Typestate-Based Semantic Code Search over Partial Programs

Alon Mishne
Technion
amishne@cs.technion.ac.il

Sharon Shoham
Tel Aviv–Yaffo Academic College
sharon.shoham@gmail.com

Eran Yahav *
Technion
yahave@cs.technion.ac.il

Abstract

We present a novel code search approach for answering queries focused on API-usage with code showing how the API should be used.

To construct a search index, we develop new techniques for statically mining and consolidating temporal API specifications from *code snippets*. In contrast to existing semantic-based techniques, our approach handles partial programs in the form of code snippets. Handling snippets allows us to consume code from various sources such as *parts of* open source projects, educational resources (e.g. tutorials), and expert code sites. To handle code snippets, our approach (i) extracts a possibly *partial* temporal specification from each snippet using a relatively precise static analysis tracking a generalized notion of *typestate*, and (ii) consolidates the partial temporal specifications, combining consistent partial information to yield consolidated temporal specifications, each of which captures a full(er) usage scenario.

To answer a search query, we define a notion of *relaxed inclusion* matching a query against temporal specifications and their corresponding code snippets.

We have implemented our approach in a tool called PRIME and applied it to search for API usage of several challenging APIs. PRIME was able to analyze and consolidate thousands of snippets *per tested API*, and our results indicate that the combination of a relatively precise analysis and consolidation allowed PRIME to answer challenging queries effectively.

Categories and Subject Descriptors D.2.4 [*Program Verification*]; D.2.1 [*Requirements/Specifications*]
General Terms Algorithms, Verification

Keywords Specification Mining, Static Analysis, Typestate, Code Search Engine, Ranking Code Samples

* Deloro Fellow

1. Introduction

Programmers make extensive use of frameworks and libraries. To perform standard tasks such as parsing an XML file or communicating with a database, programmers use standard frameworks rather than writing code from scratch. Unfortunately, a typical framework API can involve hundreds of classes with dozens of methods each, and often requires specific sequences of operations that have to be invoked on specific objects in order to perform a single task (e.g., [6, 27, 39–41]). Even experienced programmers might spend hours trying to understand how to use a seemingly simple API [27].

To write code that uses a library correctly, one can rely on code examples from other programs that use that library. The availability of textual code search services (e.g., Koders [22], GitHub search [15]) and expert sites (e.g., [34]) exposes the programmer to a vast number of API *usage examples* in the form of *code snippets* — arbitrary segments of code. Understanding how to use an API by manually browsing through such snippets, however, is an extremely challenging task: (i) A code snippet often covers only part of the desirable use case, and extracting a full scenario may require putting several different snippets together. (ii) A code snippet may invoke methods where the method body is not available, and thus its effect is unknown. (iii) Code snippets using the API of interest may appear in different contexts, making it hard for a programmer to tease out the relevant details. (iv) While most code snippets using the API are doing so correctly, some may be erroneous. As a result, manually browsing through the massive number of snippets, searching for "the right ones", is time consuming and error-prone — making it hard for a human to benefit from this vast amount of available information. Furthermore, the same reasons also present a significant challenge for automatic analysis techniques.

Goal Our long term goal is to develop a search-engine that can answer semantic code-search queries, dealing with how an API is used, in a way that consolidates, distills, and ranks matching code snippets.

To construct a search index for a particular API, we aim to use all available snippets we can find using textual code search engines, expert sites, and other sources. Therefore, in contrast to many existing approaches (e.g., [32, 44]), we *do not* assume that we have the full code of the projects in which we are searching, and our goal is to be able to handle a large number of *code snippets* obtained from various sources *without requiring the ability to build or run entire projects*. This goal presents two major challenges: (i) analysis of snippets (partial programs) (ii) consolidation of the partial information obtained from individual snippets.

The way to address these challenges depends on the level of semantic information used as a basis for search. The semantic information maintained for each code snippet should be: rich enough to describe the usage scenarios demonstrated by the code, feasible to construct efficiently, and amenable to efficient comparison operations. In this paper, we focus on (potentially partial) temporal specifications capturing sequences of API method invocations. Technically, we use a slightly generalized notion of typestate properties (see Section 4) that we believe to be a sweet spot in that regard. In contrast to existing approaches that only track sequences of method calls in terms of their resulting types (e.g., [27, 36]), we track generalized typestate properties, providing a more accurate description of API usage. Throughout the paper, we use the term *potentially partial temporal specification* (PTS) to refer to the semantic information extracted from a code snippet.

Extracting temporal specifications from partial code One of the main challenges we face is extracting temporal specifications from partial code that is most likely non-executable, and often cannot even be compiled using a standard compiler. By nature of their incompleteness, snippets are virtually impossible to run, presenting a significant obstacle to dynamic specification mining techniques (e.g., [4, 7, 14, 43]), and motivating the use of static analysis techniques. Even when turning to static analysis, new challenges arise: While handling partial code snippets is no obstacle to static approaches that are syntactic in nature (such as textual search), it poses a significant challenge for approaches that require semantic analysis of the code (e.g.,[32, 44]), as important parts of the code such as type and method definitions may be missing.

Combining partial temporal specifications A code snippet may be partial in two respects: (i) it may be covering only part of an API usage scenario, (ii) because the snippet may only contain part of the original program's code, it may invoke client methods for which the method body is not available. As a result, obtaining a full temporal specification often requires *consolidation of information extracted from several code snippets*. Consolidating PTSs requires a representation that can capture the missing information in one PTS and use other PTSs to consistently complete it.

Our Approach We present a novel code search engine capable of answering API-usage code search queries with consolidated results showing how the API should be used.

Index Representation We capture temporal specifications which exhibit *relations between different API classes*. We extend the classical notion of single-object typestate properties [35] by adding a creation context that can span multiple objects. This generalization naturally captures the common case of initialization sequences building an ensemble of objects to accomplish a task [27].

To handle partial examples, we allow temporal specifications to contain edges with "unknown" labels. Such edges represent an invocation of an unknown method, possibly hiding an arbitrary sequence of API calls. The unknown edges serve as markers that information at this part of the specification is missing. Technically, a specification is represented as a deterministic finite-state automaton (DFA). Every edge in the DFA is labeled by an unknown event or by a signature of a method in the API (not necessarily all from the same API class). The DFA therefore defines the (partial) language of interactions with that API.

Index Construction To obtain temporal specifications from snippets we need to: (i) accurately track (unbounded) sequences of API calls in each snippet and use them to derive (partial) specifications. (ii) because the sequences from each snippet may be partial, consolidate them into larger specifications that capture the full behavior of the API.

Analyzing a Snippet We use a relatively precise static interprocedural analysis tracking aliasing information to analyze a snippet and produce a PTS. We use unknown edges in the PTS to represent missing information. For example, when the body of an invoked client method is not present, we capture this fact using an unknown edge in the PTS. However, when the body of an invoked client method is present in the snippet, we perform inter-procedural analysis of the invoked method.

Consolidating Partial Temporal Specifications To obtain temporal specifications that capture the full use cases of the API we need to consolidate and amalgamate the individual partial specifications. Towards that end, we develop a technique of "unknown elimination" that iteratively attempts to consolidate matching PTSs such that unknown edges in a PTS are replaced with matching paths found in other PTSs.

Code Search Given a search query in the form of a partial program using *unknowns* (similar to SKETCH [33]), our search algorithm finds consolidated temporal specifications, each of which covers the query, along with their matching code snippets. This is done by mapping the given query into the specification space and using a notion of *relaxed inclusion* to match its PTS with PTSs in the index. By keeping a mapping from each point in the specification space back to its corresponding code snippet, we can report code snippets to the programmer. The ranking of the results is done by counting the number of similar snippets. This measure can also give the programmer an indication of whether her use of the API agrees with common usage patterns (and thus is likely correct) or not (and is thus often incorrect).

Related Work The problems of code search and code recommendation systems have seen increasing interest in recent years, and many inspiring solutions have been developed targeting different aspects of these problems (e.g., [2, 17, 18, 31, 32, 36, 44]).

The closest works to ours are [44] and [32], which target static mining of temporal API specifications. These works, as most previous work on semantic code search and static specification mining, rely on the ability to compile entire projects for complete type information, which prevents them from exploiting many examples available on the internet. This is a critical difference, as the challenge of obtaining all the code required for successfully building and analyzing a large-scale project remains a significant barrier to semantic indexing of large code bases. Assuming that the full code is available dodges the problem of consolidation which is a central challenge in our work.

Other works such as [36] and [27] can only answer queries about how to obtain one type from another via a sequence of method calls. The relatively shallow analysis employed by these can handle partial programs, but it produces a large number of answers that lack the temporal information about how a component is to be used, making it hard for the programmer to pick the right sequence.

An elaborate discussion of related work appears in Section 8.

Main Contributions The contributions of this paper are:

- We present a novel semantic code search algorithm capable of answering API-usage code search queries in the form of partial programs. Queries are answered with consolidated code showing how an API should be used.
- To obtain semantic information from code we develop new techniques for statically mining and consolidating temporal API specifications from *code snippets*. The mined specifications are generalized typestate properties that contain a creation context potentially spanning multiple objects, and may be partial, possibly containing "unknown" edges.
- We consolidate partial temporal specifications by a novel alignment technique that eliminates unknowns in a partial specification using information from other (closely related) specifications. To our knowledge, we are the first to apply such consolidation techniques in the context of code search or specification mining.
- We introduce a notion of *relaxed inclusion* and corresponding techniques to match PTSs to a query and produce a consolidated collection of snippets that cover the desired usage scenario.
- We have implemented our approach in a tool called PRIME, and evaluated it on a number of challenging APIs. We show that PRIME can be used to successfully answer expressive search queries.

```
FTPClient connectTo(String server, String user,String pass) {
   FTPClient ftp = new FTPClient();
   ftp.connect(server);
   if(ftp.getReplyCode() != 230) return null;
   ftp.login(user, pass);
   return ftp;
}
```

```
void disconnectFrom(FTPClient ftp) {
   if (ftp.isConnected()) {
      ftp.logout();
      ftp.disconnect();
   }
}
```

```
void storeFile(FTPClient ftp, String username,
       String password, String remotePath, InputStream input) {
   ftp.login(username, password);
   ftp.storeFile(remotePath, input);
   ftp.logout();
}
```

```
void upload(String server, String user,
       String pass, String remotePath, InputStream input) {
   FTPClient ftp = new FTPClient();
   ftp.connect(server);
   if(ftp.getReplyCode() == 230) {
      MyFTPUtils.uploadFile(ftp, user, pass, remotePath, input);
      ftp.disconnect();
   }
}
```

Figure 1: Code snippets using FTPClient.

```
FTPClient ftp = new FTPClient();
ftp.connect(server);
ftp.?;
ftp.storeFile(rem, in);
ftp.?;
ftp.disconnect();
```

Figure 2: A partial-code query written by a user.

2. Overview

2.1 Motivating Example

Given a task such as uploading a file to an FTP server, and a Java API such as the `FTPClient` class from the Apache Commons Net library capable of doing that, the question is, *how do we use that API to perform the task?* `FTPClient` exposes around a hundred methods and actually requires a specific method invocation sequence to successfully upload a file, leaving us lost and forcing us to seek help. Manually searching for code examples online requires time and effort, and finding the precise example that matches our needs may not be easy.

Instead, we run PRIME, which allows us to search for relevant examples based on our partial knowledge of what the code should look like. PRIME first uses textual search to download thousands of code snippets using the `FTPClient` API, among them partial snippets similar to those shown in Fig. 1. The obtained snippets are used for construction of a search index. Snippets are typically entire methods, or even entire classes, though PRIME can also handle snippets that contain "holes". PRIME then receives a query in the form of partial code, for instance as in Fig. 2 — this query is typical of a user who generally understands how to communicate with a server, but is not aware of the fine details of the API.

In this example, no code snippet demonstrates the full use case of an `FTPClient` all the way from `connect` to `storeFile` and eventually to `disconnect`. Therefore, when observing the individual code snippets and comparing them to the query, no match is found. This problem reflects the challenge in dealing with partial programs as the basis for search.

