17, 52] for capturing the output of environment functions to minimize recording overhead.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Instrumentation</th>
<th>Default assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>exclusion</td>
<td>none</td>
<td>native methods</td>
</tr>
<tr>
<td></td>
<td></td>
<td>arrays and primitive types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ASM and SEMERU classes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java.lang.<em>, java.security.</em> core system classes</td>
</tr>
<tr>
<td>library</td>
<td>lightweight</td>
<td>java.util.* (collections)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java.math.* (math functions)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java.sql.<em>, javax.sql.</em> (database)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>java.text.* (natural language)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>javax.xml.<em>, org.xml.</em>, org.w3c.dom.* (XML)</td>
</tr>
<tr>
<td>application</td>
<td>full</td>
<td>everything else</td>
</tr>
</tbody>
</table>

Table 2.1: Instrumentation level domains

2.2 Collector runtime

The collector runtime accepts static method calls from the instrumented application and outputs a sequential log of serialized records. An external control client communicates with the runtime over a socket and has the ability to start and stop log recording interactively. Table 2.2 lists the arguments to the runtime from the instrumentation domains. For example, for each field access bytecode instruction (e.g. getfield) in a class in the application domain, an additional method call instruction (invokestatic) is inserted by the instrumentation agent that passes the receiver value, the field identifier, and the assigned/accessed value. Below we describe the implementation details of the
serialization of these values into the log records.

<table>
<thead>
<tr>
<th>Location</th>
<th>Arguments to runtime</th>
<th>library</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method m enter</td>
<td>m, receiver, arguments</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Method m exit</td>
<td>m, exit value</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Library field f read or write</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application field f read or write</td>
<td>f, receiver, old/new value</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Array access</td>
<td>array, index</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 2.2: Parameters to the collector runtime from the instrumented bytecode

Object identity The runtime boxes all values passed from the instrumented code. Method arguments are packed into an array and passed to the static runtime method as a single parameter. The serialized form of an object is a pair of an int value of the identity hash code of an object for non-primitive types or the integer values for the primitive types combined with the hash of the actual type of the instance. Values for string objects are recorded in the output log, once per string object.

Metadata All class, field, and method signatures are encoded with MD5 hashes, which are consistent across executions and instrumented applications. These hashes are inlined in the inserted instructions in place of the field and method identifiers. Declarations are recorded separately in a metadata log. Array types are represented as pairs of the base type and the dimension. The runtime maintains a mapping from class names to its hash to avoid hashing string names repeatedly. The mapping is populated once with signatures during the instrumentation phase, but additional class names might be added during trace collection since JVM permits referencing classes that have not been loaded. The mapping of class names to hashes is represented as a high-performance Trove collection [3].

Thread call stack The runtime maintains a stack of method origin domains for every execution thread. For library calls initiated from the library domain, the runtime skips over the method enter and exit records. However, all application domain method calls from any domain are recorded, as well as all library calls from the application
domain. A small subset of the Java library is used by the agent itself. To avoid recording these calls, each stack is guarded by a boolean flag that is flipped whenever the control transfers in or out of the runtime.

**Object initialization** Super-constructor field initialization records are handled in a special way by delaying them until after the instance is fully constructed. The reasoning behind this is that the JVM bytecode verification does not permit passing un-initialized field owner instances to the runtime static methods.

**Array instructions** The trace collector records only successful array updates. This is achieved by erasing the array assignment instruction in the application bytecode, and performing the bounds check and the actual array update inside the collector runtime.

**Trace logs** The runtime outputs the trace records into binary log files for the metadata and the trace log separately. As we show in the collector performance evaluation (see section 6.1), storing the trace data to disk is the bottleneck of the collection pipeline. We offload ingestion of these log files into the relational database to a separate processing step after the instrumented application terminates.

### 2.3 Collection overhead

The general trade-off that SEMERU makes is relaxing the tracking of the control flow inside the bytecode and focusing on the interaction of the object instances. It is important to note that the local variable instructions are not recorded in the trace log, greatly reducing the amount of the log data. The reason behind this design decision is twofold. First, the target applications are interactive and excessive collection overhead would interfere with their behavior by exceeding the internal application timers commonly encountered in large applications. Second, our slicing algorithm is able to recover information about how references to objects are acquired and passed down without having the local variable assignments available in the trace.
Concurrency  SEMERU handles concurrent execution threads by serializing trace records into a single incremental log. Synchronization statements and lock management statements are not recorded. We have not found it problematic since frameworks like Eclipse manage concurrent jobs internally using thread pools and workers, relieving the user supplied code from manual concurrency control. While SEMERU supports recording of concurrent program threads, the synthesis algorithm cannot generate locking and synchronization statements.

Demonstration traces  In the DEMOMATCH setting, a demonstration captures only a fragment of the execution of a user application \( U \) utilizing framework \( F \) while the normal collection captures the whole trace from the start to the end of the execution. SEMERU uses different configurations for the two modes of collection (see fig. 2-2).

<table>
<thead>
<tr>
<th>Configuration</th>
<th>demo</th>
<th>full</th>
</tr>
</thead>
<tbody>
<tr>
<td>toggle recording</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>record method calls</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>record arrays</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>record fields</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>record strings</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>application domain</td>
<td>( F \cup U )</td>
<td>( F \cup U )</td>
</tr>
<tr>
<td>library domain</td>
<td>( F \cup U )</td>
<td>( F \cup U )</td>
</tr>
</tbody>
</table>

(a) Configurations

![Code partitioning](image)

(b) Code partitioning

Figure 2-2: Full and demonstration trace configurations

DEMOMATCH makes an assumption about the framework \( F \), that its programming interface is based on method calls and not direct state updates (via fields or arrays). Therefore, we only record method calls in the demonstration traces. To capture the internal details of the framework execution, we assign the framework packages to the application domain. However, as a performance optimization, we re-assign a small subset of the framework packages for the well-defined functionality (such as collections) to the library domain. We identify these framework parts via trial-and-error by statistical analysis of sample executions and consulting with the framework modules documentation.
Chapter 3

Trace data model

In this chapter, we describe the formal model of the execution traces and its database representation in SEMERU. We outline the structure of the Scala implementation of the core model functionality and the programming interfaces. SEMERU model and its query language were designed to support code generation and trace matching as components of the programming assistants such as DEMOMATCH.

3.1 Events

The atomic unit of an execution trace in SEMERU is an event. Events correspond to trace log records sent by the instrumentation code to the runtime, which in turn correspond to instructions in the code. The full list of trace event types is in fig. 3-1a. Each event holds a set of properties derived from the log records. In addition to the basic ones, such as the method ID for the enter events, events contain additional properties as shown in fig. 3-1b. Figure 3-1c describes the value domains of our formal model.

A trace in SEMERU is fundamentally a sequence of events ordered by counter IDs. To facilitate finding specific events and constructing sub-traces, SEMERU provides several trace views that add structure to the basic trace event sequence.

DeclView

A subset of trace events selected by a logical predicate (see section 3.2).
Here $a \in \mathcal{V}$, $b \in \mathcal{O}$, $i \in \text{int}$, $f \in \text{Field}$, $m \in \text{Method}$, $p$ is a vector over $\mathcal{V}$.

(a) Types of trace events.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>int</td>
<td>Sequential ID in the trace, initialized to 1</td>
</tr>
<tr>
<td>thread</td>
<td>long</td>
<td>Thread ID</td>
</tr>
<tr>
<td>depth</td>
<td>int</td>
<td>Stack depth</td>
</tr>
<tr>
<td>parent</td>
<td>Enter</td>
<td>Optional direct parent Enter event</td>
</tr>
<tr>
<td>succ</td>
<td>Exception</td>
<td>Optional exit event for Enter events</td>
</tr>
<tr>
<td>receiver</td>
<td>$\mathcal{O}$</td>
<td>this in Enter events or null for static</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Receiver in field events or null for static</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Array receiver in array events</td>
</tr>
<tr>
<td>value</td>
<td>$\mathcal{V}$</td>
<td>Value read from or assigned to a field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value read from or written to an array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exit value for Exit events</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value of the corresponding exit event for Enter</td>
</tr>
<tr>
<td>member</td>
<td>Method</td>
<td>Method ID for Enter events</td>
</tr>
<tr>
<td></td>
<td>Field</td>
<td>Field ID for field events</td>
</tr>
</tbody>
</table>

(b) Trace event properties

<table>
<thead>
<tr>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{E}$</td>
<td>Trace events indexed by counter</td>
</tr>
<tr>
<td>Enter</td>
<td>Enter events in $\mathcal{E}$</td>
</tr>
<tr>
<td>$\mathcal{O}$</td>
<td>Instance values including special values null and unknown</td>
</tr>
<tr>
<td>int</td>
<td>Primitive values represented by their integer values</td>
</tr>
<tr>
<td>$\mathcal{V} = \mathcal{O} \cup \text{int}$</td>
<td>Primitive and instance values</td>
</tr>
<tr>
<td>Type</td>
<td>Java types</td>
</tr>
<tr>
<td>Method</td>
<td>Java methods</td>
</tr>
<tr>
<td>Field</td>
<td>Java fields</td>
</tr>
<tr>
<td>$\mathcal{B} = (\mathcal{U}, \mathcal{F})$</td>
<td>Framework boundary of disjoint subsets $\mathcal{U} \subseteq \text{Type}$, $\mathcal{F} \subseteq \text{Type}$</td>
</tr>
</tbody>
</table>

(c) Formal model domains

Figure 3-1: Formal model of SEMERU
CallGraph

A forest of call trees organized by the caller-callee relationship (see section 3.3).

HeapSeries

A sequence of heap snapshots indexed by the event counter (see section 3.4).

A full trace supports all the views while a demonstration trace lacks HeapSeries and has a partial CallGraph of the full trace.

Ingestion In SEMERU, the log output of the instrumentation is ingested in bulk into a relational database (we use MySQL and MyISAM engine.) Metadata log is ingested first (to create IDs for Java types, fields, and methods), and the binary trace log is then converted to a tabular data file. Figure 3-2 outlines the schemata of the physical representation of events and traces. SEMERU follows the nested set model technique by adding a column succ to Enter events pointing to the corresponding Exit or Exception event. The values for succ and value are updated as a post-processing step after the initial bulk ingest (see fig. 3-2c). The counter IDs are generated by incrementing an integer counter. Note that we store the first two parameters for Enter events as a performance optimization, avoiding joins for commonly used parameters.

3.2 Declarative trace view

SEMERU offers a domain-specific query language for filtering a subset of events from the execution traces. The DSL is a boolean algebra over property-based atomic predicates (see table 3.1). The result of application of method select(q: Query) to a trace is a declarative view that fetches events on demand by translating query q to a SQL query and streaming the result set. Query operator is compositional, since select(a).select(b) is equivalent to select(a & & b). DeclView defines method foreach(f: Event => _) for iterating over events in their counter order.
CREATE TABLE 'LOG' (  'counter' int NOT NULL,  'event_type' tinyint NOT NULL,  'member' bigint NOT NULL,  'stack_depth' int NOT NULL,  'receiver' int DEFAULT NULL,  'value' int DEFAULT NULL,  'thread' bigint NOT NULL,  'parent' int DEFAULT NULL,  'param0' int DEFAULT NULL,  'param1' int DEFAULT NULL,  'succ' int DEFAULT NULL, PRIMARY KEY ('counter'))

CREATE TABLE 'PARAMS' (  'counter' int NOT NULL,  'id' int NOT NULL,  'arg' tinyint NOT NULL, PRIMARY KEY ('counter', 'arg'))

CREATE TABLE 'OBJECTS' (  'id' int NOT NULL,  'type' bigint NOT NULL,  'dims' int NOT NULL, PRIMARY KEY ('id'))

(a) Trace log

UPDATE 'LOG' L1, 'LOG' L2
SET L1.succ = L2.counter, L1.value = L2.value
WHERE L1.event_type = 1 AND L2.parent = L1.counter
AND (L2.event_type = 2 OR L2.event_type = 4)

(b) Enter parameters and object types

(c) Update operation for Enter events

Figure 3-2: Trace schemata

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member(m)</td>
<td>( m \in \text{Method} \cup \text{Field} )</td>
</tr>
<tr>
<td>Receiver(o)</td>
<td>( o \in \mathcal{O} )</td>
</tr>
<tr>
<td>Argument(v)</td>
<td>( v \in \mathcal{V} )</td>
</tr>
<tr>
<td>Value(v)</td>
<td>( v \in \mathcal{V} )</td>
</tr>
<tr>
<td>Children(e)</td>
<td>( e \in \mathcal{E} )</td>
</tr>
<tr>
<td>Thread(t)</td>
<td>( t \in \text{long} )</td>
</tr>
<tr>
<td>Depth(i)</td>
<td>( i \in \text{int} )</td>
</tr>
<tr>
<td>At(i)</td>
<td>( i \in \text{int} )</td>
</tr>
<tr>
<td>Before(e)</td>
<td>( e \in \mathcal{E} )</td>
</tr>
<tr>
<td>After(e)</td>
<td>( e \in \mathcal{E} )</td>
</tr>
<tr>
<td>Stack(e)</td>
<td>( e \in \mathcal{E} )</td>
</tr>
<tr>
<td>Enter, Exit, ...</td>
<td>matches the event type (see fig. 3-1a)</td>
</tr>
</tbody>
</table>

Table 3.1: Atomic predicates in DecView query language. For Argument(v), the corresponding predicate in SQL is 'counter in (select counter from PARAMS where id = ?)'.

30
3.3 Call graph view

A call trace organizes trace events into a forest of CallTrees per each execution thread. Nodes in a call tree are Enter events \( e_i \in E \) represented by pairs \((m, i)\) where \( m \in \text{Method} \) is the called method, and edges connect callers to their direct callees. SEMERU uses an in-memory data structure for call traces and a document index for the method names (using Lucene [1]).

The framework and the user code are specified by a pair \((U, F)\) of disjoint subsets of Type corresponding to the user and the framework classes. Members of those classes are categorized by their definition sites. The subsets do not necessarily form a partition of Type, since we allow classes to be neither, as in the case of multiple user applications co-existing in a framework platform (such as Eclipse RCP). For this reason, we refer to the pair \((U, F)\) as the framework boundary.

Let us define the cover of an event \( e \) with respect to a framework boundary \( B \) to be the top-most Enter event in the same code partition. Formally speaking,

\[
\text{cover}(e) = \begin{cases} 
\text{cover}(e, \text{parent}) & \text{if } e \in \text{Enter} \land e, \text{member} \in U \land e, \text{parent}, \text{member} \in U \\
\text{cover}(e, \text{parent}) & \text{if } e \in \text{Enter} \land e, \text{member} \in F \land e, \text{parent}, \text{member} \in F \\
e & \text{if } e \in \text{Enter} \\
\text{cover}(e, \text{parent}) & \text{if } e \notin \text{Enter} 
\end{cases}
\]

A projection of a call trace onto a framework boundary \( B \) is a call sub-trace consisting of boundary cover events only. The algorithm to compute the projection proceeds as follows:

- Remove nodes not in the user or the framework code.
- Iteratively remove intermediate users node whose parents are also user nodes and make the grandparent inherit the children.
- Iteratively remove intermediate framework nodes.

The projection of a call trace maintains the caller-callee relationship between the cover calls on the framework boundary, but eliminates internal calls in the user and
framework code. We extend these definitions to a depth metric on the call tree nodes. A node has depth 0 if it lies on the projection boundary, and depth $i$ if its parent has depth $i - 1$.

Figure 3-3a shows the call trace as an indented tree with the root at the top. Each node is annotated with the ID of the event and the execution thread. Call events for methods that reside in the user code partition are highlighted in blue. The projection sub-trace is shown on the right in fig. 3-3b.

(a) An example of the call trace representation  
(b) Projection of the call trace on the left

### 3.4 Heap series view

In addition and complementary to the call trace, SEMERU provides a view of the evolution of the program state that we refer to as HeapSeries. Abstractly, HeapSeries for a time interval $[l, h)$ is a sequence of heap snapshots $\mathcal{H}_l, \mathcal{H}_{l+1}, \ldots, \mathcal{H}_{h-1}$ where each heap snapshot $\mathcal{H}_i$ is the state of the Java heap just before event $e_i \in \mathcal{E}$. A heap snapshot is a set of triples:

$$(a, f, b) \in \mathcal{O} \times \text{Field} \times \mathcal{O}$$

denoting that the value of the non-static field $f$ of object $a$ is object $b$.

SEMERU extends the heap model with abstract fields that represent abstract relationships between objects in the heap (see section 3.4.2), such as containment relationship between a container (such as a java.util.List) and its elements. To capture multiplicities of abstract fields, the heap snapshots $\mathcal{H}_i$ are extended from sets to
multi-sets. Visually, one can think of \( \mathcal{H}_i \) as a directed multi-graph with nodes at objects \( \mathcal{O} \) and edges labelled by Field.

### 3.4.1 Heap series construction

Heap series is constructed by a sequential application of trace events to the initial empty heap snapshot. The semantics of an event \( e_i \in \mathcal{E} \) is defined by its effects on the multi-set of tuples in \( \mathcal{H}_{i-1} \):

\[
\mathcal{H}_i = \left[ e_i \right] \mathcal{H}_{i-1}
\]

Field write events \( e_i = b.f \leftarrow a \) (where \( a, b \in \mathcal{O}, \mathcal{H}_i \)) add and remove tuples rooted at \( b \) from \( \mathcal{H}_{i-1} \):

\[
\left[ b.f \leftarrow a \right] \mathcal{H}_{i-1} = \mathcal{H}_{i-1} \setminus \{(b) \times \{f\} \times \mathcal{O}\} \cup \{(b, f, a)\}
\]

whenever \( a \neq \text{null} \). If \( a \) is \( \text{null} \), then all tuples \( \{b\} \times \{f\} \times \mathcal{O} \) are removed from the heap \( \mathcal{H}_{i-1} \).

### 3.4.2 Container abstraction

SEMERU introduces an abstract field array to all array instances. The value of the field is the set of all elements in the array. SEMERU assigns HeapSeries semantics to the array write events as follows:

\[
\left[ b[j] \leftarrow a \right] \mathcal{H}_{i-1} = \mathcal{H}_{i-1} \setminus \{(b, \text{array}, c)\} \cup \{(b, \text{array}, a)\}
\]

where \( c \) is \( b[j] \) in \( \mathcal{H}_{i-1} \).

For methods in Java library that implement bulk array operations, we define abstract semantics in terms of HeapSeries. For example, utility method arraycopy copies a contiguous region of a source array into another array. SEMERU applies tuple removals and additions to \( \mathcal{H}_{i-1} \) for each call \( e_i \) of arraycopy according to its specification.
In addition to arrays, SEMERU constructs abstract models of the containers provided by Java collections frameworks by relying on method call input and output pairs (see table 3.2). For each method in the collection base interface, SEMERU provides an abstract semantics in terms of tuple updates in HeapSeries. For methods converting one type of collection into another, such as toArray, SEMERU relies on values of one abstract field (contents) to add tuples for another field (array).

<table>
<thead>
<tr>
<th>Abstract field</th>
<th>Container base type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents</td>
<td>java.util.Collection</td>
<td>multi-set of container elements</td>
</tr>
<tr>
<td></td>
<td>java.util.List</td>
<td></td>
</tr>
<tr>
<td></td>
<td>java.util.ArrayList</td>
<td></td>
</tr>
<tr>
<td>iterator</td>
<td>java.util.Iterator</td>
<td>multi-set of initial elements</td>
</tr>
<tr>
<td>values</td>
<td>java.util.Map</td>
<td>multi-set of values</td>
</tr>
<tr>
<td>array</td>
<td>java.lang.Object[]</td>
<td>multi-set of array elements</td>
</tr>
</tbody>
</table>

Table 3.2: Container abstractions

Table 3.3 shows a sample of the update rules for field contents in java.util.List in terms of multi-set operations. SEMERU scans for all concrete implementations of the interface methods provided by java.util.* and applies the rules for their Enter events.

