Lightweight Syntactic API Usage Analysis with U Cov
Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, Jean-Rémy Falleri

To cite this version:
Gustave Monce, Thomas Couturou, Yasmine Hamdaoui, Thomas Degueule, Jean-Rémy Falleri. Lightweight Syntactic API Usage Analysis with U Cov. 32nd IEEE/ACM International Conference on Program Comprehension (ICPC 2024), Apr 2024, Lisboa, Portugal. 10.1145/3643916.3644415 . hal-04463475

HAL Id: hal-04463475
https://hal.science/hal-04463475
Submitted on 17 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Lightweight Syntactic API Usage Analysis with UCov

Gustave Monce
gustave.monce@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Thomas Couturou
thomas.couturou@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Yasmine Hamdaoui
yasmine.hamdaoui@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Thomas Degueule
thomas.degueule@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

Jean-Rémy Falleri
falleri@labri.fr
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
France

ABSTRACT
Designing an effective API is essential for library developers as it is the lens through which clients will judge its usability and benefits, as well as the main friction point when the library evolves. Despite its importance, defining the boundaries of an API is a challenging task, mainly due to the diverse mechanisms provided by programming languages that have non-trivial interplays. In this paper, we present a novel conceptual framework designed to assist library maintainers in understanding the interactions allowed by their APIs via the use of syntactic usage models. These customizable models enable library maintainers to improve their design of APIs and to reflect on the adequacy of their tests and documentation. We implement these models for Java libraries in a new tool UCov and demonstrate its capabilities on three libraries exhibiting diverse styles of interaction: JSOUP, COMMONS-CLI, and SPARK. Our exploratory case study shows that UCov provides valuable information regarding API design and fine-grained analysis of client code to identify under-tested and under-documented library code.

CCS CONCEPTS
- Software and its engineering → Software libraries and repositories, Software maintenance tools; Software design engineering.

KEYWORDS
software library, API design, API usage, API documentation

ACM Reference Format:

1 INTRODUCTION
Application Programming Interfaces (APIs) govern the interactions between a library and client projects using it. In particular, they specify which elements of a library (e.g., types, methods, fields) can be accessed from the outside and the valid interactions with them. Designing an effective API is essential for library developers as it is the lens through which clients judge its usability and benefits and the main friction point when the library evolves [23, 28]. Despite its importance, defining the boundaries of an API is a challenging task, mainly due to the diverse mechanisms provided by programming languages and paradigms that either aim specifically at designing APIs or indirectly affect them as a side effect (e.g., visibilities, subclass restriction and sealing, name binding rules, overloading and overriding, polymorphism).

Given the complex interplay among those mechanisms, it is difficult for library maintainers to maintain an exhaustive and accurate mental model of their API and the interactions it enables. This complexity is compounded by the various styles libraries can adopt (e.g., frameworks and inversion of control, fluent interfaces) and the variety of ways that client code can interact with individual API elements. For instance, methods may either be invoked directly or overridden, and non-abstract classes may be instantiated, extended, referenced, etc. This impairs the maintainers’ ability to fully understand how their APIs are—or could be!—utilized in the wild. As a result, libraries sometimes inadvertently allow some kinds of interactions that their developers did not intend. This led to best API design practices such as “minimize accessibility” or “forbid subclassing by default” that are widespread in different communities (e.g., in Java [1]). To minimize the likelihood of bad outcomes on the client side, library developers should restrict the possible interactions with their APIs to those that are intended and validated. Indeed, unintended interactions are a recipe for buggy client code and frustrated developers who may opt for an alternative library providing similar services with better support. However, there is no lightweight, effective way of extracting and maintaining a transparent, comprehensive API model, nor is there a way to validate the extent to which the interactions with an API are validated, documented, and representative of actual uses. Current approaches in the literature can successfully analyze the use of API symbols but do not account for the variety of possible interactions with an individual symbol [12, 26].
In this paper, we introduce a novel conceptual framework aimed at helping library maintainers better understand the boundaries of their APIs via the use of so-called **syntactic usage models** (SUMs). SUMs enable maintainers to reason about the interactions allowed by their APIs throughout their evolution. Based on the SUMs, we define **syntactic usage footprints** (SUFs), which measure the extent to which a given piece of code using the library (e.g., third-party clients, documentation samples, library tests) exercises its API and covers its possible uses. We show how different SUFs can be easily compared, enabling library maintainers to answer questions such as “do our tests validate the interactions found in third-party clients?” or “do the examples in our documentation align with in-the-wild uses of our API?” (Section 3).

We present UCov (Usage COVerage), an implementation of these models for the Java programming language that leverages static analysis to automatically infer them from Java source code (Section 4). We illustrate the benefits of UCov and the underlying models with an exploratory case study of three libraries implementing various styles of interaction: jsoup, a library for HTML manipulation; Apache commons-cli, a library for implementing command-line interfaces; and spark, a simple web framework. From our experiments with UCov, we lay out a research agenda for syntactic models and envision application scenarios in other areas (Section 5).

Overall, our syntactic models and UCov provide a novel means for library maintainers to understand the extent of possible interactions allowed by their APIs and to oversee their use in client code. By offering a robust conceptual framework for usage coverage and a first implementation for Java, we aim to empower library maintainers with the insights needed to improve their API’s design, documentation, and tests, thereby reducing friction with client code and fostering long-term maintenance.

