DemoMatch: API Discovery from Demonstrations

Kuat Yessenov Ivan Kuraj Armando Solar-Lezama

MIT CSAIL, USA

{kuat,ivanko,asolar}@csail.mit.edu

Abstract

We introduce DEMOMATCH, a tool for API discovery that allows the user to discover how to implement functionality using a software framework by demonstrating the functionality in existing applications built with the same framework. DEMOMATCH matches the demonstrations against a database of execution traces called SEMERU and generates code snippets explaining how to use the functionality. We evaluated DEMOMATCH on several case studies involving Java Swing and Eclipse RCP.

CCS Concepts • **Software and its engineering** → *Automatic programming*; *Object oriented frameworks*; • **Human-centered computing** → *User interface programming*

Keywords software engineering tools, data-driven, demonstrations, traces, slicing

1. Introduction

Software libraries and frameworks are ubiquitous in modern programming practice, delivering rich functionality that simplifies the development of full-featured applications. However, the expressive power of these frameworks comes at the cost of a difficult initial learning curve. Mastering a framework requires understanding the concepts comprising the framework and the ability to select the right combination of components for a given task. 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 [1].

We present a new technique to help programmers discover the APIs necessary to leverage functionality available from a framework. Specifically, our technique addresses one of the major challenges in API discovery: how does the tool know what functionality the programmer wants if the programmer cannot even name it? This can happen if the terminology that a programmer uses to describe the functionality is different from the terminology used by the framework. For example, the functionality that many programmers refer to as auto-complete is called content-assist in Eclipse. The framework may also implement the functionality at a different level of abstraction from the way the programmer understands it. For example, what the programmer understands as syntax highlighting, the Eclipse framework implements by combining functionality for lexical scanning with functionality to maintain consistency between a model of the document and its view.

The key observation behind our approach is that a common way for programmers to learn about functionality available in a framework is by observing it in other applications built from the same framework. For example, even if the programmer does not know how to use the syntax highlighting in the Eclipse framework, the programmer knows that it must be there because many different Eclipse plugins use it. Our central claim is that demonstrations of functionality from existing applications can serve as an effective interface for an API discovery tool.

From the user's perspective, our technique involves three steps. In the first step, the programmer uses the trace-recording functionality in DEMOMATCH to record a short *demonstration trace* of the relevant functionality being exercised. In general, the demonstration from the programmer will involve many different pieces of functionality in addition to the one the programmer is actually interested in, so in the second step the programmer needs to help the system identify what aspect of the demonstration is actually relevant. In the third step, the system uses that information to generate a list of code samples that illustrate how to use the demonstrated functionality.

A major challenge for DEMOMATCH is that the *setup code* required to use the functionality in the framework often executes during initialization of the application, long before the demonstration of functionality takes place. Therefore, the API calls that the programmer is interested in are not actually part of the demonstration trace. DEMOMATCH solves this problem by leveraging a *trace database* called SEMERU that collects, aggregates, and analyzes execution traces of reference applications utilizing the framework. The database consists of billions of low-level execution event details obtained by instrumenting reference applications and recording their heap updates and method calls. Given a demonstration trace recorded by the user, DEMOMATCH identifies traces in the database that exercise

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062386

the same functionality as the demo trace. Once those matching fragments are found, DEMOMATCH then use the information in the database to reconstruct the setup code that was necessary to enable that functionality. This code is then sanitized down to a small code snippet that is presented to the user.

Contributions The ability of DEMOMATCH to discover the setup code necessary to use a framework by demonstrating its functionality goes beyond what has been done before both in the fields of API discovery and programming by demonstration. Recent work on API discovery requires the programmer to provide either a desired type or a partial expression, both of which are hard to provide without some prior knowledge of the API [7, 11, 17, 37]. On the other hand, recent work on programming by demonstration relies on careful design of a restricted language for possible programs as well as domain specific search procedures [9, 19, 38]. This approach is too restrictive for API discovery where we expect the system to work with existing frameworks and APIs.

A third alternative to our approach is to search for documentation on the Internet using a standard search engine. For widely used frameworks like Swing or Eclipse, Google can give very good answers to very high-level queries because there are numerous blog posts, online tutorials, Stack Overflow posts and even books explaining the usage of these APIs in natural language. One of the major benefits of DEMOMATCH, however, is that it does not rely on this enormous social infrastructure, so it can be useful even for brand new APIs or for APIs that are not as widely used and not as well supported as Eclipse or Swing.

Our approach to API discovery required a number of technical contributions which are summarized below.

- We introduce a novel user interaction model for API discovery based on short demonstrations of framework features.
- We develop a new ranking technique to help programmers identify a feature in their demonstration trace corresponding to the functionality they are actually interested in.
- We develop a technique to match demonstration traces against a database of complete program traces.
- We adapt dynamic slicing techniques to generate sanitized code samples from the traces.
- We demonstrate viability of our approach through an empirical evaluation of DEMOMATCH on the Swing and Eclipse frameworks.

Limitations DemoMatch was designed for API discovery tasks, where the goal is to discover the setup code that is necessary in order to access functionality in a framework. This means that any setup that happens outside the code—for example, setup that is done through XML files—will be invisible to DemoMatch.

The second limitation stems from our reliance on dynamic execution traces. Despite significant curation effort, the 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 DEMOMATCH are backed by *concrete evidence* of execution of some user code utilizing the framework.

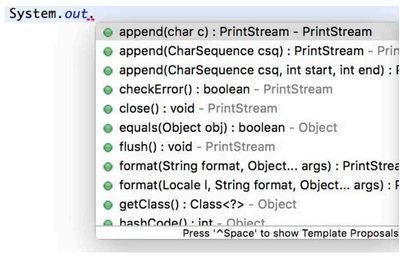
Finally, while DEMOMATCH produces code, it is not a program synthesis tool; it is an API discovery tool. The code snippets produced by DEMOMATCH are not intended to be code that programmers can directly incorporate into their application; they still need to read the documentation of the retrieved APIs to understand any subtle details about how to use these APIs. The goal is to help with the initial discovery phase, where the user does not even know what the relevant APIs are.

1.1 Illustrative Example

In this section, we walk through an example interaction of the programmer with DEMOMATCH. 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 (RCP). The goal of this programming task is to extend a blank editor plugin with the auto-completion functionality illustrated in fig. 1(a).