<table>
<thead>
<tr>
<th>Event method</th>
<th>Returns</th>
<th>Application to $H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.add(Object $o$)</td>
<td>true</td>
<td>$H \cup {(l, \text{contents}, o)}$</td>
</tr>
<tr>
<td>l.add(int $i$, Object $o$)</td>
<td>normal</td>
<td>$H \cup {(l, \text{contents}, o)}$</td>
</tr>
<tr>
<td>l.addAll(Collection $l_2$)</td>
<td>normal</td>
<td>$H \cup {l_1} \times {\text{contents}} \times l_2.\text{contents}$</td>
</tr>
<tr>
<td>l.remove(Object $o$)</td>
<td>true</td>
<td>$H \setminus {(l, \text{contents}, o)}$</td>
</tr>
<tr>
<td>l.clear()</td>
<td>normal</td>
<td>$H \setminus {l} \times {\text{contents}} \times l.\text{contents}$</td>
</tr>
<tr>
<td>l.toArray()</td>
<td>$a$</td>
<td>$H \cup {a} \times {\text{array}} \times l.\text{contents}$</td>
</tr>
</tbody>
</table>

Table 3.3: Abstract semantics of java.util.List methods

The rules of the abstract container models are designed to construct an under-approximation of the concrete traces: each abstract field membership has a corresponding Enter event adding the element to the container. Note, however, that the concrete traces are not necessarily complete and are not guaranteed to contain all container updates.
3.4.3 Heap series representation

HeapSeries is represented in SEMERU as a single graph labelled by fields and time intervals $\text{Field} \times 2^{\text{int}}$. Instead of storing each heap snapshot individually, only one snapshot is stored, but each connection is indexed by the set of counter values $i$ for which $\mathcal{H}_t$ contains the connection:

$$(a, (f, T), b) \in \text{HeapSeries}$$

for

$$T = \{i \mid (a, f, b) \in \mathcal{H}_i\} \land |T| > 0$$

The heap series model and the call tree model are connected by the IDs of the events. This allows us to quickly jump from a time on some edge in HeapSeries to the call stack for the corresponding event and vice versa. Time intervals are represented either as decision trees or as unions of disjoint segments. The motivation for using time intervals instead of multiple heaps and the real gain in compacting heap series come from a simple observation: most fields are not updated frequently.

SEMERU relies on a graph database Neo4J to store HeapSeries as a graph. Queries on HeapSeries are translated into graph traversals using a method outbound which returns the list of all pairs $((f, T), b)$ for an object $a$.

3.4.4 Heap abstractions

The heap series graph is quite large: the number of nodes and edges easily reaches millions. To support efficient computation on HeapSeries, SEMERU provides several heap abstractions aimed to further reduce the size of the graph using semantic domain knowledge while preserving properties of interest. We use heap abstractions to derive approximate answers to queries which we then refine by selectively querying HeapSeries. The critical property is the heap connectivity: if two values are connected by a path in $\mathcal{H}_t$ for some $t$, then they are connected in the heap abstraction $\mathcal{A}(\text{HeapSeries})$ of HeapSeries.
In the heap series $\text{HeapSeries}$, we call a simple path

$$(a_0, (f_1, T_1), a_1), \ldots, (a_{k-1}, (f_k, T_k), a_k)$$

viable if the edge intervals share a common time. The lifetime of the path is the intersection $\cap_i T_i$ of these time intervals. Viable paths have non-empty lifetimes.

Our abstraction techniques are all based on the idea of summarizing groups of objects and groups of field connections akin to type-based field-aware static analysis [10]. The research question is what makes a good group of objects and how to merge edges in $\text{HeapSeries}$. Our requirement for the abstraction is somewhat different from static analysis since our goal is to optimize queries rather than prove properties. We use terminology of graph homomorphisms to describe our technique.

Graph homomorphisms are natural graph transformations that preserve connectedness, and they are characterized by two functions: cluster that maps nodes of $\text{HeapSeries}$ to nodes of $\mathcal{A}(\text{HeapSeries})$ and relabel that maps edge labels of $\text{HeapSeries}$ to edge labels of $\mathcal{A}(\text{HeapSeries})$. The soundness of these abstractions rests on the following property: for any viable path $a_i$ in $\text{HeapSeries}$ there exists a viable path $b_i$ in $\mathcal{A}(\text{HeapSeries})$ such that both paths have the same length and $b_i = \text{cluster}(a_i)$. The reverse is not always true: there are false candidate abstract paths that have no concrete counterparts. We call this process of taking a candidate viable path in the abstraction and attempting to find a concrete path concretization.

Since viability of paths is determined by the time intervals on the edges, relabel must respect these intervals. A simple way to ensure that is to only allow over approximation where a time interval is mapped to a larger set of times. If a group of edges is mapped to the same edge in the abstraction, then the abstract edge time interval must at the very least include the union of the concrete time intervals.

Our graph homomorphisms are composed in the order they are described. The composition respects the property as long as each individual transformation does. The quality of abstraction is the measure of how many false candidate paths are introduced. We found the following techniques effective in reducing the size of the
heap abstraction, our main goal, but having high quality of abstraction.

**Type-based abstraction** Type-based clustering is appealing since a typical heap connectivity query is concerned with the types of the end-points rather than concrete instances. Intuitively, objects of the same type play the same role in the way they interact with the rest of the heap.

This abstraction introduces many false paths for containers, for which the type carries very little information. For example, if all maps are deemed equivalent, any object pointing to a map reaches any value of any map. Therefore, we parametrize our abstraction by *exception types*, for which instances are not merged. Types that implement the collection interfaces are the only exception types in our implementation.

To summarize, type-based abstraction is defined as:

\[
\text{cluster}(o) = \begin{cases} 
  o & \text{for exception types} \\
  \text{type}(o) & \text{otherwise}
\end{cases}
\]

**Field abstraction** Edge labels in the heap series are pairs of a time interval and a field. Field abstraction strips the field from the pair, leaving only the time interval:

\[
\text{relabel}((f, T)) = T
\]

The effect is that viable paths remain viable but they lack information about fields between every two consecutive clusters. The job of the concretizer is to select fields that would connect objects belonging to the clusters of the candidate path. Multiple such fields are possible if the fields are declared in the same class and have the same value type.

**Time abstraction** The edges in the resulting abstraction are labelled with just time intervals. Conceptually, this abstraction graph is now simple since the time intervals can be merged together with the set union operation. Time abstraction amounts to expanding the merged time intervals to larger sets: \(\text{relabel}(T) = T'\), where \(T \subseteq T'\). This operation increases lifetimes of all paths, and so it preserves viable paths.
The gain comes from the smaller representation of the time intervals. We represent them with a list of disjoint half-open ranges like \([l, h)\). A frequently updated field has an irregular time interval with many such ranges. Determining viability of a path requires taking set intersection of edge time intervals along the path. When a complex time interval occurs on the path, it adversely affects the computation time. Moreover, since we are dealing with large data structures, memory representation becomes important. Complex time intervals cannot be simply represented in memory and so require more space. Time abstraction applies a simple expansion to every range \([l, h)\) in the interval: \((\lfloor \frac{l}{r} \rfloor \cdot r, \lceil \frac{h}{r} \rceil \cdot r)\) (where \(r\) is a tunable parameter). When the distance between two consecutive ranges is smaller than \(r\), they collapse into a single range.

Figure 3-4 (red) shows the number of such ranges in time intervals in a real heap series built from a full Eclipse trace. After expanding edges with more than 10 ranges with the parameter \(r = 128 \times 1024\), the edges with irregular time intervals disappear from the abstraction (blue). The edges with many ranges constitute the fat tail in the distribution (red) that adversely affects the search algorithm. The abstraction eliminates this fat tail by abstracting the edges in the tail to larger time intervals.

![Figure 3-4: Effect of the time abstraction: Y-axis is the number of edges with \(N\) or less contiguous ranges in their time intervals.](image-url)
3.5 Programming interfaces

SEMERU is implemented in 10K lines of Java and Scala code. Bytecode instrumentation is implemented in Java to minimize library dependencies, while the rest including trace ingestion and the trace model is in Scala. We utilize implicit conversions to embed the query language as a domain specific language in Scala. Figure 3-5 lists the core interface definitions of the model. Properties of trace events are represented as fields of Scala classes, and the atomic predicates of DeciView are represented as subtypes of Query.

SEMERU provides a Scalatra web interface with a Scala REPL interpreter for searching and interactive analysis of the trace data. Interactive applications, such as DEMOMATCH, are implemented on top of the web interface.

```scala
/** Trace interface */
trait Trace extends Traversable[Event] {
  /** Apply a filter to the trace */
  def select(f: Query): Trace
  /** Apply a function to every event in the order of ID */
  def foreach[U](f: Event => U)
  /** Trace size */
  def size: Int
  /** Heap graph view */
  def heaps: HeapSeries
  /** Call graph view */
  def trees: CallTrace
}

/** Declarative view */
case class DeclTrace(c: Connection, cond: Query) extends Trace
/** List of events */
case class ConcTrace(c: Connection, events: List[Event]) extends Trace
/** Call graph view */
case class CallTrace(c: Connection, roots: List[CallTree]) extends Trace
class CallTree(val method: Method, val c: Connection, val counter: Int)
  extends LinkedListTree[CallTree, Enter]
/** Heap graph view */
trait HeapSeries extends Trace with Graph[Object, HeapLabel]
  case class HeapLabel(field: Field, interval: TimeSegment)

/** Data model */
type Object = Int
trait Event extends Ordered[Event]

/** Declarative SQL query */
trait Query extends (Event => Boolean) {
  def sql: String
  def &&(that: Query): Query =
      And(this :: Nil) && that
  def ||(that: Query): Query =
      Or(this :: Nil) || that
  def unary.!(this: Query): Query =
      Not(this)
}
```

Figure 3-5: Core definitions from Scala implementation of the trace model

Here are some examples of queries and their corresponding Scala DSL query:

- Object o is returned by a call event:
  
  Enter & Value(o)

- A method is called on object o:
  
  Enter & Receiver(o)
— Object \( o \) is passed as a parameter to a call event:

\[
\text{Enter} \land \text{Argument}(o)
\]

— Entire call history of an object \( o \):

\[
\text{Enter} \land (\text{Value}(o) \lor \text{Receiver}(o) \lor \text{Argument}(o))
\]

— Direct descendants of an event \( e \):

\[
\text{Children}(e)
\]

— Indirect descendants of \( e \):

\[
\text{Thread}(e.\text{thread}) \land \text{After}(e.\text{counter}) \land \text{Before}(e.\text{succ} + 1)
\]
Chapter 4

Trace matching

In this chapter, we study the problem of matching execution traces against each other, as in the case of DEMOMATCH demonstration traces compared against SEMERU full traces. We explore an approach based on software feature extraction from the demonstration traces using lattice-based techniques, inspired by the software reconnaissance work [51, 11]. We reduce the problem of matching traces to a problem of human-assisted selection of database queries, and provide ranking metrics for the queries.

Let us assume that the user provides a demo trace $D$ with the intent of extracting the source code for the demonstrated framework features. Taking the call graph view, $D$ is a collection of call trees with some calls residing at the projection boundary – calls to the framework and callbacks to the user code. The goal of our analysis is to connect these low-level call events with the abstract and high-level notion of a software feature. This connection is what we rely on to search SEMERU database to search for traces that exercise the same feature, and from which we can extract code to replicate the functionality of the feature.

We call the signature set of events in the demonstration trace that uniquely characterizes the framework feature as the call query for the feature. Call queries prescribe patterns of method calls between the framework and the user code, that are present in the demonstration traces but can also be used as queries to match against the trace database. The core assumption that makes our analysis possible is that
the call queries exist despite the incompleteness of the demonstration traces and the informal nature of the notion of a software feature. We refer to this existence claim as the DEMOMatch assumption.

In the following section, we analyze the relationship between the framework features, the software artifacts (classes and methods) they employ, and the demonstration execution traces of these artifacts. When we think of software features, we view them as external capabilities of the framework, and not in terms of their representation as code. The challenge for our analysis is to identify the features from their demonstration traces without knowing what code is used by the feature or what the feature is.

4.1 Framework feature analysis

Framework features are implemented on top of the features of the lower-level libraries. For example, the Swing framework is layered on top of its predecessor Java AWT library. Therefore, we can analyze the behavior of an application at different levels of the software stack depth. An application that has a Swing table must also draw pixels to a AWT canvas at a lower-level, and both are valid uses of Swing and AWT frameworks.

Applications are free to implement their functionality at any level of the software stack. However, the correct depth is necessary for DEMOMatch to generate useful code snippets. It is possible to generate the complete sequence of calls to AWT canvas from a trace of a Swing table, but the resulting code carries less value to the end-user learning to use Swing. On the other hand, an arbitrary Java GUI application may choose to bypass the framework entirely. This means that it does not utilize any of the dedicated framework classes and methods, and instead invokes lower-level libraries. For example, a Java application may use a 2-D canvas for all its graphical rendering. In fact, we have observed an application drawing triangles for the table sort indicator and bypassing the native Swing table sorting functionality (see section 6.2.4). In cases like this, the projection boundary for Swing misses all the calls to AWT.

Observation 1 The projection boundary is insufficient for identifying the call queries
if the feature implementation utilizes lower-level libraries.

Note that in the alternative case of the application relying on a higher-level custom library that abstracts the usage of Swing, DEMOMATCH assists the programmer in extracting parts of the library that are relevant to the demonstration (see fig. 4-1).

![Figure 4-1: Relationship between the user code and the framework features](image)

Consider a short trace of a user typing into an editor in the Eclipse IDE. While methods related to the editor processing the input are expected to appear in the trace, the demo trace also contains the simultaneous compilation activity (if auto-building is enabled) and update manager requests. Given a trace, separating these distinct features from one another is a complex problem. In particular, while the update manager activity trace might be a coincidence, the build activity necessarily overlaps with the code input in Eclipse.

**Observation 2** Since there can be many features exercised during the demonstration, a single demonstration trace contains many candidate call queries.

In addition to the overlap of feature demonstrations, features themselves overlap in their implementations. For example, the auto-completion code uses classes and methods for the pop-up widget. Therefore, given a demonstration of the auto-completion, the code necessary for the auto-completion and the code used to draw the pop-up are both valid solutions to the DEMOMATCH query. To differentiate between the two, we propose to assign a degree of specificity to call queries reflecting the
specificity of the features they represent. For example, one can imagine the entire demonstration trace as one call query that matches perfectly the exact trace execution, but which is unlikely to match any of the complete traces due to minor variations in the executions. On the other hand, individual method signatures can be good markers for features as long as they are not shared among other features.

**Observation 3** *Call queries may correspond to multiple features, but the more specific call queries are more likely to represent the specific features in the demonstration traces.*

To summarize, each demonstration trace has many call queries, and each call query may represent a subset of features at varying levels of specificity. Therefore, to isolate a feature from its demonstration, we are seeking to isolate the most specific call queries from the demonstration trace.

### 4.2 Concept analysis

Formal concept analysis [11] provides a general framework to reason about the binary relationship between objects and their attributes. In our case, the objects are execution traces, the attributes are patterns in the call trees, and the formal concepts capture the intuitive notion of a software features. Some of the trace attributes we are interested in are:

- the definitions of all extension points (methods, interfaces, packages) for methods in the call events;
- all framework methods invoked by the trace, or the classes containing these definitions;
- the framework methods at the projection boundary, or the classes containing them;
- the types of all object instances.

The binary relationship $R$ between the set of traces $T$ and their attributes $A$ captures the description of traces as sets of overlapping attributes. This relationship $R$ has an associated formal concept lattice. To define this lattice, consider the Galois
operator \( \sigma : 2^T \rightarrow 2^A \) that produces an intersection of the set of attributes for each trace in the set \( S \):

\[
\sigma(S) = \{ a \in A \mid \forall t \in S : (t, a) \in R \}
\]

and the operator \( \tau : 2^A \rightarrow 2^T \) that produces the common set of traces for a set of program attributes \( B \):

\[
\tau(B) = \{ t \in T \mid \forall a \in B : (t, a) \in R \}.
\]

The concepts in the lattice are pairs \( (S, B) \) of traces \( S \subseteq T \) and attributes \( B \subseteq A \) that are closed under these operators:

\[
\sigma(S) = B \quad \land \quad \tau(B) = S.
\]

Intuitively, each trace in a concept \( (S, B) \) has every program attribute \( B \), and for any other trace, there is an attribute in \( B \) that the trace does not have. Similarly, for any other attribute not in \( B \), there is a trace in \( S \) that does not have it. In principle, these concepts correspond to the framework features across a set of traces. In practice, the number of the concepts for \( n \) traces grows exponentially with \( n \) in the worst case, making it difficult to navigate the entire concept lattice.

Taking the ideas of the concept analysis, we adopt a simple approach to specify a framework feature from the demonstrations. A specific framework feature typically manifests itself across several user implementations. In DEMOMATCH, the user of the tool provides the following sets of traces as input:

- **Positive traces** \( D^+ \): the set of traces exercising the same framework feature (for example, auto-completion invocation in several Eclipse editors).

- **Negative traces** \( D^- \): the set of traces that definitively do not exercise the feature (for example, a *baseline trace* is a trace in which the user does not take any action).

- **Domain traces** \( T \): the set of the framework-specific traces that hold the
superset of all feature attributes (for example, the Swing tutorials cover a large fraction of its functionality).

The program attributes that are specific to the demonstrated framework feature are then contained in the set intersection of the positive trace attributes and all domain attributes sans the attributes in the negative traces:

\[ (\sigma(D^+) \cap R(T)) \setminus R(D^-) \]

(here \( R(S) = \{ a \in A \mid \exists t \in S : (t, a) \in R \} \).)

Note that this definition does not capture conditionally specific feature attributes, that is attributes that are specific to the feature but not executed in all demo scenarios. For example, the auto-complete can be triggered via a keyboard shortcut or automatically by typing the dot character. These two cases constitute two conditionally specific behaviors for the auto-complete feature.

The shared attributes that occur in both positive and negative traces are discarded. Some of these shared attributes may potentially be relevant to the feature at hand but they lack the necessary specificity to locate the feature in the domain traces \( T \). Our slicing algorithm, however, discovers the shared attributes on which the feature depends given the specific attributes, and puts them back into the synthesized code (see section 5.1).

### 4.3 Types of call queries

Now that we set-up the concept analysis for the demonstrations and their call queries, let us formalize the types of call queries that are useful for DEMOMATCH. A call query is a pattern of method calls in the demonstration traces. These patterns should be easy to extract and enumerate from the trace call trees. For each call query type, we define a SEMERU query that performs the inverse search for events inside traces that match the pattern. In DEMOMATCH, we have two basic types of call queries for invocation and extension attributes.
\[ Q ::= \text{Invokes} (m) \mid \text{Extends} (m) \mid \text{Group} (Q, Q, \ldots) \mid \text{Nested} (Q_{\text{parent}}, Q_{\text{child}}) \]

Figure 4-2: SEMERU call queries as a formal grammar

*Invocation attribute* corresponds to a call to a framework method \( m \). The query to the SEMERU database is \( \text{Enter} \&\& \text{Member}(m) \).

*Extension attribute* corresponds to a method override by the user code of the framework method \( m \). The query to the SEMERU database is \( \text{Enter} \&\& \text{Member}(m^U) \) where \( m^U \) is the set of user methods overriding \( m \).

Basic call queries can be composed into *group* and *nested* queries (see fig. 4-2). A group query corresponds to a logical OR query for attributes. For example, similar methods belonging to the same class form a group call query for the class. Nested query is a pair of a parent query \( a \) and a child query \( b \). SEMERU resolves this query by first matching against \( a \), selecting the top-most call node \( c \) in the match, and then matching \( b \) inside the call node \( c \).

### 4.3.1 Ranking scheme

The concept analysis reduces the number of candidate call queries by computing the set intersection. At this step in the DEMOMATCH process, we rely on the human assistance to identify the specific call query from the list of the candidate call queries. We augment each attribute with a *ranking score* derived from the combination of several factors to prioritize the feature-specific call queries.

The primary factor is the *call tree depth*. This is the distance from the framework boundary projection call in the call tree. Calls on the boundary have depth 0, while calls deep inside the framework or the user code get assigned high depth. For a method \( m \), we take the minimum of the depths for all calls to \( m \).

The second factor is the *inverse document frequency (IDF)*, which is the inverse function of the number of domain traces \( T \) containing the attribute \( a \) and determines
the term specificity relative to the domain traces:

$$\log \frac{\#T}{\#\{t \in T \mid (t, a) \in R\}}$$

In combination with IDF, the call tree depth fills the role of the metric on the relative importance of the method to a trace in a trace corpus, similar to the term frequency in TF-IDF document retrieval technique [43].

We prioritize the extension attributes over the invocation attributes since the complex part of a feature typically involves callbacks from the framework to a user extension as opposed to a simpler direct library call.

Additionally, DEMOMATCH supports traditional keyword searches using the standard identifier word part and synonym analyses. A match against a keyword increases the ranking score of an attribute. We also supply method name heuristics to fine-tune the ranking scheme. The heuristics identify common design patterns from the method, namely:

- `equals` and `hashCode` are the object contract methods;
- `java.lang.*` and `java.util.*` host the common utility classes;
- `get*`, `is*`, `can*`, `has*` are all names for method getters;
- `Listener` is the standard name for the listener pattern;

### 4.3.2 Domain traces

In DEMOMATCH, the call queries are resolved by matching them against the domain traces $T$, which are full execution traces for a given framework. In cases where the framework provides a set of executable tutorials for its features, we can construct a framework feature index by collecting domain traces from these tutorials. Provided that the tutorials broadly cover the feature set, the IDF metric aids in identifying feature specificity for the call queries. Tutorials that match a demonstration trace contain sample code that exercise the same framework features and provide description of this code and the feature. Comparing executions instead of the tutorial code snippets directly from the documentation makes our analysis more precise and targeted, since
our slicing algorithms extract only the relevant parts of the tutorial code snippets.