To address this challenge, PRIME applies *consolidation* techniques during index construction for combining PTSs extracted from individual snippets. Intuitively, each PTS can be thought of as a piece of a puzzle, and consolidation can be understood as putting these pieces together to obtain the full picture. Technically, PRIME analyzes each snippet to produce an automaton as shown in Fig. 3 and consolidates the individual automata together to create two summaries of usage, as shown in Fig. 4. Generally, the state numbering in the figures does not affect the meaning of the specification and is only used for presentation purposes. In these figures, we use the state numbering to show correspondence between the individual (partial) and the consolidated automata.

The query is now matched by one of the consolidated results. The relevant pieces of code, e.g. `connectTo`, `storeFile` and `disconnectFrom` from Fig. 1, are then returned, aligned together, giving us a simple yet effective visual guide for writing the code.

In addition to consolidation of partial specifications, PRIME assists the programmer in identifying common use cases by ranking the results based on the number of snippets that correspond to each specification. This is useful even when the query can be matched by a PTS before consolidation.

In the simple example of this section, each code snippet corresponds to a simple sequence of method invocations. However, in practice, many of the obtained automata have a more complex structure, for example if some method can be invoked repeatedly, or if two different methods can follow the same method invocation.

2.2 Our Approach

PRIME downloads thousands of code snippets automatically using a textual code search engine. These code snippets (*partial programs*) are used to construct a search index for a set of APIs of interest, as defined by the user. When a query is given, PRIME evaluates it against the index.

Index Representation PRIME extracts from each snippet a potentially partial temporal specification that captures sequences of API method invocations. We use a deterministic finite-state automaton (DFA) to represent a temporal specification. We refer to such a DFA as a *history*. The histories generated for our example are depicted in Fig. 3. There are several points to observe here:

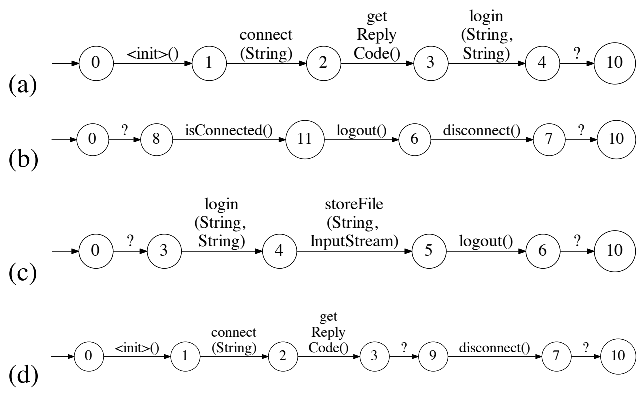


Figure 3: Partial Specifications obtained from (a) `connectTo()`, (b) `disconnectFrom()`, (c) `storeFile()` and (d) `upload()`.

```
void listFiles(String server, String username,
    String password, String dir, int n) {
  FTPClient ftp = new FTPClient();
  ftp.connect(server);
  ftp.login(username, password);
  FTPListParseEngine engine = ftp.initiateListParsing(dir);
  while (engine.hasNext()) {
    FTPFile[] files = engine.getNext(n);
    printFiles(files);
  }
  ftp.logout();
  ftp.disconnect();
}
```

Figure 6: Listing all the files in a remote directory, `n` at a time.

Partial method sequences In the absence of a clear entry point for an example (e.g. if the snippet was an entire class), we consider each method as a possible entry point. Thus, the API methods invoked on each object do not necessarily contain its creation phase. Similarly, a single snippet does not necessarily capture the full sequence of events in an object's lifetime. For example, `connectTo()` by itself leaves an `FTPClient` object in an intermediate state, without properly logging out and disconnecting, while `disconnectFrom()` does not show the prefix of a common usage. In such cases, we use a special *unknown* event, denoted ?, to model an unknown sequence of events (e.g. Fig. 3(b)). The unknown event records the fact that the partial specification can agree with other specifications that match it up to unknowns. Our goal is for the missing sequence to be filled-in by other examples.

Unknown methods The method `upload()` invokes the method `MyFTPUtils.uploadFile()` whose code is inaccessible and its defining class `MyFTPUtils` is unknown. Similarly to the treatment of partial method sequences, we use an unknown event to denote the invocation of an unknown method.

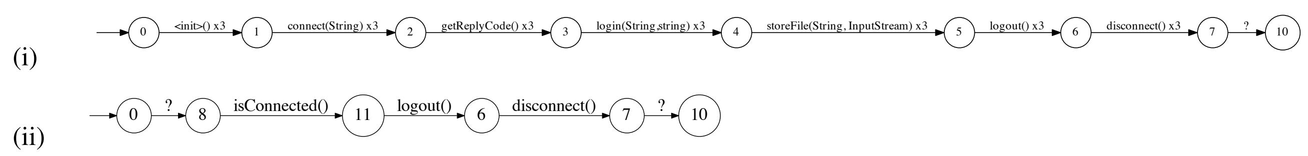


Figure 4: Consolidated specifications.

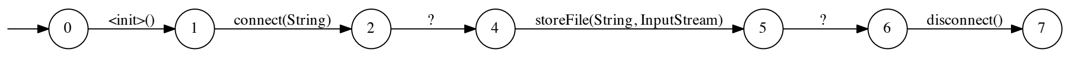


Figure 5: Partial specification extracted from a search query.

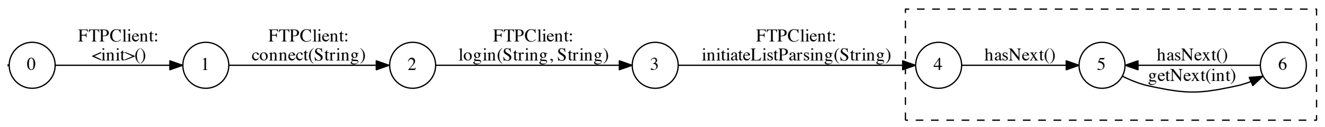


Figure 7: Creation-context-enabled result for the object of type `FTPListParseEngine`

Method sequences across multiple types Some API usage scenarios span multiple objects of different types. While the relation between objects can potentially be complex, one common pattern is that of objects being created by API methods invoked on different objects. For example, the code in Fig. 6 demonstrates how to list all the remote files in an FTP server directory, `n` entries at a time, using an object of type `FTPListParseEngine`.

Our histories exhibit the creation-relation between objects, adding the history of the creating object as the prefix of the history of the created object. This allows us to see the entire flow of API method invocations over multiple objects of related types required to create an API object and perform a certain task. For example, Fig. 7 shows a creation-context-enabled DFA recorded for the object of type `FTPListParseEngine` when analyzing the code of Fig. 6. The `logout()` and `disconnect()` methods that are invoked on the creating object of type `FTPClient` will only be part of the creating object's DFA.

Note that techniques that mine single-object typestate specifications (e.g., [32]) can only capture specifications such as the small part highlighted in Fig. 7. Techniques that only track type conversions (e.g., [27, 36]) cannot track state changes such as the fact that `FTPClient` needs to be connected and logged-in before creating an `FTPListParseEngine`, or like the state changes in the specification of Fig. 4i.

Index Construction To tackle the challenges arising when considering arbitrary code snippets, PRIME separates the construction of the search index into two phases: the *analysis* phase and the *consolidation* phase.

Analysis phase During the analysis phase, each code snippet is analyzed separately to distill and gather relevant semantic data. Some of the code snippets cannot be compiled, let alone executed. PRIME therefore analyzes the downloaded code snippets using interprocedural static analysis with points-to and aliasing information, and tracks the sequences of API method invocations observed in them for each API object, in order to derive the PTSs. In particular, PRIME has a special treatment for unknown types and methods to allow us to work around them and extract the knowable information, while clearly marking the unknowns. We emphasize that only non-API methods whose implementation is either missing or unresolved are treated as unknown. In any other case, an interprocedural analysis takes place.

In order to capture creation-context, the analysis maintains a relation between objects at the point of creation, copying the prefix of the creating object into the created object.

Unbounded sequences and sets of allocated objects The analysis has to address two sources of unboundedness: unbounded number of allocated objects (e.g., objects of type `FTPFile` in Fig. 6), and an unbounded length of API method invocation sequences (e.g., the `while` loop calling `hasNext` and `getNext` on an `FTPListParseEngine` in Fig. 6). To address the former, we use a heap abstraction based on access paths, similar to the ones used in [12]. To address the latter, we introduce a new abstraction representing sequences (with unknowns) in a bounded way using DFAs, as described in Section 4.2. The abstraction is responsible for transforming the tracked sequences into PTSs.

Consolidation phase The consolidation phase is responsible for making sense of the partial specifications obtained from individual code snippets, by completing their unknowns when possible and amalgamating them together. As our experiments indicate, this is a crucial ingredient in a successful search-engine. To our knowledge, we are the first to apply such consolidation techniques in the context of code search or specification mining.

Unknown Elimination In many cases, unknowns in one history can be resolved based on other histories. For example, the unknown event in Fig. 3a follows a `login()` event. It can therefore be matched to the sequence `storeFile()`, `logout()`, `?` from Fig. 3c, which also follows a `login()` event. This matching based on shared context implies that the unknown event most likely represents the above sequence and can therefore be replaced by it. Our approach generalizes the same principle to perform unknown elimination in DFAs, where an unknown-edge can be replaced by a DFA.

The unknown-elimination process is iterated until no further eliminations can take place (special care is needed to ensure termination). Elimination of some unknown-transitions can enable elimination of others. For example, the first unknown event in history Fig. 3d cannot be eliminated at first, since no other history contains a matching context of both a preceding `getReplyCode()` event and a following `disconnect()` event. However, as a result of other eliminations it is eventually eliminated. In this example, all histories except for Fig. 3b are consolidated into Fig. 4i, which describes a correct usage of an `FTPClient` to store files. We therefore managed to mine a correct spec for `FTPClient` even though no single snippet contained the complete specification.

Summarization Histories that are isomorphic or included in one another are merged together. In this process, method invocations (edges) which appear at the same point in the history of more than one sample are assigned increased weights (exemplified by edges labeled $\times 3$ in Fig. 4i). With a high enough number of samples, the edge weights allow us to identify the more likely full sequences which are performed on objects of type `FTPClient`.

Query Language We consider a straightforward query language which is nearly identical to Java, except that we allow a question mark character to follow the dot operator. A call `x.?` is interpreted as an unknown sequence of API method invocations on the object pointed by `x` (resembling the interpretation of an unknown client method invocation to which `x` was passed as a parameter). If this call is a part of an assignment `y = x.?`, then it is interpreted as an unknown initialization sequence of the object pointed by `y`, starting from the object pointed by `x` (and possibly referring to other objects of other types as well). Alternative query languages are possible as long as queries can be translated to partial specifications in the form of DFAs with unknowns.

Query Evaluation To answer a query in the form of a partial program, PRIME first uses similar static analysis techniques to extract a PTS from the query's partial code. For example, for the query given in Fig. 2, the obtained partial specification is depicted in Fig. 5. Matches to the query are found based on a novel notion of *relaxed inclusion*, tailored to handle partial specifications with unknown edges.

Relaxed Inclusion Relaxed inclusion resembles automata inclusion, except that unknown-edges of the included automaton can be replaced by paths (or sub-automata) of the including automaton. This captures the intuition that a match to the query should include it, but should also complete it in the sense of replacing its unknowns with more complete sequences of events.

In our example, Fig. 5 is included in Fig. 4i by the relaxed notion — even though it is not included in it by standard automata-inclusion — and is therefore returned as a match to the query. Recall that while this example demonstrates the idea on simple sequences, we in fact handle the more general notion of an automaton.

Search Results Before we present the user with results, we distill the obtained matches (in the form of consolidated histories) from parts that are irrelevant to the query and break them into linear sequences, for clarity. These sequences are ranked based both on the number of specifications summarized into the matching history, and on the likelihood of the particular sequence within its history (reflected by the weights of the corresponding history edges).

In order to present the user with code snippets, we keep a mapping from specifications back to the snippets from which they were created. In particular, each edge in a (consolidated) specification is mapped to a set of relevant snippets. For example, the `storeFile()` edge of Fig. 4i is mapped to the `storeFile` snippet only, while the `login()` edge is mapped to both `connectTo` and `storeFile`. The user can browse through the relevant code snippets accordingly.