The first step in using DEMOMATCH is to identify an existing plugin that uses the auto-complete functionality in the Eclipse RCP. For this example, we use the Eclipse Java editor as the source of the demonstration. Once the Java editor has been launched, the user issues a command to DEMOMATCH to indicate that a demonstration is about to begin. The command initiates the trace recording functionality, which relies on online bytecode modification to record all method entries and exits performed by the running application, excluding those inside the standard Java libraries.

After issuing the trace recording command, the programmer types into the editor, triggering the auto-completion functionality as illustrated in fig. 1(a). Once the functionality has been observed, the user issues a second command to DEMOMATCH to indicate the end of the demonstration. For this example, there are over 150K method calls inside this trace that are triggered by the demonstration. Many of these method calls are directly relevant to the auto-completion functionality, but the trace also includes calls to draw additional widgets, and even to compile the code in the background during the demonstration. The next step is to identify, with the help of the user, a subset of the trace that is actually relevant to the functionality of interest. Obviously, asking the user to classify 150K method calls by hand is out of the question; instead, DEMOMATCH presents to the user a ranked list of *call queries*, signatures of methods or sequences of methods that can serve as representatives for the different pieces of functionality in the demonstration trace. The user can aid this ranking by providing a keyword or by providing an additional demonstration of functionality that can be combined with the first one.



```
System.out.
append(char c) : PrintStream - PrintStream
append(CharSequence csq) : PrintStream - Prin
append(CharSequence csq, int start, int end) : P
checkError() : boolean - PrintStream
close() : void - PrintStream
equals(Object obj) : boolean - Object
flush() : void - PrintStream
format(String format, Object... args) : PrintStrea
format(Locale l, String format, Object... args) : P
getClass() : Class<?> - Object
hashCode() : int - Object
Press '^Space' to show Template Proposals
```

a) Auto-completion functionality in eclipse.

```
class USourceViewerConfiguration extends SourceViewerConfiguration {
 @Override IContentAssistant getContentAssistant(ISourceViewer a0) {
  ContentAssistant ca = new ContentAssistant();
  UIContentAssistProcessor uicap = new UCompletionProcessor();
  ca.setContentAssistProcessor(uicap, ??);
  return ca;
}}
class UCompletionProcessor implements IContentAssistProcessor {
 @Override ICompletionProposal[] computeCompletionProposals(
       ITextViewer viewer, int offset) {
  ICompletionProposal[] result = new ICompletionProposal[??];
  result[??] = new UCompletionProposal();
  return result;
}}
class UAbstractDecoratedTextEditor extends
       AbstractDecoratedTextEditor {
 @Override void doSetInput(IEditorInput a0) {
  USourceViewerConfiguration usvc = new USourceViewerConfiguration()
       ;
  setSourceViewerConfiguration(usvc);
}}
class UCompletionProposal implements ICompletionProposal {}
```

b) User code required to add content assist to an editor.

Figure 1: Auto-completion example

For this running example, the user provides a second demonstration of the same functionality with a different Eclipse editor, such as the ANT build file editor. From the two traces, DEMOMATCH provides a ranked list of likely call queries. At the top of the list is the method computeCompletionProposals, which DEMOMATCH identifies as a good representative of the functionality that is common to the two demonstration traces. If the user selects this call query, the system then automatically generates the code fragment in fig. 1(b).

The actual code for the auto-complete functionality, as suggested by the Eclipse documentation, requires the following steps. First, the SourceViewerConfiguration must be extended to return a ContentAssistant. The assistant provides a IContentAssistProcessor for a source fragment. 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. Not all of the steps listed above took place during the demonstration of the functionality; several took place when the plugin was first launched, long before the demonstration started. In order to discover this code, DEMOMATCH had to match the demonstration trace with complete traces stored in its SEMERU database which did include those setup steps, and identify in those traces the calls and actions that were essential to enable the functionality observed in the demonstration.

DEMOMATCH depends on an assumption about the design of object-oriented frameworks in order to be able to identify the intended framework feature and connect the demo trace and the traces in the database exercising the same feature. This DEMOMATCH assumption is that *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 DEMOMATCH assumption. However, the assumption does not always hold. For example, if the feature is implemented at a very low level of abstraction (*e.g.* as a sequence of drawing commands to a rendering interpreter), then there is no such set of distinguishing methods to separate the feature from other features built from the same commands. In this paper, we focus on Swing and Eclipse RCP graphical toolkits and a variety of off-the-shelf applications. The framework features best suited for DEMOMATCH are reactive by nature, requiring user interaction to trigger a behavior, and involve multiple callbacks from the framework to the user code as well as set-up code.

We now describe in detail how each of the key components in DEMOMATCH works, starting with the trace collection and storage infrastructure.

2. Trace Data Model

DEMOMATCH relies on execution traces to query and generate code snippets. There are two types of traces: *demonstration traces*, which are short and incomplete and are used to interact with DEMOMATCH, and *full traces*, which are collected once and stored permanently as part of the SEMERU database, and which contain very detailed execution information. In this section, we describe the common formal model of the execution traces, their representation in our trace database SEMERU, and the collection framework used to populate SEMERU. We also briefly explain the embedded domain-specific language, implemented in Scala, for querying SEMERU.

2.1 Instrumentation Framework

DEMOMATCH 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 the ASM bytecode instrumentation framework [4] and the Java agent facility of the Java virtual machine (JVM).

DEMOMATCH provides a Java *agent* to be loaded alongside the target Java application into the Java virtual machine. DEMOMATCH modifies bytecode by inserting instructions into the method bodies that invoke static methods from the *collector runtime*. The agent assigns every class to one of three instrumentation domains and applies distinct transformation rules to classes from different domains. These domains are: (1) application, (2) library, and (3) exclusion.

Classes in the application domain and the exclusion domain are respectively instrumented fully and not instrumented at all. The library domain includes the common utility classes in java.util and java.lang. Classes in this domain are instrumented at the *top-most* level: only calls to the library and from the library are recorded, while all internal field, array, and method accesses are ignored. This is crucial for performance given the fraction of heap updates that occur inside java.util and java.lang in some of these applications. 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 [10, 35] for capturing the output of environment functions to minimize recording overhead.

Our recording infrastructure trades off precision for efficiency in several ways. It relaxes the tracking of control flow inside the bytecode and focuses on the interaction among object instances. Additionally, local variable instructions are not recorded in the trace log; instead, our system later infers how references to objects are acquired and passed down.