DEMOMATCH can be applied to executions from a single application as well. In this case, the domain traces are all full traces for a given application (or platform in the case of Eclipse where all plugins share the execution environment.) The feature extraction algorithm acts as a mechanism to pinpoint the source code lines that are activated during the demonstration, and the slicing algorithm extracts the code that enables this activation. The code generation results are all versions of the feature code across executions that are invariant modulo our slicing and code simplification algorithms.

4.3.3 Query refinement

DEMOMATCH supports interactive query refinement by guided composition of Nested queries. The user of the tool provides a parent or a child part of the nested query, and DEMO MATCH restricts the set of positive attributes to the sub-trace that matches the supplied query part. For example, to restrict the call queries to the call trees rooted at all calls to method $m_1$, we supply the following scope to DEMO MATCH:

$$\text{Nested ( Invokes}(m_1), \_ )$$

This scope selects a sub-trace using a declarative query:

$$\text{select( select(Enter \\&\\& Member}(m_1)).map(_\text{contains}).reduce(_\text{or } \_ ) )$$

where contains stands for the predicate for the indirect descendants of an event.

Similarly, we can restrict attributes to the events on the stack trace for all calls to methods overriding $m_2$:

$$\text{Nested ( } _. \text{ Extends}(m_2) )$$

with the attributes extracted from a restricted trace:

$$\text{select( select(Enter \\&\\& Member}(m_2^R)).map(Stack).reduce(_\text{or } \_ ) ).$$

The refined query provides contextual information and improves specificity of the matching features in the full traces. For example, a call to the generic action invocation method actionPerformed occurs for both button clicks and keyboard shortcuts. We can specify a parent context query ($\text{Invokes}(\text{processKeyBindings})$, for example) to narrow the search down to just the keyboard shortcuts by restricting the scope down to
Extends(actionPerformed). Alternatively, we can proceed in the other direction, and start from the call query Extends(actionPerformed), and discover the call to Invokes(processKeyBindings) by restricting the scope upwards in DEMOMATCH.

The refinement process requires two selections from the user of the tool. First, the user selects a part of the nested query, and DEMOMATCH generates a set of candidate call queries that extend the selected call query. Afterwards, DEMOMATCH computes the common call queries, ranks them, and presents the list of the refined call queries for the final selection of the call query.

4.3.4 Binary attributes

In addition to the interactive process of query refinement, DEMOMATCH can automatically extract binary Nested call queries as program attributes and perform comparative analysis and scoring on them. Given a call trace, we extract invocation-under-extension attributes as follow. First, we split the trees at the user call projection boundaries. For each tree, we remove all user calls except the root user call which overrides framework method \( m \), and compute a set of nested call queries:

\[
\text{Nested} \left( \text{Extends}(m), \text{Invokes}(n) \right)
\]

where \( n \) is the framework method for any of the remaining framework call events. We assign the depth of the framework call to be the depth of the composed query, and take the minimum depth for a given query across all nodes and trees.

Note that the maximum number of the nested call queries is bounded by the number of all calls in the call trace (since for each tree partition, each node produces at most one call query, and only if it is a framework call). However, the maximum number of binary call queries is quadratic in the number of single invocation and extension attributes. The worst case happens if every framework invocation occurs under every user method extension. In practice, we have found it to not be the common case, and the number of binary attributes is comparable to the number of single attributes (see section 6.4.3).

The additional specificity of the binary attributes is useful for more complex features that are not identifiable from either the parent or the child query alone. The
main drawback of the more specific queries is the reduced number of occurrences in the full traces, especially in cases where the composition of two features is trivial in terms of the synthesized glue code.

4.4 Summary

DEMO\textsc{MATCH} reduces the problem of trace matching to a problem of human-assisted call query selection. We rely on techniques from software reconnaissance [51, 11] and information retrieval to prioritize the feature-specific call queries as the top candidate queries. We show that the human involvement is necessary since a single demonstration trace does not provide sufficient information to identify the framework features from the execution traces. In addition to the ranking metric, DEMO\textsc{MATCH} provides tools for the query refinement and binary query extraction as part of the DEMO\textsc{MATCH} user interface. In the evaluation section, we empirically test our ideas on a sample of Swing applications (see section 6.2) and conclude that the DEMO\textsc{MATCH} analysis is capable of identifying the key call queries from demonstrations without the knowledge of either the application source code or the framework.
Chapter 5

Synthesis from traces

In this chapter we describe a method of synthesizing code from SEMERU execution traces. The algorithm relies on techniques from program slicing (see section 5.1) to extract a subset of trace events necessary for reproducing program behavior, and compilation strategies (see section 5.2) to generate symbolic code from concrete trace events.

The input to SEMERU synthesis is a full execution trace with a set of goal seed events. The third important input is the framework boundary $B$ specifying the framework $F$ and the user client code $U$ in the trace. The output is a code snippet that reproduces the usage of the framework by the client modulo implementation details in the user code and in the framework, that triggers execution of the seed events. In other words, the algorithm approximates the required glue code for the framework to enable execution of the goal seed events.

SEMERU synthesis algorithm consists of the following phases applied in that order:
1. EXPAND computes a slice expansion from a set of initial seeds (see section 5.1)
2. PROJECT projects the slice onto the framework boundary $B$ (see section 5.2.1)
3. GENERATE generalizes the projected slice to code (see section 5.2.2)
4. SIMPLIFY erases redundant instructions in the code (see section 5.2.3)
5. COMBINE generalizes and aggregates code across traces (see section 5.2.4)
5.1 Dynamic slicing

Slicing is a technique for computing semantics-preserving sub-programs. In the case of dynamic traces of SEMERU, we operate on execution events and track data and inter-method control dependencies back from goal seeds. A seed is a pair \((e, o)\) of an event \(e \in E\) and an object \(o \in O\) participating in \(e\) (for example, the receiver instance of a method call). Each seed \((e, o)\) is an obligation for the algorithm to answer the following question:

\[
\text{how does object } o \text{ arrive to event } e? 
\]

Augmenting an event with an object of interest disambiguates between two kinds of dependencies:

- \textit{thin} dependency of an element in a container on the container update
- \textit{strong} dependency of an element in a container on the producer of the container reference

Starting from a set of initial seeds, EXPAND algorithm computes the fixed point of \textit{slicing rule} applications. Each slicing rule expands an individual seed to a set of its prior dependency seeds. Table 5.1 lists all rules used by SEMERU. The algorithm relies on SEMERU trace views to formulate queries to resolve the output seeds.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Computes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>local producer within the method body</td>
</tr>
<tr>
<td>Cover</td>
<td>top-level method call boundary arguments</td>
</tr>
<tr>
<td>Static</td>
<td>write to a static field</td>
</tr>
<tr>
<td>Heap</td>
<td>write to a local field</td>
</tr>
<tr>
<td>Container</td>
<td>addition of an element to a container</td>
</tr>
</tbody>
</table>

Table 5.1: SEMERU slicing rules

**Local rule** Given a seed \((e, o)\) where \(o \in O\) and \(e\) uses \(o\), the rule queries for a local producer in the same method call body (using declarative view from section 3.2):

\[
\text{Before}(e) \&\& \text{Children}(e.parent) \&\& \text{Produces}(o)
\]
The predicate $\text{Produces}(o)$ for the event producing a reference to $o$ is a disjunction of the following cases:

- **Field**: $\text{Read} \&\& \text{Value}(o)$
  
  — finds the field access

- **Array**: $\text{ArrayRead} \&\& \text{Value}(o)$
  
  — finds the array access

- **Return**: $\text{Enter} \&\& \text{Value}(o)$
  
  — finds the method call that returns $o$

- **Constructor**: $\text{Enter} \&\& \text{Receiver}(o) \&\& o.\text{typ.constructors}$
  
  — finds the constructor call of object $o$ (for the actual type of $o$)

- **Argument**: $e.\text{parent} \in \text{Argument}(o) \| \text{Receiver}(o)$
  
  — lastly, $o$ can be passed as a parameter to the parent method call of $e$

These cases are derived from the rules for the locally produced and used object instance values in table 5.2.

**SEMERU** resolves the query to the *last event* $d$ satisfying one of the above conditions. The output of the rule is the seed $(d, o)$ except for the non-container return case, in which case it is $(d.\text{succ}, o)$ (since $d$ is the call event and not the exit event.)

Since **SEMERU** does not collect local control information, the *last event* heuristics combined with aliasing may potentially result in false dependency inference. Consider
the sequence of events and the original code statements in fig. 5-1. Given the seed $(e_3, o)$, the algorithm infers the $(e_2, o)$ as the producer since both $v_1$ and $v_2$ alias the same instance $o$. Incompleteness of the tracing data is inherent to the dynamic approach of SEMERU and requires the synthesis algorithm to make the best effort in handling missing information. It is also the reason why Local rule does not follow seeds for primitive values.

<table>
<thead>
<tr>
<th>Event</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_1$: $o \leftarrow a.f$</td>
<td>$H v_1 = v_a.f$</td>
</tr>
<tr>
<td>$e_2$: $o \leftarrow b.g$</td>
<td>$H v_2 = v_b.g$</td>
</tr>
<tr>
<td>$e_3$: $c \leftarrow o.h$</td>
<td>Object $v_c = v_1.h$</td>
</tr>
</tbody>
</table>

Figure 5-1: Local slicing rule heuristics in presence of aliasing

**Cover rule** resolves directly to the interface boundary between the framework and the client code. For events deep inside the framework, it outputs the top-most call to the framework. For events residing inside the client code, it outputs the user method call that the framework performs. For the cover calls, the rule outputs the parent call. Given a seed $(e, o)$, let $d = \text{cover}(e)$. If $e$ and $d$ are not the same, the rule resolves to the seeds of $d$; otherwise, it outputs the seeds of $d.parent$ (the seeds of an Enter event consists of the receiver and all object arguments.)

**Static rule** The three remaining rules compute data dependencies. For static fields, the rule applies to the seeds of the type $(e, o)$ where $e = o \leftarrow f$, $o \in \mathcal{O}$, and the query is:

$$\text{Before}(e) \& \& \text{Write} \& \& \text{Member}(f)$$

The output are the seeds of the last event satisfying the query.

**Heap rule** The heap slicer relies on the trace HeapSeries to resolve field reads to field writes. Given a seed $(e, o)$ where $e = o \leftarrow a.f$, $o \in \mathcal{O}$, the rule finds tuples in HeapSeries using $\text{outbound}(a)$ that are active during event $e$:

$$((f, T), b) \text{ where } e.\text{counter} \in T$$
and outputs seeds of the earliest event in $T$ as well as the strong dependency on the receiver seed $(e, a)$.

**Container rules** Each container abstraction (see section 3.4.2) provides a slicing rule for the thin dependencies on the container updates. For every seed $(e, o), e \in \text{Enter}, o \in O,$ for which $[e]$ is a read of $o$ from an abstract field $f$ of $a$, the slicer finds preceding events $d$ for which $[d]$ adds tuples $(a, f, b)$ to $\text{HeapSeries}$. The result of the expansion is the seeds $(d, o)$ together with the receiver seed $(e, a)$.

### 5.2 Code generation

The result of the slicing rule expansion is a set of trace records $\{e\}$ from an individual execution trace. SEMERU code generation algorithm synthesizes code snippets from the slice and addresses the inherent challenges of generalizing concrete execution to code:

1. abstracting values to variables and holes
2. erasing user-specific and framework-specific details
3. name assignment
4. combining multiple slices into single code

The algorithm is structured as a sequence of passes: **PROJECT** (section 5.2.1), **GENERATE** (section 5.2.2), **SIMPLIFY** (section 5.2.3), and **COMBINE** (section 5.2.4).

The process starts by reducing the slice to a subset of user and top-level framework events and then generating abstract syntax from them. The derived code is subjected to simplification passes to eliminate user-specific details. Finally, **COMBINE** generates symbolic names and aligns multiple synthesis results into one.

#### 5.2.1 Projection to user events

For each event $e$ in the slice, consider its parent cover $d = \text{cover}(e.\text{parent})$. If $e$ is a non-call event produced by framework code, then $d$ is the top-level call to the framework from the user code. The algorithm **PROJECT** removes framework-internal
events like the above $e$ but keeps the top-level call in the slice. On the other hand, if $e$ is a non-call user code event, then $d$ is the top-level user call from the framework (the main function or an overriding method). In this case, user events are maintained in the slice but grouped under their parent covers $d$. Top-level framework calls $e$ are also placed under their user parent covers $d$.

Formally, PROJECT partitions the slice by the parent covers. If the parent cover call event method is not a user method, then the partition is removed. Otherwise, an event $e$ is kept in its partition at $d = \text{cover}(e.\text{parent})$ ($d.\text{member} \in \mathcal{U}$) only if one the following conditions holds:

Field events: $\text{Read}(e) \lor \text{Write}(e)$

Array events: $\text{ArrayRead}(e) \lor \text{ArrayWrite}(e)$

Top-level return: $e = d.\text{succ}$

Framework call: $\text{Enter}(e) \land e.\text{member} \in \mathcal{F}$

User constructor: $\text{Enter}(e) \land e.\text{member} \in \mathcal{U} \land e.\text{member}.\text{name} == "\text{<init>}"$

Notice that the internal calls from the user code to the user code are eliminated (except for constructors) and the framework calls are flattened. Each partition is turned into a trace by adding the partition head call $d$ and sorting events.

### 5.2.2 Generating statements from events

The algorithm GENERATE operates on individual concrete traces by translating events to instructions in a simple subset of Java language statements shown in table 5.3. The output is a method body for the partition head call method $m$.

Translation proceeds in the natural event order. The local variable environment is initialized with this and fresh argument variables for the method $m$ bound to the literal object values. For each event, a fresh variable is allocated for the produced value of the inferred declaration type, and the consumed values are replaced by the latest bound variables with the type casts inserted if necessary (see table 5.3). Primitive values and string objects are replaced by their actual values, and unbound literal objects are kept as unknown holes ??.
### Table 5.3: SEMERU Java-like syntax for statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
<th>Source event</th>
<th>Uses</th>
<th>Produces</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T \ v = w.m(w_1, \ldots); )</td>
<td>method call</td>
<td>Enter</td>
<td>( w, w_1, \ldots )</td>
<td>( v )</td>
</tr>
<tr>
<td>( T \ v = \textbf{new} \ T(w_1, \ldots); )</td>
<td>constructor call</td>
<td>Enter</td>
<td>( w_1, \ldots )</td>
<td>( v )</td>
</tr>
<tr>
<td>( T \ v = w.f; )</td>
<td>field dereference</td>
<td>Read</td>
<td>( w )</td>
<td>( v )</td>
</tr>
<tr>
<td>( T \ v = w[??]; )</td>
<td>array dereference</td>
<td>ArrayRead</td>
<td>( w )</td>
<td>( v )</td>
</tr>
<tr>
<td>( v.f = w; )</td>
<td>field assignment</td>
<td>Write</td>
<td>( v, w )</td>
<td></td>
</tr>
<tr>
<td>( v[??] = w; )</td>
<td>array assignment</td>
<td>ArrayWrite</td>
<td>( v, w )</td>
<td></td>
</tr>
<tr>
<td>return ( v; )</td>
<td>return statement</td>
<td>Exit</td>
<td>( v )</td>
<td></td>
</tr>
<tr>
<td>throw ( v; )</td>
<td>throw statement</td>
<td>Exception</td>
<td>( v )</td>
<td></td>
</tr>
<tr>
<td>( T[\ldots] \ v = \textbf{new} \ T[??]; )</td>
<td>array allocation</td>
<td>—</td>
<td>—</td>
<td>( v )</td>
</tr>
</tbody>
</table>

Here \( T \in \text{Type}, m \in \text{Method}, \text{and} f \in \text{Field} \)

The constructor calls appear in the byte code as a chain of call events to \(<\text{init}>\) methods. Before applying the translation above, the entire chain of constructor calls is replaced by the last call in place of the first call. User constructors are replaced by the default empty constructors for simplicity (since the entire user call is flattened, field initialization becomes the responsibility of the caller).

Finally, the unbound literal array objects are replaced by array variables and their allocation statements are prepended to the method body. The receiver variables are substituted by \texttt{this} and \texttt{super}.

### 5.2.3 Code simplification

At this stage, the synthesized code is a set of method bodies. A single method may have multiple bodies if the trace slice expands to multiple executions of the same statements. Moreover, the method bodies are derived by flattening the internal user calls rendering them larger and adding duplicate statements from executions of the internal user methods.

To address these issues, \textsc{Simplify} algorithm attempts to reduce the size of the synthesized code in a semantics-preserving way. The goal of this two-stage process of unrolling and compressing is that it erases \textit{user-specific} code details such as user helper methods and user-defined fields, and makes it possible to \textit{combine} the synthesized code across executions.
Algorithm *Simplify* is a fixed-point algorithm of a set of *simplification phases* each reducing the total number of statements in the method bodies. Below is the description of these phases. All except the last two operate on individual method bodies.

**Remove double-dereferences**  Since field read events are turned into field dereferences statements by default, a method body may contain two statements $T \ v_1 = w.f;$ and $T \ v_2 = w.f;$ even though the literal object value $o$ for $v_1$ and $v_2$ is the same. To remove such double dereferences, the following pass over each method body is introduced. Starting from the set $S$ of input argument variables, the pass scans every statement in order. For a field dereference, if the set $S$ already contains variables with the same literal object value, the field dereference statement is erased. Otherwise, the set $S$ is augmented with the variable produced by the statement. At the end, the erased variables are substituted by the corresponding variables in $S$.

**Remove unused return variables**  The call statements introduce a fresh variable for the return value. This pass eliminates variables that are not consumed by any of the subsequent statements.

**Remove unused constructors**  The slicing algorithm introduces constructors aggressively for all objects but the code generation phase turns them into default empty constructors and moves the initialization statements outside of the constructor. This pass simply erases constructors for values which are not used.

**Remove trivial bodies**  Method bodies that consists of the call to *super* overridden method need not be present in the code.

**Remove local containers**  This pass eliminates locally defined constructors that do not escape the local method body context. Consider the method bodies in fig. 5-2. On the left, the generated code creates two containers $a$ and $b$ holding object $o$ and then uses iterator $i$ to retrieve $o$. These containers appear since the slicing flattens the
user helper method that pass a and b as parameters. Since container references do not escape the local context in this particular case, they can be safely erased, only leaving the code on the right.

```
Object o = new Object();
Set a = new HashSet();
List b = new ArrayList();
b.add(o);
a.add(b);
Iterator i = a.iterator();
List c = i.next();
Object d = c.get(0);
return d;
```

(a) Generated code

```
Object o = new Object();
List b = new ArrayList();
b.add(o);
Object d = b.get(0);
return d;
```

(b) Code after one iteration

```
Object o = new Object();
return o;
```

(c) Simplified code

Figure 5-2: Simplification pass removes local containers from the method body

The algorithm to identify local containers that are safe to erase works as follows. We scan the statements and identify receiver variables for container constructors. We group them together with the containers that are obtained by calling getter calls such as iterator. If any of these variables escape the context by being assigned to a field, passed as parameter, or returned, we cancel the entire group. Otherwise, we erase all call and constructor statements for these container variables. In the example above, we identify two groups \{a, i\} and \{b\}. Notice that b is passed as a parameter in add, and so its group is cancelled. Therefore, the algorithm erases statements for a and i only (highlighted in fig. 5-2a). Finally, the return variables to calls to the erased containers are substituted by preceding variables with the same literal values (that is c is replaced by b.) In the next iteration, similar procedure erases the list b and only leaves the constructor of o.

**Remove owned fields** One common code pattern is adding a field to a user class to pass state between two methods. However, if after flattening, both field write and read appear in the same method body, the field is not needed. This pass eliminates field assignments if the value is not read anywhere else but the local method body.
**Remove empty bodies** Finally, after applying all these simplification passes, some method bodies have no statements remaining. In this case, the entire method is removed from the code.

### 5.2.4 Code combination

Once the set of method bodies is completed, the synthesis algorithm generates symbols for the class, method, and variable declarations. In order for multiple synthesis results for similar code to coincide, the symbols must rely only on the framework definitions as opposed to any user code specific symbols.

**Class declarations**

For each user class type reference in the method code, the algorithm computes the typing constraints from the generated code as follows:

- user-defined sub-typing on the user classes;
- methods called on, field dereferences from, and field assignments to instances of a given user type required the type to be a sub-type of the declared method and field receiver type;
- assignments to arrays of the given user type require it to be a subtype of the base type;
- return variables of the given declared user type require it to be a subtype of the method return type.

Initially, each user class type corresponds to a class declaration record extending the set of user and framework type bound from the typing constraints and holds all its method and field declarations. However, if the user code has two extensions \( U_1 \text{ extends } U_2 \) of a framework type \( F \) that split method extensions of \( F \) between each other, then type collapsing is applied. Declarations of \( U_1 \) and \( U_2 \) are merged and a new type bound is computed for the combined declaration \( U \).
Field declarations

Fields are sorted by a global sorting order using their full names, and assigned names \( f_i \) with index \( i \).