2 BACKGROUND AND MOTIVATION

Understanding how client code interacts with APIs is instrumental in prioritizing development and documentation efforts. Therefore, numerous studies have examined methods to extract usage data from client code to identify notable **hotspots** and **coldspots**, i.e., parts of the APIs that are over- or under-utilized [30–32]. In a recent study, Qiu et al. delved into the usage of Java’s standard library and third-party libraries across a corpus of over 5,000 projects, encompassing 150M+ lines of code [26]. Various tools and studies adopt distinct methodologies for gathering usage data, whether from bytecode [12], source code and resolved ASTs [6, 18, 26], or other sources [31]. A unifying theme across these works and their underlying models of API usage is their focus on determining whether a specific API symbol is accessed in client code, rather than exploring how it is used. This emphasis stems from the primary objectives of these studies: assess the frequency, popularity, and coverage of API symbols in client code, rather than exploring the diverse uses of these symbols. Indeed, the way a symbol is used has no influence on its popularity. Extending, instantiating, or simply referencing a class all contribute equally to its popularity.

We argue that this approach to modeling library usage is too coarse-grained to allow library maintainers to understand fully the implications of the design of their API. For instance, let us consider a library designer providing a simple `public class C` and a client instantiating it with `C c = new C()`. Existing usage models would indicate that there is one exported symbol (C) and that this symbol is used in client code. Based on this information, library maintainers could conclude that the design of their API is satisfactory. However, this declaration does not prevent clients from subclassing the provided class with `class ClientC extends C`. The aforementioned models do not provide any means to distinguish this particular use from other uses in client code. This is a missed opportunity, as library maintainers could have reconsidered the API early on to prevent subclassing, disallowing clients from extending the class, and thus freeing themselves from supporting this scenario. The approaches mentioned above are oblivious to this distinction, while our work emphasizes it.

There is a wide variety of literature addressing the problem of mining and recommending API usage patterns and protocols [35, 42]. These studies typically analyze library code and client code to infer automata and probabilistic graphs that represent legal sequences of API invocations and check whether client code complies with them. While these approaches go beyond our objectives and consider some kind of semantic relation between API elements, they are unable to differentiate between the different interactions allowed for a specific element. In contrast to previous works, our approach emphasizes the diverse ways a particular symbol may be utilized in client code, leveraging the **syntactic usage models** introduced in the next section.

3 SYNTACTIC API USAGE

Our analysis of API usage is built upon two distinct models: syntactic usage models, which are extracted from library code and represent the **legal** (possible) uses of an API (Section 3.1), and syntactic usage footprints, which are extracted from client code and represent the **actual** uses of an API (Section 3.2). From these two models, we derive various metrics, including a dedicated API coverage metric (Section 3.3). These models are formulated independently of any specific programming language and can be adapted for diverse analysis scenarios. However, we reference Java syntax and semantics throughout this section for illustrative purposes.

3.1 Syntactic Usage Model (SUM)

We consider that a library defines a set of symbols $S$ (i.e., named entities), each of a particular kind and with a declaration that specifies its properties. For instance, the Java standard library defines the symbol `public class ArrayList<E>` of kind `Class`, the symbol `public final static PrintStream out` of kind `Field`, and the symbol `public void println()` of kind `Method`. These symbols can be exported, allowing client code to access them. Together, the exported symbols of a library form its API. Given a function $\text{exported} : S \rightarrow \{\top, \bot\}$ indicating whether a given symbol is exported ($\top$) or not ($\bot$), derived from the language’s semantics, the API of a library is the set $\mathcal{A} \subseteq S$ such that $\mathcal{A} = \{s \in S : \text{exported}(s) = \top\}$. In Java, for instance, visibilities and modules are the primary mechanisms for controlling symbol exports. As an

---

1. https://jsoup.org/
2. https://commons.apache.org/proper/commons-cli/
While the SUM holds the legal, possible uses of an API, the SUF materializes actual uses of the API in a given piece of client code. Formally, the footprint of a client on a SUM is the set of triples \( F = \{ (s, u, l) : s \in \mathcal{A}, u \in \text{uses}(s), l \in \mathcal{L} \} \) where \( s \) denotes the API symbol being used, \( u \) the kind of use (e.g., \{Referenced, Invoked\}), and \( l \) the location of the use in client code (e.g., a physical line-column location). As the definition implies, there can be multiple identical uses of the same symbol in different locations.

Syntactic usage footprints materialize the diversity of ways a given API can be used in client code. For instance, Listing 2 depicts a classical interaction with the \textit{ArrayList} API through instantiation and invocations. On the other hand, Listing 3 depicts a typical framework-like interaction with the same API through extension and overriding. This latter style is prevalent in frameworks such as web servers or batch processing frameworks (e.g., Spring, Hadoop, Spark) that heavily employ inversion of control and the Hollywood principle ("Don’t call us, we’ll call you"). As Table 1 shows, these two snippets yield two disjoint and complementary SUFs. Importantly, when looking at multiple clients, their SUFs can trivially be joined through set union and compared through set difference.