The code snippets returned for the query in Fig. 2 appear in the thesis version of this work [28].

3. Background

We first define what we mean by the terms *API* and *client program.*

Library API: A library API is a collection of class names $T_1, \ldots, T_n$, where each class has an associated set of method signatures corresponding to methods provided by the class.

Client: We use the term *client program* of an API to refer to a program that uses a given API by allocating objects of API classes and invoking methods of the API classes. We assume that API methods only affect the internal state of the library and do not change other components of the global state of the client program.

Concrete Semantics We assume a standard imperative object-oriented language, and define a program state and evaluation of an expression in a program state in the standard manner. Restricting attention to reference types (the only types in Java that can receive method invocations), the semantic domains are defined in a standard way as follows:

$$\begin{array}{lcl}
L^\natural & \in & 2^{\mathit{objects}^\natural} \\
v^\natural & \in & \mathit{Val} = \mathit{objects}^\natural \cup \{\mathit{null}\} \\
\rho^\natural & \in & \mathit{Env} = \mathit{VarId} \to \mathit{Val} \\
\pi^\natural & \in & \mathit{Heap} = \mathit{objects}^\natural \times \mathit{FieldId} \to \mathit{Val} \\
\mathit{state}^\natural = \langle L^\natural, \rho^\natural, \pi^\natural \rangle & \in & \mathit{States} = 2^{\mathit{objects}^\natural} \times \mathit{Env} \times \mathit{Heap}
\end{array}$$

where $\mathit{objects}^\natural$ is an unbounded set of dynamically allocated objects, *VarId* is a set of local variable identifiers, and *FieldId* is a set of field identifiers. To simplify notation, we omit typing from the definition. In practice, objects are typed and all components admit correct typing.

A *program state* keeps track of the set $L^\natural$ of allocated objects, an *environment* $\rho^\natural$ mapping local variables to values, and a mapping $\pi^\natural$ from fields of allocated objects to values.

Partial Programs Our approach requires handling of partial programs. To simplify presentation, we assume that the code of each method we inspect is known in full, which means that all modifications of local state are known. An inspected method may invoke unknown methods, and refer to unknown types and fields. Our implementation also handles the more general (but less common) case of a method in which part of the code is missing.

We assume a standard semantics of partial programs updating a program state $\langle L^\natural, \rho^\natural, \pi^\natural \rangle$, where an invocation of an unknown method can allocate any number of fresh objects and can modify its reachable objects arbitrarily (e.g. [8]).

4. From Snippets to Partial Specifications

The first step of the index construction phase of our approach is analyzing each code snippet individually to extract a partial temporal specification.

In this section we first define an instrumented concrete semantics for partial programs that tracks "histories" for each tracked object, representing the way the API has been used. The notion of a history defines our choice of a formalism for index representation. Then, we describe an analysis responsible for deriving histories from individual snippets in terms of an abstraction of the instrumented concrete semantics.

4.1 Instrumented Semantics Tracking Partial Specs

Events We refer to the invocation of an API method as an *event*. An event has a receiver object and a method signature. The receiver is the object whose method is invoked. Since static API method calls have no receiver, we instead treat those as an event for each of the method's arguments, where the receiver is the argument and the signature is that of the static method. Other than this exception, we ignore the arguments since they lie beyond the focus of this paper.

Representing Unknowns A partial program might invoke a client (non-API) method where the method body is not available. Since an unknown client method may perform an arbitrary sequence of API operations on its arguments, an invocation of such a method may hide an arbitrary sequence of events. To model this behavior, we introduce special events, called *unknown* events. An unknown event stands for any possible sequence of events (including the empty sequence).

An unknown event has a receiver and a signature, denoted `T` : ?, where `T` is the type of its receiver. For each tracked object passed as an argument to an unknown method, we generate an unknown event with that tracked object as a receiver. When `T` is clear from the context, it is omitted.

We denote by U the set of all unknown events, and by $\Sigma_? = \Sigma \cup U$ the set of events extended by the unknown events. This set defines the alphabet over which program histories are defined.

Histories In our instrumented semantics, sequences of events that have occurred on the tracked objects are recorded by "concrete histories". Technically, we define the notion of a *history* as capturing a regular language of event sequences.

DEFINITION 4.1. *Given a set of events $\Sigma_?$, a* history *h is a finite automaton $(\Sigma_?, \mathcal{Q}, init, \delta, \mathcal{F})$, where $\mathcal{Q}$ is a set of states, init is the initial state, $\delta : \mathcal{Q} \times \Sigma_? \to 2^{\mathcal{Q}}$ is the transition relation, and $\mathcal{F} \neq \emptyset$ is a set of final states. We define the* traces represented by *h, $Tr(h)$, to be the language $\mathcal{L}(h)$.*

A *concrete history* $h^\natural$ is a special case of a history that encodes a single finite trace of events, that is, where $Tr(h^\natural)$ consists of a single finite trace of events. In Section 4.2 we will use the general notion of a history to describe a regular language of event sequences. We refer to a history that possibly describes more than a single trace of events as an *abstract history*.

In this section, we use the histories of Fig. 3 as examples for concrete histories. Despite the fact that these histories result from our analysis, in this special case they are just sequences and can be thus used as example for concrete histories.

A history describes a partial temporal specification. In principle, a history may be associated with different levels of a state, such as: (i) *Fully relational history:* a single history may be associated with a global state and track the *global* sequence of events over all API objects. Because a global history maintains the order between all events, it may create artificial ordering between events of independent objects. (ii) *Per-object history:* a history may be associated with a single object in the state. Such a per-object history does not capture any ordering between events on different objects, even when such relationship is important (cf. Fig. 7 in Section 2).

Our instrumented semantics offers a compromise, where it maintains a *creation context* for an object as part of its history, allowing to observe the sequences of events observed on other objects that lead to its creation.

Creation-Context Histories We maintain a *creation relation* between objects. When an object is allocated by a method of a tracked object, e.g. `x = y.m(...)`, then we consider the sequence of events of the receiver object o_1 (pointed by y) up until the assignment into o_2 (pointed by x), called the *creation context* of o_2, as part of the events invoked on o_2. Note that o_1 might also have a creation context, which will therefore also be a part of o_2's creation context. This approach allows us to record the concrete history of each object separately, while maintaining the creation context. We refer to such histories as *per-object histories with creation context*. The creation context replaces the `init` event, which typically exists upon allocation of objects.

EXAMPLE 4.2. *Consider our `FTPClient` example. In Fig. 6, an invocation of `initiateListParsing` on an `FTPclient` object (pointed by `ftp`) returns an `FTPListParseEngine` object (pointed by `engine`). Up until the creation time of the `FTPListParseEngine` object, the following sequence of events were invoked on the `FTPclient` object:*

```
<FTPClient:<init>(),
FTPClient:connect(String),
FTPClient:login(String, String),
FTPClient:initiateListParsing(String)>
```

This sequence of events, which is the concrete history of `ftp`, is therefore considered the creation context of the `FTPListParseEngine` object, pointed by `engine`, and it initializes its concrete history. Later, when `engine` invokes `hasNext()`, its history is extended by the corresponding event, resulting in:

```
<FTPClient:<init>(),
FTPClient:connect(String),
FTPClient:login(String, String),
FTPClient:initiateListParsing(String),
FTPListParseEngine:hasNext()>
```

Note that the prefix of this history consists of events that refer to a different receiver than the suffix. In this example, a single history combines two receiver types, but in more complex examples, more complex creation contexts, involving more receivers, will arise.

Instrumented Semantic: State We denote the set of all concrete histories by $\mathcal{H}^\natural$. We augment every concrete state $\langle L^\natural, \rho^\natural, \pi^\natural \rangle$ with an additional mapping $his^\natural \colon L^\natural \rightharpoonup \mathcal{H}^\natural$ that maps an allocated object of a tracked type to its concrete history. A state of the instrumented concrete semantics is a tuple $\langle L^\natural, \rho^\natural, \pi^\natural, his^\natural \rangle$.

Instrumented Semantics: Transformers Dealing with partial programs adds nondeterminism to the semantics due to the arbitrary effect of unknown client methods and API methods on the heap. Thus, each statement generates a *set* of possible successor states for a concrete state. We focus on the $his^\natural$ component of the states. The $his^\natural$ component of the concrete state tracks the histories for all tracked objects. It is updated as a result of allocation and calls to methods of a tracked type in a natural way. Here, we only show the treatment of creation context and invocations of unknown methods:

- *Creation Context:* If an object is *created* by a statement `y = x.m(...)` where both x and y are of tracked types, the concrete history of the object pointed by y is initialized to the concrete history of the object pointed by x, reflecting its creation context: $his^{\natural\prime}(\rho^{\natural\prime}(y)) = his^\natural(\rho^\natural(x))$.
  The return value of `x.m(...)` can also be an existing object, in which case no history update is needed (this is taken into account by the nondeterministic semantics updating $\langle L^\natural, \rho^\natural, \pi^\natural \rangle$).
- *Unknown Client Methods:* For an invocation `foo(y1,...,yn)`, where `foo` is an unknown client method: Let $Reach$ denote the set of arguments of `foo`, objects reachable from them, and fresh objects allocated in `foo`, restricted to tracked types. Note that the interpretation of $Reach$ is nondeterministic due to the nondeterministic semantics updating $\langle L^\natural, \rho^\natural, \pi^\natural \rangle$. The concrete history of each object $o \in Reach$ is extended by an *unknown* event `T` $: ?$, where `T` is the type of o. If the concrete history of o already ends with an unknown event, it remains unchanged.

Since the semantics considers each method as a potential entry point, we assume implicitly unknown prefixes/suffixes for the generated histories, unless an `init` event is observed. `init` "grounds" a prefix because we know that no event can precede it. Histories that start with any event other than `init` are considered *non-initialized*, and an unknown-transition is prepended to them. Similarly, an unknown-transition is appended to all histories, reflecting the unknown suffix.

EXAMPLE 4.3. *The histories depicted by Fig. 3b and Fig. 3c are non-initialized. Indeed, these histories result from tracking `FTPClient` objects when considering the methods `disconnectFrom` and `storeFile` (see Fig. 1) as entry points, thus not exhibiting the allocation and initialization of the corresponding objects.*

Fig. 3d depicts the history extracted from the `upload` code snippet for the `FTPClient` object pointed by `ftp`. This history contains an $?$-transition, resulting from the unknown client method `MyFTPUtils.uploadFile(ftp,...)`.

4.2 Abstractions for Partial Programs

The instrumented concrete semantics uses an unbounded description of the program state, resulting from a potentially unbounded number of objects and potentially unbounded histories. We now describe an abstract semantics that conservatively represents the instrumented semantics using a bounded program state, and provides a basis for our analysis. Specifically, the analysis propagates a sound approximation of program state which is based on a heap abstraction and a history abstraction. The heap abstraction is in fact quite subtle, but applies standard concepts from modular analyses (e.g., [8]). In this paper, we focus on the abstraction of histories.

History Abstractions A concrete history simply encodes a sequence of events. A conservative abstraction for it can be defined in many ways, depending on the information we wish to preserve.

In practice, automata that characterize API specifications are often simple, and further admit simple characterizations of their states (e.g. their incoming or outgoing sequences). This motivates using quotient structures of automata to reason about abstract histories.

Quotient Structure Given an equivalence relation $\mathcal{R}$ on the states of a history automaton, the quotient structure of the automaton overapproximates the history by collapsing together equivalent states: Let $[q]$ denote the equivalence class of q. Then $Quo_{\mathcal{R}}(h) = (\Sigma_?, \{[q] \mid q \in \mathcal{Q}\}, [init], \delta', \{[q] \mid q \in \mathcal{F}\})$, where $\delta'([q], \sigma) = \{[q'] \mid \exists q'' \in [q] : q' \in \delta(q'', \sigma)\}$.

k-Past Abstraction with Unknowns API usage sequences often have the property that a certain sequence of events is always followed by the same behaviors. This motivates an equivalence relation in which states are considered equivalent if they have a common incoming sequence of length k (e.g., [32]).

In order to differentiate between unknown transitions in different contexts, we increase the length of the sequence to be considered for equivalence when the last event is unknown.