Method declarations

The generated code may contain several method bodies for a method declaration, in case the method is called multiple times in the extracted graph. We output all versions of the generated method bodies. We skip bodies that are incomplete prefixes of other versions (which happens if the method exits in the middle of its execution due to a conditional check.)

Variable declarations

Fresh local variables are named by combining upper case letters from their declared types and adding integer suffixes incremented within the body.

The result of the COMBINE pass is a set of generic class declarations and statements within the method bodies. This allows us to compare the textual output across multiple executions or across multiple seeds within the same execution. In our experimental evaluation, we evaluate the quality of the code relative to the intended tutorial code, the trace from which the code is generated, and the seeds in the trace.

5.3 Programming interfaces

SEMERO applications utilize the code synthesis algorithm by providing a set of input seeds. These seeds are extracted from the matches to the call queries in the case of DEMOMATCH, or the events establishing the chains in MATCHMAKER (see fig. 5-3). The interface to the algorithm is the following function:

```scala
// Object o participates in the event e as a producer or a consumer
case class Seed(e: Event, o: Object)
def compile(seeds: List[Seed])(p: Parameters): CodeResult
```
The applications supply parameters to the synthesis process to limit the extent of the slicing and the symbol generation:

1. bounds on the total number of expansion steps;
2. bound on the cover depth limiting the number of times Cover rule crosses the user-framework code boundary;
3. flags to control whether the primitive values and strings are replaced by holes.

The bounds on the slicing reduce the number of the generated statements and provide the applications with a simple knob to control the depth of the expansions. Without the bounds, the slicing algorithm and, consequently, the pre-simplification synthesis output is strictly monotonic on the dependency relationship between the events. (The container elimination pass may, however, eliminate more lines from larger slices if the entire object histories of containers get resolved.) In some cases, the slicing algorithm cannot distinguish between essential and accidental values. One common example of accidental values is the use of the parameter objects that are passed as arguments to method calls. These objects sometimes determine the method behavior, or in other times are completely optional. For the latter case, the bound on the expansion restricts tracking of the dependencies for the accidental parameter objects.

Figure 5-3: DEMOMATCH and MATCHMAKER extract code snippets from SEMERU traces by providing initial seeds to the slicer.
Lastly, it is important to emphasize, that the entire process of code generation is wholly \textit{data-parallel}. Each slicing expansion step operates locally near the event in the sequential trace or near the object in the heap view. The subsequent simplification passes are polynomially bounded by the sizes of the input slice dependency graphs. Therefore, the code snippets can be generated in parallel across a collection of traces and/or input seeds from the traces, and the code generation procedure is predominantly limited by the total bandwidth of the I/O access.
Chapter 6

Experimental evaluation

In this chapter, we describe our evaluation of SEMERU and DEMOMATCH. The user study for MATCHMAKER is explained in the prior work [56, 53]. We show the validity of our approach to program synthesis from execution traces and benefits of our user interaction models by applying our prototype implementation, SEMERU, to a sample of real world applications.

We start our evaluation study by addressing performance of the instrumentation and trace processing components of SEMERU. In section 6.1, we show that SEMERU can observe execution of several large applications without breaking them or causing them to time-out. We then show SEMERU can ingest large traces and process them into MySQL and Neo4J representations. This demonstrates feasibility of construction of comprehensive trace databases.

To address the central question of how much assistance DEMOMATCH provides to the programmer utilizing a framework, we break down the analysis into two parts. First, we apply our trace matching algorithms (see chapter 4) to a collection of demo traces from sample Swing applications. We validate our assumption that it is possible to identify features from short demonstration traces, and show that DEMOMATCH identifies the key feature methods as one of the top-scored call queries.

In the second part, we address the code generation component of DEMOMATCH by seeding the synthesis algorithm (see chapter 5) with a set of call features extracted from the demonstration traces. We evaluate the code quality relative to the programming
intent behind the originating demonstration traces. We also analyze robustness of the slicing algorithm to different implementations of the features across applications and executions of the same application.

The end-to-end usability of DEMOMATCH relies on extraction of the feature-specific call queries from demo traces and generation of high-quality code from these queries. The additional human guidance is restricted to selection of the call query, for which DEMOMATCH provides a web-based user interface. We show that DEMOMATCH ranking scheme selects the best call queries as one of the top scored proposals in the interface.

**Generality** The ability of DEMOMATCH to provide good answers for both Eclipse and Swing without awareness of the specific mechanisms of either framework shows that our methods generalize to two user interface Java frameworks. We have restricted our analysis to graphical applications both because it is easy to supply user actions and observe effects, and because of the complexity of the user code and framework interaction.

**Repeatability** SEMERU derives its results solely from the trace data; therefore, the computation is completely deterministic. In our examples, we examine variability of the trace executions, and the degree to which code simplification elides execution details from multiple runs of the same application, multiple feature invocations in the same run, and multiple implementations of the same feature across applications.

**Threats to validity** We have analyzed third-party applications utilizing the common frameworks that were selected without bias. Demonstrations were selected based on our external observations about what the application does without inspecting the source code. In particular, we encounter cases where the user code does not use the framework, and thus, falls out of scope of our synthesis approach.

**Experimental set-up** The experimental results are obtained on a machine with 3.1GHz Intel Core i7 CPU, 512 GB SSD drive, and 8GB of RAM allocated to SEMERU.
JVM and Neo4J. MySQL database hosting the trace data and SEMERU are co-located on the same machine.

## 6.1 Trace collector performance

We have applied the instrumentation framework to several large Java applications (Eclipse, RText, JEdit, jGnash, etc.). We have checked that the instrumented applications do not have obvious malfunctions. In the case of resilient platforms such as Eclipse, we have inspected the internal error logs.

To understand the impact of the full instrumentation, we have recorded a trace of Eclipse with the plugins from section 6.4 installed. With Eclipse under observation, we have performed typical user actions such as opening files, and editing the code in 3 minutes of execution time. All user actions succeeded within the internal time bounds.

![Figure 6-1a](image)

(a) Instrumentation output as a function of time

![Figure 6-1b](image)

(b) Execution times of the processing steps

<table>
<thead>
<tr>
<th>Processing step</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>3 min</td>
</tr>
<tr>
<td>Metadata ingestion</td>
<td>7s</td>
</tr>
<tr>
<td>Trace log ingestion</td>
<td>14 min</td>
</tr>
<tr>
<td>SQL updates and indexing</td>
<td>18 min</td>
</tr>
<tr>
<td>Heap construction</td>
<td>17 min</td>
</tr>
<tr>
<td>Heap ingestion</td>
<td>22 min</td>
</tr>
<tr>
<td>Lucene index</td>
<td>4s</td>
</tr>
</tbody>
</table>

| # events, # objects in HeapSeries    | 127M, 2M, 5M |

Figure 6-1a shows the instrumentation output as a function of execution time. The log file size increases linearly at a rate of about 40 MB per second. The resulting trace (consisting of 127 M events) is then processed by SEMERU in several steps shown in fig. 6-1b. As we can see, the processing time is significantly larger than the trace collection step. This validates our design choice to separate the trace log collection from processing as opposed to online ingestion. The high cost of processing traces amortizes across future queries and analysis.

One optimization we undertake to reduce the amount of trace data, in addition
of marking low-level packages as libraries, is to skip instrumentation of two pure
methods, Object.hashCode and Object.toString, for all method overrides. To detect
reflective invocations, we force instrumentation of Method.invoke. The resulting traces
appear to be shallow in terms of the frequencies of the events by the methods or fields
they reference. In this example trace, the most frequently accessed member is the
field HashtableOfObject.keyTable used in 2.5% of all events.

The second optimization is to delay the resolution of the fields to the processing
phase. In Java bytecode, fields are identified by their name and require the type
hierarchy to resolve to a unique field member. This is complicated by the fact that
the virtual applies instrumentation out of class sub-typing order. In SEMERU, we
process the metadata file with all declarations first, before resolving field members in
the trace file.

For demonstration traces, we apply the instrumentation at the time of loading
the byte code. Demo traces are recorded by triggering the internal instrumentation
runtime guarding the output of the event records. A typical demonstration trace does
not exceed hundreds of thousands of method calls.

Experimental data  In the rest of the chapter, we evaluate DEMOMATCH on an
instance of SEMERU database with over 200 traces, more than a billion events, and
around 300,000 methods in the metadata storage. In terms of distribution, about
a third of all events are Enter events, a third are Exit events, a quarter are Read
events, and less than 5% are of each of the remaining ArrayRead, ArrayWrite, Write, and
Exception events.

6.2 Swing feature extraction

In this section, we evaluate performance of the DEMOMATCH feature extraction and
matching for a sample of real-world Swing applications by providing answers to the
following experimental question:

- Can DEMOMATCH extract the key methods from the demo traces with respect
to the demonstrated feature? Does DEMOMATCH provide an insight into how to accomplish a programming task using these methods?

We analyze three open-source Swing applications:

1. Movies (1520 SLOC) is a movie log application (section 6.2.2).
2. Passwordstore (5494 SLOC) is a password storage application (section 6.2.3).
3. Stocks (6330 SLOC) is a stock monitor that groups stocks by their dynamically updated performance (section 6.2.4).

For each application, we identify demonstrable features and record short traces by manually triggering them with user actions. For each demo trace, we annotate the plausible programming intent, a set of framework features exercised in the demo traces that we expect DEMOMATCH to isolate. We report the quality of the call queries produced by DEMOMATCH trace matching. We omit the execution times for the computation of the queries since our implementation produces results within seconds.

### 6.2.1 Swing feature trove

We use the official Swing Tutorial [4] as the target set of traces $T$ for matching. The tutorial consists of short code projects that illustrate various aspects of the Swing widget toolkit. For each code project, we recorded a full trace by performing the user actions described on the associated documentation page. These actions consist of clicking buttons, entering text in a field, or pressing shortcut key. The total number of lines of code in the tutorials is 19 663. The size of javax.swing library itself is 191 984 SLOC, and its underlying java.awt library is 66 324 SLOC (as measured for JDK 7). The total number of the trace events for the tutorials is 252 011 903 in over 100 traces.

For the purposes of DEMOMATCH, we are only interested in the interactive features of the toolkit as opposed to the constructive features. For example, widget layout, widget styling and borders, font settings, theme settings are performed during the application start-up and cannot be easily demonstrated. Interactive features use listeners, validators, and triggers to respond to the user actions.
6.2.2 Movies application

Movies is a Swing application for keeping a movie log. The interface is a table of movie entries with columns for the title, view date, rating, and comments. We have identified several demonstrable actions listed in table 6.1. Each action has an interface trigger, an element of the interface through which the user can invoke the action. For each action, we propose a plausible programmer intent, a framework feature of interest that the demo recording is expected to showcase.

Note that as we discussed in section 4.1, the application is not required to utilize the framework to achieve its functionality. To identify cases where the application bypasses the Swing framework, we record the internal calls in javax.* and java.*. We report the number of method calls and the number of unique methods. We also include the baseline trace which does not have any action except for moving mouse around the application interface.

<table>
<thead>
<tr>
<th>#</th>
<th>Action and trigger</th>
<th>Plausible intent</th>
<th># events / methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Baseline</td>
<td></td>
<td>719093 / 896</td>
</tr>
<tr>
<td>1</td>
<td>Show the add dialog via Menu</td>
<td>Menu action</td>
<td>458680 / 2365</td>
</tr>
<tr>
<td>2</td>
<td>Add a movie via OK button</td>
<td>Button listener, Field validation, Table update</td>
<td>170637 / 1510</td>
</tr>
<tr>
<td>3</td>
<td>Show the edit dialog via Double click on a table row</td>
<td>Double click listener, Movie entry dialog</td>
<td>389975 / 2232</td>
</tr>
<tr>
<td>4</td>
<td>Enter an incorrect rating via OK button</td>
<td>Button listener, Field validation, Message dialog</td>
<td>312807 / 1863</td>
</tr>
<tr>
<td>5</td>
<td>Delete a movie via Menu</td>
<td>Menu action, Table update</td>
<td>300653 / 1592</td>
</tr>
<tr>
<td>6</td>
<td>Sort movies via Click on the column header</td>
<td>Table sort, Click listener, Table header update, Table update</td>
<td>235963 / 1032</td>
</tr>
<tr>
<td>7</td>
<td>Reorder columns via Drag-and-drop of the column header</td>
<td>D&amp;D listener, Table header update</td>
<td>419215 / 943</td>
</tr>
</tbody>
</table>

Table 6.1: Movies demonstrations

For each intent, we perform a DEMOMATCH query by selecting the traces exercising the feature as the positive demo traces, and treating the rest of the traces as the negative demo traces. In particular, the baseline traces allow us to filter the focus
activity (resulting from switching to the application at the OS level) as well as the logging activity.

The output of the DEMOMATCH query is a list of ranked call queries corresponding to invocations and extensions of the framework methods. For this study, we use a lexicographic ordering by the following metrics (starting from the highest importance):

1. Queries for extending the framework are prioritized over the invocations.
2. High IDF metric (term specificity) is preferred.
3. Low tree depth (proximity to the framework boundary) is preferred.

For framework extensions, we specify the interface definition method and ignore the IDF score. For larger result sets, we summarize the list by grouping queries by the declaration classes of the associated framework method. For each group, we choose the best score, and apply the ranking order to the groups.

**Menu action (#1, #5)** The top 10 groups of the 227 call queries are the following:

<table>
<thead>
<tr>
<th>Declaration class</th>
<th># queries</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt.Rectangle</td>
<td>2</td>
<td>30</td>
<td>4.796</td>
</tr>
<tr>
<td>java.awt.BasicStroke</td>
<td>7</td>
<td>31</td>
<td>3.186</td>
</tr>
<tr>
<td>java.awt.event.KeyEvent</td>
<td>1</td>
<td>37</td>
<td>2.85</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicMenuUI</td>
<td>13</td>
<td>23</td>
<td>2.398</td>
</tr>
<tr>
<td>javax.swing.JMenu</td>
<td>12</td>
<td>24</td>
<td>2.311</td>
</tr>
<tr>
<td>javax.swing.JMenuItem</td>
<td>6</td>
<td>13</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicMenuUI</td>
<td>5</td>
<td>22</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.event.MenuDragMouseEvent</td>
<td>1</td>
<td>24</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.JMenuBar</td>
<td>6</td>
<td>25</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.AbstractButton</td>
<td>5</td>
<td>26</td>
<td>2.231</td>
</tr>
</tbody>
</table>

The first three groups are low-level AWT classes that appear to be relatively unused in the Swing tutorials, which causes them to have high term specificity. The subsequent groups list the components of the Swing framework involved in the menu action. Of particular importance is the method `BasicMenuUI.doClick` which appears to be the key implementation method of this functionality.

**Field validation (#2, #4)** Here are all 12 call queries:

Note that the button mouse click functionality has the same positive demo traces as the field validation (see table 6.1). Therefore, the call queries for both features
appear in the results. Apart from the BasicButtonListener responsible for the button click, the key method for Movies field validation appears to be BigDecimal constructor which parses the contents of the text field (obtained via the getText methods). This constructor does not appear in the domain traces, giving it zero IDF score despite it being on the projection boundary. Swing provides means to perform form validation, but Movies is not using them and opts to bypass the framework and implement a custom way to achieve the same effect.

Table update (#2, #5, #6) DEMOMatch produces 61 call queries with the following top 8 groups:

<table>
<thead>
<tr>
<th>Declaration class</th>
<th># queries</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.swing.table.AbstractTableModel</td>
<td>2</td>
<td>0</td>
<td>4.103</td>
</tr>
<tr>
<td>javax.swing.event.TableModelEvent</td>
<td>6</td>
<td>1</td>
<td>4.103</td>
</tr>
<tr>
<td>javax.swing.table.DefaultTableColumnModel</td>
<td>2</td>
<td>3</td>
<td>3.697</td>
</tr>
<tr>
<td>javax.swing.JTable</td>
<td>22</td>
<td>2</td>
<td>2.157</td>
</tr>
<tr>
<td>javax.swing.table.TableColumn</td>
<td>4</td>
<td>19</td>
<td>2.088</td>
</tr>
<tr>
<td>javax.swing.DefaultListSelectionModel</td>
<td>3</td>
<td>4</td>
<td>2.023</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicTableUI</td>
<td>2</td>
<td>18</td>
<td>2.023</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>3</td>
<td>19</td>
<td>1.132</td>
</tr>
</tbody>
</table>

Since the supplied demo traces perform different updates (add, remove, sort), the answers we obtain are generic table model and UI updates. We find these results acceptable given that the feature of updating a table in Swing involves not only the view changes, but also the abstract model updates and event exchanges. It appears Movies uses a Vector-backed representation for the table model.

One important limitation is that DEMOMatch does not show cues for implement-
ing the table model (javax.swing.table.TableModel) as answers. This happens because the other traces read the data from the model and negate these getter methods from the results. If we supply an individual positive table update trace without the negative traces, we see extension call queries for TableModel.

**Table sort (#6)**  DEMOMATCH shows the following queries:

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends Comparator.compare</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes javax.swing.table.DefaultTableModel.abs method</td>
<td>24</td>
<td>3.186</td>
</tr>
<tr>
<td>Invokes Collections.sort</td>
<td>0</td>
<td>0.864</td>
</tr>
<tr>
<td>Invokes BigDecimal.compareTo</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Movies adds a mouse listener to the table header that performs sorting of the table model data representation. Swing provides methods for table sorting, but the application bypasses them in this case.

**Mouse click listeners (#3, #6)**  DEMOMATCH shows the correct interface method as the result and two associated mouseClicked methods:

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends MouseListener.mouseClicked</td>
<td>0</td>
<td>3.186</td>
</tr>
<tr>
<td>Invokes java.awt.AWTEventMulticaster.mouseClicked</td>
<td>22</td>
<td>0.735</td>
</tr>
<tr>
<td>Invokes java.awt.event.MouseAdapter.mouseClicked</td>
<td>24</td>
<td>0.735</td>
</tr>
</tbody>
</table>

**Dialog (#1, #3)**  DEMOMATCH shows 263 call queries with the following top 10 declaration classes:

<table>
<thead>
<tr>
<th>Declaration class</th>
<th># queries</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.swing.JDialog</td>
<td>2</td>
<td>0</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.Box</td>
<td>2</td>
<td>0</td>
<td>4.103</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicTableHeaderUI</td>
<td>31</td>
<td>2</td>
<td>3.186</td>
</tr>
<tr>
<td>java.util.Collections</td>
<td>6</td>
<td>7</td>
<td>2.599</td>
</tr>
<tr>
<td>javax.swing.text.FieldView</td>
<td>2</td>
<td>10</td>
<td>2.231</td>
</tr>
<tr>
<td>java.awt.Dialog</td>
<td>2</td>
<td>0</td>
<td>2.157</td>
</tr>
<tr>
<td>javax.swing.JTextField</td>
<td>1</td>
<td>0</td>
<td>2.088</td>
</tr>
<tr>
<td>javax.swing.text.View</td>
<td>4</td>
<td>8</td>
<td>2.088</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicLabelUI</td>
<td>1</td>
<td>14</td>
<td>1.905</td>
</tr>
<tr>
<td>javax.swing.text.JTextComponent</td>
<td>27</td>
<td>0</td>
<td>1.185</td>
</tr>
</tbody>
</table>

The first class is the correct answer to create a dialog in Swing. The large number of additional classes are used to populate the dialog with the form fields and buttons.
**Table header update (#6, #7)** DEMOMATCH shows 14 call queries in the Swing table header and column classes:

<table>
<thead>
<tr>
<th>Declaration class</th>
<th># queries</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.swing.table.DefaultTableColumnModel</td>
<td>2</td>
<td>24</td>
<td>3.004</td>
</tr>
<tr>
<td>javax.swing.table.JTableHeader</td>
<td>5</td>
<td>24</td>
<td>3.004</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicTableHeaderUI</td>
<td>5</td>
<td>24</td>
<td>3.004</td>
</tr>
<tr>
<td>javax.swing.JTable (method columnMoved)</td>
<td>1</td>
<td>26</td>
<td>3.004</td>
</tr>
<tr>
<td>javax.swing.event.TableColumnModelEvent</td>
<td>1</td>
<td>25</td>
<td>2.023</td>
</tr>
</tbody>
</table>

Notice that the mostly identical IDF score suggests that these methods are all parts of the same feature (with no separating tutorial).

**D&D listener (#7)** DEMOMATCH shows 10 call queries with the first one being the most relevant to the drag-and-drop of the table header functionality:

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes BasicTableHeaderUI$MouseInputHandler.mouseDragged</td>
<td>23</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes BasicTableHeaderUI.selectColumn</td>
<td>24</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes JTableHeader.getDraggedDistance</td>
<td>25</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes DefaultListSelectionModel.removeSelectionInterval</td>
<td>25</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes DefaultListSelectionModel.removeIndexInterval</td>
<td>25</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes DefaultListSelectionModel.insertIndexInterval</td>
<td>25</td>
<td>2.311</td>
</tr>
<tr>
<td>Invokes DefaultListSelectionModel.setIndexInterval</td>
<td>26</td>
<td>2.311</td>
</tr>
<tr>
<td>Invokes ToolTipManager.mouseDragged</td>
<td>23</td>
<td>2.088</td>
</tr>
<tr>
<td>Invokes AWTEventMulticaster.mouseDragged</td>
<td>22</td>
<td>1.66</td>
</tr>
<tr>
<td>Invokes Vector.insertElementAt</td>
<td>25</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Because dragging the table header also requires selecting it, the selection methods also appear in the demo trace.