### 3.3 Usage Coverage and Metrics

The SUM gives the universe of legal uses to cover for a given API and the SUF pinpoints those that are actually covered in client code. The set of API symbols that are covered in a SUF is denoted \( C_R = \{ s : (s, u, l) \in F \} \) and the set of covered API uses is denoted \( C_M = \{ u : (s, u, l) \in F \} \). Then, the API symbol coverage score of the SUF with respect to the API is \( \frac{|C_R|}{|\mathcal{A}|} \) and its API use coverage score is \( \frac{|C_M|}{|\mathcal{A}|} \). Following these definitions, Listing 2 covers 50% of Listing 1’s API uses and 100% of its symbols, so does Listing 3. Together, they cover 100% of the API uses. One can identify the exported symbols that are not covered in client code by exploring \( \mathcal{A} \setminus C_R \) and the legal uses that are not realized by exploring \( \mathcal{M} \setminus C_M \). Interestingly, one may reproduce the popularity metrics used to identify hotspots and coldspots in the literature [26, 30] by counting the occurrences of \((s, u, l) \in F\) for a given symbol of interest \( s \).
We distinguish between three levels of coverage for each individual API symbol. An API symbol \( s \in \mathcal{A} \) is fully covered if all of its uses are present in the SUF, i.e., if \( \forall u \in \text{uses}(s), \exists(s, u, l) \in \mathcal{F} \). Conversely, an API symbol is not covered if none of its uses are present in the SUF, and it is partially covered if only some of its uses are present in the SUF.

4 UCov: Java syntactic usage analysis

UCov is our proof-of-concept implementation of syntactic API usage analysis for Java libraries.\(^4\) It parses and analyzes the source code of Java libraries to produce syntactic usage models and their coverage. When implementing such a tool, one must decide on the symbols of interest to include in the API and the types of uses to represent, based on language semantics and analysis goals, as highlighted in Section 3. In this section, we discuss some of the choices we made while implementing UCov to support our case studies and the overall architecture of the tool, depicted in Figure 1.

4.1 Exported Symbols

SUM models employ a simple exported : \( S \to \{\top, \bot\} \) function that indicates whether a given symbol \( s \) is exported. From the library’s code, UCov first extracts the list of all API symbols (namely, for Java, types, methods, and fields) uniquely identified by their fully qualified name (and their signature in the case of methods). Then, it distinguishes those that can be accessed from the outside, in client code, and those that cannot. Following the rules described in the Java 17 Language Specification \([11]\), the following symbols can be accessed from client code:

**Public symbols:** (transitively) public types, as well as public methods and fields within public types, can be accessed from anywhere without restriction;

**Protected symbols:** protected fields and methods within effectively extensible public types can be accessed through subtyping or by code located in the same package. An effectively extensible type is one that is not final nor sealed and that, in the case of classes, possesses a public or protected constructor the subclass can access;

**Package-private symbols:** package-private symbols (Java’s default visibility in the absence of a visibility modifier) can be accessed by code located in the same package.

The only legal visibilities for a top-level type declaration are public and package-private, so API designers can only use the latter when they want to hide a type declaration from the outside. UCov’s implementation of the exported function thus considers the public and protected symbols and discards the package-private ones, as client code intentionally using a package of the same name as the library’s to access its package-private symbols is a pathological case that most likely breaches the API’s intent.

4.2 Supported API Symbol Uses

Once the exported symbols of a library are identified, UCov maps each of these to a set of legal uses, based on its declaration specifics, thereby implementing the uses function. Table 3 gives an excerpt of the uses we consider for each kind of symbol.

In a nutshell, UCov looks for every possible use of type references (as parameter types, return types, exceptions, etc.) and categorizes them all as Type reference uses. Classes can be instantiated if they provide a public (possibly default) constructor and are not abstract. They can be inherited unless declared final. Interfaces can be implemented by client classes or extended by other interfaces. Methods can be invoked (statically or dynamically) and overridden unless declared final. When polymorphism comes into play in client code and the static analysis cannot determine the dynamically invoked method beyond the static type of its receiver object, UCov registers an invocation to the corresponding method in the static type of the receiver object, as well as invocations to all super-methods with the same signature in its hierarchy. As a result, abstract methods in abstract classes and interfaces are considered covered whenever a more concrete implementation is invoked in client code. For instance, an invocation of \( \text{ArrayList.size}() \) in client code would cover \( \text{List.size}() \). Finally, UCov distinguishes between read and write accesses to an exported field.

Note that these choices aim to reflect customary uses of Java APIs and are, to some extent, arbitrary. Another implementation may, for instance, distinguish between various references to a type or unify read and write accesses to a field under the same kind of use. These choices will vary depending on the task at hand and analysis objectives.

4.3 Implementation

UCov is implemented in Java and uses the Spoon framework \([25]\) to parse Java code and create visitors that implement the necessary static analyses.

When supplied with library code, UCov uses Spoon to build a resolved AST and applies a dedicated visitor on types, methods, and fields to list the exported API symbols, thereby implementing the exported function (upper part of Figure 1). Then, it maps each of these symbols to the corresponding uses (\(\times\)) according to the rules in Section 4.2, forming the final syntactic usage model.

When supplied with client code (library tests, library samples, code of third-party clients, or any arbitrary snippet using the library), UCov builds its resolved AST and visits it to identify every
node that references, in one way or another, symbols of the API. For example, it analyzes method invocations, field accesses, and type references to determine if they correspond to any of the uses identified in the syntactic usage model. Table 3 lists an excerpt of the elements in client code that the static analyzer searches for and the information it collects for each. The resulting set of actual uses in client code constitutes its syntactic usage footprint.

### 5 EXPLORATORY CASE STUDY

In this section, we evaluate the capability of syntactic usage models and footprints, as well as their implementation for Java libraries in UCov, to produce meaningful information that assists maintainers in understanding the boundaries of their libraries and their usage in the wild. To this end, we follow an exploratory case study approach [27] to analyze three popular Java libraries that exhibit diverse interaction styles: Javasoup, a library for HTML manipulation; Apache Commons-CLI, a library for implementing command-line interfaces; and Spark, a simple web framework. We introduce our subject libraries in Section 5.1 and our methodology in Section 5.2, and then detail our results in Section 5.3. We direct the reader to our reproduction package to access the data, scripts, and analyses conducted in this section [20].