Collector runtime The collector runtime accepts static method calls from the instrumented application and outputs a sequential log of serialized records. The runtime methods are synchronized so that the calls from concurrent threads are serialized into the sequential trace log. An external *control client* communicates with the runtime over a socket and has the ability to start and stop log recording interactively.

Ingestion SEMERU ingests the log output generated by DEMOMATCH into a relational database (we use MySQL and MyISAM engine). A 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.

2.2 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 given in fig. 2b. A *trace* in SEMERU is a sequence of events ordered by counter IDs. SEMERU provides three different *views* of a trace: a Declarative view, a Call graph view and a Heap series view. Each view presents the information in a trace in a different way suitable for different steps within DEMOMATCH. The key difference between demonstration traces and full traces is that demonstration traces only include Call events and only support the Declarative view and a partial Call graph view.

2.3 Declarative Trace View

The declarative trace view provides ordered access to the individual events. This view is supported by a simple embedded DSL we call DeclView where the programmer can invoke a method select(q: Query) to get an iterator that lazily fetches events from the database on demand. The query is a boolean combination of the property-based atomic predicates listed in

Domain	Description
$\mathcal{E}$	Trace events indexed by counter
Enter	Enter events in $\mathcal{E}$
$\mathcal{O}$	Instance values including **null** and unknown
int	Primitive values
$\mathcal{V}$	Primitive and instance values ($\mathcal{O}\cup$int)
Type	Java types
Method	Java methods
Field	Java fields
$\mathcal{B}=(\mathcal{U},\mathcal{F})$	Framework boundary of disjoint subsets $\mathcal{U}\subseteq$Type, $\mathcal{F}\subseteq$Type

(a) Formal model domains

	Type	Notation	Description
Call	Enter	call m($\mathbf{p}$)	method entrance
	Exit	return a	method exit returning a
	Exception	throw e	exceptional exit
Field	Read	$a \leftarrow b.$f	value a from field f of object b
		$a \leftarrow$ f	value a from static field f
	Write	$b.$f $\leftarrow a$	Assignment to field f of b
		f $\leftarrow a$	Assignment to static field f
Array	ArrayRead	$a \leftarrow b[i]$	Read of value a from array b
	ArrayWrite	$b[i] \leftarrow a$	Write to array b at index i

(b) Types of trace events, where $a \in \mathcal{V}$, $b \in \mathcal{O}$, $i \in$ int, f $\in$ Field, m $\in$ Method, and $\mathbf{p}$ is a vector of $\mathcal{V}$.

Predicate	Semantics
Member(m)	member $\overset{?}{=} m$ for $m \in$ Method$\cup$Field
Receiver(o)	receiver $\overset{?}{=} o$ Where $o \in \mathcal{O}$
Argument(v)	Enter with a non-receiver parameter $v \in \mathcal{V}$
Value(v)	value $\overset{?}{=} v$ for $v \in \mathcal{V}$
Children(e)	parent $\overset{?}{=} e$ for $e \in \mathcal{E}$
Thread(t)	thread $\overset{?}{=} t$ for $t \in$ long
Depth(i)	depth $\overset{?}{=} i$ for $i \in$ int
At(i)	counter $\overset{?}{=} i$ for $i \in$ int
Before(e)	counter $\overset{?}{<} e.$counter for $e \in \mathcal{E}$
After(e)	counter $\overset{?}{>} e.$counter for $e \in \mathcal{E}$
Stack(e)	succ.counter $\overset{?}{\geq} e.$succ for $e \in \mathcal{E}$
Enter, Exit, ...	matches the event type (see fig. 2b)

(c) Atomic predicates in DeclView query language.

Figure 2: Formal model of SEMERU trace data

fig. 2c. The DSL also provides a foreach method to iterate over events in execution order.

2.4 Call Graph View

SEMERU also provides a view of traces as forests of call trees (one per each execution thread). Nodes in a call tree are Enter events $e_i \in \mathcal{E}$ and edges connect callers to their direct callees. SEMERU stores call traces in memory and indexes method names using Lucene [2].

DEMOMATCH uses the call graph view to compute the cover of an event e. The cover is defined in terms of the boundary between two categories of classes distinguished by SEMERU depending on the package where they are defined: *user* classes (Type $\mathcal{U}$) and *framework* classes (Type $\mathcal{F}$). The cover of e is an Enter event satisfying the following properties: (a) the cover of e is a parent of e in the call tree; (b) the cover and all the events between it and e in the tree are in the same category as e (either $\mathcal{U}$ or $\mathcal{F}$); finally (c) the cover is either the root of the tree, or its parent is in a different category. The covers allow DEMOMATCH to capture the caller-callee relationships while eliminating internal calls in the respective domains (see section 4.2).

2.5 Heap Series View

SEMERU also provides a view of the evolution of the program state that we refer to as HeapSeries (similar to the heap series used in MATCHMAKER [37]). HeapSeries for a *time interval* $[l,h)$ is a sequence of heap snapshots $\mathcal{H}_l, \mathcal{H}_{l+1}, ..., \mathcal{H}_{h-1}$ where each heap snapshot $\mathcal{H}_i$ is the state of the heap just before event $e_i \in \mathcal{E}$. A heap snapshot is a set of triples: $(a, \mathsf{f}, b) \in \mathcal{O} \times \mathsf{Field} \times \mathcal{O}$ denoting that the value of the field f of object a is object b. Intuitively, one can think of $\mathcal{H}_i$ as a directed multigraph with nodes at objects $\mathcal{O}$ and edges labelled by Field.

HeapSeries is represented in SEMERU as a single graph labelled by *fields* and *time intervals* $\mathsf{Field} \times 2^{\mathsf{int}}$ stored in a Neo4J graph database [31]. 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 $\mathcal{H}_i$ contains the connection: $(a, (\mathsf{f}, T), b) \in$ HeapSeries for $T = \{i \mid (a, \mathsf{f}, b) \in \mathcal{H}_i\} \wedge |T| > 0$. 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. The HeapSeries view is important for slicing, which requires knowledge of the heap structure in order to be effective (see section 4.1).

3. Trace Matching

DEMOMATCH takes demonstration traces as input and matches them against a collection of complete traces in the SEMERU database. The basic idea behind the matching procedure is to identify in the demonstration trace a pattern of method calls between the framework and the user code that is characteristic of the feature that the user is interested in[1]. Once the characteristic pattern has been identified, the tool can search for other traces in the database that contain this pattern. The underlying assumption, which we call the DEMOMATCH assumption, is that there is indeed a characteristic pattern of calls for each feature that the programmer is interested in.