**Message dialog (#4)** Movies shows an error message dialog upon entering an invalid rating. Here are the top 10 queries out of 188 queries:

Method showMessageDialog is the correct answer for the query. Since our ranking is biased towards IDF and this method appears in many Swing tutorials, DEMOMATCH shows it as the 5th result.

**Summary** Table 6.2 summarizes the results for Movies feature matching. For each feature, we have identified the key implementation methods in the top 10 ranked queries. We also indicate whether the application avoids using the framework, and
whether modifying the initial choice of the positive demo traces improves results (see column $D^+$).

<table>
<thead>
<tr>
<th>Feature</th>
<th>Traces</th>
<th>Key call queries</th>
<th>Bypass</th>
<th>$D^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Menu action</td>
<td>1, 5</td>
<td>\texttt{Invokes(BasicMenuItemUI.doClick)}</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Field validation</td>
<td>2, 4</td>
<td>\texttt{Invokes(BigDecimal.&lt;init&gt;)}</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Table update</td>
<td>2, 5, 6</td>
<td>\texttt{Extends(TableModel), Invokes(JTable)}</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Table sort</td>
<td>6</td>
<td>\texttt{Invokes(TableHeader.mouseClicked)}</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mouse click listeners</td>
<td>3, 6</td>
<td>\texttt{Extends(mouseClicked)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dialog</td>
<td>1, 3</td>
<td>\texttt{Invokes(JDialog.&lt;init&gt;)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table header update</td>
<td>6, 7</td>
<td>\texttt{Invokes(JTableHeader)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D&amp;D listener</td>
<td>7</td>
<td>\texttt{Invokes(TableHeader.mouseDragged)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Message dialog</td>
<td>4</td>
<td>\texttt{Invokes(JOptionPane.showMessageDialog)}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: \textsc{DemoMatch} feature extraction results for Movies

Remarks: \textsc{DemoMatch} is unable to extract methods that use arguments or the return value to enable a framework feature, because these methods appear in both positive and negative traces. For example, method \texttt{JTableHeader.setDraggedColumn} takes \texttt{null} argument to indicate no column is being dragged. Thus, a call to this method appears in traces that lack the drag-and-drop feature. Another limitation stems from the \texttt{a priori} decision not to track internal library calls that return integer types in the demo traces. This causes \textsc{DemoMatch} not to record methods such as \texttt{MouseEvent.getClickCount} that differentiate between the single and double clicks.
6.2.3 Passwordstore application

Passwordstore is a password management application. The interface consists of a list of the passwords and an information pane. Each password has fields for the host, the account, the password, and the user notes. The application shows the strength of the password as an animated histogram. A menu option switches the list view to a table view with columns for each field. Without inspecting the source code, we have demonstrated several features in this application (see table 6.3), and proposed the plausible programming intent for each demo trace.

<table>
<thead>
<tr>
<th>#</th>
<th>Action and trigger</th>
<th>Plausible intent</th>
<th># events / methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Baseline</td>
<td>–</td>
<td>773943 / 1660</td>
</tr>
<tr>
<td>1</td>
<td>Select a list item via Mouse click</td>
<td>List selection</td>
<td>2387927 / 2043</td>
</tr>
<tr>
<td>2</td>
<td>Add a list item via `n shortcut</td>
<td>Keyboard shortcut List update</td>
<td>444971 / 2490</td>
</tr>
<tr>
<td>3</td>
<td>Cut a list item via `x shortcut</td>
<td>Keyboard shortcut List update Buffer operation</td>
<td>338725 / 2487</td>
</tr>
<tr>
<td>4</td>
<td>Paste a list item via `v shortcut</td>
<td>Keyboard shortcut List update Buffer operation</td>
<td>327629 / 2343</td>
</tr>
<tr>
<td>5</td>
<td>Edit the host field in the list view via Text field</td>
<td>List update Text field edit</td>
<td>178209 / 1693</td>
</tr>
<tr>
<td>6</td>
<td>Filter the list via Filter text field</td>
<td>List update List filter</td>
<td>461551 / 2237</td>
</tr>
<tr>
<td>7</td>
<td>Switch to the table view via Menu</td>
<td>Table update Menu action</td>
<td>227214 / 2050</td>
</tr>
<tr>
<td>8</td>
<td>Sort the table view via Click on column header</td>
<td>Table update Table sort</td>
<td>373821 / 1912</td>
</tr>
<tr>
<td>9</td>
<td>Generate a password via Menu</td>
<td>Menu action List update Password generation</td>
<td>219819 / 1819</td>
</tr>
<tr>
<td>10</td>
<td>Undo cut of a list item via `z shortcut</td>
<td>Undo Buffer operation Keyboard shortcut List update</td>
<td>471447 / 2477</td>
</tr>
</tbody>
</table>

Table 6.3: Passwordstore demonstrations

Keyboard shortcut (#2, #3, #4, #10) Selecting these four traces does not produce any queries related to the shortcut keys. The reason is that the baseline
trace appears to process the application switching shortcut using the same framework feature. Therefore, we reduce the set of negative traces to a single trace #1 (list selection) and obtain the following call queries (first 10 out of 418):

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes AWTEventMulticaster.keyPressed</td>
<td>25</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes BasicListUI$Handler.keyPressed</td>
<td>26</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes BasicListUI$Handler.isNavigationKey</td>
<td>27</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes AWTEventMulticaster.keyReleased</td>
<td>25</td>
<td>3.186</td>
</tr>
<tr>
<td>Invokes BasicListUI$Handler.keyReleased</td>
<td>26</td>
<td>3.186</td>
</tr>
<tr>
<td>Invokes JMenuBar.processKeyBinding</td>
<td>28</td>
<td>2.85</td>
</tr>
<tr>
<td>Invokes JMenuBar.processBindingForKeyStrokeRecursive</td>
<td>29</td>
<td>2.85</td>
</tr>
<tr>
<td>Invokes ToolTipManager$AccessibilityKeyListener.keyPressed</td>
<td>26</td>
<td>2.311</td>
</tr>
<tr>
<td>Invokes KeyStroke.getKeyStrokeForEvent</td>
<td>27</td>
<td>2.311</td>
</tr>
<tr>
<td>Invokes KeyAdapter.keyReleased</td>
<td>26</td>
<td>1.905</td>
</tr>
</tbody>
</table>

Method JMenuBar.processKeyBinding is the key method for the shortcut feature (specifically, for the menu item shortcut). The remaining queries are key listeners inside Swing.

**List filter (#6)** DEMOMATCH fails to produce queries related to filtering of a list. Passwordstore does not use Swing features for filtering a view; instead, it relies on a custom data model backed by a collection and a simple string containment check.

**Buffer operation (#3, #4, #10)** Selecting all three traces as the positive traces produces 22 queries in classes ParagraphView, BoxView, CompositeView, etc. involved in styling, and are not related to the buffer operation. It appears that our initial judgement of assigning the feature to trace #10 (undoing the cut) is incorrect. Selecting only traces #3 and #4 produces 16 call queries shown in table 6.4.

Transferable and TransferHandler are the key classes for the cut, copy, and paste functionality in Swing. To identify the specific methods for the cut vs. the paste, we find call queries for each trace separately. DEMOMATCH successfully identifies key methods for the import and export of the buffer data separately. The additional queries in the cut trace are mostly from collection classes for managing the list model data.
Table 6.4: Queries for cut and paste in Passwordstore

**Menu action (#7, #9)** DEMOMATCH produces 48 call queries with the menu-related classes in the top 10 query classes (see table 6.5). The action extension method is not in the results, since it is shared with non-menu actions in Swing. The menu item acts like a button; hence, the button methods appear in the results.

<table>
<thead>
<tr>
<th>Declaration class for Menu Action</th>
<th># queries</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.awt.MenuComponent</td>
<td>4</td>
<td>1</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.AbstractButton</td>
<td>7</td>
<td>13</td>
<td>3.697</td>
</tr>
<tr>
<td>javax.swing.JMenu</td>
<td>6</td>
<td>10</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.DefaultButtonModel</td>
<td>3</td>
<td>11</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicMenuBarUI</td>
<td>1</td>
<td>13</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.JMenuBar</td>
<td>4</td>
<td>14</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.JMenuItem</td>
<td>1</td>
<td>13</td>
<td>1.851</td>
</tr>
<tr>
<td>javax.swing.DefaultSingleSelectionModel</td>
<td>1</td>
<td>14</td>
<td>1.269</td>
</tr>
<tr>
<td>java.awt.event.ItemEvent</td>
<td>1</td>
<td>1</td>
<td>0.753</td>
</tr>
</tbody>
</table>

Table 6.5: Queries for menu action in Passwordstore
**Table update** (#7, #8) DEMOMATCH produces top 10 (out 144) queries (see table 6.6). First choices, `TableModel`, are using the correct interface to define a data model for Swing table values and updates.

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends <code>TableModel.getColumnClass</code></td>
<td>0</td>
<td>2.493</td>
</tr>
<tr>
<td>Extends <code>TableModel.getRowCount</code></td>
<td>0</td>
<td>2.493</td>
</tr>
<tr>
<td>Extends <code>TableModel.getValueAt</code></td>
<td>0</td>
<td>2.493</td>
</tr>
<tr>
<td>Extends <code>TableModel.getColumnCount</code></td>
<td>0</td>
<td>2.493</td>
</tr>
<tr>
<td>Invokes <code>JTable.getSelectedRow</code></td>
<td>0</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes <code>DefaultRowSorter.getViewRowCount</code></td>
<td>2</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes <code>JTable$SortManager.viewSelectionChanged</code></td>
<td>4</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes <code>TableRowSorter$TableRowSorterModelWrapper.getColumnCount</code></td>
<td>6</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes <code>DefaultRowSorter.convertRowIndexToModel</code></td>
<td>1</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes <code>DefaultRowSorter.convertRowIndexToView</code></td>
<td>1</td>
<td>3.409</td>
</tr>
</tbody>
</table>

Table 6.6: Queries for table update in Passwordstore

**List update** We select traces with a list view update as the positive set (#2, #3, #4, #6) and the traces with a table view as the negative set (#7, #8). Table 6.7 shows top 10 queries out of 153. First choice tells us that the list uses a custom rendered. Second choice, `ListModel`, is the correct interface for a custom list data model in Swing. Since all our positive traces use key input and redraw the list, we obtain related queries as well.

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends <code>ListCellRenderer.getListCellRendererComponent</code></td>
<td>0</td>
<td>4.103</td>
</tr>
<tr>
<td>Extends <code>ListModel.getElementAt</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Extends <code>ListModel.getSize</code></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes <code>AffineTransform.invert</code></td>
<td>0</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes <code>BufferedImage.getProperty</code></td>
<td>1</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes <code>BufferedImage.getProperty</code></td>
<td>2</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes <code>AWTEventMulticaster keyPressed</code></td>
<td>25</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes <code>BasicListUI$Handler keyPressed</code></td>
<td>26</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes <code>BasicListUI$Handler isNavigationKey</code></td>
<td>27</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes <code>ImageIcon.&lt;init&gt;</code></td>
<td>0</td>
<td>3.186</td>
</tr>
</tbody>
</table>

Table 6.7: Queries for list update in Passwordstore

**Table sort** (#8) DEMOMATCH produces 100 call queries in the top 10 groups shown in table 6.8. The top choice is the call to the framework method `DefaultRowSorter.toggleSortOrder` (at depth 24), which is the correct key method involved in the Swing
feature. JTable appears first since its group contains a method at the projection boundary. In comparison to Movies, this application utilizes the framework feature for table sorting.

<table>
<thead>
<tr>
<th>Declaration class</th>
<th># queries</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.swing.JTable</td>
<td>18</td>
<td>0</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.DefaultRowSorter</td>
<td>17</td>
<td>24</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.SortOrder</td>
<td>3</td>
<td>25</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.RowSorter</td>
<td>6</td>
<td>25</td>
<td>4.796</td>
</tr>
<tr>
<td>java.util.Collections</td>
<td>3</td>
<td>26</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.event.RowSorterEvent</td>
<td>6</td>
<td>27</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.table.TableRowSorter</td>
<td>5</td>
<td>28</td>
<td>4.796</td>
</tr>
<tr>
<td>java.text.Collator</td>
<td>3</td>
<td>30</td>
<td>4.796</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicTableHeaderUI</td>
<td>18</td>
<td>23</td>
<td>3.186</td>
</tr>
<tr>
<td>javax.swing.table.JTableHeader</td>
<td>5</td>
<td>24</td>
<td>3.004</td>
</tr>
</tbody>
</table>

Table 6.8: Queries for table sort in Passwordstore

**Password generation (#9)** Table 6.9 lists the top call query choices (out of 17). DEMOMATCH successfully isolates calls to Random that are used to implement password generation. Menu action handling in this demo differs from trace #7, resulting in MenuItem methods present in the list.

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes MenuItem.processEvent</td>
<td>11</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes MenuItem.eventEnabled</td>
<td>11</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes MenuItem.getActionCommandImpl</td>
<td>11</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes MenuItem.processActionEvent</td>
<td>12</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes Random.seedUniquifier</td>
<td>0</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes Random.nextInt</td>
<td>0</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes Random.&lt;init&gt;</td>
<td>0</td>
<td>3.697</td>
</tr>
</tbody>
</table>

Table 6.9: Queries for password generation in Passwordstore

**Undo (#10)** Table 6.10 shows all 8 call queries. The first choice is the characteristic method for the undo functionality in Swing.

**Text field edit (#5)** Supplying this trace to DEMOMATCH gives a single query to framework method DeletePrevCharAction.actionPerformed from DefaultEditorKit. This is the exact method called on entering ‘backspace’ in a text field in Swing.
Table 6.10: Queries for undo in Passwordstore

<table>
<thead>
<tr>
<th>Query</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends UndoableEdit.undo</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes UndoManager.undo</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes AbstractUndoableEdit.undo</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes DropTarget.&lt;init&gt;</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes UndoManager.undoTo</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Invokes CompoundEdit.canRedo</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Invokes CompoundEdit.undo</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Invokes AbstractUndoableEdit.canRedo</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Summary Table 6.11 summarizes the results of DEMOMATCH queries for all demonstrated features in Passwordstore. DEMOMATCH successfully identifies the key implementation methods in the top 10 call queries for all features but one. The failure for the list filter feature is caused by the application bypassing the framework and performing the operation on the custom list model class. The remaining features utilize the framework facilities without bypassing them. For each identified key implementation method, we show the rank of the call query in the total number of queries. We also indicate whether we have modified the set of positive and negative traces in cases where the initial assignment of traces to features is insufficient (see column $D^+$).

<table>
<thead>
<tr>
<th>Feature</th>
<th>Traces</th>
<th>Rank</th>
<th>Key methods</th>
<th>$D^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard shortcut</td>
<td>2,3,4,10</td>
<td>6/418</td>
<td>Invokes (JMenuBar.processKeyBinding)</td>
<td>✓</td>
</tr>
<tr>
<td>List filter</td>
<td>6</td>
<td>70</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>Buffer operation (just paste)</td>
<td>3,4,10</td>
<td>1/16</td>
<td>Extends (Transferable.getTransferData)</td>
<td>✓</td>
</tr>
<tr>
<td>(just cut)</td>
<td></td>
<td>1/7</td>
<td>Extends (TransferHandler.importData)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1/36</td>
<td>Extends (TransferHandler.exportToClipboard)</td>
<td></td>
</tr>
<tr>
<td>Menu action</td>
<td>7,9</td>
<td>7/48</td>
<td>Invokes (JMenu.setSelected)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>8/48</td>
<td>Invokes (DefaultButtonModel.setSelected)</td>
<td></td>
</tr>
<tr>
<td>Table update</td>
<td>7,8</td>
<td>1/144</td>
<td>Extends (TableModel)</td>
<td></td>
</tr>
<tr>
<td>List update</td>
<td>2,3,4,6</td>
<td>2/153</td>
<td>Extends (ListModel)</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1/153</td>
<td>Extends (getListCellRendererComponent)</td>
<td></td>
</tr>
<tr>
<td>Table sort</td>
<td>8</td>
<td>1/100</td>
<td>Invokes (DefaultRowSorter.toggleSortOrder)</td>
<td></td>
</tr>
<tr>
<td>Password generation</td>
<td>9</td>
<td>7/17</td>
<td>Invokes (Random.&lt;init&gt;)</td>
<td></td>
</tr>
<tr>
<td>Undo</td>
<td>10</td>
<td>1/8</td>
<td>Extends (UndoableEdit.undo)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2/8</td>
<td>Invokes (UndoManager.undo)</td>
<td></td>
</tr>
<tr>
<td>Text field edit</td>
<td>5</td>
<td>1/1</td>
<td>DeletePrevCharAction.actionPerformed</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.11: Summary of feature extraction for Passwordstore
6.2.4 Stocks application

Stocks is a Swing application for monitoring stock. The interface consists of a table of stocks and a tree of filters. The stocks are updated from a web service at a regular interval. The user can load, save, or manage a stock portfolio via the toolbar buttons or the menu. Table 6.12 lists the actions, triggers, and the proposed programming intent for demonstrations. The baseline trace does not include the stock refresh activity.

<table>
<thead>
<tr>
<th>#</th>
<th>Action and trigger</th>
<th>Plausible intent</th>
<th># events / methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Baseline</td>
<td>-</td>
<td>564707 / 778</td>
</tr>
<tr>
<td>1</td>
<td>Refresh stocks <em>via</em> keyboard shortcut</td>
<td>Web service connection</td>
<td>51287 / 1261</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Keyboard shortcut</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table update</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Refresh stocks <em>via</em> toolbar button</td>
<td>Web service connection</td>
<td>123324 / 1355</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Toolbar button action</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table update</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Enter invalid stock in a text field</td>
<td>Field validation</td>
<td>174433 / 1001</td>
</tr>
<tr>
<td>4</td>
<td>Select a stock filter <em>via</em> click on a tree node</td>
<td>Tree node action</td>
<td>128249 / 1194</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table update</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Collapse stock filters <em>via</em> click on a tree folder</td>
<td>Tree collapse</td>
<td>161721 / 1349</td>
</tr>
<tr>
<td>6</td>
<td>Save dialog <em>via</em> toolbar button</td>
<td>Open dialog</td>
<td>164864 / 2190</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Toolbar button action</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Portfolio dialog <em>via</em> ^E shortcut</td>
<td>Open dialog</td>
<td>111159 / 1832</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Keyboard shortcut</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Sort stocks <em>via</em> click on table header</td>
<td>Table sort</td>
<td>142461 / 992</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table update</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Drag toolbar out of the window</td>
<td>Draggable toolbar</td>
<td>208371 / 1555</td>
</tr>
<tr>
<td>10</td>
<td>Stock name tooltip <em>via</em> click on table header</td>
<td>Table tooltip</td>
<td>154281 / 1181</td>
</tr>
<tr>
<td>11</td>
<td>Stock text field tooltip <em>via</em> click on table header</td>
<td>Text field tooltip</td>
<td>236257 / 1086</td>
</tr>
</tbody>
</table>

Table 6.12: Stocks demonstrations

**Tooltip (#10, #11)** DEMOMATCH produces 96 queries with the top 10 being highly relevant to the tooltip functionality (see table 6.13). Note that these methods appear at high tree depth, since the text field tooltip and the table tooltip have different high-level interfaces in Swing. The results for the text field tooltip demo (#11) in isolation involve classes `javax.swing.PopupFactory` and `javax.swing.Popup`. The results for
the table tooltip demo (#10) in isolation involve calls to `JTableHeader.getToolTipText`.

<table>
<thead>
<tr>
<th>Query for tooltip</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes JComponent.getTopLevelAncestor</td>
<td>26</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes ToolTipManager.checkForTipChange</td>
<td>24</td>
<td>2.85</td>
</tr>
<tr>
<td>Invokes ToolTipManager.showTipWindow</td>
<td>11</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes JToolTip.setTipText</td>
<td>12</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes SwingUtilities.windowForComponent</td>
<td>12</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes JComponent.createComponent</td>
<td>12</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes ToolTipManager.getPopupFitHeight</td>
<td>12</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes ToolTipManager.getPopupFitWidth</td>
<td>12</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes ToolTipManager.getDrawingGC</td>
<td>12</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes JTableHeader.setComponent</td>
<td>13</td>
<td>2.716</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query for table tooltip</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes JTableHeader.getToolTipText</td>
<td>24</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes JTable.getToolTipText</td>
<td>11</td>
<td>2.85</td>
</tr>
<tr>
<td>Invokes BasicSplitPaneDivider.setMouseOver</td>
<td>24</td>
<td>2.311</td>
</tr>
</tbody>
</table>

Table 6.13: Call queries for the tooltip feature in Stocks

**Toolbar D&D (#9)** DEMOMATCH produces 120 call queries with the top 10 shown in table 6.14 listing methods in `BasicToolBarUI` for mouse press and drag activity.