DEFINITION 4.4. *The* k-past abstraction with unknowns *is a quotient-based abstraction w.r.t. the* k-past relation with unknowns $R[k]$ *defined as:* $(q_1, q_2) \in R[k]$ *if one of the following holds:*

- q_1, q_2 *have a common incoming sequence of length* k *whose last event* $\neq ?$, *or*
- q_1, q_2 *have a common incoming sequence of length* $k+1$ *whose last event is* $?$, *or*
- q_1, q_2 *have a common* maximal *incoming sequence of length* $< k$, *where an incoming sequence of length* l *is* maximal *if it cannot be extended to an incoming sequence of length* $> l$.

In particular, the initial states of all histories generated by the k-past abstraction have no incoming transitions and are therefore equivalent by $R[k]$. In addition, the produced histories are *deterministic*.

When $k = 1$, all the incoming transitions of a state q are labeled by the same event a, and all transitions labeled by a point to q. States of the automaton with an incoming ?-transition are characterized by their incoming sequences of length 2. In particular, there can be multiple history states with an incoming ?-transition, but each of these states has exactly one incoming transition. Therefore, it is characterized by its (unique) predecessor. As a result, the number of states of the history automaton is bounded by twice the number of events, which ensures a bounded description of histories.

In this paper, we consider histories obtained with the 1-past abstraction with unknowns.

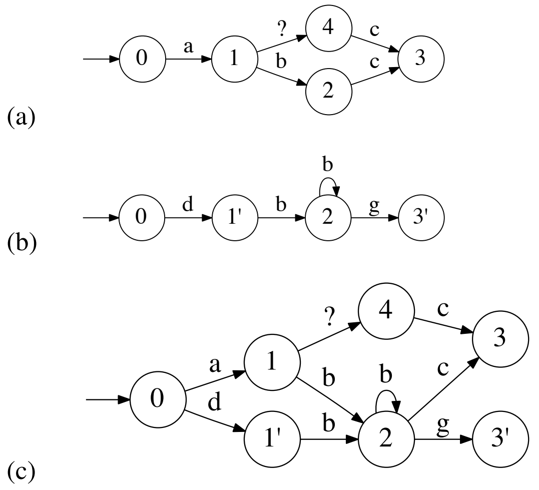


Figure 8: Abstract histories (a) and (b) merged into (c).

Abstract Histories: Transformers In the concrete semantics, a concrete history is updated by either initializing it to a given history, or appending an event to it. The abstract semantics is defined via the following transformers:

- *Abstract extend transformer:* appends the new event σ to the final states, and constructs the quotient of the result, with one exception: Final states that have an incoming unknown transition, are not extended by (possibly other) unknown events.
- *Merge operator:* constructs the union of the given automata and returns the quotient of the result.

The abstract history component for a fresh object is initialized to a history reflecting an `init` event, or to the abstract history of the object that created it. When an observable event occurs, the semantics updates the relevant histories using the extend transformer.

As long as the domain of abstract histories is bounded, the abstract analysis is guaranteed to terminate. Yet, in practice, it can easily suffer from an exponential blowup due to branching control flow. The merge operator will mitigate this blowup, accelerating convergence. Specifically, at control flow join points, all abstract histories associated with the same abstract object (representing different execution paths) are merged. This introduces additional overapproximation but reduces the number of tracked histories.

EXAMPLE 4.5. *Histories (a) and (b) from Fig. 8 are merged into history (c), where the two states labeled* 2 *are collapsed into one. As a result, additional sequences that do not exist in the union of histories (a) and (b) are introduced, such as the sequence* $\langle$`a`, `b`, `g`$\rangle$.

5. From Partial Specs to Search Index

The analysis of the previous section works on a large number of code snippets and produces a large number of partial specifications (abstract histories). These are the building blocks of the search index. Since the generated specifications are partial and contain many unknowns, and their number might be enormous, their consolidation is essential.

In this section, we discuss the consolidation phase of the index construction and present techniques for consolidating partial specifications. Consolidation has the following goals:

Completion of partial specifications Specifications obtained from a single example may contain unknown transitions and may describe a non-initialized and non-finalized behavior. Our approach relies on having a large number of examples, and on combining the partial specifications obtained from different examples. Putting partial specifications together is an *alignment* problem.

Amalgamation of partial specifications To answer code queries, we would like to present a user with a manageable number of relevant code snippets based on our analysis. For that purpose we need to collapse together similar results, while maintaining the support of each result.

Noise reduction The analysis will inevitably infer some spurious usage patterns, due to either analysis imprecision or rare (and thus likely incorrect) examples. Such rare specifications should be treated as noise and discarded.

5.1 General Approach

In the consolidation phase, we redefine the notion of an event (and accordingly of the alphabet $\Sigma_?$) to refer to the signature of the invoked method (or unknown event), while omitting the identity of the receiver. At this phase the identity of the receiver is irrelevant since we are only interested in the invoked methods for each type.

As partial specifications are being consolidated, we maintain the total number of times a transition appears in a partial specification. We refer to this number as the *weight* of a transition, and it indicates how common a transition (or a sequence of transitions) is.

Consolidation is performed by applying the following techniques:

(a) *unknown elimination*, which makes partial specifications more complete by aligning them with others to match their unknowns with alternative known behaviors, and

(b) *summarization*, which merges together histories where one is included in the other (by the standard automata-theoretic notion of inclusion or some relaxation of it, such as the one defined in Section 6.1), while increasing the weights of the shared transitions.

These two techniques can be repeated in any order. Noise reduction is intertwined with these steps, and is performed by discarding transitions (or histories) whose weights are below a given threshold. In the rest of this section we focus on the unknown elimination procedure.

5.2 Elimination of Unknowns

We mine partial histories which contain ?-transitions, modeling unknown sequences of events. In this section we develop a technique for eliminating ?-transitions by replacing them with suitable candidates. Roughly speaking, candidates are found by aligning histories containing ?-transitions with other histories that share the context of the ?-transition.

Given a ?-transition $(q_s, ?, q_t)$, candidates to replace it are sequences that appear in the same *context*, i.e. following the same prefix and followed by the same suffix. To identify such sequences, we define the set of *past-equivalent* states $\overleftarrow{[q]}$ and the set of *future-equivalent* states $\overrightarrow{[q]}$ for a state q. The set of candidates is then computed as the union of all paths between a past equivalent state of q_s and a future equivalent state of q_t, as described in Alg. 1.

Algorithm 1: Alternative Paths Computation

```
Input: Set of histories $H$, Transition $t$
Output: History $h$ summarizing alternative paths
$QS$ = FindPastEquivalentStates(Source($t$), $H$)
$QT$ = FindFutureEquivalentStates(Target($t$), $H$)
$h$ = EmptyHistory()
for $q_s$ : $QS$ do
    for $q_t$ : $QT$ do
        $h$ = Union($h$, GetRestrictedAutomaton($q_s$, $q_t$, $H$)
return $h$
```

The procedure GetRestrictedAutomaton(q_s, q_t, H) looks for a history h_{st} in H that contains both q_s and q_t, if such a history exists (otherwise, an empty automaton is returned). It then restricts the history h_{st} to a sub-automaton whose initial state is q_s and whose final state is q_t. If the restricted automaton contains an ?-transition from q_s to q_t, the transition is removed.

Given an automaton h representing the set of alternatives for the ?-transition (as returned by Alg. 1), we eliminate the transition by replacing it with that automaton h, and constructing the quotient of the result.

Past- and Future-Equivalent States For a state q, the set of *past-equivalent* states and the set of *future-equivalent* states are:

$$\overleftarrow{[q]} = \{q' \mid q \text{ and } q' \text{ have a joint initialized incoming sequence}\}$$
$$\overrightarrow{[q]} = \{q' \mid q \text{ and } q' \text{ have a joint terminating outgoing sequence}\}$$

Approximations Computing the sets of past- and future-equivalent states is not feasible. We therefore overapproximate these sets by overapproximating the equivalence relations. This is done by parameterizing the relations by a limit k on the length of incoming and outgoing sequences. The resulting relations are the k-past relation of Section 4.2, and its dual, called the k-future relation. Overapproximating the past- and future-equivalent states results in an overapproximation of the set of candidates to replace the unknown.

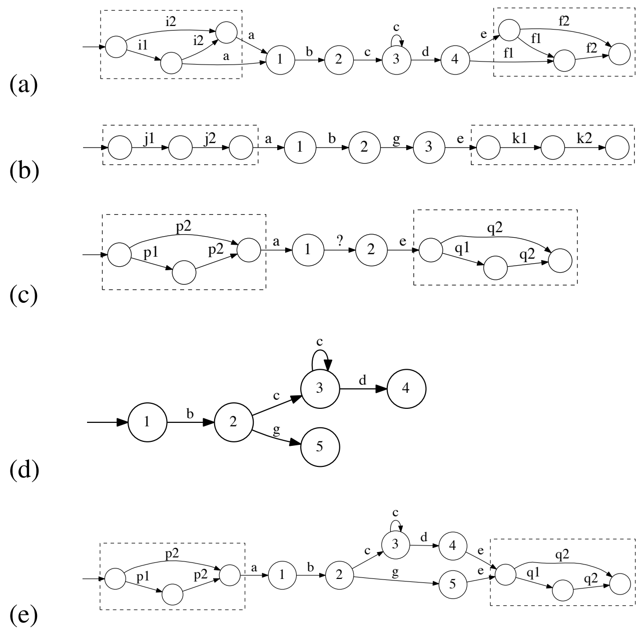


Figure 9: Partial histories (a-c), the automaton (d) representing the alternative paths for the ?-transition in (c), and the history (e) obtained after elimination of the unknown transition in (c).

EXAMPLE 5.1. *Consider the toy example depicted in Fig. 9. Histories (a-c) depict the input histories. In particular, history (c) contains an* ?*-transition, preceded by an* `a` *event and followed by* `e`*. This defines the context of the* ?*-transition. The alternative paths computation finds two past-equivalent states to the source of the transition and two future-equivalent states to the target of the transition, resulting in two alternative sub-automata. The result of the alternative paths computation is therefore their union, as presented by automaton (d). Replacing the unknown transition with this automaton results in history (e).*

Iterative Unknown Elimination Alg. 1 eliminates a single ?-transition. It is iterated until no further ?-transitions can be eliminated. Special care is given to the order in which ?-transitions are eliminated in order to guarantee termination (a naive approach could lead to non-termination, example is available in [28]).

To guarantee termination, we handle all ?-transitions that share a context simultaneously. This ensures termination of the algorithm since it ensures that once all ?-transitions with some context are eliminated, they can never reappear. However, a single ?-transition can have several contexts (e.g., it can be followed by two distinct events). As a preliminary step in the iterative unknown-elimination algorithm we therefore "split" each such transition to several ?-transitions, each with a unique context (i.e., a unique combination of preceding and following events). We say that two ?-transitions, each with a unique context, are *equivalent* if their context is the same.

We partition the set of split ?-transitions from all histories into sets of equivalent transitions (based on the unique context). In each iteration, one of these sets is considered and all ?-transitions within it are eliminated simultaneously.

Let UT be the currently eliminated set of (equivalent) ?-transitions. We use Alg. 1 to compute the set of alternative paths for an arbitrary transition, say $(q_s, ?, q_t)$, in UT. Recall that this computation disregards paths consisting of an ?-transition only (such paths result from the equivalent ?-transitions in UT). The obtained automaton h describing the set of alternative paths is common to all ?-transitions in UT. We therefore continue with the application of the unknown-elimination algorithm for each of the transitions in UT based on the same automaton h representing the alternative paths.

On top of its importance for correctness, handling equivalent ?-transitions simultaneously is also more efficient, since it prevents repeating the same computation for each of them separately.

As a result of previous unknown-elimination steps, ?-transitions that could previously not be eliminated, might become amenable for elimination. Therefore, the elimination process continues until no further ?-transitions (or sets) can be eliminated.