#### 5.1 Subject Libraries

The implementation of syntactic usage models and footprints in UCov aims to explore the boundaries of APIs and their actual uses in client code. Every API is unique, and it is neither possible nor desirable to study all of them. Instead, we establish a set of criteria to select subject libraries that are diverse and representative of mature Java libraries.

### Table 3: The kinds of uses considered in UCov, inspired and refined from the work of Qiu et al. [26]

<table>
<thead>
<tr>
<th>Symbol Type</th>
<th>Use</th>
<th>Example</th>
<th>Resolved API Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Reference</td>
<td>String s</td>
<td>java.lang.String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void f(Integer i)</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integer f()</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void f() throws IOException</td>
<td>java.io.IOException</td>
</tr>
<tr>
<td></td>
<td></td>
<td>catch (IOException e)</td>
<td>java.io.IOException</td>
</tr>
<tr>
<td></td>
<td></td>
<td>List&lt;String&gt; l</td>
<td>java.lang.String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etc.</td>
<td>etc.</td>
</tr>
<tr>
<td>Class</td>
<td>Instantiation</td>
<td>new Integer(42)</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td></td>
<td>Inheritance</td>
<td>class T extends Thread</td>
<td>java.lang.Thread</td>
</tr>
<tr>
<td>Interface</td>
<td>Implementation</td>
<td>class R implements Runnable</td>
<td>java.lang.Runnable</td>
</tr>
<tr>
<td></td>
<td>Extension</td>
<td>Runnable r = ()-&gt;{...}</td>
<td>java.lang.Runnable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>interface R extends Runnable</td>
<td>java.lang.Runnable</td>
</tr>
<tr>
<td>Constructor</td>
<td>Invocation</td>
<td>new Integer(42)</td>
<td>java.lang.Integer(int)</td>
</tr>
<tr>
<td>Method</td>
<td>Invocation</td>
<td>&quot;a&quot;.length()</td>
<td>java.lang.String.length()</td>
</tr>
<tr>
<td></td>
<td>Static invocation</td>
<td>String.valueOf(42)</td>
<td>java.lang.String.valueOf(int)</td>
</tr>
<tr>
<td></td>
<td>Overriding</td>
<td>@Override void run()</td>
<td>java.lang.Thread.run()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Runnable r = ()-&gt;{...}</td>
<td>java.lang.Runnable.run()</td>
</tr>
<tr>
<td>Field</td>
<td>Field read</td>
<td>Integer.MAX_VALUE</td>
<td>java.lang.Integer.MAX_VALUE</td>
</tr>
<tr>
<td></td>
<td>Field write</td>
<td>Point.x = 2</td>
<td>java.awt.Point.x</td>
</tr>
</tbody>
</table>

#### Interaction styles

Based on our modeling of syntactic usage, we hypothesize that different interaction styles favor different kinds of uses in client code and yield different usage profiles. To explore the diversity of uses, we aim for libraries that expose diverse styles. We categorize interaction styles into three types that are widespread in practice: classical, framework, and fluent. Below, we provide an example of each and explain their significance in the context of API usage. Naturally, these styles are not mutually exclusive and a library may offer different styles to interact with the same features.

In the **classical style**, APIs expose a set of public classes that are then instantiated in client code to invoke their methods and access their services. Client code retains control of the execution flow. The first snippet of Listing 4 depicts a typical classical use of Commons-Cli, where a parser and options are instantiated and configured using their methods. The classical style is the most common and is especially popular in libraries that offer a collection of utilities (e.g., Google’s Guava and Apache’s Commons).

In the **framework style**, APIs implement inversion of control and the Hollywood principle to allow client code to extend and specialize types of the API (usually interfaces and abstract classes) by providing implementation code. The library retains control over the execution flow and only hands it to client code when needed. This style is prevalent in frameworks (e.g., Spring, Hadoop), hence its name. The second snippet of Listing 4 shows a typical framework-like interaction with Spark, where users configure routes for their application by passing lambda expressions that implement the functional interface Route and its unique method Handle(Request, Response) which gives the implementation of endpoints.

In the **fluent style**, APIs heavily employ programming tricks such as method chaining and cascading, static methods, the Builder design pattern, and static imports to mimic the look and feel of a domain-specific language [9]. The third snippet of Listing 4 shows...
Listing 4: Diverse API interaction styles exemplified using actual documentation samples from our subject libraries

```java
// Classical usage of commons-cli
// https://commons.apache.org/proper/commons-cli/usage.html
CommandLineParser parser = new DefaultParser();
Options options = new Options();
options.addOption("a", "all", false, "do not hide entries");
options.addOption("C", false, "list entries by columns");
try {
    CommandLine line = parser.parse(options, args);
    if (line.hasOption("block-size")) { /* ... */ }
} catch (ParseException exp) { /* ... */ }

// Framework-like usage of Spark
// https://sparkjava.com/documentation#routes
get("/", (request, response) -> { /* ... */ });
post("/", (request, response) -> { /* ... */ });
put("/", (request, response) -> { /* ... */ });
delete("/", (request, response) -> { /* ... */ });
options("/", (request, response) -> { /* ... */ });

// Fluent-like usage of jsoup
// https://jsoup.org/cookbook/
Document doc = Jsoup.connect("http://example.com")
      .data("query", "Java")
      .userAgent("Mozilla")
      .cookie("auth", "token")
      .timeout(3000)
      .post();
```

a typical fluent interaction with jsoup that shows how the Builder pattern is used to configure a jsoup connection and load a document from a remote URL.