<table>
<thead>
<tr>
<th>Query for toolbar drag-and-drop</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes BasicToolBarUI$DragWindow.paint</td>
<td>13</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes Container.update</td>
<td>13</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes BasicToolBarUI.paintDragWindow</td>
<td>14</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes BasicToolBarUI$DragWindow.getBorderColor</td>
<td>15</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes BasicToolBarUI$DragWindow.getInsets</td>
<td>16</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes MetalToolBarUI$MetalDockingListener.mousePressed</td>
<td>22</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes BasicToolBarUI$DockingListener.mouseReleased</td>
<td>22</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes MetalToolBarUI$MetalDockingListener.mouseDragged</td>
<td>22</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes BasicToolBarUI$Handler.mouseReleased</td>
<td>23</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes MetalToolBarUI.setDragOffset</td>
<td>23</td>
<td>4.796</td>
</tr>
</tbody>
</table>

Table 6.14: Queries for toolbar D&D in Stocks

**Open dialog (#6, #7)** DEMOMATCH produces 320 results with the calls to `Dialog` as one of the top 10 choices (see table 6.15). The top-level call to `JOptionPane.showInputDialog` (used by #7) is not exercised in the tutorial suite giving it IDF score 0. The call to the constructor of `JDialog` is lower on the list since dialogs are common in the tutorials.
Table 6.15: Queries for dialog display in Stocks

Field validation (#3) DEMOMATCH produces 141 queries with the top shown in table 6.16. The extension calls are correct. The large number of queries is due to trace #3 being the only one exercising edits in a text field.

Table 6.16: Queries for field validation in Stocks

Keyboard shortcut (#1, #7) DEMOMATCH produces the queries in table 6.17. It appears that the keyboard shortcuts are also exercised in other traces and, also, keyboard shortcuts are implemented using button actions. We re-run the query with a smaller negative trace set (#5, #6, #8) and obtain results in table 6.18. DEMOMATCH correctly identifies method processKeyBinding for handling keyboard shortcuts.

Toolbar button action (#2, #6) DEMOMATCH produces 8 queries in table 6.19. If we reduce the negative trace set to just the baseline trace, we obtain the extension method used to implement the button action (see table 6.20).
### Table 6.17: Queries for keyboard shortcut in Stocks

<table>
<thead>
<tr>
<th>Query for keyboard shortcut</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes BasicMenuItemUI$Actions.actionPerformed</td>
<td>33</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes AbstractButton.doClick</td>
<td>34</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes AbstractButton.doClick</td>
<td>35</td>
<td>2.157</td>
</tr>
<tr>
<td>Invokes JComponent.paintImmediately</td>
<td>36</td>
<td>2.157</td>
</tr>
<tr>
<td>Invokes MenuSelectionManager.clearSelectedPath</td>
<td>34</td>
<td>1.538</td>
</tr>
</tbody>
</table>

### Table 6.18: Queries for keyboard shortcut in Stocks (re-run)

<table>
<thead>
<tr>
<th>Query for keyboard shortcut</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes BasicMenuItemUI$Actions.actionPerformed</td>
<td>33</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes AbstractButton.doClick</td>
<td>34</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes JMenuBar.processKeyBinding</td>
<td>27</td>
<td>2.85</td>
</tr>
<tr>
<td>Invokes JMenuBar.processBindingForKeyStrokeRecursive</td>
<td>28</td>
<td>2.85</td>
</tr>
<tr>
<td>Invokes JMenuItem.getSubElements</td>
<td>31</td>
<td>2.599</td>
</tr>
<tr>
<td>Invokes KeyStroke.getKeyStrokeForEvent</td>
<td>25</td>
<td>2.311</td>
</tr>
<tr>
<td>Invokes KeyboardManager.fireBinding</td>
<td>26</td>
<td>2.157</td>
</tr>
<tr>
<td>Invokes JMenuBar.getSubElements</td>
<td>28</td>
<td>2.157</td>
</tr>
<tr>
<td>Invokes JMenu.getComponent</td>
<td>29</td>
<td>2.157</td>
</tr>
<tr>
<td>Invokes JMenuItem.getSubElements</td>
<td>29</td>
<td>2.157</td>
</tr>
</tbody>
</table>

### Table 6.19: Queries for toolbar action in Stocks

<table>
<thead>
<tr>
<th>Declaration class</th>
<th># queries</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.swing.plaf.basic.BasicButtonUI</td>
<td>1</td>
<td>27</td>
<td>2.311</td>
</tr>
<tr>
<td>javax.swing.AbstractButton</td>
<td>2</td>
<td>24</td>
<td>2.231</td>
</tr>
<tr>
<td>javax.swing.plaf.metal.MetalButtonUI</td>
<td>2</td>
<td>26</td>
<td>0.989</td>
</tr>
<tr>
<td>javax.swing.plaf.basic.BasicButtonListener</td>
<td>2</td>
<td>23</td>
<td>0.606</td>
</tr>
<tr>
<td>javax.swing.plaf.metal.MetalUtils</td>
<td>1</td>
<td>27</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 6.20: Queries for toolbar action in Stocks (re-run)

<table>
<thead>
<tr>
<th>Query for toolbar action (total 278)</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends ActionListener.actionPerformed</td>
<td>0</td>
<td>0.653</td>
</tr>
<tr>
<td>Extends Handler.publish</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes BasicToolBarUI$Handler.focusLost</td>
<td>22</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes BasicToolBarUI$Handler.focusGained</td>
<td>22</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes JToolBar.getComponentIndex</td>
<td>23</td>
<td>3.409</td>
</tr>
</tbody>
</table>

85
Table sort (#8) DEMOMATCH produces 19 results with top 10 in table 6.21. It appears the application implements custom table sorting using a mouse listener on the table header.

<table>
<thead>
<tr>
<th>Query for table sort (total 19)</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends MouseListener.mouseClicked</td>
<td>0</td>
<td>3.186</td>
</tr>
<tr>
<td>Extends Observer.update</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes MouseAdapter.mousePressed</td>
<td>24</td>
<td>3.409</td>
</tr>
<tr>
<td>Invokes BasicTableHeaderUI$MouseInputHandler.mouseClicked</td>
<td>23</td>
<td>3.186</td>
</tr>
<tr>
<td>Invokes BasicTableHeaderUI$MouseInputHandler.mousePressed</td>
<td>23</td>
<td>3.004</td>
</tr>
<tr>
<td>Invokes BasicTableHeaderUI$MouseInputHandler.mouseReleased</td>
<td>23</td>
<td>3.004</td>
</tr>
<tr>
<td>Invokes JTableHeader.setDraggedColumn</td>
<td>24</td>
<td>3.004</td>
</tr>
<tr>
<td>Invokes JTableHeader.setDraggedDistance</td>
<td>24</td>
<td>3.004</td>
</tr>
<tr>
<td>Invokes JTableHeader.setResizingColumn</td>
<td>24</td>
<td>3.004</td>
</tr>
<tr>
<td>Invokes BasicTableHeaderUI$MouseInputHandler.setDraggedDistance</td>
<td>24</td>
<td>3.004</td>
</tr>
</tbody>
</table>

Table 6.21: Queries for table sort in Stocks

Table update (#1, #2, #4, #8) For this demo, we use only the baseline trace as the negative set since the majority of the demos update the table. DEMOMATCH gives us 317 call queries with top 10 presented in table 6.22. DEMOMATCH identifies the extension methods for defining a table model.

<table>
<thead>
<tr>
<th>Query for table update (total 317)</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends TableCellRenderer.getTableCellRendererComponent</td>
<td>0</td>
<td>4.796</td>
</tr>
<tr>
<td>Extends TableModel.getRowCount</td>
<td>0</td>
<td>2.493</td>
</tr>
<tr>
<td>Extends TableModel.getValueAt</td>
<td>0</td>
<td>2.493</td>
</tr>
<tr>
<td>Extends TableModel.getColumnCount</td>
<td>0</td>
<td>2.493</td>
</tr>
<tr>
<td>Extends DefaultTableCellRenderer.setValue</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Extends Comparator.compare</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes Collections$UnmodifiableList.get</td>
<td>0</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes Collections$UnmodifiableCollection.size</td>
<td>0</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes JTable$2.getLowerBoundAt</td>
<td>21</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes AbstractTableModel.fireTableDataChanged</td>
<td>0</td>
<td>4.103</td>
</tr>
</tbody>
</table>

Table 6.22: Queries for table update in Stocks

Web service connection (#1, #2) DEMOMATCH produces 48 queries with the top 10 shown in table 6.23. Socket communication using java.net is not shown since the instrumentation agent does not apply to packages used internally by the agent.
Table 6.23: Queries for web service connection in Stocks

Tree selection (#4) and Tree collapse (#5) Table 6.24 shows the queries for the tree selection and tree collapse actions. DEMOMATCH correctly identifies the calls to `JTree.setSelectionPath` and `JTree.collapsePath`, respectively.

<table>
<thead>
<tr>
<th>Query for refresh (total 48)</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends SwingWorker.done</td>
<td>0</td>
<td>3.697</td>
</tr>
<tr>
<td>Extends SwingWorker.doInBackground</td>
<td>0</td>
<td>3.409</td>
</tr>
<tr>
<td>Extends Runnable.run</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Invokes ProgressMonitor.close</td>
<td>1</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes ProgressMonitor.setProgress</td>
<td>1</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes ProgressMonitor.isCanceled</td>
<td>1</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes ProgressMonitor.&lt;init&gt;</td>
<td>1</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes ProgressMonitor.&lt;init&gt;</td>
<td>1</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes Pattern.compile</td>
<td>0</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes SwingWorker.execute</td>
<td>0</td>
<td>2.716</td>
</tr>
</tbody>
</table>

Table 6.24: Queries for tree selection and tree collapse actions in Stocks

<table>
<thead>
<tr>
<th>Query for tree selection (total 11)</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes BasicTreeUI$Handler.mouseDragged</td>
<td>22</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes BasicTreeUI.startEditing</td>
<td>24</td>
<td>2.85</td>
</tr>
<tr>
<td>Invokes BasicTreeUI.selectPathForEvent</td>
<td>24</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes JTree.setSelectionPath</td>
<td>25</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes BasicTreeUI.isToggleEvent</td>
<td>25</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes BasicTreeUI.isToggleSelectionEvent</td>
<td>25</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes BasicTreeUI.isMultiSelectEvent</td>
<td>25</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes DefaultTreeSelectionModel.setSelectionPath</td>
<td>26</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes JTree.getToggleClickCount</td>
<td>26</td>
<td>2.716</td>
</tr>
<tr>
<td>Invokes InputEvent.isShiftDown</td>
<td>26</td>
<td>1.132</td>
</tr>
<tr>
<td>Invokes BasicGraphicsUtils.isMenuShortcutKeyDown</td>
<td>26</td>
<td>1.132</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query for tree collapse (total 61)</th>
<th>Depth</th>
<th>IDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invokes JTree.isPathSelected</td>
<td>29</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes DefaultTreeSelectionModel.isPathSelected</td>
<td>30</td>
<td>4.796</td>
</tr>
<tr>
<td>Invokes JTree.collapsePath</td>
<td>27</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes JTree.fireTreeWillCollapse</td>
<td>29</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes JTree.fireTreeCollapsed</td>
<td>29</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes DefaultTreeSelectionModel.removeSelectionPaths</td>
<td>30</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes BasicTreeUISHandler.treeCollapsed</td>
<td>30</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes DefaultTreeSelectionModel.canPathsBeRemoved</td>
<td>31</td>
<td>4.103</td>
</tr>
<tr>
<td>Invokes VariableHeightLayoutCache$TreeStateNode.collapse</td>
<td>32</td>
<td>3.697</td>
</tr>
<tr>
<td>Invokes VariableHeightLayoutCache$TreeStateNode.collapse</td>
<td>33</td>
<td>3.697</td>
</tr>
</tbody>
</table>

Table 6.25 summarizes feature extraction results for Stocks. DEMOMATCH extracts key implementation methods relative to the programming intent for each case. In the case of table sorting, the application does not utilize the framework.
mechanism (framework bypass). The remaining features utilize framework-specific methods. In several cases, we have modified the set of positive traces or the set of negative traces due to shared attributes (indicated by column $D^+$). For each feature, we show the rank of the query in the total number of queries. For all but one case, the queries are in the top 10 proposed scored queries.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Traces</th>
<th>Rank</th>
<th>Key methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tooltip</td>
<td>10, 11</td>
<td>3/96</td>
<td>Invokes(ToolTipManager.showTipWindow)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4/96</td>
<td>Invokes(JToolTip.setTipText)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2/3</td>
<td>Invokes(JTable.getToolTipText)</td>
</tr>
<tr>
<td>Toolbar D&amp;DD</td>
<td>9</td>
<td>1/120</td>
<td>Invokes(DragWindow.paint)</td>
</tr>
<tr>
<td>Open dialog</td>
<td>6,7</td>
<td>17/320</td>
<td>Invokes(Dialog.modalShow)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>-/581</td>
<td>Invokes(JOptionPane.showInputDialog)</td>
</tr>
<tr>
<td>Field validation</td>
<td>3</td>
<td>1/141</td>
<td>Extends(InputVerifier.verify)</td>
</tr>
<tr>
<td>Keyboard shortcut</td>
<td>1,7</td>
<td>3/72</td>
<td>Invokes(JMenuBar.processKeyBinding)</td>
</tr>
<tr>
<td>Toolbar button</td>
<td>2,6</td>
<td>1/278</td>
<td>Extends(ActionListener.actionPerformed)</td>
</tr>
<tr>
<td>Table sort</td>
<td>8</td>
<td>4/19</td>
<td>Invokes(BasicTableHeaderUI.mouseClicked)</td>
</tr>
<tr>
<td>Table update</td>
<td>1,2,4,8</td>
<td>2/317</td>
<td>Extends(TableModel)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1/317</td>
<td>Extends(getTableCellRendererComponent)</td>
</tr>
<tr>
<td>Web request</td>
<td>1,2</td>
<td>2/48</td>
<td>Extends(SwingWorker.doInBackground)</td>
</tr>
<tr>
<td>Tree node action</td>
<td>4</td>
<td>4/11</td>
<td>Invokes(JTree.setSelectionPath)</td>
</tr>
<tr>
<td>Tree collapse</td>
<td>5</td>
<td>3/61</td>
<td>Invokes(JTree.collapsePath)</td>
</tr>
</tbody>
</table>

Table 6.25: Summary of feature extraction for Stocks

6.2.5 Concluding remarks

We have applied DEMOMATCH call query extraction to 28 black-box demonstrations in three third-party applications. Our results provide empirical evidence in support of DEMOMATCH hypothesis, and indicate that the lexicographic ranking score is sufficient to identify the key implementation methods as one of the top 10 proposed queries for almost all features. We show that the failure of DEMOMATCH analysis is due to one of the following reasons:

- Incorrect association between a demo trace and the programming intent requires modification to the sets of positive and negative traces.

- Application bypassing the framework leads to generic methods in the demo results list.

- Incomplete coverage of the framework features by the domain traces leads to
zero IDF score. This can be mitigated by expanding the set of domain traces.

- Framework methods that rely on the arguments or the return value to modify the framework behavior appear as common features.

### 6.3 Swing feature synthesis

In this part, we evaluate the search and synthesis components of DEMOMATCH. We take the call queries, extracted from the demonstration traces, as input, and address the following experimental question:

- Does DEMOMATCH generate useful code snippets given call queries extracted from demonstrations?

We structure our analysis as a series of case studies testing distinct aspects of the synthesis output.

#### 6.3.1 Text component example

Consider TextAreaDemo tutorial from Swing [4] demonstrating JTextArea usage by implementing simple auto-complete functionality. Whenever the user types ‘sp’, for example, the tutorial completes it to a full word by appending and highlighting ‘ecial’ immediately after the cursor (demo #1). To accept the proposed word ‘special’, the user presses the enter key (demo #2). We recorded both demonstrations, and extracted the following call queries:

1. **Nested(Extends(Runnable.run), Invokes(JTextArea.insert))**
   - for showing a proposal by inserting text;

2. **Nested(Invokes(processKeyBindings), Extends(actionPerformed))**
   - for responding to a key press.

Matching these queries against the full trace of TextAreaDemo produces two code snippets shown in fig. 6-2a (32 lines) for demo #1 and fig. 6-2b for demo #2 (18 lines). The source code for the whole tutorial consists of 152 lines of code. Both code snippets showcase two disjoint valid uses of Swing framework (the only common code

89
is the main method.) Example in fig. 6-2a adds a document listener that appends to a document on every update. Example in fig. 6-2b creates an action and adds it to an action map. The code is incomplete since the slicing misses the two lines in blue to bind the action to a key stroke due to differences in the way Enter key stroke is processed. Note, however, that a matching result for query #2 in TextFieldDemo tutorial produces the complete code in fig. 6-2b except for using JTextField instead of JTextArea.

class URunnable0 implements Runnable {
    @Override void run() {
        UDocumentListener udl =
            new UDocumentListener();
        JTextArea jta = new JTextArea();
        udl.f1 = jta;
        Document d = jta.getDocument();
        d.addDocumentListener(udl);
    }
}
class UDocumentListener
    implements DocumentListener {
    JTextArea f1;
    static void main(String[] a0) {
        URunnable0 ur = new URunnable0();
        SwingUtilities.invokeLater(ur);
    }
    @Override void insertUpdate(DocumentEvent a0) {
        URunnable ur = new URunnable();
        ur.f2 = this; ur.f3 = ?;
        SwingUtilities.invokeLater(ur);
    }
}
class URunnable implements Runnable {
    UDocumentListener f2;
    String f3;
    @Override void run() {
        UDocumentListener udl = this.f2;
        JTextArea jta = udl.f1; String s = this.f3;
        jta.insert(s, ??);
    }
}

(a) Synthesized code from TextAreaDemo for showing a proposal

class URRunnable implements Runnable {
    @Override void run() {
        JTextArea jta = new JTextArea();
        ActionMap am = jta.getActionMap();
        UAction ua = new UAction();
        am.put("id", ua);
        InputMap im = jta.getInputMap();
        im.put(KeyStroke.getKeyStroke(?), "id");
    }
}
class UObject {
    static void main(String[] a0) {
        URRunnable ur = new URRunnable();
        SwingUtilities.invokeLater(ur);
    }
}
class UAction implements Action {
    @Override
    void actionPerformed(ActionEvent a0) {
    }
}

(b) Synthesized code from TextAreaDemo for accepting a proposal

Summary The slicing and synthesis algorithms are able to extract small slices of code from the tutorial examples given feature-specific call queries. In the case of TextAreaDemo, we obtain reduction to 20% and 12% of the total tutorial code size for
two exercised features.

### 6.3.2 Field validation example

Consider the call query extracted from the field validation example in Stocks:

```
Extends(InputVerifier.verify)
```

DEMONMATCH matches this query to a single tutorial on the focus subsystem that has two example code projects. There are 15 matches to the call query. We synthesize code for each match to the call query. The resulting code snippets form 5 distinct groups, that describe two distinct behaviors:

- Figure 6-3a shows code for binding an input verifier to a text field. This code has 4 variations in the run method depending on which field gets assigned the verifier, and the statement ordering. Note that method shouldYieldFocus has two versions in the method body. This happens whenever the method is called twice in the slice graph. SEMERU does not infer the missing conditional statement that combines the two versions of the code. Note also that one variation of this code is obtained from both tutorial projects, showing that the synthesis algorithm successfully elides project-specific details.

- Figure 6-3b shows a simpler behavior, in which an action listener, that is bound to a text field, manually performs a call to verify.

**Summary** SEMERU synthesizes code for distinct behaviors from a single execution trace depending on the matching seed events. The combination algorithm effectively groups snippets even if they are derived from several executions, since the simplification pass successfully elides execution-specific details.