The key challenge for this approach to work is to identify this characteristic pattern, which we refer to as the *call query* for the feature. Call queries prescribe patterns of *method calls* at the *projection boundary*—the boundary between the framework and the user code. At a high-level, our approach uses lattice-based techniques, inspired by the software reconnaissance work [6, 34] to produce a ranked list of possible call queries and asks the user to select the best one. The rest of this section focuses on the problem of identifying a call query from the thousands of call events that make up a demonstration trace.

[1] We use feature in the sense of software features, not in the machine learning sense of the term.

3.1 Framework Feature Analysis

Our approach to identifying call queries is informed by three observations. First, *the projection boundary is useful but not sufficient for identifying call queries.* Calls at the boundary between user code and framework code tend to be good call queries because they tend to correspond to concepts that are meaningful to a user of the framework. However, framework features are implemented on top of the features of the lower-level libraries, and an application written on top of the framework may choose to *bypass* the framework and invoke lower-level libraries directly. For example, the Swing framework is layered on top of its predecessor Java AWT library. Some applications will combine calls to Swing with calls to the low-level AWT library. It is also common for applications to rely on a higher-level custom library that abstracts the usage of Swing (see fig. 3). As a result, many applications that seem to exercise the same feature will actually have different calls at the projection boundary.

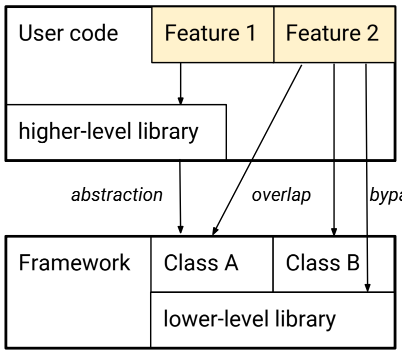


Figure 3: Relationship between the user code and the framework features

Second, *Since there can be many features exercised during the demonstration, a single demonstration trace contains many candidate call queries.* 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 is a complex problem. In particular, while the update manager activity trace might be a coincidence, the build activity *overlaps* with the code input in Eclipse.

And third, *call queries may correspond to multiple features.* In addition to the overlap of feature demonstrations, features themselves overlap in their implementations. We assign a degree of specificity to call queries reflecting the specificity of the features they represent. More specific queries are likely to have high precision but low recall, whereas less specific queries will have higher recall but lower precision. For example, one can imagine the entire demonstration trace as a call query with very high specificity; it will only match traces exercising the exact same functionality in an identical way, but 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 if they are not shared among other features.

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 trace.

3.2 Concept Analysis

Formal concept analysis [6] 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 feature. Specifically, we are interested in the following trace attributes:

- 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 $\sigma: 2^T \to 2^A$ that produces an intersection of the set of attributes for each trace in a set of traces $S \subset T$:

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

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. These attributes are derived from the following kinds of traces.

- **Positive traces D^+:** a set of traces exercising the same framework feature (for example, auto-completion invocation in several Eclipse editors).
- **Negative traces D^-:** a set of traces that definitively do not exercise the feature (*e.g.*, a *baseline trace* is a trace in which the user does not take any action).
- **Domain traces T:** a 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); these would correspond to the traces stored in the database.

In DEMOMATCH, the user of the tool only needs to provide the positive and negative traces; the domain traces are automatically extracted from the database. 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:

$$\big(\sigma(D^+) \cap R(T)\big) \setminus R(D^-)$$

here $R(S) = \{a \in A \mid \exists t \in S: (t,a) \in R\}$, which is different from $\sigma(S)$ due to the different quantifier.

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.

It is important to emphasize that the goal is not to identify all attributes relevant to a feature; only the most specific ones which can be used to find other traces with the same feature. In particular, *shared* attributes that occur in both positive and negative traces are discarded, even if some of these shared attributes may potentially be relevant to the feature at hand. The loss of these attributes is not a problem because the later phases of code generation will recover them if they need to be part of the generated code (see section 4.1).

3.3 Types of Call Queries

Now that we set-up the concept analysis framework 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. For each call query type, we define a SEMERU query that searches for events inside traces that match the pattern. In DEMOMATCH, we have two basic types of call queries based on invocation and extension attributes. *Invocation attributes* (**Invokes**(m)) correspond to a call to a framework method m. The query to the SEMERU database is Enter && Member(m). *Extension attributes* (**Extends**(m)) correspond to a call to a method in user code that overrides a framework method m. The query to the SEMERU database is Enter && Member($\mathsf{m}^{\mathcal{U}}$) where $\mathsf{m}^{\mathcal{U}}$ is the set of user methods overriding m. In both cases, the result of the query is a set of method entry events.

SEMERU also includes two forms of query composition, nesting and grouping. A *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 c. 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.

From the demonstration traces, DEMOMATCH exhaustively generates invocation and extension attributes as basic call queries, and then for every pair of basic call queries a and b, SEMERU proposes a binary *nested* query consisting of a and b. For example, if the demonstration trace includes a user call which overrides framework method m, and which internally calls a framework method n, DEMOMATCH generates a nested call query: **Nested** (**Extends**(m), **Invokes**(n)). 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. SEMERU also generates a high-level summary of all queries in the form of group queries and shows them side by side to the ranked list of the basic and nested queries.

The concept analysis reduces the number of candidate call queries by computing the set intersection, but this process still results in a large number of potential call queries. At this point, DEMOMATCH relies on human assistance to identify the specific call query from the list of candidate queries, but DEMOMATCH aids this process by ranking the call queries based on the heuristics described in the next section.

3.4 Call Query Ranking

DEMOMATCH ranks queries based on three factors. The primary factor is the *call tree depth*. This is the distance from the cover in the call tree to the event corresponding to this attribute. Attributes on the projection boundary have depth 0, while attributes deep inside the framework or the user code get assigned high depth. For a method m, DEMOMATCH uses the minimum of the depths for all calls to m.

The second factor is the *inverse document frequency (IDF)*, the inverse function of the number of domain traces T containing the attribute a. The IDF 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 a role similar to the term frequency in the TF-IDF document retrieval technique [29]; it provides a metric of the relative importance of the method to a trace in a trace corpus.