EXAMPLE 5.2. *We summarize this section by demonstrating the consolidation phase on our motivating example. Consider the histories depicted in Fig. 3. We describe in detail the iterative unknown-elimination is this example. The intermediate histories are depicted in Fig. 10.*

Throughout the example we denote the context of an ?*-transition as a pair* (*pre*,*post*) *where pre precedes the* ?*-transition, and post follows it. In this example each* ?*-transition has a unique context since all histories describe simple sequences. Therefore the preliminary splitting step is not needed.*

We first eliminate the ?*-transition from state* 4 *to state* 10 *in (a) whose context is* (`login`, −)*. We find a match in history (c), and history (a) is transformed to the history (a') depicted in Fig. 10. Now, we simultaneously eliminate the two equivalent* ?*-transitions with context* (`logout`, −)*, in histories (a') and (c) (each of these transitions is from state* 6 *to state* 10 *in the corresponding history). The transitions are eliminated based on history (b). The resulting histories (a") and (c') are depicted in Fig. 10. Next, the* ?*-transition from state* 0 *to state* 3 *in (c') with context* (−, `login`) *is eliminated based on history (a") resulting in the history (c"). Last, the* ?*-transition from state* 3 *to state* 9 *in (d) with context* (`getReplyCode`, `disconnect`) *is eliminated based on histories (a") and (c"), resulting in history (d'). No further* ?*-transitions can be eliminated. In particular, the* ?*-transitions in history (b) are not eliminated.*

Note that in this example, the ?-transition from state 3 to state 9 in history (d) cannot be eliminated at first, but is eliminated after other elimination steps are performed.

When the iterative unknown-elimination algorithm terminates, a summarization step is applied. Since histories (a"), (c") and (d') are all isomorphic, they are merged into one. The consolidated results are depicted in Fig. 4.

6. Querying the Index

The previous sections discussed index representation and construction. In this section, we describe our approach for answering code search queries.

Given a query in the form of a partial program, we first use the analysis described in Section 4 to transform it into an automaton. We then compare the query automaton to the (consolidated) automata in the index in order to find matches. We observe that a match for the query should "include" it in the sense of including all the temporal information present in the query, and filling in its unknowns. Our approach therefore looks for automata that include the query automaton by some relaxed notion of inclusion as described in Section 6.1. The obtained automata are simplified, ranked and transformed back to code snippets.

6.1 Query Evaluation by Relaxed Inclusion

A similarity measure is a key ingredient in the development of a search algorithm, as it determines the matches found for the given query.

Our notion of similarity is motivated by automata inclusion. Intuitively, a good match for the query should include it. To handle ?-transitions, we relax the standard notion of automata inclusion. The intuition is that an unknown event in the query stands for some sequence of events, which might also contain unknowns (since the index we are searching in might still contain partial specifications, even after the unknown-elimination step). Therefore, we say that the query is included in some automaton from the search index if its ?-transitions can be completed in some way such that standard automata-inclusion will hold.

Before we turn to the formal definition, we demonstrate the intuition on our motivating example. In this example, the automaton representing the query, depicted in Fig. 5, is included in the first automaton from Fig. 4 if the first ?-transition is "completed" by ⟨ `getReplyCode`, `login` ⟩ and the second ?-transition is "completed" by the sequence ⟨`logout` ⟩. Note that in this example all automata represent simple sequences, but in practice more complex automata arise.

This leads us to the following relaxed definition of inclusion. Let $\mathcal{Q}^? \subseteq \mathcal{Q}$ denote the targets of unknown transitions in a history. Recall that each such state $q \in \mathcal{Q}^?$ has a unique predecessor, which we denote $pred(q)$. Then

DEFINITION 6.1 (Relaxed Inclusion). *A history* $h_1 = (\Sigma_?, \mathcal{Q}_1, init_1, \delta_1, \mathcal{F}_1)$ *is* relaxly-included *in a history* $h_2 = (\Sigma_?, \mathcal{Q}_2, init_2, \delta_2, \mathcal{F}_2)$ *if there is a mapping* $f : \mathcal{Q}_1 \to \mathcal{Q}_2 \cup 2^{\mathcal{Q}_2}$ *such that (1)* $f(q_1) \in 2^{\mathcal{Q}_2}$ *iff* $q_1 \in \mathcal{Q}^?{}_1$*, in which case we require that all states in* $f(q_1)$ *are reachable from* $f(pred(q_1))$*, (2) If* $q_1 \notin \mathcal{Q}^?{}_1$*, then whenever* $q_1' \in \delta_1(q_1, a)$ *for* $a \in \Sigma$*, then* $f(q_1') \in \delta_2(f(q_1), a)$ *as well, and (3) If* $q_1 \in \mathcal{Q}^?{}_1$*, then whenever* $q_1' \in \delta_1(q_1, a)$ *for* $a \in \Sigma$*, then there exists* $\hat{q} \in f(q_1)$ *such that* $f(q_1') \in \delta_2(\hat{q}, a)$ *as well.*

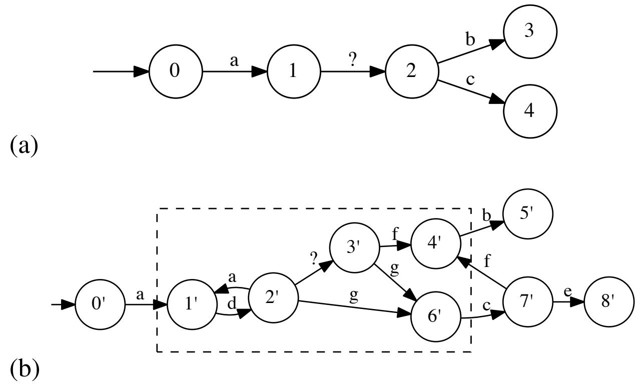


Figure 11: Relaxed inclusion of history (a) in history (b).

Intuitively, a history is relaxly-included in another if it can be "embedded" in it where unknown transitions are replaced by "more complete" sub-automata. This definition resembles the problem of *subgraph isomorphism*, which is NP-complete. Fortunately, due to the simple characterization of the history states when using the k-past abstraction (with unknowns), checking if a history is relaxly-included another is easy in our setting: the characterization of the states induces the (potential) mapping for all states except for the mapping of $\mathcal{Q}^?$. The candidates for the mapping of states in $\mathcal{Q}^?$ are found by examining their successors. Thus it remains to check if this mapping fulfills the requirements.

EXAMPLE 6.2. *History (a) depicted in Fig. 11 is relaxly-included in history (b). Intuitively speaking, history (b) includes history (a) when the ?-transition in (a) is "completed" by the rectangle in (b). Note that this rectangle still contains an ?-transition, but it is "more complete" than the ?-transition of (a).*

Technically, the mapping of states in this example is the following: $0 \mapsto 0'$, $1 \mapsto 1'$, $2 \mapsto \{4', 6'\}$, $3 \mapsto 5'$, $4 \mapsto 7'$.

EXAMPLE 6.3. *Histories (a), (c), (d) depicted in Fig. 3 are all included by our notion of relaxed inclusion in the top consolidated history from Fig. 4. Note that standard automata-inclusion does not hold here.*

6.2 Search Results

In order to transform the consolidated histories that match the query into code snippets and present them to the user in a useful way, we use the following steps.

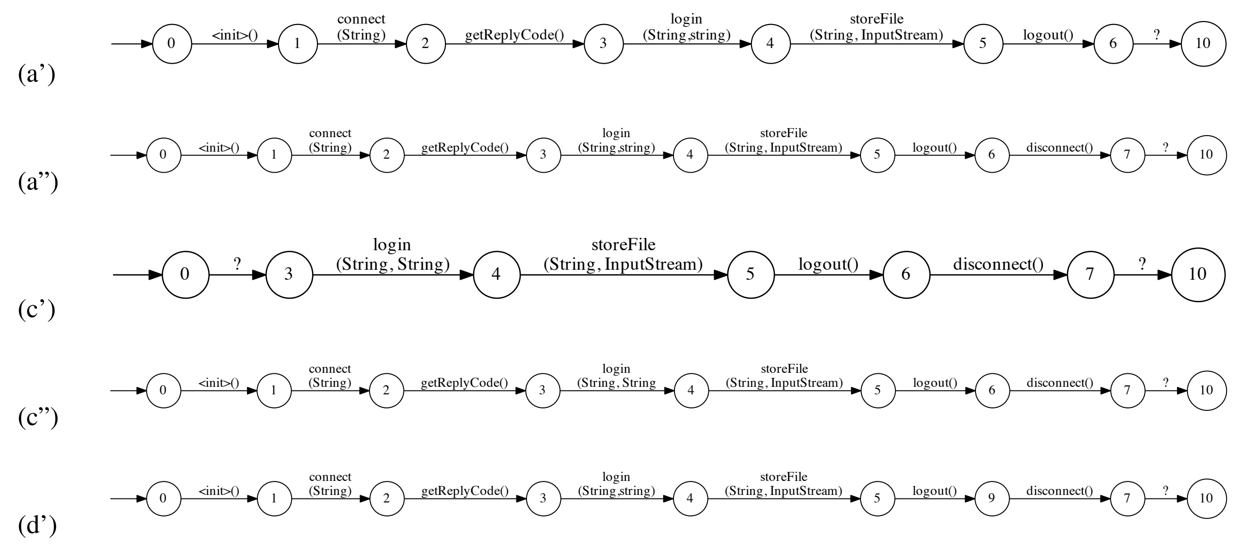


Figure 10: Intermediate unknown elimination results.

Extracting Sequences from Consolidated Automata The histories matching the query represent a multitude of possible method call sequences, some of which are irrelevant to the given query, and would not be useful as (a part of) a search result. In addition, some of the obtained automata are large and complex, and do not lend themselves well to being returned as human-readable search results.

To address these two issues, we first extract from each history h that matches the query (i.e., relaxly-includes it), a sub-automaton h_{min} representing all *minimal* sub-automata that still relaxly-include the query. h_{min} still matches the query and it summarizes all the information relevant for the query. It is computed as the intersection of h with the result of applying the unknown-elimination algorithm on the query automaton based on h. The history h_{min} is then decomposed into simple *paths*, each of which encodes a single sequence of events, by a repetition-free depth-first-traversal of h_{min}. This means that although the consolidated automata may contain loops, the final human-friendly results never do.

Ranking The history-paths extracted from the automata that match the query are ranked according to:

- Their *support*: the support of a history-path is inherited from the history that the path was extracted from. It corresponds to the number of histories summarized into the history during the consolidation phase of the index construction (see Section 5).
- Their *probability*: this is the probability of following the history-path (and observing the sequence of events labeling it) in its source history. It is computed by normalizing all the weights on the source history transitions such that the sum of all outgoing transitions for each state is 1. This turns the history into a probabilistic automaton, where the probability of each path corresponds to the multiplication of all probabilities along its transitions.

Since it is possible that two or more similar paths will be returned from different histories, the paths are *summarized* (merged together by inclusion) to avoid redundant search results. In such cases, the supports are summarized as well, and the probability is chosen to be the maximal probability among the summarized paths. Paths are first sorted according to their (accumulated) support. Paths with the same support are then ranked by their (maximal) probabilities.

From Automata to Code Starting at the index construction phase and up until the query evaluation phase, we maintain for each transition of a history the set of code snippets responsible for introducing it. This set is initialized during the analysis of individual snippets (where each transition is associated with the snippets that produced it), and it is updated during unknown-elimination and summarization. This enables us to transform history-paths and their corresponding sequences of events back to code snippets. The problem of finding a smallest set of snippets that cover all of the relevant transitions is NP-complete. Still, in many cases it is possible to find such small sets.

7. Evaluation

7.1 Prototype Implementation

We have implemented our approach in an open-source tool called PRIME. The tool and the data used for our experiments are available from `http://priming.sourceforge.net/`.

For collecting snippets we have used automatic scripts to search and download from GitHub, Stackoverflow and (before it was closed) Google Code Search. Partial snippet compilation, needed for simplifying Java to an intermediate language for analysis, is done using a specialized partial compiler [9]. Code analysis relies on the Soot static analysis framework [37], but adds custom-made points-to analysis which can handle partial code snippets, as well as an interprocedural analysis mechanism which can also deal with recursion and takes parameters and return values into consideration.

PRIME can track objects across upcasts and downcasts, and can handle method calls on sub- or super-types of the tracked object's type even when it is not aware of the relations between the types. This is done by collapsing parallel edges together if they correspond to different implementations of the same method.

Our textual search queries return many thousands of examples, which PRIME then proceeds to analyze. When operating on pre-downloaded, locally-available snippets, PRIME is capable of analyzing hundreds of thousands and even millions of snippets. We assume that downloading and constructing the index for a family of APIs is done before queries are evaluated.