### 6.3.3 Cut-copy-paste examples

Consider the call query for the buffer paste feature extracted from passwordstore:

```
Extends(TransferHandler.importData)
```
class UlnputVerifier extends InputVerifier {
    @Override
    boolean shouldYieldFocus(JComponent aO) {
        aO.setInputVerifier(this);
        // --
        verify(aO);
    }
    @Override
    boolean verify(JComponent aO) {
        return true; // Example verification logic
    }
}

class URunnable implements Runnable {
    @Override
    void run() {
        JFrame jf = new JFrame();
        UContainer uc = new UContainer();
        UlnputVerifier uiv = new UlnputVerifier();
        JLabel jl = new JLabel();
        JTextField jtf = new JTextField();
        JTextField jtf0 = new JTextField();
        jl.setInputVerifier(uiv);
        jl.setForeground(jtf0);
        JPanel jp = new JPanel();
        jp.add(jtf);
        uc.add(jp); // Place the components on the container
        jf.setContentPane(uc);
        jf.pack();
    }
}

class UContainer extends Container {
    static void main(String[] aO) {
        UIManager.put(null);
        URunnable ur = new URunnable();
        SwingUtilities.invokeLater(ur);
    }
}

class UObject {
    static void main(String[] aO) {
        URunnable ur = new URunnable();
        SwingUtilities.invokeLater(ur);
    }
}

(a) Binding input validation to a text field
(32 LOC, 10% of the tutorial code)

(b) Manual field validation in a listener (21 LOC, 8% of the tutorial code)

Figure 6-3: Synthesized input validation examples from the focus subsystem tutorial

The synthesized code for this call query forms 8 distinct variants from 33 matches in 7 different tutorials on drag-and-drop. One tutorial (ListCutPaste) generates three distinct code snippets. On the other hand, the generated code is identical for several tutorials. Figure 6-4 shows the two distinct groups of variants:

- Figure 6-4a shows a simple case of binding via method setTransferHandler (see highlighted line). The difference lies in the choice of the host component.

- Figure 6-4b shows an example of manual management of cut and paste actions in Swing from ListCutPaste tutorial. In this case, the call to importData is triggered manually by the highlighted line a.actionPerformed(ae). The rest of the code establishes the action, the menu item that holds the action, and accessors for the default cut and paste actions.
class URunnable implements Runnable {
    @Override
    void run() { /* variant 1 */
        String[] s = new String[??];
        DefaultTableModel dtm = new DefaultTableModel(null, s);
        JTable jt = new JTable(dtm);
        UTransferHandler uth = new UTransferHandler();
        jt.setTransferHandler(uth);
    }
    @Override
    void run() { /* variant 2 */
        DefaultTreeModel dtm = new DefaultTreeModel(??);
        JTree jt = new JTree(dtm);
        UTransferHandler uth = new UTransferHandler();
        jt.setTransferHandler(uth);
    }
    @Override
    void run() { /* variant 3 */
        DefaultListModel dim = new DefaultListModel();
        JList jl = new JList(dim);
        UTransferHandler uth = new UTransferHandler();
        jl.setTransferHandler(uth);
    }
    @Override
    void run() { /* variant 4 */
        UTransferHandler uth = new UTransferHandler();
        JTextField jtf = new JTextField(??, ??);
        jtf.setTransferHandler(uth);
    }
    @Override
    void run() { /* variant 5 */
        UIManager.put(??, null);
        UTransferHandler uth = new UTransferHandler();
        JTextField jtf = new JTextField(??, ??);
        jtf.setTransferHandler(uth);
    }
    @Override
    void run() { /* variant 6 */
        UTransferHandler uth = new UTransferHandler();
        DefaultListModel dim = new DefaultListModel();
        JList jl = new JList(dim);
        UTransferHandler uth = new UTransferHandler();
        jl.setTransferHandler(uth);
    }
    @Override
    void run() { /* variant 7 */
        UTransferHandler uth = new UTransferHandler();
        DefaultListModel dim = new DefaultListModel();
        JList jl = new JList(dim);
        UTransferHandler uth = new UTransferHandler();
        ActionMap am = jl.getActionMap();
        Action a = TransferHandler.getCutAction();
        am.put(??, a);
        Action a0 = TransferHandler.getPasteAction();
        am.put(??, a0);
    }
}
class UActionListener implements ActionListener, PropertyChangeListener {
    JComponent fl;
    @Override
    void actionPerformed(ActionEvent ae) {
        String s = ae.getActionCommand();
        JComponent jc = this.fl;
        ActionMap am = jc.getActionMap();
        Action a = am.get(s);
        ActionEvent ae = new ActionEvent(jc, ??, null);
        a.actionPerformed(ae);
    }
    @Override
    void propertyChange(...) {
        Object o = ae.getNewValue();
        this.fl = o;
    }
}
class URunnable implements Runnable {
    @Override
    void run() {
        UIManager.put(??, null);
        JFramejf = new JFrame(??);
        UTransferHandler uth = new UTransferHandler();
        DefaultListModel dim = new DefaultListModel();
        JListjl = new JList(dim);
        UTransferHandler uth = new UTransferHandler();
        jl.setTransferHandler(uth);
        ActionMap am = jl.getActionMap();
        Action a = TransferHandler.getCutAction();
        am.put(??, a);
        Action a0 = TransferHandler.getPasteAction();
        am.put(??, a0);
        JMenuBar jmb = new JMenuBar();
        JMenu jm = new JMenu(??);
        UActionListener ual = new UActionListener();
        ActionListener aul = new ActionListener();
        KeyboardFocusManager kfm = KeyboardFocusManager.
            getCurrentKeyboardFocusManager();
        kfm.addPropertyChangeListener(??, ual);
        JMenuItem jmi = new JMenuItem(??);
        jm.add(jmi);
        JMenuItem jmi0 = new JMenuItem(??);
        jmi0.setActionCommand(??);
        jmi0.addActionListener(ual);
        jm.add(jmi0);
        jmb.add(jm);
        jf.setJMenuBar(jmb);
        jf.pack();
    }
}
class UTransferHandler extends TransferHandler {
    @Override
    boolean importData(TransferSupport aO) {
        return true;
    }
}
class UObject {
    static void main(String[] aO) {
        URunnable ur = new URunnable();
        SwingUtilities.invokeLater(ur);
    }
}

(a) Binding a transfer handler to Swing components

Figure 6-4: Synthesized buffer import examples from 7 Swing tutorials

(b) Manual management of the paste action from ListCutPaste (54 LOC, 32% of the two tutorial source files)
The call queries for the cut operation and for the combined cut-copy-paste feature are not present in the tutorial suite.

Summary SEMERU is able to match multiple tutorials for a call query and generate similar code for a wide variety of the example projects.

6.3.4Tooltip examples

The tooltip demonstrations from section 6.2.4 contain three call queries listed in table 6.26 that are related to the tooltip functionality in Swing. SEMERU identifies 126 matches for these call queries and synthesizes 25 distinct code snippets from these matches. Synthesis tasks complete in 587ms on average (and at most 4363ms for the longest task, with 14 tasks taking longer than 1s). The simplification pass reduces the snippets by 14% of the number of lines on average (up to 42% maximum).

Remark that we refined the first call query for getToolTipText to return only the calls that do not return null using the predicate Not(Values(null)), since it appears that in Swing this method is invoked whether or not the tooltip functionality is used.

<table>
<thead>
<tr>
<th>Call query</th>
<th>Matches</th>
<th>Snippets</th>
</tr>
</thead>
<tbody>
<tr>
<td>invokes(JTable.getToolTipText) (non-null)</td>
<td>86</td>
<td>2</td>
</tr>
<tr>
<td>invokes(JToolTip.setTipText)</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>invokes(ToolTipManager.showTipWindow)</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>126</strong></td>
<td><strong>25</strong></td>
</tr>
</tbody>
</table>

Table 6.26: Call queries for the tooltip functionality

Consider the synthesized code for the first query (see Figure 6-5). These code snippets are derived from two code projects in the single tutorial on how to use Swing tables \(^1\), and utilize two distinct ways to add tooltips to tables:

1. Call method setToolTipText on the default table cell renderer.
2. Extend JTable and override method getToolTipText.

The remaining forty code snippets for the two call queries are similar (pairwise) since the calls to the query methods appear in pairs. However, since method setTipText

\(^1\)https://docs.oracle.com/javase/tutorial/uiswing/components/table.html
class URunnable implements Runnable {
    @Override
    void run() {
        JFrame jf = new JFrame();
        UContainer uc = new UContainer();
        Object o = new Object();
        JFrame jt = new JFrame(o);
        JScrollPane jsp = new JScrollPane(jt);
        TableColumnModel tcm = jt.getColumnModel();
        TableColumn tc = tcm.getColumn();
        DefaultTableCellRenderer dtcr = new DefaultTableCellRenderer;
        dtcr.setToolTipText(??);
        tc.setCellRenderer(dtcr);
        uc.add(jsp);
        jf.setContentPane(uc);
        jf.pack();
    }
}

class UContainer extends Container {
    static void main(String[] a0) {
        URunnable ur = new URunnable();
        SwingUtilities.invokeLater(ur);
    }
}

(a) Two snippets from TableRenderDemo tutorial (one snippet is a strict subset of the presented code)

(b) Snippet from TableRenderDemo

Figure 6-5: Synthesized code for the table tooltip call query

takes a string as an argument, the synthesizer generates more code to compute the argument value for this method. Below we summarize what the 11 code snippets for showTipWindow call query prescribe to do and why they appear in the result set:

- 3 code snippets invoke method setToolTipText on three distinct Swing components (JToggleButton, JLabel, and JComponent) with no other code but to instantiate these components.

- 4 code snippets pass an object to the JButton constructor that holds the description of the button; Swing uses this description for the tooltip.

- 1 code snippet utilizes JColorChooser which by default provides tooltips for colors.

- 2 code snippets use JTable but no code for the tooltip is present; these snippets correspond to the table tooltip examples we described earlier.

- 1 code snippet makes a call to ToolTipManager.sharedInstance() to register a JTree; it then installs a custom renderer for the tree component, which in turns makes a call to setToolTipText when requested to render a node.

95
**Summary**  The selection of the call query in DEMOMATCH affects the synthesized output in two ways. In one case, the more specific call query shows how a feature is used in specific contexts (like in the case of the JTable tooltip). In the other case, the output shows general usage across a variety of contexts, but depending on the arguments involved in the query method the code differs slightly between the call queries.

### 6.3.5 Table sorting example

The characteristic call query for the table sorting Swing feature is:

```java
Invokes(DefaultRowSorter.toggleSortOrder)
```

There is one matching tutorial on “how to use tables” with 8 invocations of the method. The synthesized code, shown in fig. 6-6, is identical for all matches. This code constructs a table and a container for it, and enables the feature in the highlighted line. Note that the boolean argument (true) is replaced by a hole value by the general policy of hiding primitive values from the synthesized code.

```java
class URunnable implements Runnable {
    @Override void run() {
        JFrame jf = new JFrame();
        UContainer uc = new UContainer();
        Object o = new Object();
        JTable jt = new JTable(o);
        jt.setAutoCreateRowSorter();
        JScrollPane jsp = new JScrollPane(jt);
        uc.add(jsp);
        jf.setContentPane(uc);
        jf.pack();
    }
}
class UContainer extends Container {
    static void main(String[] ao) {
        URunnable ur = new URunnable();
        SwingUtilities.invokeLater(ur);
    }
}
```

Figure 6-6: Synthesized code for the table sorting feature

**Summary**  DEMOMATCH discovers the activation method for a framework feature from its demonstration call query.
6.4 End-to-end examples from Eclipse RCP

In this section, we study the quality of the synthesized code snippets for the Eclipse platform from demonstrations in existing plugins. Eclipse provides a comprehensive platform for developing rich desktop applications called Rich Client Platform (RCP). The applications are structured as collections of plugins that are connected to each other via OSGI modular system. Some of the notable RCP applications are integrated development environments (IDEs) consisting of editors, builders, debugging tools, navigation tools, and etc. In our study, we focus on the common framework functionality that Eclipse provides for constructing IDEs. We have selected the following language plugins:

1. Java development tools (JDT);
2. ANT plugin for project build files;
3. Mylyn\(^2\) task management framework and its WikiText editor;
4. PyDev\(^3\) environment for Python;
5. TeXlipse\(^4\) plugin for \LaTeX{} files.

We concentrate on four features present in some or all of these plugins: auto-completion, editor folding, auto-edit, and outline navigation. For each feature, we record demonstrations and identify the call queries using DEMOMATCH. Then we synthesize code from full traces of the Eclipse plugins. We compare the resulting code to the reference code provided by Eclipse documentation.

6.4.1 Eclipse auto-completion

Consider the development task from the introductory example (see fig. 1-3) of adding an auto-completion feature to a plugin. According to the Eclipse documentation, the glue code for this task must perform the following steps. First, the editor must extend AbstractTextEditor class which defines createSourceViewer method to instantiate a source viewer. Second, the editor sets a custom SourceViewerConfiguration, which

\(^2\)http://www.eclipse.org/mylyn/
\(^3\)http://www.pydev.org/
\(^4\)http://texlipse.sourceforge.net/
in turn declares a content assistant. This content assistant registers another user class extending IContentAssistProcessor, which contains the actual implementation to compute an array of completion proposals. We show below that DEMOMATCH is able to synthesizes this glue code from a simple call query. We have extracted variations of the code from each of the 5 test plugins.

Table 6.27 shows the number of events and unique methods for the auto-completion demo traces. Demonstration involves pressing `Space to show a completion pop-up display. We also list the number of events and methods on the framework boundary. Figure 6-7 shows the concept lattice Hasse diagram of call queries for these 5 traces against a baseline Eclipse. The top-most node contains 31 call queries common to all 5 demo traces, and which does not appear in the baseline trace. Levels below show concepts with 4, 3, and 2 traces, respectively, annotated by the number of the call queries. The total number of concepts (18 out of maximum 26) in the diagram indicates significant differences in the implementation of this functionality across plugins.

<table>
<thead>
<tr>
<th>Demo plugin</th>
<th># events / methods</th>
<th># B events / B methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDT</td>
<td>277521 / 2700</td>
<td>9751 / 255</td>
</tr>
<tr>
<td>ANT</td>
<td>125675 / 1051</td>
<td>11319 / 105</td>
</tr>
<tr>
<td>PyDev</td>
<td>107039 / 1666</td>
<td>7811 / 243</td>
</tr>
<tr>
<td>TeXclipse</td>
<td>51342 / 929</td>
<td>420 / 69</td>
</tr>
<tr>
<td>WikiText</td>
<td>59590 / 939</td>
<td>1185 / 88</td>
</tr>
</tbody>
</table>

Table 6.27: Demonstration traces for the Eclipse auto-completion task

Figure 6-7: Eclipse auto-complete demo traces call query lattice

Assume we select the highest ranked query among 31 common call queries:
Table 6.28 shows results of matching and synthesis for 5 full traces of Eclipse plugins, exercising auto-complete calls. For each full trace, we indicate the total synthesis execution time, number of seeds and their dependencies in the expanded slice graphs, the reduction in the number of statements by the code simplification algorithm, and the number of events in the full traces.

The synthesized code snippets are shown in fig. 6-8 and fig. 6-9. All results define user extensions to IContentAssistProcessor and USourceViewerConfiguration, and perform all the necessary steps to provide the auto-completion feature in Eclipse. The snippets differ in the following aspects:

1. Attaching a content assist processor to a content assistant. Note that a content assistant may have several processors applied to categories of syntax elements (the unknown value ?? stands for the category descriptor.) The complexity in WikiText editor follows from the fact that the category descriptor is retrieved from a separate document provider feature as opposed to a constant value in the other three plugins.

2. Location of the call to setSourceViewerConfiguration inside TextEditor: constructor, method doSetInput, or initializeEditor.


<table>
<thead>
<tr>
<th>Full trace</th>
<th>Time</th>
<th>Seeds/deps</th>
<th>Final/initial stmts</th>
<th>% simplified</th>
<th># events</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDT</td>
<td>676 ms</td>
<td>133 / 376</td>
<td>8 / 10</td>
<td>20%</td>
<td>29 269 599</td>
</tr>
<tr>
<td>ANT</td>
<td>818 ms</td>
<td>353 / 1018</td>
<td>11 / 19</td>
<td>42%</td>
<td>28 610 756</td>
</tr>
<tr>
<td>PyDev</td>
<td>4.865 s</td>
<td>341 / 1076</td>
<td>12 / 17</td>
<td>29%</td>
<td>52 671 499</td>
</tr>
<tr>
<td>TeXlipse</td>
<td>961 ms</td>
<td>513 / 1619</td>
<td>18 / 26</td>
<td>31%</td>
<td>19 471 138</td>
</tr>
<tr>
<td>WikiText</td>
<td>1.041 s</td>
<td>432 / 1313</td>
<td>20 / 30</td>
<td>33%</td>
<td>21 731 296</td>
</tr>
</tbody>
</table>

Table 6.28: Performance of the synthesis algorithm on the Eclipse auto-completion task for full traces of Eclipse plugins
class UlContentAssistProcessor
  implements IContentAssistProcessor {
    @Override
    ICompletionProposal[]
    computeCompletionProposals(...) {} 
  }
}
class USourceViewerConfiguration
  extends SourceViewerConfiguration {
    @Override
    IContentAssistant
    getContentAssistant(ISourceViewer aO) {
      JD
      T
      version
    ContentAssistant ca = new ContentAssistant();
    UIContentAssistProcessor uicap = new UIContentAssistProcessor();
    ca.setContentAssistProcessor(uicap, ??);
    return ca;
  }
}
class UTextEditor extends TextEditor {
  @Override
  void createActions() {
    super.createActions();
    UObject uo = UObject.f2;
    ResourceBundle rb = uo.fl;
    TextOperationAction toa = new TextOperationAction(rb, ??, this, ??);
    setAction(??, toa);
  }
  @Override
  ISourceViewer
  createSourceViewer(...) {
    ProjectionViewer pv = new ProjectionViewer(aO, al, ??, ??, ??);
    return pv;
  }
  @Override
  void initializeEditor() {
    USourceViewerConfiguration usvc = new USourceViewerConfiguration();
    setSourceViewerConfiguration(usvc);
  }
}
class UObject {
  ResourceBundle fl;
  static UObject f2;
  UObject() {
    UObject.f2 = this;
    ResourceBundle rb = ResourceBundle.getBundle(??);
    this.fl = rb;
  }
}

Figure 6-8: Synthesized code from the Eclipse auto-completion query
Rest of WikiText synthesized code

class UTextEditor extends TextEditor {
    UTextEditor() {
        UTextFileDocumentProvider utfdp = new UTextFileDocumentProvider();
        setDocumentProvider(utfdp);
        USourceViewerConfiguration usvc = new USourceViewerConfiguration();
        setSourceViewerConfiguration(usvc);
    }
    @Override
    void createActions() {
        super.createActions();
        Object o = new Object();
        ContentAssistAction caa = new ContentAssistAction(o, ??, this);
        setAction(??, caa);
    }
    @Override
    ISourceViewer createSourceViewer(Composite aO, IVerticalRuler al, int a2) {
        ISourceViewer isv = new ISourceViewer();
        return isv;
    }
} class UTextFileDocumentProvider extends TextFileDocumentProvider {
    @Override
    void connect(Object aO) {
        String[] s = new String[??];
        s[??] = ??;
        UObject.fl = s;
    }
} class UObject {
    static String[] fl;
}

Figure 6-9: Synthesized code from the Eclipse auto-completion query for WikiText plugin

6.4.2 Eclipse editor folding

Consider the task of implementing a code folding feature in an Eclipse editor. Using DEMOMATCH, we demonstrate this functionality by clicking ‘+’ icon on the left margin in 4 different editors (see table 6.29). Comparing these demonstrations against the other demonstration, we obtain 73 call queries, with the following key call query in the top 15 (at depth 4):

\textbf{Invokes(ProjectionAnnotationModel.toggleExpansionState)}

<table>
<thead>
<tr>
<th>Demo plugin</th>
<th># events / methods</th>
<th># B events / B methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDT</td>
<td>4968177 / 2625</td>
<td>10341/313</td>
</tr>
<tr>
<td>ANT</td>
<td>104933 / 1519</td>
<td>77/9</td>
</tr>
<tr>
<td>PyDev</td>
<td>182202 / 1591</td>
<td>2886/123</td>
</tr>
<tr>
<td>TeXlipse</td>
<td>186095 / 1792</td>
<td>200/71</td>
</tr>
</tbody>
</table>

Table 6.29: Demonstration traces for the Eclipse folding task

According to the Eclipse tutorial on editor folding\textsuperscript{5}, extending an editor with code folding requires the following steps:

\textsuperscript{5}https://eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html
1. Override method `createPartControl` of the `TextEditor` class, and introduce the following lines:

```java
ProjectionViewer viewer = (ProjectionViewer) getSourceViewer();
projectionSupport = new ProjectionSupport(viewer.getAnnotationAccess(),getSharedColors());
projectionSupport.install();
viewer.doOperation(ProjectionViewer.TOGGLE); // value 19
this.model = viewer.getProjectionAnnotationModel();
```

2. Modify method `createSourceViewer` to return a `ProjectionViewer`.

3. Update annotations via a call to the annotation model `model.modifyAnnotations`.

   Note that step 3 is not exercised during the demonstrations; folding regions are updated whenever the editor text is updated, not during the actual region folding. Figure 6-10 shows the synthesis results derived by resolving the call query above against the plugin full traces. The code snippets miss the call in step 3 since the slicing fails to track pairs of the primitive integer position values representing annotations. Table 6.30 shows the performance results alongside the comparison of the synthesized code against the tutorial code. For the last criterion, we put a question mark in cases where the synthesized code implies the missing return statement due to either instantiating the returned value, or casting the returned value in another method. Also note that JDT plugin uses an alternative way to toggle `doOperation` using method `enableProjection`, that is not mentioned in the tutorial.