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.

The third factor is based on optional keywords provided by the user. If the user provides DEMOMATCH with keywords, the system generates a score based on whether the keywords or their synonyms match against a method name or its documentation. DEMOMATCH also uses *method name heuristics* to fine-tune the ranking scheme. The heuristics identify common design patterns from the method names to de-prioritize the specificity of the associated call queries.

Overall, DEMOMATCH uses a lexicographic ranking, so it ranks call queries first on call tree depth, then on IDF score, and then on the keyword/method-name score.

4. Generating Snippets from Traces

The algorithm for generating code snippets from SEMERU relies on techniques from *program slicing* (see section 4.1) to extract a subset of relevant trace events, and *compilation strategies* (see section 4.2) to generate symbolic code from concrete trace events.

The input to SEMERU snippet generation is a full execution trace with a set of *goal seed events*. These events are matches to the call queries derived by DEMOMATCH from the demonstration traces. The third important input is the framework boundary $\mathcal{B}$ specifying the framework $\mathcal{F}$ and the user client code $\mathcal{U}$ in the trace. The output is a code snippet that aims to summarize the *setup code* implemented by the client that triggered the execution of the seed events.

The SEMERU snippet generation algorithm consists of the following phases: (1)EXPAND computes a slice expansion from a set of initial seeds. (2) PROJECT projects the slice onto the framework boundary. (3) GENERATE generalizes the projected slice to code. (4) SIMPLIFY erases redundant instructions. (5) COMBINE normalizes and aggregates snippets across matching traces.

4.1 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 $e \in \mathcal{E}$ and an object $o \in \mathcal{O}$ 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: *how does object o arrive to event e?*

SEMERU dynamic slicing relies on the container abstractions to avoid slicing the object histories for the container objects similar to the thin slicing technique [30]. Specifically, augmenting an event with an object of interest disambiguates between two kinds of dependencies: *thin* dependency of an element in a container on the container update, and *strong* dependency of an element in a container on the producer of the container reference.

Starting from a set of initial seeds from the call query, DEMOMATCH computes the fixed point of *slicing rule* applications. Each slicing rule expands an individual seed to a set of its prior dependency seeds. SEMERU uses four rules: Local finds the local producer event within the method body (field read, for example); Static resolves to the write to a static field; Heap resolves to the write to a local field; Cover implements the idea of asymmetric slicing [22], which improves the slice quality by categorizing data flows around the user-framework code boundary; specifically, the slicing rule for the cover events utilizes this distinction to skip internal events and produce succinct snippets, and Container resolves to the method call that adds an element to a container skipping intermediate operations. The implementation relies on SEMERU trace views to formulate queries to resolve the output seeds.

4.2 Code Generation

The result of the slicing rule expansion is a set of trace records $\{e\}$ from an individual execution trace. The SEMERU code generation algorithm synthesizes code snippets from the slice and addresses the inherent challenges of generalizing concrete execution to code: (1) abstracting concrete values to variables and holes; (2) erasing user-specific and framework-specific details; (3) name assignment; (4) combining multiple slices into a single code.

Projection to user events 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 $d = \mathsf{cover}(e.\mathsf{parent})$ ($d.\mathsf{member} \in \mathcal{U}$) only if one the following conditions holds:

$$\begin{array}{rl} \text{Field events:} & \mathsf{Read}(e) \mid\mid \mathsf{Write}(e) \\ \text{Array events:} & \mathsf{ArrayRead}(e) \mid\mid \mathsf{ArrayWrite}(e) \\ \text{Top-level return:} & e = d.\mathsf{succ} \\ \text{Framework call:} & \mathsf{Enter}(e) \;\&\&\; e.\mathsf{member} \in \mathcal{F} \\ \text{User constructor:} & \mathsf{Enter}(e) \;\&\&\; e.\mathsf{member} \in \mathcal{U} \;\&\& \\ & e.\mathsf{member.name} == \texttt{"<init>"} \end{array}$$

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.

Generating statements from events GENERATE operates on individual concrete traces by translating events to symbolic instructions. 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. Primitive values and string objects are replaced by their actual values, and unbound literal objects are kept as unknown holes ??.

Constructor calls appear in the byte code as a chain of call events to <init> methods. Before applying the translation above, the entire chain of <init> calls is replaced by the call to the most specific constructor. User constructors are replaced by 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 **this** and **super**.

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.

DEMOMATCH iteratively runs a set of simplification phases, each reducing the total number of statements in the method bodies. These phases include removal of double field dereferences, unused return variables, unnecessary object instantiations, and empty method bodies. Locally defined containers that do not escape the scope of a method body are detected and erased together with their operations. Similarly, fields that are only accessed within a single method body are recognized and their declarations and statements are also removed.

Code combination Once the set of method bodies is completed, the snippet generation algorithm produces symbols for the class, method, and variable declarations. In order for multiple snippets for similar code to coincide, the symbols must rely only on the framework definitions as opposed to any user code specific symbols. For each user class type reference in the method code, the algorithm computes the typing constraints based on the usage of the user class type in the code snippet, and derived the name from the type bound on the framework super-types. Fields are sorted by a global sorting order using their full names, and assigned names f_i with index i. Multiple *versions* of the method body are shown as part of the code snippet in case the generated code contains several bodies for the same method declaration (due to multiple method executions in the slice).

5. Experimental Evaluation

In this section, we summarize our empirical evaluation of the DEMOMATCH approach and the capability of SEMERU to support DEMOMATCH queries. Specifically, the evaluation focuses on three key questions:

- How likely is it that a demo from an application straight from the Internet will find a match in a database previously created from other reference applications?
- How effective is the mechanism to select and rank call queries?
- How effective is code generation mechanism?

In order to address the first two questions, we downloaded a small set of Swing applications from the web and used them to produce demo traces without having seen their source code or knowing how they were implemented (other than knowing they used Swing). For our selected applications, we performed a set of demos and labeled each demo with possible intents (for example, a demo that cuts and pastes could have as intent cutting or pasting). For each intent, we took a set of traces that matched that intent and used them as a query. The results of this experiment are outlined in section 5.4.

In the second part, we address the code generation component of DEMOMATCH by seeding the code generation algorithm (see section 4) 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. This was done both in the context of the Swing applications mentioned above, as well as in the context of a set of end-to-end case studies for editor features in the Eclipse platform.