Analysis We have used the k-past abstraction with unknowns, setting $k = 1$.

Performance Index construction was run on a 2.66 GHz Intel Xeon X5660 CPU machine with 48 GB memory. Each group of 1000 snippets took approximately 1 hour to compile, 30 minutes to analyze and 5 minutes to consolidate. Querying was done on a 1.7 GHz Intel Core i5 machine with 4 GB memory. It took 1 to 3 seconds to find matches to a query, but extracting sequences and ranking them takes between 1 and 45 seconds, depending on the number and size of the found matches. Optimizing the performance of querying and extracting sequences could be done but was outside the scope of our initial work.

7.2 Benchmarks and Methodology

For the purpose of evaluation we selected several high-profile APIs in which method sequence order is important. We used PRIME to construct an indexed data set from them and to query them. For the sake of the evaluation of PRIME's results, we chose APIs for which tutorials exist, and can provide familiarity with the desired outcome.

Benchmarks The APIs we have chosen to show are taken from the popular libraries:

- **Apache Ant** A project building library.
- **Apache Commons CLI** A library for parsing command-line options (and in particular `GnuParser`, its command line parser).
- **Apache Commons Net** A network communications library (and particularly its `FTPClient` component).
- **Eclipse** The popular modular Java IDE (in particular its UI, JDT and GEF plugins).
- **JDBC** A built-in Java library for interacting with databases.
- **WebDriver** A widely-used browser automation framework.

These libraries are all widely-used, and our personal experience indicates that some of them are quite tricky to use.

Experiments We ran several experiments on our collected data set:

- **Distilling.** To evaluate the ability of PRIME to distill a large number of snippets into illustrative examples, we ran PRIME and checked whether the few top results include the examples present in a tutorial of the API. We show that a user can use PRIME to browse a small number of distilled examples instead of sifting through thousands of results returned from a textual search engine.
- **Prediction.** To evaluate the quality of our results, we used PRIME to answer prediction queries. We gathered example method invocation sequences for several of our selected APIs, representing a common use of that API. The examples were either produced from online tutorials or manually crafted according to the API documentation. They were transformed into prediction queries by replacing their suffixes with unknown events.
- **Consolidation.** To evaluate the importance of consolidation, we gathered several queries which represent parts of common use-cases over high-profile APIs, and show that these produce high-quality results after consolidation is applied, but no results or poor results without consolidation.

We now elaborate on these experiments and their results.

7.3 Results

7.3.1 Distilling

To show the advantage of distilling and ranking search results according to history support, we use a simple 1-method query for each API and check whether the top results returned by PRIME include the tutorial example, and if so, how it was ranked. This experiment is similar to the evaluation approach used in [44]. However, our queries used what we considered to be the key method in each tutorial, not just the first one. The value of PRIME is not only in finding the tutorial result, but also in establishing the support of this result in term of commonality.

The results are summarized in Table 1. The first column shows the API used and how many of its snippets PRIME analyzed. In this specific case, each snippet is an entire class (the granularity of files obtained from GitHub). The second column shows the query used (not including surrounding unknowns). The third column shows the number of results returned from GitHub's textual search, i.e., the number of possible matches the user has to browse (notice this is less than the number of snippets given to PRIME, as not all snippets actually contain the method in question directly, though they may of course call it indirectly). The last two columns shows

API used for the query, num of downloaded snippets	**Query description**	**Query method**	**Number of textual matches**	**Tutorial's rank**	**Tutorial's support**
WebDriver 9588 snippets	Selecting and clicking an element on a page	WebElement.click()	2666	3	2k
Apache Commons CLI 8496 snippets	Parsing a getting a value from the command line	CommandLine.getValue(Option)	2640	1	873
Apache Commons Net 852 snippets	"connect -> login -> logout -> disconnect" sequence	FTPClient.login(String, String)	416	1	446
JDBC 6279 snippets	Creating and running a prepared statement	PreparedStatement.executeUpdate()	378	1	431
	Committing and then rolling back the commit	Connection.rollback()	177	4	28
Eclipse UI 17,861 snippets	Checking whether something is selected by the user	ISelection.isEmpty()	1110	2	411
Eclipse JDT 17,198 snippets	Create a project and set its nature	IProject.open(IProgressMonitor)	3924	1	1.1k
Eclipse GEF 5981 snippets	Creating and setting up a ScrollingGraphicalViewer	GraphicalViewer.setEditPartFactory (EditPartFactory)	219	1	14

Table 1: Experimental Results - Distilling. In this case, each "snippet" is a complete class.

the rank that the tutorial example was found in, and the support for that tutorial.

As can be seen from Table 1, PRIME always returned the tutorial example as one of the top few results, and in many cases as the first result. This means that a user can rely solely on looking at the top few results returned from PRIME instead of manually browsing through the hundreds or thousands of results returned from regular textual search.

The `WebElement.click()` tutorial was only ranked 3rd because it is very common to click on some element only when a certain condition holds. Therefore, the two preceding matches contained calls to getter methods before `click()`. `Connection.rollback()` was ranked 4th because although there are required preceding and succeeding method calls that appeared in all top results, there are often other various calls on that object in-between. The `GEF` tutorial was matched as first but has poor support because there are many different use-cases for the `setEditPartFactory` method.

7.3.2 Prediction

Table 3 lists the sequences used to obtain prediction queries, their description, and their prediction accuracy by PRIME. Accuracy was measured by checking the prediction PRIME gave for each method in the sequence when the suffix starting with it was replaced by an unknown. We adapted PRIME to use the same principles described in Section 6 in order to return a ranked list of possible next methods, which we compared to the actual method from the sequence. We use the average, median, and max rank of the prediction over the entire sequence to evaluate accuracy. Any rank less than half the number of methods defined in the type is better than blind guess, and any rank less than 1.5 means that *most* of the time PRIME's prediction was perfect.

Table 3 shows that PRIME was generally successful in predicting the next method in a sequence given the previous ones. In *all* cases the correct method was within the first dozen methods presented by PRIME, and in the majority of times it was actually the 1st ranked, giving a perfect prediction.

Prediction did not fare as well in JDBC as in the other APIs, mainly because there are various popular ways to use a Connection and ResultSet, leaving no clear advantage to the way described in the tutorial.

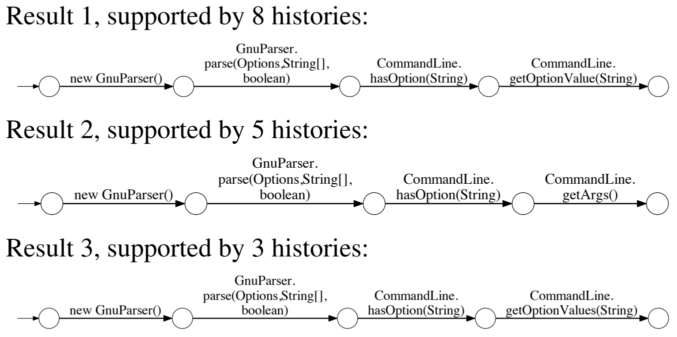
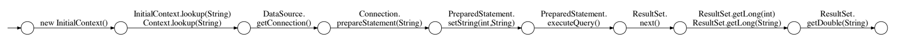
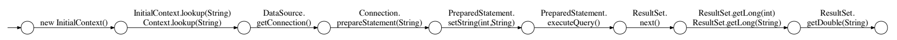
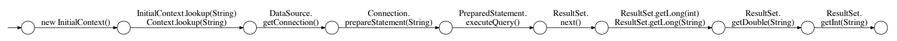

Result 1, supported by 8 histories:
new GnuParser()
GnuParser. parse(Options,String[], boolean)
CommandLine. hasOption(String)
CommandLine. getOptionValue(String)
Result 2, supported by 5 histories:
new GnuParser()
GnuParser. parse(Options,String[], boolean)
CommandLine. hasOption(String)
CommandLine. getArgs()
Result 3, supported by 3 histories:
new GnuParser()
GnuParser. parse(Options,String[], boolean)
CommandLine. hasOption(String)
CommandLine. getOptionValues(String)

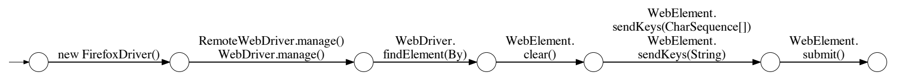
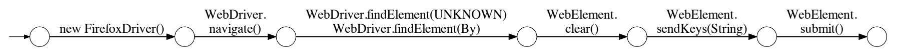
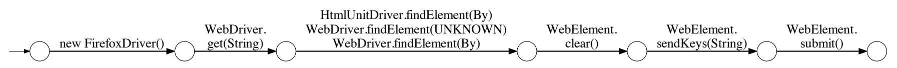
Figure 12: Results of `GnuParser` search query of Table 2 on an index without consolidation. Note that the support is significantly lower. (Difference between results 1 and 3 is in the last call.)

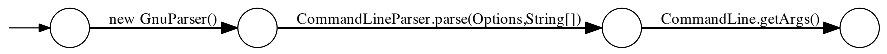
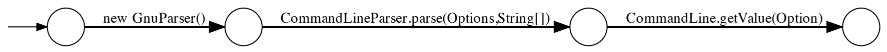
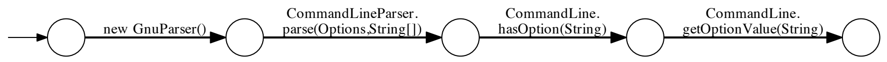
7.3.3 Effect of Consolidation

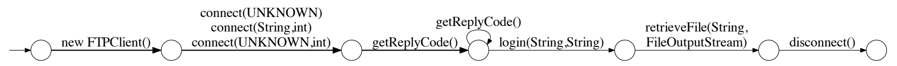
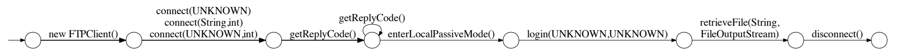
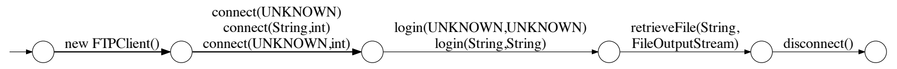
Table 2 shows a number of search queries and the top-ranked results obtained from PRIME when using a consolidated index. When evaluated over a non-consolidated index, only one of these queries can be answered, and its result is shown in Fig. 12. The queries were hand-crafted from official tutorials and documentations, so they always represent correct sequences, but portions of them have been replaced by unknown edges. This replacement is done to demonstrate how a user only partially familiar with the API can still get accurate results even with just partial information about her goals. For each query we show the three highest-ranked sequences extracted from the results matching it.

Query	Consolidated Results

A query which users a cached database manager to obtain a connection, then creates and runs a prepared statement and finally gets a double value form a specific field.
 	
`InitialContext ic =`
`  new InitialContext();`
`ic.?;`
`PreparedStatement ps = ic.?;`
`ResultSet rs = ps.executeQuery();`
`rs.?;`
`double d = rs.getDouble("column");`
	Result 1, supported by 36 histories:
	 new InitialContext()
	InitialContext.lookup(String)
	Context.lookup(String)
	DataSource.
	getConnection()
	Connection.
	prepareStatement(String)
	PreparedStatement.
	setString(int,String)
	PreparedStatement.
	executeQuery()
	ResultSet.
	next()
	ResultSet.getLong(int)
	ResultSet.getLong(String)
	ResultSet.
	getDouble(String)
	Result 2, supported by 36 histories:
	 new InitialContext()
	InitialContext.lookup(String)
	Context.lookup(String)
	DataSource.
	getConnection()
	Connection.
	prepareStatement(String)
	PreparedStatement.
	setString(int,String)
	PreparedStatement.
	executeQuery()
	ResultSet.
	next()
	ResultSet.getLong(int)
	ResultSet.getLong(String)
	ResultSet.
	getDouble(String)
	Result 3, supported by 36 histories:
	 new InitialContext()
	InitialContext.lookup(String)
	Context.lookup(String)
	DataSource.
	getConnection()
	Connection.
	prepareStatement(String)
	PreparedStatement.
	executeQuery()
	ResultSet.
	next()
	ResultSet.getLong(int)
	ResultSet.getLong(String)
	ResultSet.
	getDouble(String)
	ResultSet.
	getInt(String)

Looking for any sequence which uses a specific implementation of a WebDriver, that creates a WebElement from the WebDriver via a specific method, and that clears and then submits a web form.
 	