**Client code.** To make the analysis possible, we look for libraries that have sufficient amounts of client code available: third-party clients, tests, and documentation samples. This rule out immature libraries that lack sufficient documentation or clients, or that are not well-tested.

While we expect to find third-party clients and tests that can be parsed and analyzed with UCov, documentation and samples suffer from additional issues. Some libraries include their samples as proper compilable files in their source directory, while others include their samples in readme files (e.g., on GitHub) or dedicated documentation websites. These samples are typically simple snippets cleaned from the surrounding boilerplate code (imports, structure, type declarations, etc.) that often cannot be parsed on their own. When these cases arise, we create a Java file for each case, encapsulating the snippet within a main method and manually inserting the missing imports. For our subject libraries, with the help of the surrounding documentation, the missing imports are always unambiguous.

**Selected libraries.** To obtain high-quality libraries that meet these requirements, we start from the Duets dataset [7]. Duets contains 395 libraries and 2,874 clients extracted from GitHub that compile, can be executed, have passing test suites and a minimum of five stars. To identify libraries with documentation samples, we narrow our search to libraries with a readme file and an official documentation website. If these contain documentation samples, we collect them. Otherwise, we follow their hyperlinks and repeat the process. We also explore the source tree of each library to find samples stored alongside the library code. Finally, we handpick a set of three libraries that exhibit diverse interaction styles and have sufficient test cases, documentation samples, and third-party clients: jsoup (classical and fluent styles), commons-cli (classical and fluent styles), and spark (framework style). Table 4 provides some descriptive statistics for these libraries.

For these three libraries, we retrieve the following documentation samples:

- For commons-cli, its readme links to the official documentation, which includes a page showcasing some sample uses. This page contains partial code snippets without imports or structure. We manually reconstruct parseable Java files for these snippets.
- For jsoup, we find a set of samples directly in its source code (src/main/java/org/jsoup/examples). Its readme file also leads to the official website and a cookbook that presents different sample snippets. We reconstruct complete Java files with proper imports and structure for these samples.
- For spark, its readme includes fully parseable sample code and links to the official documentation. The documentation points to a list of user-authored tutorials, which consist of parseable Java files, and a list of template projects, which we also include.

| Descriptive statistics of the subject libraries, extracted from GitHub on October 30, 2023 |
|-------------------------------------|-----------------|-----------------|-----------------|
| **COMMONS-CLI** | **JSOUP** | **SPARK** |
| Last release date | 2021/10/29 | 2023/04/29 | 2020/10/08 |
| Version | 1.5.0 | 1.16.1 | 2.9.3 |
| Since | 2002 | 2010 | 2011 |
| Stars | 300 | 10.4k | 9.5k |
| Commits | 1,374 | 1,889 | 1,067 |
| Size (LoC) | 6,307 | 27,817 | 11,298 |
| Contributors | 46 | 97 | 100 |
| Clients | 49k | 131k | 30k |

5https://commons.apache.org/proper/commons-cli/usage.html
6https://jsoup.org/cookbook/
7https://sparkjava.com/documentation
8https://sparkjava.com/tutorials/application-structure
9https://commons.apache.org/proper/commons-cli/usage.html
of the libraries of interest. After discarding repositories that cannot be retrieved and forks, we obtain 11,159 clients for jSoup, 11,612 for commons-cli, and 12,530 for spark. Naturally, it is not necessary to analyze all of these, so we apply the standard Cochran formula with a confidence level of \( c = 95\% \), an error margin of \( e = 5\% \), and a conservative proportion of \( p = 0.5 \). We obtain sample sizes of 372, 372, and 373, which we draw at random from the corresponding client sets. These sets contain the third-party clients, which we analyze in the remainder of this section.

We use UCov to construct the SUM of each library. Following the specification in Section 4, UCov extracts the list of all exported API symbols and maps them to their corresponding legal uses. The resulting models materialize the universe of interactions enabled by the libraries (Table 5). For each library, we classify client code as either tests, documentation samples, or third-party clients, and we run UCov on these three corpora to produce their SUF. Since SUFs can be easily joined through set union, we also construct a merged SUF for each library representing every use from all client code, denoted All in Table 5.

5.3 Analysis and Results
This section discusses the results obtained with UCov for the three subject libraries and their client code.

Syntactic usage models and footprints. Table 5 presents simple statistics for the SUMs and SUFs extracted from our three subject libraries. jSoup exposes the most extensive API, with 1,138 symbols and 2,941 legal uses in its SUM, followed by Spark and Commons-cli. This is consistent with their overall size in lines of code (Table 4). The three libraries exhibit a similar ratio of exported API symbols to legal uses, averaging around 2.5 legal uses per symbol.

The coverage scores for uses are consistently lower than for API symbols. This is expected, as covering a use implies covering the corresponding symbol. However, the sizeable difference in coverage indicates that many of the interactions legally allowed in the APIs are not realized in either tests, documentation samples, or third-party clients, even when the corresponding symbols are known and used. Although the relatively low coverage of API symbols in client code has been extensively studied in the literature (e.g. [12, 26]), this suggests that symbol coverage does not fully reflect the extent of interactions permitted by APIs.