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. However, we have restricted our analysis to graphical applications. These are a good match for our approach both because it is easy to supply user actions and observe effects, and because of the complexity of the user code and framework interaction. The generality beyond this class of applications remains an open question.

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 The biggest threat to validity of our evaluation comes from the fact that it focuses on only two frameworks, Swing an Eclipse RCP. Within those frameworks, we selected third-party applications 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 Experimental results are obtained on a machine with 3.1GHz Intel Core i7 CPU, 512 GB SSD drive, and 8GB of RAM for SEMERU, JVM and Neo4J. MySQL database hosting the trace data and SEMERU are co-located on the same machine.

5.1 Trace Collector Performance

Data collection is a core aspect of our system; the entire approach relies on our ability to efficiently collect and store execution data. In this section, we quantify some of the details of data collection.

Data collection happens offline, but it still must be efficient enough for the applications to be usable and avoid triggering timeouts. To illustrate the cost of trace collection and processing, we recorded a trace of Eclipse while we performed typical user actions such as opening files, and editing the code over 3 minutes of execution time. All user actions succeeded without triggering timeouts.

During trace recording, the log file size increased linearly at a rate of about 40 MB per second. The resulting trace (consisting of 127 M events) was then processed by SEMERU in several steps shown in the table below. 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 since it only has to be done once per complete trace.

Processing step	Duration
Execution	3 min
Metadata ingestion	7s
Trace log ingestion	14 min
SQL updates and indexing	18 min
Heap construction	17 min
Heap ingestion	22 min
Lucene index	4s
Statistics	
# events	127M
# objects, # edges in HeapSeries	2M, 5M

5.2 Experimental Data

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. We use the official *Swing Tutorial* [3] to collect reference traces to populate SEMERU for the Swing study. The tutorial consists of short code projects that illustrate various aspects of the Swing widget toolkit. For each tutorial, 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.

5.3 Swing Experiment Setup

For our Swing experiments, we analyze three open-source Swing applications downloaded from the web:

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

For each application, we identified demonstrable features and recorded short traces by manually triggering them with user actions—constructive features such as widget layout or styling are not demonstrable, but interactive features are. 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. Table 1 lists all the demonstration traces together with their intent. For each application we also recorded a baseline trace that was used as a negative trace and it involved simply moving the mouse over the application without clicking on anything. Except for one trace which is 51k events, all other traces have over 100k events and involve over seven hundred methods each.

5.4 Ranking Call Queries

Using the recorded traces above, we conducted the following experiment to evaluate the ability of DemoMatch to return

	Action and trigger	Plausible intent
Movies	Show the add dialog *via* Menu	Menu action Movie entry dialog
	Add a movie *via* OK button	Button listener Field validation Table update
	Show the edit dialog *via* Double click on a table row	Double click listener Movie entry dialog
	Enter an incorrect rating *via* OK button	Button listener Field validation Message dialog
	Delete a movie *via* Menu	Menu action Table update
	Sort movies *via* Click on the column header	Table sort Click listener Table header update Table update
	Reorder columns *via* Drag-and-drop of the column header	D&D listener Table header update
Passwordstore	Select a list item *via* Mouse click	List selection
	Add a list item *via* ^N shortcut	Keyboard shortcut List update
	Cut a list item *via* ^X shortcut	Keyboard shortcut List update Buffer operation
	Paste a list item *via* ^V shortcut	Keyboard shortcut List update Buffer operation
	Edit the host field in the list view *via* Text field	List update Text field edit
	Filter the list *via* Filter text field	List update List filter
	Switch to the table view *via* Menu	Table update Menu action
	Sort the table view *via* Click on column header	Table update Table sort
	Generate a password *via* Menu	Menu action List update Password generation
	Undo cut of a list item *via* ^Z shortcut	Undo Buffer operation Keyboard shortcut List update
Stocks	Refresh stocks *via* keyboard shortcut	Web service connection Keyboard shortcut Table update
	Refresh stocks *via* toolbar button	Web service connection Toolbar button action Table update
	Enter invalid stock in a text field	Field validation
	Select a stock filter *via* click on a tree node	Tree node action Table update
	Collapse stock filters *via* click on a tree folder	Tree collapse
	Save dialog *via* toolbar button	Open dialog Toolbar button action
	Portfolio dialog *via* ^E shortcut	Open dialog Keyboard shortcut
	Sort stocks *via* click on table header	Table sort Table update
	Drag toolbar out of the window	Draggable toolbar
	Stock name tooltip	Table tooltip
	Stock text field tooltip	Text field tooltip

Table 1: Demonstrations of Swing features in Movies, Passwordstore, and Stocks

meaningful call queries and rank them close to the top. First, for each application, we took each of the intents from table 1 and took as positive traces all the demonstration traces that included that intent; all the traces that did not include that intent were used as negative traces. So for example, for Passwordstore , one experiment consisted on assuming that the intent was to learn how to do a Menu action, so the two traces that involved the Menu served as positive traces; all those that did not include that intent served as negative traces. The table below summarizes the result. For each application, the table lists how many such experiments were performed, and in what fraction of them the relevant call query appeared in the top 5 ranked call queries and in the top 10 ranked call queries. The ranking takes only a couple seconds despite the fact that computing the IDF measure involves a series of queries into the SEMERU database.

Application	Experiments	top 5	top 10
Movies	9	7	8
Passwordstore	10	6	9
Stocks	11	10	10

Figure 4a shows in more detail the result of the experiment on Passwordstore. Only the ‘list filter’ feature failed to match because the application bypasses the framework and performs the operation on the custom list model class. The remaining features utilize the framework facilities without bypassing them and the correct code query is always in the top 10 results, and often in the top 5.

In general, the few cases where DEMOMATCH failed to rank a desired call query highly were due to one of the following reasons:

– Incorrect association between a demo trace and the programming intent. In some of these cases, modifying the sets positive and negative traces improved the result.

– 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.

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

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

The first two failure modes can be mitigated with additional input. The third falls is scope of our tool, while the last one is a current limitation of DEMOMATCH. Nevertheless, our results provide empirical evidence in support of DEMOMATCH hypothesis in the context of Swing, and indicate that the lexicographic ranking score is sufficient to identify the key implementation methods as one of the top 10 proposed queries.