`WebDriver d = new FirefoxDriver();`
`d.?;`
`By by = ?;`
`WebElement e = d.findElement(by);`
`e.?;`
`e.clear();`
`e.?`
`e.submit();`
	Result 1, supported by 739 histories:
	 new FirefoxDriver()
	RemoteWebDriver.manage()
	WebDriver.manage()
	WebDriver.
	findElement(By)
	WebElement.
	clear()
	WebElement.
	sendKeys(CharSequence[])
	WebElement.
	sendKeys(String)
	WebElement.
	submit()
	Result 2, supported by 478 histories:
	 new FirefoxDriver()
	WebDriver.
	navigate()
	WebDriver.findElement(UNKNOWN)
	WebDriver.findElement(By)
	WebElement.
	clear()
	WebElement.
	sendKeys(String)
	WebElement.
	submit()
	Result 3, supported by 478 histories:
	 new FirefoxDriver()
	WebDriver.
	get(String)
	HtmlUnitDriver.findElement(By)
	WebDriver.findElement(UNKNOWN)
	WebDriver.findElement(By)
	WebElement.
	clear()
	WebElement.
	sendKeys(String)
	WebElement.
	submit()

Aiming to obtain a String from command-line arguments that adhere to the GNU style. The query only requires initialization of a GnuParser instance, and that at a certain point a String will be obtained from it.
 	
`GnuParser p = new GnuParser();`
`p.?;`
`String s = p.?;`
`s.?;`
	Result 1, supported by 312 histories:
	 new GnuParser()
	CommandLineParser.parse(Options,String[])
	CommandLine.getArgs()
	Result 2, supported by 273 histories:
	 new GnuParser()
	CommandLineParser.parse(Options,String[])
	CommandLine.getValue(Option)
	Result 3, supported by 20 histories:
	 new GnuParser()
	CommandLineParser.
	parse(Options,String[])
	CommandLine.
	hasOption(String)
	CommandLine.
	getOptionValue(String)

A query typical of a user who wants to retrieve a file via FTP and only knows she needs to login first.
 	
`FTPClient c = ?;`
`c.login("username", "password");`
`c.?;`
`c.retrieveFile("file path", out);`
	Result 1, supported by 74 histories:
	 new FTPClient()
	connect(UNKNOWN)
	connect(String,int)
	connect(UNKNOWN,int)
	getReplyCode()
	getReplyCode()
	login(String,String)
	retrieveFile(String,
	FileOutputStream)
	disconnect()
	Result 2, supported by 74 histories:
	 new FTPClient()
	connect(UNKNOWN)
	connect(String,int)
	connect(UNKNOWN,int)
	getReplyCode()
	getReplyCode()
	enterLocalPassiveMode()
	login(UNKNOWN,UNKNOWN)
	retrieveFile(String,
	FileOutputStream)
	disconnect()
	Result 3, supported by 74 histories:
	 new FTPClient()
	connect(UNKNOWN)
	connect(String,int)
	connect(UNKNOWN,int)
	login(UNKNOWN,UNKNOWN)
	login(String,String)
	retrieveFile(String,
	FileOutputStream)
	disconnect()

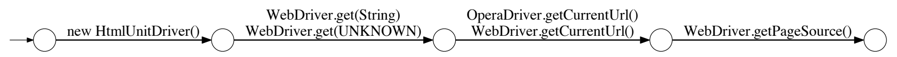
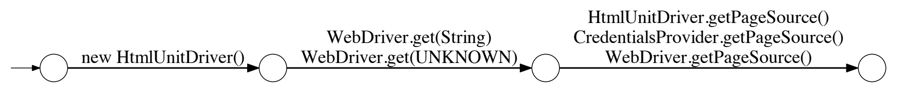
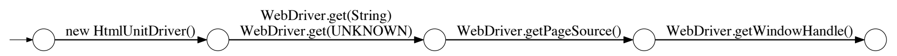
This query tries to find out how to get the web page's source using the simple HtmlUnit browser.
 	
`WebDriver d = new HtmlUnitDriver();`
`d.?;`
`d.getPageSource();`
	Result 1, supported by 4k histories:
	 new HtmlUnitDriver()
	WebDriver.get(String)
	WebDriver.get(UNKNOWN)
	OperaDriver.getCurrentUrl()
	WebDriver.getCurrentUrl()
	WebDriver.getPageSource()
	Result 2, supported by 3k histories:
	 new HtmlUnitDriver()
	WebDriver.get(String)
	WebDriver.get(UNKNOWN)
	HtmlUnitDriver.getPageSource()
	CredentialsProvider.getPageSource()
	WebDriver.getPageSource()
	Result 3, supported by 603 histories:
	 new HtmlUnitDriver()
	WebDriver.get(String)
	WebDriver.get(UNKNOWN)
	WebDriver.getPageSource()
	WebDriver.getWindowHandle()

Table 2: Results of search queries over an index with consolidated partial specifications. For all of these queries but the 3rd no results were found when the index construction does not use consolidation. For the 3rd query, results without consolidation are shown in Fig. 12.

```
GnuParser p = new GnuParser();
p.?;
String s = p.?;
s.?;
```

Figure 13: Obtaining a String from a command-line parser

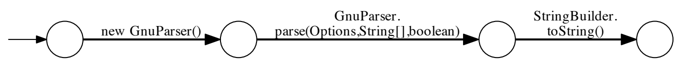


Figure 14: A result for the query of Fig. 13 when the analysis was done on a single global object. Results of PRIME for this query are shown in Table 2.

For `FTPClient`, we omitted the prefix `FTPClient` from all calls so we can fit the histories into the figure.

Of interest are the first row and the last row. In the first row, PRIME demonstrates the ability to complete very long sequences when required to do so by the query. The event names are difficult to read in this example, but the important property is the length of the sequences. In the last row, the results exhibit very high support values, since the query describes an extremely common use-case of scraping the source of a web page (extracting information from the HTML) using the simplest available browser. Note that despite of the popularity of this usage scenario, a match for the query was not found on a non-consolidated index.

As shown in Fig. 12, in the (only) case where matches to the query were found also in the non-consolidated index, the support of each result is still much higher when searching in a consolidated index.

7.4 Importance of Tracking Typestate

To demonstrate the importance of tracking typestate for individual (abstract) objects in the context of search we show an example of the results returned when the distinction between abstract objects is removed and all objects are considered to be the same single global object. We do that by running a modified version of PRIME which does not distinguish between abstract objects. This has the effect of tracking global event sequences. The query in Fig. 13, for instance, was answered by the sequence in Fig. 14, which is not helpful, as it contains the unrelated `StringBuilder.toString()` call after the parse, which cannot be invoked on a `GnuParser`. In contrast, the results of PRIME for this query are shown as part of Table 2. We have observed similar loss of precision for most other APIs we considered.

Attempting to address this limitation by filtering the objects by types is not feasible, since with partial samples types are not known and the type hierarchy may not be available.

7.5 Comparison with Related Work

Prospector Since the analysis is aware of creation context, PRIME can theoretically find many of the type conversions found by Prospector ([27]), by composing a query containing both in input type and output type. For instance, [27] shows a conversion between `IWorkbenchPage` and `IDocumentProvider`; given the query containing both types, PRIME was able to find and analyze a relevant example as seen in Fig. 15. With this approach, running on the 20 examples provided in [27], PRIME found 8 of the 18 conversions identified by Prospector, as well as one conversion not identified by Prospector (`IWorkspace` $\rightarrow$ `IFile`). Prospector could identify the remaining conversions primarily because they rely on a connection between method *arguments* and return value, which our concept of creation context ignores, though it can be extended to include it.

MAPO We have experimented with the GEF framework (e.g. Table 1), which was also used as the experimental evaluation of MAPO ([44]). PRIME's ability to deal with partial code snippets allowed PRIME to easily obtain a lot more snippets to work with for the GEF API than was possible with MAPO [44], enabling a higher support for each result. Note that despite the partialness of the snippets being used, PRIME still performs a more precise typestate analysis with aliasing support.

8. Related Work

Our work mines temporal specifications as the basis for code search. There has been a lot of work on specification mining, recommendation systems, and various forms of semantic code-search. In this section, we survey some of the closely related work. We note that there are other lines of related research such as clone detection and code comparison (e.g. [19]) that can provide alternative similarity measures between snippets. For example, some works on clone detection considered syntactic information such as the tokens that appear in each sample (e.g., [11, 20]), other works are based on ASTs which maintain some structural information (e.g., [5, 19, 25, 38]), or on more semantic information based on program dependence graphs (e.g., [13, 23, 24]). However, this is not the focus of this work.

Code Search and Recommendation

Several approaches addressing the problem of semantic code search and its variation were proposed in the literature.

MAPO [44] uses API usage patterns as the basis for recommending code snippets to users. Their work differs from our work in several crucial aspects: (i) MAPO does not deal with missing parts of an implementation. As a result it does not handle arbitrary code snippets, such as many of the examples found online, nor their challenges. Our approach handles arbitrary partial programs, and uses consolidation techniques to derive from them a much more complete view of the API than obtained from individual methods. (ii) MAPO's analysis tracks global sequences of method invocations on various types, disregarding their association with individual objects, resulting in noise reflecting mixed usage patterns of multiple objects. Our work tracks the receiver of an event even in the presence of aliasing,

API/Type	Nature of sequence (source)	Length	Avg Rank	Median Rank	Max Rank
Apache Commons Net / FTPClient	Upload a file (official tutorial)	7	1.14	1	2
Apache Commons CLI / CommandLine	Parse a command-line and get values from it (official usage guide)	4	1.50	1	2
Apache Ant / CommandLine.Argument	Prepare the executable and argument of a command-line (online tutorial)	4	1.25	1	2
Apache Ant / Path	Create a path element and append it to existing and boot paths (authors)	6	1.33	1	3
JDBC / ResultSet	Run a query and iterate over the results (many online tutorials)	8	3.13	2	10
JDBC / Connection	Commit and then rollback a transaction (official tutorial)	5	4.60	3	12
Eclipse JDT / ITypeBinding	Get the key of an array element type (authors)	5	4.20	1	12
Eclipse JDT / IProjectDescription	Get the description and nature IDs of a Java project (online tutorial)	4	2.00	1	4
Eclipse UI / PluginAction	Create a new action (online tutorial)	5	1.00	1	1
Eclipse UI / IEditorInput	Get the input for the current editor (online tutorial)	5	2.80	3	5

Table 3: Experimental Results - Prediction

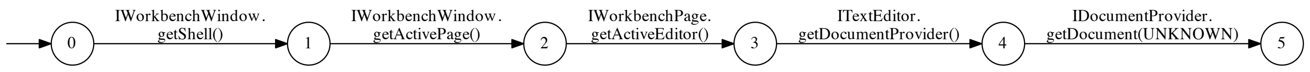

IWorkbenchWindow. getShell()
IWorkbenchWindow. getActivePage()
IWorkbenchPage. getActiveEditor()
ITextEditor. getDocumentProvider()
IDocumentProvider. getDocument(UNKNOWN)
0
1
2
3
4
5

Figure 15: `IDocumentProvider` example found by querying over both `IWorkbenchPage` and `IDocumentProvider`. The conversion between the types appears between states 2 and 4.

through method calls, drastically reducing noise from surrounding objects and method calls. (iii) MAPO mines simple sequences ignoring loops, whereas we mine generalized typestate. (iv) While we use relaxed inclusion to find similarities between typestates, MAPO clusters the mined sequences by various clustering techniques. The consideration of such techniques for typestates is the subject of future work.

Strathcona [17] matches the structure of the code under development to the code in the examples. The query in this case is implicit and consists of the prefix of the currently written code. The search is performed over a sample repository (e.g., the existing project), thus no partial code fragments are considered. The search is based on structural context which comprises details of the method being written, its containing class and previous methods invoked. Temporal information such as the order of method invocations is not considered.