Across all libraries, tests achieve the highest coverage scores for symbols and uses, followed by third-party clients and samples. This suggests that the tests cover a sizeable subset of the APIs. According to Hyrum’s law, “with a sufficient number of users of an API [...] all observable behaviors [...] will be depended on by somebody” [36], suggesting that third-party clients might eventually achieve the highest coverage score, assuming a sufficient number of clients. This has already been empirically verified for the libraries hosted in Maven Central [12]. Documentation samples are sparse, so naturally they cover much less than tests and third-party clients. Spark stands out with a much better coverage score of its API by samples, thanks to the rich documentation on its official website.

Overall, commons-cli is the most focused library, with fewer exported symbols and possible uses, which results in better coverage of its symbols and uses in client code. Its API also exhibits repetitive patterns: client code often instantiates numerous command-line options and configures them similarly, resulting in frequent and uniform usage of identical symbols (averaging 216 instances per use, compared to 29 for jSoup and 50 for spark).

Library profiles. Figure 2a depicts the usage profiles of our three libraries. The profile of a library is the distribution of the legal uses in their SUMs, such that the proportion of each kind of use adds up to 1. It highlights which kinds of uses are allowed and which are the most frequent. The profiles are mostly similar, with some specificities for each library. In all cases, (virtual) method invocation and method overriding dominate the frequencies. This is expected as methods are by far the most frequent kind of symbol in the three libraries, with fewer types and fields exposed. Commons-cli and jSoup share a very similar profile. Indeed, they both implement the classical and fluent styles in their interactions. Interestingly, spark exposes a larger number of methods that can be statically invoked and interfaces that can be extended and overridden in client code. These correspond to the primary interfaces and methods used in client code to configure routes (get(), post(), path(), etc.), as shown in Listing 4. This is consistent with its framework-like interaction style.

Usage analysis. Figure 2b shows the usage profiles of the three libraries, defined as the distribution of actual uses in their SUFs (including third-party clients, tests, and samples). Comparing Figure 2a and Figure 2b reveals the discrepancies between what the APIs allow and what client code actually uses.

Among the three libraries, method overriding is the least covered type of use by a large margin. This suggests that the APIs offer possibilities for extension and specialization that are not yet utilized in client code or, more likely, that many of the API methods could be closed for extension using the final keyword at the level of the method or its containing type. Until version 17 (2021) and the introduction of sealed classes, Java lacked the ability to restrict extension and overriding to a predetermined set of types. Without

<table>
<thead>
<tr>
<th>SUM</th>
<th>API symbols</th>
<th>Legal uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI</td>
<td>291</td>
<td>755</td>
</tr>
<tr>
<td>JSOP</td>
<td>1,138</td>
<td>2,941</td>
</tr>
<tr>
<td>SPARK</td>
<td>771</td>
<td>1,960</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbols used</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>205 (70%)</td>
<td>608 (53%)</td>
</tr>
<tr>
<td>Clients</td>
<td>153 (53%)</td>
<td>282 (25%)</td>
</tr>
<tr>
<td>Tests</td>
<td>179 (62%)</td>
<td>586 (51%)</td>
</tr>
<tr>
<td>Samples</td>
<td>24 (8%)</td>
<td>47 (4%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUF Unique uses</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>367 (49%)</td>
<td>1,028 (35%)</td>
</tr>
<tr>
<td>Clients</td>
<td>266 (35%)</td>
<td>456 (16%)</td>
</tr>
<tr>
<td>Tests</td>
<td>315 (42%)</td>
<td>967 (33%)</td>
</tr>
<tr>
<td>Samples</td>
<td>38 (5%)</td>
<td>69 (2%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total uses</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>79,284</td>
<td>29,979</td>
</tr>
<tr>
<td>Clients</td>
<td>74,453</td>
<td>15,246</td>
</tr>
<tr>
<td>Tests</td>
<td>4,690</td>
<td>14,511</td>
</tr>
<tr>
<td>Samples</td>
<td>141</td>
<td>222</td>
</tr>
</tbody>
</table>
this possibility, library maintainers had to open types for extension and specialization to everyone, even when their intent was to allow extension and specialization in library code only.

Here again, COMMONS-CLI and JSOUP expose a similar profile. However, JSOUP’s clients rely more on its fluent style, with fewer class instantiations and more frequent use of static invocations and builders to create objects. We also observe that although COMMONS-CLI and JSOUP expose some fields that can be read and written, as well as some types that can be extended or implemented, clients rarely use these interactions. The only uses of these interfaces in client code are through type references.

SPARK, on the other hand, exhibits a different profile with many static invocations, method overrides, and interface implementations. Indeed, as advocated in its documentation, the preferred method of declaring routes in SPARK is to pass a lambda expression that implements a single method in the Route interface, resulting in each route declaration involving one static invocation, one interface implementation, and one method override.

Digging deeper into the most popular uses for each library, we find that the SUFs accurately reflect the expected uses of each library, as documented in their samples. For COMMONS-CLI, the top three most popular interactions involve referencing and instantiating the symbols Option and Options, and invoking the method addOption, which developers use to build their command-line interface. For JSOUP, the top three most popular interactions involve referencing the symbols Document, Element, and invoking the method Element.select(String), which developers use to navigate HTML documents. For SPARK, the top three most popular interactions involve implementing the interface Route and overriding its method Route.handle(), which developers use to declare routes, as well as referencing the types Request and Response, which are passed to and from the routes. We refer the reader to the reproduction package for a comprehensive list of the most and least popular API symbols and interactions.