5.5 Code Generation for Swing

We evaluate the search and snippet generation components of DEMOMATCH by taking as input the call queries for five separate tasks, extracted from the demonstration traces, and

Feature	Traces	Rank
Keyboard shortcut	2,3,4,10	6/418
List filter	6	70
Buffer operation	3,4,10	1/16
(just paste)		1/7
(just cut)		1/36
Menu action	7,9	7/48
		8/48
Table update	7,8	1/144
List update	2,3,4,6	2/153
		1/153
Table sort	8	1/100
Password generation	9	7/17
Undo	10	1/8
		2/8
Text field edit	5	1/1

(a) Summary of feature extraction for Passwordstore

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

(b) Evaluation of the synthesis results for the Eclipse editor folding demonstration across full traces of the plugins

Figure 4: Results from feature extraction of passwordstore and eclipse editor folding experiments.

addressing the following experimental question: Does DEMOMATCH generate useful code snippets given call queries extracted from demonstrations?

Text component example This experiment focuses on auto-complete functionality available through the JTextArea. The two parts of the functionality involve the system proposing a completion for a word (demo #1) and pressing enter to accept it (demo #2). We recorded both demonstrations, and extracted their respective call queries.

The two demonstrations matched against the swing demo named TextAreaDemo, producing 32 and 18 lines of code respectively. The code snippets are missing two important lines of code, but otherwise convey the right classes and methods to use. Moreover, the original tutorial was 152 lines of code, so DEMOMATCH does a good job of focusing attention on the parts of the tutorial that are relevant for the desired functionality. Query #2 also matches on the TextFieldDemo tutorial and produces code that is correct and illustrates the functionality in the context of JTextField instead of JTextArea used in the demonstration.

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

Extends(InputVerifier.verify)

DEMOMATCH 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:

– 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.

– Binding an action listener to a text field which then explicitly performs a call to verify.

An interesting observation in this case is that the combination algorithm effectively groups snippets even if they are derived from several executions, since the simplification pass successfully elides execution-specific details.

Cut-copy-paste examples Consider the call query for the buffer paste feature extracted from passwordstore:

Extends(TransferHandler.importData)

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. Thanks to simplification, though, the generated code is identical for several tutorials. In general, the variants fell into two categories: one where the binding happens via the method setTransferHandler, and one where the cut and paste actions were managed manually by a listener. However, the combined cut-copy-paste feature did not produce a match in tutorial suite.

Tooltip examples The tooltip demonstrations from the Stocks application produced three call queries that are related to the tooltip functionality in Swing. DEMOMATCH identifies 126 matches for these call queries and synthesizes 25 distinct code snippets from these matches.

The call queries proved not to be specific enough, because in many tutorial applications, the getToolTipText was invoked whether or not the tooltip functionality was used. Filtering for those where the returned value is not **null**, however, produced useful code.

There is significant variety on how tooltip functionality is used, leading to many different code snippets. For example, even within a single tutorial on how to use tables [2], DEMOMATCH identified two distinct ways to add tooltips to tables: (a) Call method setToolTipText on the default table cell renderer, and (b) extend JTable and override method getToolTipText.

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

Invokes(DefaultRowSorter.toggleSortOrder)

