The Journal of Systems and Software 210 (2024) 111974

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

API usage templates via structural generalization ☆

May Mahmoud, Robert J. Walker *, Jörg Denzinger

Department of Computer Science, University of Calgary, Calgary, Canada

ARTICLE INFO

Dataset link: 10.6084/m9.figshare.25040405

Keywords:
API usage
Coding templates
E-generalization

ABSTRACT

APIs matter in software development, but determining how to use them can be challenging. Developers often refer to a small set of API usage examples, analysing the information there to understand and adapt them to their own context. Generalization over many examples may aid in understanding commonalities and differences, reducing information overload while including greater variety.

We propose ASGard, a novel approach that generates API usage templates from examples. Approximating the formal problem of E-generalization, ASGard generalizes all syntactic and some semantic information within the examples to arrive at pseudocode representations that retain the commonality of the usage examples but abstract the varying aspects.

We evaluate the templates from our approach and the patterns generated from PAM and MUDetect (two existing tools for API data mining), using a total of 1954 API usage examples across 59 different APIs. We measure the quality of the resulting templates: ASGard's templates have superior completeness and compression.

We perform a user study on ASGard with 12 participants to compare the use of these templates in solving programming tasks, compared to MUDetect. We find that participants solved the programming tasks in significantly less time with ASGard. Participants expressed a general preference for using ASGard templates.

1. Introduction

Developers rarely code everything from scratch; reuse is part-and-parcel of the software development process, and application programming interfaces (APIs) are one of the most common forms of code reuse (McLellan et al., 1998; Robillard, 2009). An API is a collection of pre-existing code, abstracted behind a programmatic interface, that developers can call from their code to accomplish programming tasks (Stylos and Myers, 2007). APIs are an essential component of modern software development; they are constantly increasing in number as well as becoming more complex (Robillard, 2009; Zhang et al., 2020). Much literature investigates the challenges faced by developers to learn and use a new API, asserting that it be a common challenge facing developers every day (Ko et al., 2004; Robillard, 2009; Robillard and Deline, 2011; Myers and Stylos, 2016; Zhang et al., 2020).

When learning to use a new API, developers often seek examples of the API usage to understand how to use the API in different contexts (Nasehi et al., 2012). However, due to limited time and attention, developers rarely look beyond five examples when searching for code to complete a task (Starke et al., 2009). Developers try to analyse such examples to understand what is common and what is different and to decide on ways to adapt this information to their needs. Manual analysis with concrete examples tends to be problematic as it can be difficult to determine which aspects of a given example are necessary, incidental, or common (Sillito and Begel, 2013; Sushine et al., 2015; Thayer et al., 2021). Correct use of an API requires conformance to various constraints – such as preconditions, exception handling, and specific method-call order – in order to accomplish the tasks that it supports. In addition, when dealing with an unfamiliar API, developers often have difficulties unmasking its relationships with other APIs where the interdependencies are not obvious or not documented (Duala-Ekoko and Robillard, 2012). Research has shown that API designers still have difficulty in gathering user feedback regarding their API designs; this suggests that automated extraction of API usage from data produced by the community would be helpful (Murphy et al., 2018; Zhang et al., 2020). Thus, providing a way to better generalize usage examples might help developers understand API usage better and more efficiently (Chatterjee et al., 2019).

Work on API usage mining seeks recurrent information in usage examples. Some approaches seek frequent subsequences of method calls (e.g., Monperrus et al., 2010; Wasylkowski and Zeller, 2011; Fowkes and Sutton, 2016). Others use graph-based representations, applying frequent subgraph mining techniques (e.g., Nguyen et al., 2009;

☆ Editor: Yan Cai.