Interestingly, we observe that JSOUP exposes parts of its internal API publicly (org.jsoup.internal), most likely for technical reasons: isolating types in a package forces them to be public to allow other packages of the library to access them. While the maintainers take great care to discourage clients from using these APIs using source code comments ("Jsoup internal use only, please don’t depend on this API"), we find several uses in the code of third-party clients. Perhaps more surprisingly, JSOUP’s official documentation also uses these internal APIs in one of its samples.\(^{11}\) UCov allows library maintainers to immediately identify these problematic cases at a glance.

Comparative analysis of third-party clients, tests, and samples. The SUF profiles of third-party clients, tests, and samples (not shown here for conciseness but available in the reproduction package) are very similar for each library. However, when looking at individual symbols and uses, we observe that each has its specificities. Figure 3 depicts the individual contributions to unique uses of third-party clients, tests, and samples, represented as UpSet plots to visualize the size of their intersections. This visualization enables library maintainers to immediately identify gaps in the coverage of client code by their tests and documentation.

The official samples of COMMONS-CLI\(^{12}\) are particularly sparse compared to JSOUP and SPARK: all the uses they document are also found in tests and third-party code. Conversely, we identify 176 uses that are common to tests and third-party clients but not documented (23% of legal uses). These include the deprecated OptionBuilder type, which is unsurprisingly not documented, but also symbols that are popular in client code such as org.apache.commons.cli.Option.isRequired(). As the API of COMMONS-CLI is well-focused, there is still a significant amount of uses that are common to all three. However, we observe that 52 uses found in third-party clients

\(^{11}\)https://github.com/jhy/jsoup/blob/84f6d4376b992a01057b37840acee4882f1e3ddd6/src/main/java/org/jsoup/examples/HtmlToPlainText.java

\(^{12}\)https://commons.apache.org/proper/commons-cli/usage.html
are not exercised: for instance, the method `org.apache.commons.cli.Option.setValueSeparator(char)` is neither tested nor documented.

The samples of `jsoup` are more varied, but there is still a significant amount of uses that are undocumented. Among these 338 uses, we identify some popular interactions such as invoking the `org.jsoup.nodes.Node.hasAttr(java.lang.String)` method. Its tests are comprehensive, and only a few uses are exclusive to third-party clients. Among these, we find clients extending and overriding the `org.jsoup.Connection` type to implement their own logic and innocuous cases such as invocations to the many `toString()` methods that are (predictably) untested and undocumented. Interestingly, some of the uses we find in third-party clients are also documented in its samples, but absent from the tests. This is, for instance, the case for the internal APIs of `jsoup` discussed previously.

Despite the extensive samples found in its readme and tutorials, `Spark` still presents 39 uses that are common to clients and tests but undocumented. Even more surprisingly, we find some uses that are documented but neither tested nor used in third-party clients, such as invocations of the `spark.Request.bodyAsBytes()` method. In contrast with `Commons-CLI` and `JSouP`, we find some uses that are documented, tested, but for which we could not find any instance in client code: we did not find any static invocation of the method `spark.Spark.modelAndView(Object, String)` which is the documented way of instantiating a `ModelAndView` object. Similarly to `jsoup`, we find 27 API interactions that are documented in samples and used by third-party clients but remain untested. These include setting the listening port of the web server (`spark.Spark.port(int)`) and examining the underlying raw Java requests beneath `spark`’s `Request` objects (`spark.Request.raw()`).

A significant part of the value of syntactic models residues in the intersection of the uses of different kinds of client code. `UCov` allows library maintainers to identify undocumented or untested interactions in their APIs and take appropriate action. Naturally, it is not practical to expect that all API interactions are covered by the library’s tests or samples. Most developers use IDEs to discover features, and the associated `JavaDoc` is often sufficient. Besides, trivial APIs such as the `toString()` methods mentioned above may not necessarily require documentation. Nevertheless, we believe that `UCov` can help maintainers prioritize their documentation and testing efforts by focusing on the interactions that are commonly used in the code of third-party clients, and identify blind spots in their design.

### 6 DISCUSSION

Syntactic usage models and syntactic usage footprints offer a lightweight and intuitive way to analyze the interactions allowed by an API and their coverage in client code. While our case study primarily explores their ability to accurately represent different library profiles and identify unused, undocumented, and untested interactions, they can inform and benefit other scenarios.

**Support for API design in programming languages.** SUM models represent the interactions permitted by an API. When using them, it becomes clear that the capabilities provided by Java for designing APIs are limited. Library developers are often required to expose numerous legal interactions to client code for purely technical reasons that have little to do with API design. For example, although library developers can prevent extension and overriding for everyone by using the `final` modifier, they have only recently gained the ability to open extension to a predetermined set of implementers using `sealed` classes. On the other hand, it is not possible to fine-tune the interactions allowed in client code, such as allowing method overriding but not method invocation, which can violate the principles of inversion of control in framework-like libraries. Similarly, there are no mechanisms to restrict access to certain types, fields, and methods beyond the basic scoping mechanisms provided by visibilities. As a result, developers must sometimes declare types as package-private or even public for technical reasons and rely on code comments and naming conventions such as internal packages and `@Internal` annotations to warn their users. SUM models make all these issues explicit and could facilitate reflection on language design after adapting to other programming languages.