[2] `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();
  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[] a0) {
  URunnable ur = new URunnable();
  SwingUtilities.invokeLater(ur);
}}
```

Figure 5: Synthesized code for the table sorting feature

There is one matching tutorial on "how to use tables" which generates the code in fig. 5 which constructs a table and a container for it, and enables the feature. There is only one boolean argument which is presented as a hole value due to the general policy of hiding primitive values from the synthesized code (highlighted in the figure).

5.6 End-to-End Eclipse Examples

We also evaluated the code generation capabilities of the system in the context of Eclipse. Eclipse applications are structured as collections of plugins that are connected to each other via OSGI. The Rich Client Platform (RCP) is a minimal set of plugins that are used to develop feature-rich applications. 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 for our evaluation:

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

For this phase, the evaluation strategy was as follows. First, we selected four features present in some or all of these plugins: editor folding, auto-completion, auto-edit, and outline navigation. For each feature, we recorded demonstrations and identified the call queries using DEMOMATCH. Then we synthesized code from full traces of the Eclipse plugins. Independently, we searched the web for tutorials and documentation for each of these features and made a checklist for each feature listing all the elements that are important in order to use that functionality. We then evaluated the generated code based on how many of the important elements in the checklist appear on each code snippet. Note that unlike the Swing examples, in this case the demonstrations and reference traces came from the same set of plugins.

editor folding Figure 4b summarizes the results from the editor-folding example. One important observation is that some of the elements outlined in the checklist execute during setup time, long before the demonstration begins. Another interesting observation is that the JDT plugin does not call the doOperation method, but instead calls another method enableProjection that achieves the same effect but is not mentioned on the tutorial (hence the star next to the check mark in the figure). The most common missing element was a missing **return** statement in one of the relevant methods.

auto-completion This is the introductory example. The top results from all the plugins perform all the necessary steps to enable the auto-completion feature. However, there are some differences between them depending on the plugin they come from. For example, the location of the call to setSourceViewerConfiguration inside TextEditor was sometimes in the constructor, sometimes in the method doSetInput, and sometimes in initializeEditor. The way the content assist processor is attached to the content assistant was also different for the WikiText because the category descriptor is retrieved from a separate document provider feature as opposed to a constant value in the other three plugins.

auto-edit This corresponds to the functionality where a user enters an opening bracket, and the closing bracket is inserted automatically immediately after the cursor. We demonstrated this functionality by typing "(" or "{" in three different editors; the plugins for ANT and WikiText do not provide this feature. On average, each search and synthesis task takes 450ms (with two outliers at around 2s) execution time.

The most common generated code (generated by 13 separate matches) 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 (the return statement is missing, however).

Interestingly, this example was not a complete success according to our criteria, since the approach used by the plugins was different from the approach suggested by the Eclipse documentation, which suggests using extensions of IAutoEditStrategy as the mechanism for the auto-insertion of brackets. Nevertheless, the generated snippets provide insight into an alternative approach to implement the functionality. We believe this mismatch between documentation and implementation is a good example of the strength of our approach that relies purely on executions.

outline navigation In the outline navigation 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 this, we selected four editors with an outline, and demonstrated the functionality where a mouse click on an item highlights the related document section.

For this example, DEMOMATCH failed to generate a good call query. Part of the reason turned out to be that the ideal call query shares many attributes with the negative call traces; additionally, the diversity in implementations was significant enough that by taking intersections across many positive traces, we were losing important attributes. Therefore, this was a rare example where having fewer traces, both positive and negative, actually improved the resulting call query.

For the synthesis task, we select the following query: **Nested**(**Extends**(selectionChanged), **Invokes**(TextViewer.setSelectedRange))

Interestingly, Eclipse JDT plugin has a distinct implementation of this feature by relying on a helper method linkToEditor. 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.

Given the proper call query, DEMOMATCH produced code snippets from ANT, PyDev and TeXlipse. Each of the code snippets illustrated a different approach of implementing the functionality; all of them performed the necessary steps for the feature and would allow someone to search for the missing details, but all of them had structural quirks that made them distinct.

Overall, of the four experiments, the first two had strong positive results, with the system behaving exactly as expected. The third one produced code that was useful, but was significantly different from the recommended approach to implementing the functionality, and this was due to the fact that none of the reference plugins used the recommended approach. Finally, for the last experiment, DEMOMATCH was able to produce useful code, but only after significant effort was put into crafting a call query. The interested reader can find more details about these experiments, including complete code snippets in Kuat Yessenov's thesis [36].

6. Related Work

We believe our approach combines ideas from code mining, program tracing, program understanding, and synthesis.

Mining Code The idea of using large corpus of data for program understanding has seen many incarnations in the past few years. Prospector [17], XSnippet [28], MAPO [33, 39], PARSEWeb [32], Strathcona [13], and InSynth [11] 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.

Another category of tools focuses on inferring specifications from code snippets and data. PRIME [21] is a code search tool that consolidates generalized typestate-based temporal summaries. The generalized type state automata have been formalized in the subsequent work [23]. Buse[5] 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 to predict missing method calls [25], estimate types in binaries [14], and infer program properties (*e.g.* symbol de-obfuscation) from source code [26]. These approach 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.

Type-Directed Search Jungloid mining [17] 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 search has been applied to provide completions with arbitrary compositions of expressions and ranking inferred from software corpora [11], as well as to a general "partial expression" query language [24] with holes.

MatchMaker [37] uses two object types to derive the glue code that facilitates their interaction. Like DEMOMATCH, it uses detailed execution trace to search for heap connections between type instances. SEMERU database could serve as a common foundation for both tools. Unlike MatchMaker, DEMOMATCH relies on short demonstration traces as queries without requiring the knowledge of the particular types involved in the framework feature, which is the requirement for MATCHMAKER.

Programming with Keywords and Natural Language Keyword programming [16] is a technique for translating keywords to API calls. Portfolio [20] 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 [27], which is obtained by executing the snippets and testing them on the user-supplied cases. SmartSynth [15] applies programming with natural language to smart phone development environment by hand-crafting a DSL around its API. DEMOMATCH supports keyword search on method identifiers and the documentation for events in the demonstration traces as part of the ranking score for the call queries.

Program Execution Query Languages The existing work on execution query languages focuses on discovering design defects [8, 18]. While achieving similar goal, SEMERU provides a query language that is tailored for the synthesizer to analyze program executions.

Dynamic Analysis for Program Understanding FUDA is closely related in its goal of producing program templates from example traces [12]. 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 instrumentation to example programs for each query.

7. Conclusion

DEMOMATCH presents a new approach to API discovery based on direct demonstrations of behavior on existing applications build from the same APIs. The paper showed that for functionality related to the UI, this approach is viable even for complex frameworks such as Eclipse. The most significant challenge, and the biggest opportunity for future work, lies in the need to involve the user in selecting meaningful call queries in order to disambiguate among the many behaviors occurring in an application.

Acknowledgments

This work was funded by the MUSE program (Darpa grant FA8750-14-2-0242), and also in part by NSF-CCF-1116362. We would like to thank in particular Suresh Jagannathan for his leadership of the MUSE program.

References

[1] Eclipse Mars Newsletter. `https://www.eclipse.org/community/eclipse_newsletter/2015/july/article1.php`.

[2] Apache lucene. `https://lucene.apache.org/`.

[3] Swing Tutorial. `http://docs.oracle.com/javase/tutorial/uiswing/`.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[5] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792, 2012.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, 2003.

[7] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 653–663, 2014.

[8] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 385–402, 2005.

[9] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[10] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 193–208, 2008.

[11] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 27–38, 2013.

[12] A. Heydarnoori, K. Czarnecki, and T. T. Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In S. Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, pages 344–368, 2009.

[13] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 117–125, May 2005.

[14] O. Katz, R. El-Yaniv, and E. Yahav. Estimating types in binaries using predictive modeling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 313–326, 2016.

[15] V. Le, S. Gulwani, and Z. Su. Smartsynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 193–206, 2013.

[16] G. Little and R. C. Miller. Keyword programming in java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 84–93, 2007.

[17] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, 2005.

[18] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, 2005.

[19] M. Mayer and V. Kuncak. Game programming by demonstration. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 75–90, 2013.

[20] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In

Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 111–120, 2011.

[21] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016, 2012.

[22] N. Nitta, I. Kume, and Y. Takemura. Identifying mandatory code for framework use via a single application trace. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*, pages 593–617. 2014.

[23] H. Peleg, S. Shoham, E. Yahav, and H. Yang. Symbolic automata for static specification mining. In F. Logozzo and M. Fähndrich, editors, *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 63–83. 2013.

[24] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 275–286, 2012.

[25] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, 2014.

[26] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, 2015.

[27] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, 2009.

[28] N. Sahavechaphan and K. Claypool. Xsnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 413–430, 2006.

[29] G. Salton and C.-S. Yang. On the specification of term values in automatic indexing. *Journal of documentation*, 29(4):351–372, 1973.

[30] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, 2007.

[31] N. Technology. The neo4j manual v2.0.0-m02. May 2013.

[32] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213, 2007.

[33] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 319–328, 2013.

[34] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[35] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang. Language-based replay via data flow cut. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 197–206, 2010.

[36] K. Yessenov. *Program Synthesis from Execution Traces and Demonstrations*. PhD thesis, Massachusetts Institute of Technology, 2016.

[37] K. Yessenov, Z. Xu, and A. Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 65–82, 2011.

[38] K. Yessenov, S. Tulsiani, A. Menon, R. C. Miller, S. Gulwani, B. Lampson, and A. Kalai. A colorful approach to text processing by example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 495–504, 2013.

[39] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 318–343. 2009.