* Corresponding author.
E-mail addresses: may.mahmoud2@ucalgary.ca (M. Mahmoud), walker@ucalgary.ca (R.J. Walker), denzinge@ucalgary.ca (J. Denzinger).

https://doi.org/10.1016/j.jss.2024.111974
Received 2 December 2022; Received in revised form 7 June 2023; Accepted 19 January 2024
Available online 24 January 2024

Amann et al., 2019). However, all such approaches focus on frequently occurring commonalities; this results in either excluding variations in the usage of the API elements in similar contexts, or subdividing such variations across several patterns, forcing developers to manually figure out variability in the API elements' usage. Approaches that aim to select the best examples (e.g., Moreno et al., 2013) ignore variation. Approaches that generate examples (e.g., Barnaby et al., 2020) focus on producing maximally succinct examples rather than representing whatever commonality is present. Approaches that focus on the problem of code summarization seek to produce short, textual descriptions of the code (Haiduc et al., 2010; Abid et al., 2015; McBurney and McMillan, 2014, 2016); they do not aim to represent all source commonality and points of variation in a set of examples.

Our approach, called ASGard (for **A**PI usage templates via **S**tructural **G**eneralization), constructs API usage templates, a code-based representation generalizing similar API usage contexts, in order to show the commonality within the usage examples, replacing the varying aspects of the input examples with *structural variables*. API usage templates are represented as generalized abstract syntax trees (ASTs) containing structural variables. API usage templates thus support reuse of the knowledge in the input examples, in a form that summarizes commonalities and differences.

ASGard takes a set of API usage examples and a simple specification of the API of interest, as input. We proceed in two phases. (1) For the sake of improved performance, we cluster the examples based on the similarity of the API usage. (2) We then use an approximate solution to the formal problem called E-generalization (Burghardt, 2005) to infer API usage templates from the examples. We compare the nodes of the ASTs of the examples, seeking to preserve common elements in the nodes while abstracting away the differences. The generalization proceeds iteratively, permitting increasing abstraction of the template as long as no API usage information is eliminated. The final generalized ASTs represent the API usage templates. In addition to generalizing syntactically similar elements, we allow for semantically similar elements to be compared and generalized (e.g., for-loops with while-loops, variable declarations with assignments) by defining a set of *equational theories*.[1] ASGard does not start from a strong assumption about which details in source code examples are irrelevant, allowing the technique to *discover* the commonalities that exist, constructing templates to represent these. Furthermore, by approximating E-generalization, ASGard can utilize semantic similarities that less powerful techniques would ignore.

We automatically evaluate ASGard versus the patterns from PAM (Fowkes and Sutton, 2016) and MUDetect (Amann et al., 2019), two approaches that represent the main directions in API usage pattern mining in the literature. We use a total of 1954 API usage examples across 59 different APIs, in a simulated scenario in which a concrete implementation is to be constructed from a usage template. We measure the quality of the resulting sets of templates relative to two factors: (1) their *completeness*, which measures how much of each originating example is retained; and (2) their *compression*, which represents the degree to which they have compressed the set of examples. We find that ASGard yields usage templates with greater completeness and better compression than the patterns of PAM and MUDetect.

We conduct a user study on the usefulness of ASGard by having developers solve some programming tasks, either using our templates or the best alternative usage pattern miner. Through the user study we found that, compared to the best alternative, participants solved the programming tasks in less time: 48% for a coding task and 31% for a debugging task. Participants express a greater preference for using ASGard and a greater belief that the approach helped them understand the API usage better; they express a greater willingness to use the approach again than the best alternative.

[1] We must approximate E-generalization because the equational theories that we adopt result in the search space being too large to find an optimal solution.

1.1. Motivating example

We present a simple example to illustrate the extraction of an API usage template generalizing the API usage from concrete examples. Fig. 1 shows three examples[2] using the Twitter4J API,[3] a popular Java library for interacting with the social media platform Twitter. The `Twitter` class can access the Twitter data for a verified user.

The three examples show a situation in which a list of tweets is to be retrieved as `Status` objects, then processed. All three examples start by instantiating a `Twitter` object by calling `TwitterFactory.getInstance()` (line 1). They then create a `User` object by calling the `verifyCredentials()` method of the `Twitter` object (line 3). The three examples vary in the tweets they seek (line 4): example 1(a) seeks tweets from the user in their timeline; example 1(b) seeks tweets of the user that are retweeted by other users; and example 1(c) seeks tweets where the user is mentioned. The three examples then vary in how they output the list of `Status` objects (line 7): one prints to the standard output, one writes to a file, and one writes to a print stream. The examples use the same exception handling (lines 9–13).