**Compatibility and breaking changes.** When libraries evolve, they sometimes introduce breaking changes that affect clients and force them to adapt their code. How breaking changes impact client code depends on the interactions with the broken API. For example, a class that evolves into an abstract class does not break client code that references it but does break client code that attempts to instantiate it. Simply knowing which API symbol the client code...
uses is insufficient to assess the impact of breaking changes. Some approaches attempt to measure the impact of potential breaking changes by analyzing client code. While some only rely on import declarations to determine if client code may be impacted [37], more recent approaches gather usage information to improve the accuracy of impact detection [23, 24]. SUF models provide accurate information regarding the uses made in client code and we believe they could improve the accuracy of these tools.

Another interesting direction is to utilize SUM models to automatically generate synthetic client code that thoroughly exercises every possible interaction with the API. This code would exhibit a complete footprint of the API and could be compiled whenever the library is updated, allowing the compiler to automatically check if any of the interactions break, signaling a breaking change. Using the compiler as ground truth for breaking change detection would address the accuracy issues of existing detection tools [13] and automatically cope with the evolution of programming language specifications and their implementation in compilers and virtual machines.

API evolution. Predicting the consequences of API changes can be challenging. When an API evolves, it can not only introduce breaking changes but also alter the way clients interact with it, even when the changes are backward-compatible. Because SUM models can be efficiently computed (e.g., during code review or continuous integration), maintainers can reason about the impact of their changes and refactorings by exploring the differences between pre- and post-change SUMs. This could allow maintainers to easily assess the impact of external contributions on the interactions permitted by their API and determine whether to incorporate the changes.

7 RELATED WORK

In this section, we discuss related work on the analysis and applications of API usage. Specifically, we review the tools and studies that analyze usage at the symbol level, the extensive literature on usage patterns and protocols, and their applications to API evolution and breaking changes analysis.

API symbols analysis. Traditionally, the analysis of API usage has primarily been conducted at the level of individual symbols. Many studies analyze how API symbols are used to identify notable hotspots and coldspots, i.e., parts of the APIs that are over- or under-utilized [30–32]. A recent study by Qiu et al. delves into the usage of Java’s standard library and third-party libraries across a corpus of over 5,000 projects, encompassing 150M+ lines of code, and finds that usage follows a Zipf distribution [26]. Another study by Harrand et al. examines 2.2M dependencies on Maven Central and confirms the observations of Hyrum’s law: a small subset of the API attracts most usages in client code [12]. As illustrated by our exploratory case study, the coverage of uses is much smaller than that of symbols. Various tools and studies employ different methodologies for collecting usage data, whether from bytecode [12], source code and resolved ASTs [6, 18, 26], or other sources [31]. A unifying theme across these works and their underlying models of API usage is their focus on determining whether a specific API symbol is accessed in client code, rather than exploring how it is used.

Syntactic models and UCov go beyond this level of granularity by distinguishing the various interactions allowed by individual symbols. As our case study shows, the coverage of uses in client code is much smaller than that of symbols.

Usage patterns and protocols. A variety of work in the literature addresses the problem of mining and recommending unordered or sequential API usage patterns and protocols to users [3, 22, 29, 35, 42]. These studies typically analyze library code and client code to infer automata and probabilistic graphs that represent legal sequences of API invocations and check whether client code complies with them. Some even gather data from Q&A websites to infer and recommend usage patterns [33, 40], while others empirically study these usages in the wild [41]. These approaches consider sequences of API calls, often with temporal properties, and go beyond the goals of UCov. On the other hand, UCov still provides a more detailed view of interactions with API symbols, which could benefit these approaches. In our work, we leverage syntactic usage models and footprints to help developers understand which API uses are allowed and to help them improve their API design, eliminating unintended uses or documenting undocumented areas of their APIs. A promising area for future research is studying how syntactic usage models and footprints can contribute to extracting more fine-grained usage patterns and protocols.

API evolution breaking changes. There is a wide range of literature on the nature and driving forces of API evolution [14, 15, 19], as well as on breaking changes and their impact [2, 23, 37]. Some studies focus on detecting breaking changes through API usage patterns [8, 10, 12, 16, 21, 34, 38, 39]. A key enabler for these studies is the use of accurate models that can represent both the interactions allowed by an API and their uses in client code. The impact of a breaking change, for example, depends on both the symbol being used and the way it is used. We believe that the rich information provided by syntactic usage models and footprints can better inform breaking change detection tools [13] and improve their accuracy in assessing the impact of breaking changes on client code. This information can also help inform studies that aim to understand API usage and evolution by refining the level of granularity at which APIs are considered [4, 5].

8 CONCLUSION

In this paper, we introduced syntactic usage models and footprints to support library maintainers in understanding the boundaries of their APIs and the interactions they allow. We presented an implementation of these models for the Java programming language in UCov, a static analysis tool that analyzes the source code of libraries to collect their exported symbols and allowed uses, and client code to collect fine-grained uses of the library.

Our exploratory case study of three popular Java libraries that exhibit diverse interaction styles (COMMONS-CLI, Jsoup, and Spark) showed that UCov provides valuable information regarding the usage profiles of libraries and their uses in documentation samples, tests, and third-party clients. Specifically, we showed how UCov can pinpoint undocumented and untested interactions, as well as misalignments between legal uses of the API and actual uses in client code.
For future work, we will explore the suitability of syntactic usage models and footprints to support graceful API evolution and the detection of breaking changes and their impact. We will also employ U Cov for large-scale empirical analysis of the evolution of Java libraries and investigate the benefits of syntactic models in other programming languages and ecosystems.

ACKNOWLEDGMENTS

This work was partially funded by the French National Research Agency through grant ANR ALIEN (ANR–21–CE25–0007). We thank the anonymous reviewers for their precious comments.

REFERENCES