### 6.4.3 Eclipse auto-edit

In this example, we apply DEMOMATCH to the following feature of Eclipse editors: whenever a user enters an opening bracket, the closing bracket is inserted automatically immediately after the cursor. In this example, we are interested in extracting the glue code that provides this functionality from the Eclipse full traces.

We demonstrate this functionality by typing `(*)` or `{*` in three different editors (see fig. 6-11). The remaining two plugins (for ANT and WikiText) do not provide this feature. We apply DEMOMATCH for both unary and binary attributes and specify the
// JDT plugin
class UAbstractDecoratedTextEditor extends AbstractDecoratedTextEditor {
    @Override ISourceViewer createSourceViewer(...) {
        UProjectionViewer upv = new UProjectionViewer();
        IAnnotationAccess iaa = getAnnotationAccess();
        ISharedTextColors istc = getSharedColors();
        ProjectionSupport ps =
            new ProjectionSupport(upv, iaa, istc);
        ps.install();
    }
    class UProjectionViewer extends ProjectionViewer {
        @Override void setVisibleDocument(IDocument aO)
            enableProjection();
    }
    class UAbstractDecoratedTextEditor extends AbstractDecoratedTextEditor {
        @Override void createPartControl(Composite aO) {
            super.createPartControl(aO);
            ISourceViewer isv = getSourceViewer();
            IAnnotationAccess iaa = getAnnotationAccess();
            ISharedTextColors istc = getSharedColors();
            ProjectionSupport ps =
                new ProjectionSupport(isv, iaa, istc);
            ps.install();
            ((ProjectionViewer) isv).doOperation(19);
        }@
        @Override ISourceViewer createSourceViewer(...) {
            ISharedTextColors istc = getSharedColors();
            IOverviewRuler ior = createOverviewRuler(istc);
            this.fOverviewRuler = ior;
            getOverviewRuler();
            ISourceViewer isv = new ISourceViewer();
            return isv;
        }
    }
}

// ANT plugin
class UAbstractDecoratedTextEditor extends AbstractDecoratedTextEditor {
    @Override void createPartControl(Composite aO) {
        super.createPartControl(aO);
        ISourceViewer isv = getSourceViewer();
        IAnnotationAccess iaa = getAnnotationAccess();
        ISharedTextColors istc = getSharedColors();
        ProjectionSupport ps =
            new ProjectionSupport(isv, iaa, istc);
        ps.install();
        ((ProjectionViewer) isv).doOperation(19);
    }
    @Override ISourceViewer createSourceViewer(...) {
        return ??;
    }
}

// TeXlipse plugin
class UAbstractDecoratedTextEditor extends AbstractDecoratedTextEditor {
    @Override void createPartControl(Composite aO) {
        super.createPartControl(aO);
        ISourceViewer isv = getSourceViewer();
        IAnnotationAccess iaa = getAnnotationAccess();
        ISharedTextColors istc = getSharedColors();
        ProjectionSupport ps =
            new ProjectionSupport(isv, iaa, istc);
        ps.install();
        ((ProjectionViewer) isv).doOperation(19);
    }@
    @Override ISourceViewer createSourceViewer(...) {
        return ??;
    }
}

// PyDev plugin
class UAbstractDecoratedTextEditor extends AbstractDecoratedTextEditor {
    @Override void createPartControl(Composite aO) {
        super.createPartControl(aO);
        ISourceViewer isv = getSourceViewer();
        IAnnotationAccess iaa = getAnnotationAccess();
        ISharedTextColors istc = getSharedColors();
        ProjectionSupport ps =
            new ProjectionSupport(isv, iaa, istc);
        ps.install();
        ((ProjectionViewer) isv).doOperation(19);
    }
    @Override ISourceViewer createSourceViewer(...) {
        return ??;
    }
}

Figure 6-10: Synthesized code from the Eclipse editor folding query

<table>
<thead>
<tr>
<th></th>
<th>ANT</th>
<th>JDT</th>
<th>PyDev</th>
<th>TeXlipse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time</td>
<td>1.712 s</td>
<td>2.732 s</td>
<td>950.911 ms</td>
<td>3.568 s</td>
</tr>
<tr>
<td>Full trace size</td>
<td>30 760 335</td>
<td>48 492 132</td>
<td>25 671 324</td>
<td>16 205 594</td>
</tr>
<tr>
<td>Final statements / initial stmts</td>
<td>13 / 15</td>
<td>6 / 9</td>
<td>4 / 4</td>
<td>10 / 12</td>
</tr>
<tr>
<td>% eliminated statements</td>
<td>13%</td>
<td>33%</td>
<td>0%</td>
<td>17%</td>
</tr>
<tr>
<td>Number of seeds and dependencies in the slice graph</td>
<td>342 / 984</td>
<td>160 / 392</td>
<td>107 / 221</td>
<td>310 / 873</td>
</tr>
<tr>
<td>Overrides createPartControl?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Instantiates ProjectionSupport?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Calls support.install?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Calls viewer.doOperation?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Overrides createSourceViewer?</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Irrelevant statements?</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.30: Evaluation of the synthesis results for the Eclipse editor folding demonstration across full traces of the plugins

103
demonstrations in the previous examples as the negative set. There are 5 common unary queries (none relevant) and 1017 common binary (Extends–Invokes) queries, from which we select the following one from the top 10 queries as the input to the synthesizer:

\[
\text{Nested(Extends(VerifyKeyListener.verifyKey), Invokes(SynchronizableDocument.replace))}
\]

<table>
<thead>
<tr>
<th>Demo plugin</th>
<th># events</th>
<th># unary queries</th>
<th># binary queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyDev</td>
<td>285636</td>
<td>2275</td>
<td>2514</td>
</tr>
<tr>
<td>TeXclipse</td>
<td>179336</td>
<td>2607</td>
<td>2420</td>
</tr>
<tr>
<td>JDT</td>
<td>290191</td>
<td>2545</td>
<td>2897</td>
</tr>
</tbody>
</table>

Figure 6-11: Demonstrations of the Eclipse auto-edit feature

We apply this call query to the whole collection of Eclipse full traces (including the traces in which we do not exercise the auto-edit feature). We identify 18 matches in 4 traces (30-70M events in size), from which we synthesize 5 distinct code snippets. We restrict the cover depth to 7 to reduce the amount of the synthesized code for presentation purpose. On average, each search and synthesis task takes 450ms (with two outliers at around 2s) execution time. The simplification pass reduces the number of lines of code to 66% on average across 18 matches.

In fig. 6-12, we show the shortest version of the synthesized code that is identical for 13 matches (occurring in two separate traces). This code installs a VerifyKeyListener inside the editor’s createSourceViewer method body. This listener observes key events, and inserts strings into the IDocument under certain conditions. A reference for the document is obtained from the editor text viewer that is created as the return value of the method (note, the return statement is missing, however). Thus, this short snippet provides us with a general insight into how this functionality is implemented in these editors. Interestingly, Eclipse documentation suggests using extensions of IAutoEditStrategy as the mechanism for the auto-insertion of brackets, but the plugins under analysis use alternative means to accomplish the same functionality. We believe this mismatch between documentation and implementation is a good example of the strength of our approach that relies purely on executions.

The remaining 5 matches have the following minor differences from the presented code snippet:
- Instantiating the listener in the editor constructor.
- Storing a reference to the text editor instead of the text viewer in the key listener.
- Registering the listener inside the editor method createPartControl.
- Using prependVerifyKeyListener instead of appendVerifyKeyListener.

Note that these differences are not amenable to the simplification pass without additional guarantees about the ordering of the method calls and/or the mutability of the fields in between method calls. We believe enhancing the synthesizer with the inferred state specifications may potentially resolve the first three differences as structurally the same. The last point above affects the semantics of the code snippet, which demonstrates the utility of SEMERU in exploring the space of glue code variants.

The main source of irrelevant statements in the remaining matches is the expansion of the openEditor method that initiates the editor construction. This call occurs due to internal Eclipse actions which then leads to generation of the code to create and register these actions.

class UVerifyKeyListener implements VerifyKeyListener {
    TextViewer fl;
    @Override
    void verifyKey(VerifyEvent aO) {
        TextViewer tv = this.fl;
        IDocument id = tv.getDocument();
        id.replace(??, ??, ??);
    }
}

class UAbstractDecoratedTextEditor extends AbstractDecoratedTextEditor {
    @Override
    ISourceViewer createSourceViewer(Composite aO, IVerticalRuler a1, int a2) {
        TextViewer tv = new TextViewer();
        UVerifyKeyListener uvkl = new UVerifyKeyListener();
        uvkl.fl = tv;
        tv.appendVerifyKeyListener(uvkl);
    }
}

Figure 6-12: Synthesized code for the Eclipse auto-edit feature

6.4.4 Eclipse outline navigation

Consider the editor outline navigator feature in Eclipse. The navigator shows the document outline as a tree of sections and declarations. The goal is to discover the glue
for the outline view using DEMOMATCH. To accomplish it, we record the navigation functionality of the outline, where a mouse click on an item highlights the related document section. We selected four editors with an outline, and demonstrated the mouse click on the navigator window (see fig. 6-13).

<table>
<thead>
<tr>
<th>Demo plugin</th>
<th># events</th>
<th># unary queries</th>
<th># binary queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>TeXclipse</td>
<td>269391</td>
<td>2323</td>
<td>839</td>
</tr>
<tr>
<td>PyDev</td>
<td>415011</td>
<td>2205</td>
<td>1643</td>
</tr>
<tr>
<td>ANT</td>
<td>229105</td>
<td>2102</td>
<td>804</td>
</tr>
<tr>
<td>JDT</td>
<td>323482</td>
<td>2610</td>
<td>1872</td>
</tr>
</tbody>
</table>

Figure 6-13: Demonstrations of the Eclipse outline view navigation feature

**Feature extraction** In this example, the comparative analysis of unary attributes fails to identify the characteristic call queries. In response, we modify the DEMOMATCH query to account for the potential causes:

- In case the attributes are shared with other demonstrations, we reduce the negative demonstration set to the auto-complete and folding demonstrations only.
- In case of differences in implementations, we take a subset of the positive attributes (TeXlipse and ANT only)
- To improve specificity of the call query, we extract binary attributes.

The result of the modified query consists of 430 binary call queries, all sharing the parent sub-query:

```
Extends (ISelectionChangedListener.selectionChanged)
```

For the synthesis task, we select the following top 10 query:

```
Nested(Extends(selectionChanged), Invokes(TextViewer.setSelectedRange))
```

Remark that the methods in the query are not specific to the outline view. Two of the plugins use the content outline page to implement the selection listener, but it is not required by the platform. Interestingly, Eclipse JDT plugin has a distinct implementation of this feature by relying on a helper method linkToEditor, which explains why it is not possible to extract meaningful call queries by combining all four demonstrations. PyDev does follow the same implementation as ANT and TeXlipse,
but it dispatches an intermediate Runnable task in between the two calls in the query. This splits the call tree in the middle, and thus, prevents the algorithm from extracting the call query above. This is a limitation of the current algorithm for cases involving a job queue that disrupts the call tree hierarchy.

**Search and synthesis** We apply the call query to the entire collection of Eclipse full traces, and obtain 15 matches in three execution traces exercising the content outline feature (one for each plugin). We limit the cover depth to 6 to limit the amount of the generated code. The synthesis tasks complete in 377ms on average (1 outlier at 2s) and generate 13 distinct snippets with 7-36 lines of code (the large number of lines and snippets is due to the current limitations of the code simplification pass).

Eclipse documentation recommends the following three implementation steps:
1. Override method `getAdapter(Class c)` in the editor, and return an instance of `IContentOutlinePage` when `c` equals to it.
2. Subclass `ContentOutlinePage`.
3. Invoke `setInput` on the outline page.

Figure 6-14a shows the shortest generated snippet, obtained from TeXlipse trace. This snippet discovers `getAdapter` method but misses the conditional check inside it, since SEMERU does not synthesize conditional checks. It also shows the subclass of the content page, but it uses a direct reference to the editor instead of the call to `setInput`.

In fig. 6-14b, we show one variant from Eclipse ANT plugin. The other variants from Eclipse ANT differ slightly from the presented code. This snippet performs all three necessary steps for the feature (it uses a direct reference to the editor, instead of calling `setInput`.) There are substantial differences from the recommended code, however. It uses two selection listeners. One listener is added to the internal tree viewer of the outline page. This listener delegates a selection event to the listener list of the outline page, which in turn performs the text selection. The additional lines of code initialize and glue these listeners to the editor and outline page components.

The last group of snippets belongs to PyDev execution trace. It satisfies the first

\footnote{https://wiki.eclipse.org/FAQ_How_do_I_create_an_Outline_view_for_my_own_language_editor}
Figure 6-14: Synthesized code for the content outline feature in Eclipse
step for the feature by overriding `getAdapter`, but it does not use `ContentOutlinePage`, and instead extends `Page` directly and creates `FilteredTree` inside it. It uses a `MouseListener` to delegate selection events. Additionally, it does not call `setSelectedRange` inside `selectionChanged`, opting to create a `ISafeRunnable` job and schedule it to run asynchronously.

### 6.4.5 Concluding remarks

In this section, we have applied DEMOMATCH end-to-end analysis to 5 plugins for the Eclipse platform and 4 distinct features of this platform. We have shown that the trace matching analysis extends to the Eclipse framework despite the increase in size and complexity of the framework. We have also obtained encouraging results regarding the quality of the synthesized code relative to the readily available documentation and tutorials, which are the standard sources of information for novices in the Eclipse platform.

The majority of the synthesis tasks complete within 1 second and do not exceed 5 seconds. This shows that the interactive process of DEMOMATCH analysis is achievable for large scale frameworks thanks to the SEMERU database. The results of the synthesis tasks match the documentation instructions for the auto-completion, editor folding, and outline navigation demonstrations. The audio-edit demonstration reveals a distinct implementation strategy from the tutorial code. Additionally, we have discovered the expected variations in the synthesized code, some of which are potentially helpful to the user of the tool as they hint at the design space of valid implementation strategies.
Chapter 7

Related work

In this section, we review and contrast related work across several research areas. We believe our approach identifies a novel solution in the design space of interactive program assistants, that combines ideas from program tracing, understanding, and synthesis.

Program Execution Query Languages SEMERU provides a query language to allow the synthesizer to analyze program executions. The existing work on execution query languages focuses on the bug-finding applications. PQL [31] proposes a language for discovering design defects. Queries, which resemble incomplete code fragments, can be run as a static analysis or against concrete program executions using instrumentation. The PQL system finds matches against these fragments in the code or in the execution. Design defects such as resource leakage or security vulnerabilities (e.g. lack of string sanitization in SQL) are expressed as code templates and passed to the analyzer. Due to differences in requirements, PQL is unable to query the global program state at any point in execution and is not suitable as a synthesis oracle since it uses code snippets as the query language. PTQL [15] has similar goals and similar approach of generating program instrumentation from a declarative query but uses a different relational language inspired by SQL. This language appears to be close to PQL in its expressive power but the included query compiler optimizes the translation to lighter instrumentation.
Whyline [24] assists debugging by suggesting questions that relate external observations to internal method calls (e.g. “where in the program is the color of the button set to red?”). Whyline combines static analysis and dynamic instrumentation. The system designates inputs to the program and graphical updates as important events, and traces data and control flows that lead to these events. SEMERU could potentially serve as a back-end to it since Whyline relies purely on slicing queries.

Programming with Keywords and Natural Language  Keyword programming [28] is a technique for translating keywords to API calls. Portfolio [32] shows benefits of semantic knowledge for improving free-form queries using a model of functional call chains. Additional improvement to keyword search is described in [41], which is obtained by executing the snippets and testing them on the user-supplied cases. SmartSynth [26] applies programming with natural language to smart phone development environment by hand-crafting a DSL around its API. DEMOMATCH supports keyword search on the method identifiers and the associated documentation for events in the demonstration traces as part of the ranking score for the call queries.

Dynamic Analysis for Program Understanding  FUDA [19] is closely related in its goal of producing program templates from example traces. Like SEMERU, FUDA leverages the distinction between user and framework code to project slices. However, the API trace slicing used in FUDA only uses shared objects in argument lists of calls to detect dependencies in the heap. FUDA does not keep track of the heap updates. Unlike SEMERU, FUDA does not aggregate many traces; instead, it applies specialized instrumentation to example programs for each query.

extended for inference of usage protocols which are implicit in the synthesized code.

**SEMERU** dynamic slicing relies on the container abstractions to avoid slicing the object histories for the container objects similar to the thin slicing technique [45]. Asymmetric slicing [35] improves the slice quality by categorizing data flows around the user-framework code boundary. The slicing rule for the cover events utilizes this distinction to skip internal events and produce succinct code snippets.

**End-User Program Synthesis** Program synthesis has been an active topic in the recent years in the domain of end-user programming. These tools rely on a carefully crafted domain-specific language characterizing the space of programs and efficient search and enumeration procedures [16, 38]. Many of them use examples of input and output pairs to communicate the user intent. Structured end-user programming has been studied in the context of text processing [55].

**Mining Code** The idea of using large corpus of data for program understanding has seen many incarnations in the past few years. Prospector [29], XSnippet [42], MAPO [57, 50], PARSEWeb [48], and Strathcona [20] mine source code repositories and assist programmers in common tasks: finding call sequences to derive an object of one type from an object of another type, complex initialization patterns, and frequent API usage patterns. They do so by computing relevant code snippets as determined by the static program context and then applying heuristics to rank them. Since they primarily utilize static analysis, the context lacks heap connectivity information. These tools are geared towards code assistance and do not produce full templates of the program that may span multiple classes.

Jungloid mining [29] is the most relevant synthesis project in the context of large scale systems. This project focuses on the problem of chaining API calls to derive an object of the goal type from an object of the source type. The approach is to build a graph where each node corresponds to a type and each edge corresponds to API calls, and then run a reachability query on this graph. **SEMERU** attempts to provide a richer query language to enable synthesis of more expressive programs (that
may, for example, have heap effects). Type-based code completion has been extended to provide full completion lists with chained calls and statistical ranking inferred from large software corpus [18] as well as to more general “partial expression” query language [37] with holes in place of missing method names, missing arguments, or missing property lookup.

Another category of tool attempt to infer specifications from code snippets, and data. PRIME [33] is a code search tool over a large collection of code snippets. The tool constructs and consolidates generalized typestate-based temporal summaries from partial programs and lets users search against them using a notion of relaxed automata inclusion. The generalized type state automata have been formalized in the subsequent work [36]. Buse[8] synthesizes high-quality usage examples from software corpus by using program analysis techniques that make output examples sufficiently general, succinct, and representative.

Statistical language models have been applied to short sequences of call operations on an object (tracelets) to predict missing method calls [40], estimate types in binaries [22], and infer program properties (e.g. symbol de-obfuscation) from source code [39]. These approaches rely on static analysis to extract tracelets from code snippets and/or binaries and construct generative statistical models. SEMERU synthesis would benefit from incorporating these models to improve the quality of the synthesized code by predicting missing values and statements.

**Traditional Synthesis**  Traditional program synthesis uses deductive methods to synthesize programs from the domain axioms and formalization of the program specification [30]. More recent approaches employ exhaustive combinatorial search over the program text [44] or over both program structure and its invariants [47]. A machine learning technique called version space algebras has been used to synthesize short programs from input-output examples [25]. Paraglide project [49] applied verification techniques based on abstract interpretation framework to synthesize synchronization in concurrent programs. All of the above work view program synthesis as automated derivation of efficient code from the formal specification of its behavior. Given a
partial specification, the automation comes from efficient search for solutions to the formula characterizing the program. In the case of MATCHMAKER and DEMOMATCH, however, it is not clear what the formal specification for Eclipse and its plugins would be. Large software systems pose a challenge for these formal approaches due to lack of specification and scaling limitations of static analysis.
Chapter 8

Conclusion

In this thesis, we have described a novel architecture for productivity tools and two user interaction models built on top of it. Our reliance on the trace data poses a distinct set of challenges unlike the prior work on synthesis and program understanding. We propose techniques to tackle some of these challenges borrowing ideas from information retrieval, databases, and program analysis, and empirically evaluate them on real-world examples.

Our focus on the code generation sets an important bar for the precision of the entire system. We believe that the code synthesis tools provide the best form of insight to programmers. However, the level of the automated reasoning necessary to generate code for the off-the-shelf libraries has been difficult to achieve due to lack of specification, weak abstractions, and limitations of the constraint-based solvers. We have shown that SEMERU analysis is able to handle the complexity of large frameworks and achieve high precision in reasoning about them.

The great advantage of the data-oriented architecture is the ability to integrate and share results from multiple analyses and tools. The compositional aspect of SEMERU architecture shows a novel method of combining programmer assistant tools, and gives a new path to scaling these tools to the modern software systems on a common foundation of the data-oriented program understanding tools.
Bibliography