Mandelin et al. [27] use static analysis to infer a sequence of code (*jungloid*) that shows the programmer how to obtain a desired target type from a given source type. This code-sequence is only checked for type-safety and does not address the finer notion of typestate. Thummalapenta and Xie [36] introduce a tool called PARSEWeb to expand on Mandelin's approach by gathering samples online, partially-compiling them and analyzing the results with a simple static analyzer. We employ a similar technique in the first phases of our solution, and we draw from their experience. However, like with Mandelin's work, their analysis is only concerned with the object types appearing in code sequences. Moreover, their approach is AST-based and does not perform a deeper semantic analysis tracking objects.

XSnippet [31] uses queries that refer to object instantiations, possibly with some additional context from the user's code. Their analysis is based on a graph representation of the code, which describes the types, methods and fields appearing in the code, but does not track objects and sequences of operations applied on them.

Alnusair et al. [2] use ontologies to represent semantic information about object instantiation sequences. They use an interprocedural points-to analysis to obtain a precise return type for API methods based on the framework code. This allows them to rely on library-side semantic information rather than relying just on information from snippets. Considering richer ontological models of library code seems like a promising direction that can complement the semantic information we use in our approach.

Kim et al. [21] search code for the purpose of attaching code examples to documentation. Their index construction is based on intraprocedural AST-based analysis and considers each snippet as a full use case. Their search is based on method names. This approach is too crude to provide quality results for the kind of queries we address in this paper.

Reiss [30] uses a combination of class or method signatures and dynamic specifications such as test-cases and contracts supplied by a user as a basis for semantic code search. The candidate code snippets, initially obtained by textual search, undergo several transformations aimed at generating candidates that match the signature provided by the user. Matches among these candidates are then found by dynamically checking the test-cases (and additional dynamic specifications if exist). Our approach does not require the user to supply test-cases or their kind as a part of the query. In addition, we do not consider the difficult problem of synthesizing executable code, which makes the usage of test-cases inapplicable. In cases where the results are indeed executable, we can benefit from a similar dynamic approach to find matches to the query.

Specification Mining

Dynamic Specification Mining There has been a lot of past work on dynamic specification mining for extracting various forms of temporal specifications (e.g., [4, 7, 10, 26, 42, 43]). Dynamic specification mining does not suffer from the difficulties inherent to abstraction required in static analysis. Because our focus on this paper is on analysis of code snippets, employing dynamic analysis would be extremely challenging. Still, when it is feasible to run a program with adequate coverage, dynamic analysis represents an attractive option for specification mining.

Component-side Static Analysis In component-side static analysis, a tool analyzes a component's implementation, and infers a specification that ensures the component does not fail in some predetermined way, such as by raising an exception. For example, Alur et al. [3] use Angluin's algorithm and a model-checking procedure to learn a permissive interface of a given component. In contrast, client-side mining produces a specification that represents the usage scenarios in a given code-base. The two approaches are complementary, as demonstrated in [42]. Our index construction in this paper performs client-side specification mining.

Client-side Static Analysis Many papers have applied static analysis to client-side specification mining.

Weimer and Necula [41] use a simple, lightweight static analysis to infer simple specifications from a given codebase. Their insight is to use exceptional program paths as negative examples for correct API usage. We believe that our approach could also benefit from using exceptional paths as negative examples. Weimer and Necula learn specifications that consist of pairs of events $\langle a, b \rangle$, where a and b are method calls, and do not consider larger automata. They rely on type-based alias analysis, and so their techniques should be much less precise than ours. On the other hand, their paper demonstrates that even simple techniques can be surprisingly effective in finding bugs.

Monperrus et al. [29] attempt to identify missing method calls when using an API by mining a codebase and sharing our assumption that incorrect usage will be infrequent. They only compare objects that have identical type and same containing method signature, which only works for inheritance-based APIs. Their approach deals with identical histories or identical histories minus k method calls, and unlike PRIME it cannot handle incomplete programs, non-linear method call sequences, and general code queries.

Wasylkowski, Zeller, and Lindig [40] use an intraprocedural static analysis to automatically mine object usage patterns and identify usage anomalies. Their approach is based on identifying usage patterns, in the form of pairs of events, reflecting the order in which events should be used. In contrast, our work mines temporal specifications that over-approximate the usage scenarios in a code-base.

The work of [16] is similar in spirit, but more lightweight. Here too, specifications are only pairs of events, and are used to detect anomalies.

Acharya et al. [1] also mine pairs of events in an attempt to mine partial order between events. Their analysis is for C, which is a fundamental difference since it is not an object-oriented language.

Wasylkowski and Zeller [39] mine specifications (operational preconditions) of method parameters in order to detect problems in code. They use intraprocedural analysis, without any pointer analysis. The mined specifications are CTL formulas that fit into several pre-defined templates of formulas. Therefore, the user has to know what kind of specifications she is looking for. In addition, no consolidation of partial specifications is applied.

Shoham et al. [32] use a whole-program analysis to statically analyze clients using a library. Their approach is limited to single-object typestate. More importantly, their approach is not applicable in the setting of partial programs since they rely on the ability to analyze the complete program for complete alias analysis and for type information. The transition to partial programs and partial specifications is a significant departure from this work. Other than the additional challenges during the analysis, dealing with partial specifications raises new challenges while processing the results. In [32] the focus was on reducing noise, whereas a significant part of our focus is on consolidating the partial specifications into complete ones. In particular, partial specifications include unknown events (?-transitions). To that end, we suggest unknown elimination and relaxed-inclusion techniques which are different in implementation as well as general goal.

9. Conclusion

We present a semantic code search approach capable of searching over arbitrary code snippets, including *partial snippets*, such as the ones obtained from expert code sites. Our search is based on novel static analysis techniques for specification mining which (a) extract partial temporal specifications in the form of *typestate with creation context* from (partial) code snippets, and (b) *consolidate* partial specifications in a way that completes their unknowns whenever possible. In order to answer code search queries, we propose a new notion of *relaxed inclusion* tailored for partial specifications with unknowns. We show that our approach is useful for answering code search queries dealing with how an API is used.

Acknowledgements

We would like to thank Barthelemy Dagenais, whose work on a partial compiler ([9]) enabled us to approach otherwise-problematic partial code snippets, and who was generously willing to extend his tool for our needs. We would like to thank Hongseok Yang for many insightful discussions.

This research was partially supported by The Israeli Science Foundation (grant no. 965/10), and the German-Israeli Foundation for Scientific Research and Development (grant no. 2248-2045.6).

References

[1] ACHARYA, M., XIE, T., PEI, J., AND XU, J. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07*, pp. 25–34.

[2] ALNUSAIR, A., ZHAO, T., AND BODDEN, E. Effective API navigation and reuse. In *IRI* (aug. 2010), pp. 7 –12.

[3] ALUR, R., CERNY, P., MADHUSUDAN, P., AND NAM, W. Synthesis of interface specifications for Java classes. In *POPL* (2005).

[4] AMMONS, G., BODIK, R., AND LARUS, J. R. Mining specifications. In *POPL'02*, pp. 4–16.

[5] BAXTER, I. D., YAHIN, A., MOURA, L., SANT'ANNA, M., AND BIER, L. Clone detection using abstract syntax trees. In *ICSM '98*.

[6] BECKMAN, N., KIM, D., AND ALDRICH, J. An empirical study of object protocols in the wild. In *ECOOP'11*.

[7] COOK, J. E., AND WOLF, A. L. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol. 7*, 3 (1998), 215–249.

[8] COUSOT, P., AND COUSOT, R. Modular static program analysis, invited paper. April 6—14 2002.

[9] DAGENAIS, B., AND HENDREN, L. J. Enabling static analysis for partial Java programs. In *OOPSLA'08*, pp. 313–328.

[10] DALLMEIER, V., LINDIG, C., WASYLKOWSKI, A., AND ZELLER, A. Mining object behavior with ADABU. In *WODA '06*.

[11] DUCASSE, S., RIEGER, M., AND DEMEYER, S. A language independent approach for detecting duplicated code. In *ICSM '99*.

[12] FINK, S., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. Effective typestate verification in the presence of aliasing. In *ISSTA'06*, pp. 133–144.

[13] GABEL, M., JIANG, L., AND SU, Z. Scalable detection of semantic clones. In *ICSE '08*, pp. 321–330.

[14] GABEL, M., AND SU, Z. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE'08*.

[15] github code search. https://github.com/search.

[16] GRUSKA, N., WASYLKOWSKI, A., AND ZELLER, A. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *ISSTA '10*.

[17] HOLMES, R., AND MURPHY, G. C. Using structural context to recommend source code examples. In *ICSE '05*.

[18] HOLMES, R., WALKER, R. J., AND MURPHY, G. C. Strathcona example recommendation tool. In *FSE'05*, pp. 237–240.

[19] JIANG, L., MISHERGHI, G., SU, Z., AND GLONDU, S. Deckard: Scalable and accurate tree-based detection of code clones. IEEE Computer Society, pp. 96–105.

[20] KAMIYA, T., KUSUMOTO, S., AND INOUE, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng. 28*, 7 (2002).

[21] KIM, J., LEE, S., WON HWANG, S., AND KIM, S. Towards an intelligent code search engine. In *AAAI'10*.

[22] Koders. http://www.koders.com/.

[23] KOMONDOOR, R., AND HORWITZ, S. Using slicing to identify duplication in source code. In *SAS '01*, pp. 40–56.

[24] KRINKE, J. Identifying similar code with program dependence graphs. In *WCRE* (2001), pp. 301–309.

[25] LIVIERI, S., HIGO, Y., MATUSHITA, M., AND INOUE, K. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE'07*.

[26] LO, D., AND KHOO, S.-C. SMArTIC: towards building an accurate, robust and scalable specification miner. In *FSE'06*.

[27] MANDELIN, D., XU, L., BODIK, R., AND KIMELMAN, D. Jungloid mining: helping to navigate the API jungle. In *PLDI '05*, pp. 48–61.

[28] MISHNE, A. Typestate-based semantic code search over partial programs. Master's thesis, Technion-Israel Institute of Technology, Haifa, Israel, 2012.

[29] MONPERRUS, M., BRUCH, M., AND MEZINI, M. Detecting missing method calls in object-oriented software. In *ECOOP'10* (2010), vol. 6183 of *LNCS*, pp. 2–25.

[30] REISS, S. P. Semantics-based code search. In *ICSE'09*.

[31] SAHAVECHAPHAN, N., AND CLAYPOOL, K. XSnippet: mining for sample code. In *OOPSLA '06*.

[32] SHOHAM, S., YAHAV, E., FINK, S., AND PISTOIA, M. Static specification mining using automata-based abstractions. In *ISSTA '07*.

[33] SOLAR-LEZAMA, A., RABBAH, R., BODÍK, R., AND EBCIOĞLU, K. Programming by sketching for bit-streaming programs. In *PLDI '05*.

[34] stackoverflow. http://stackoverflow.com/.

[35] STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng. 12*, 1 (1986), 157–171.

[36] THUMMALAPENTA, S., AND XIE, T. PARSEWeb: a programmer assistant for reusing open source code on the web. In *ASE'07*, pp. 204–213.

[37] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot - a Java bytecode optimization framework. In *CASCON '99*, IBM Press, pp. 13–.

[38] WAHLER, V., SEIPEL, D., WOLFF, J., AND FISCHER, G. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation* (2004).

[39] WASYLKOWSKI, A., AND ZELLER, A. Mining temporal specifications from object usage. In *Autom. Softw. Eng.* (2011), vol. 18.

[40] WASYLKOWSKI, A., ZELLER, A., AND LINDIG, C. Detecting object usage anomalies. In *FSE'07*, pp. 35–44.

[41] WEIMER, W., AND NECULA, G. Mining temporal specifications for error detection. In *TACAS* (2005).

[42] WHALEY, J., MARTIN, M. C., AND LAM, M. S. Automatic extraction of object-oriented component interfaces. In *ISSTA'02*.

[43] YANG, J., EVANS, D., BHARDWAJ, D., BHAT, T., AND DAS, M. Perracotta: mining temporal API rules from imperfect traces. In *ICSE '06*, pp. 282–291.

[44] ZHONG, H., XIE, T., ZHANG, L., PEI, J., AND MEI, H. MAPO: Mining and recommending API usage patterns. In *ECOOP'09*.