Our goal is to generate API usage templates, as shown in Fig. 2 generalizing the examples in Fig. 1. The elements "V_x", where x is a unique number, represent *structural variables* which indicate variability between API usage examples. Each example can be recovered by *substituting* specific concrete values in place of the structural variables, or the template can be reused by replacing the variables with details appropriate to some novel context.

The template generalizes the three examples. It instantiates the `Twitter` object by calling the method `TwitterFactory.getInstance()` (line 1) and the call to `twitter.verifyCredentials()` to create the `User` object (line 3). It then shows that a call to a method on the `Twitter` object returns a list of tweets (line 4); the different methods to be called are represented by the structural variable V_1. The differences in the output mechanism used for the results is indicated by the structural variables V_2 and V_3 (lines 5 and 7); note that due to the use of E-generalization, the consistency of the arguments is fully preserved as is the constraint that the targets and names be the same on those two lines. The template also shows the handling of any `TwitterException` object (lines 10–14) that is thrown during the execution of the **try** clause (lines 2–9). Finally, each structural variable can be substituted by specific values in order to recover one of the original examples; these are shown in Fig. 3.

The reader should note that the template has preserved the commonality in the input examples and pointed to where the variation lies. This contrasts with data mining approaches that seek to eliminate all information not defined a priori as "important", thus preventing the discovery of unexpected commonalities. In the next section, we examine past approaches, seeing that they preserve much less commonality than an approach based on E-generalization.

The remainder of the paper is structured as follows. Section 2 overviews the related work. Section 3 details our approach. Section 4 describes our simulation-based evaluation. Section 5 presents the human study we conducted. Section 6 discusses remaining issues.

2. Related work

Two directions of related work are relevant here: (1) mining of API usage patterns, and (2) work using anti-unification. We will see that the mining approaches fail to address our goal of preserving common information across input examples without defining a priori which such information is important. We will see that prior work on anti-unification does not suffice to address our goal, though it is a promising route to build atop.

[2] Adapted from https://github.com/Twitter4J/Twitter4J/tree/master/twitter4j-examples/src/main/java/twitter4j/examples/timeline.

[3] https://twitter4j.org/en/index.html

```
Twitter twitter = new TwitterFactory().getInstance();
try {
  User user = twitter.verifyCredentials();
  List<Status> statuses = twitter.getHomeTimeline();
  System.out.println("@" + user.getScreenName());
  for (Status status : statuses) {
    System.out.println("@" + status.getUser().getScreenName() + " - "+ status
        .getText());
  }
}
catch (TwitterException te) {
  te.printStackTrace();
  System.out.println("Failed to get timeline: " + te.getMessage());
  System.exit(-1);
}
```

(a) Example A.

```
Twitter twitter = new TwitterFactory().getInstance();
try {
  User user = twitter.verifyCredentials();
  List<Status> statuses = twitter.getRetweetsOfMe();
  file.write("@" + user.getScreenName());
  for (Status status : statuses) {
    file.write("@"+ status.getUser().getScreenName() +" - "+ status.getText());
  }
}
catch (TwitterException te) {
  te.printStackTrace();
  System.out.println("Failed to get timeline: " + te.getMessage());
  System.exit(-1);
}
```

(b) Example B.

```
Twitter twitter = new TwitterFactory().getInstance();
try {
  User user = twitter.verifyCredentials();
  List<Status> statuses = twitter.getMentionsTimeline();
  output.print("@" + user.getScreenName());
  for (Status status : statuses) {
    output.print("@" + status.getUser().getScreenName() + " - " + status.
        getText());
  }
}
catch (TwitterException te) {
  te.printStackTrace();
  System.out.println("Failed to get timeline: " + te.getMessage());
  System.exit(-1);
}
```

(c) Example C.

Fig. 1. Three examples of the use of the Twitter class.

```
1  Twitter twitter = new TwitterFactory().getInstance();
2  try {
3    User user = twitter.verifyCredentials();
4    List<Status> statuses = twitter.V1();
5    V2.V3("@"+ user.getScreenName());
6    for (Status status : statuses ) {
7      V2.V3("@" + status.getUser().getScreenName() + " – "+ status.getText());
8    }
9  }
10 catch (TwitterException te) {
11   te.printStackTrace();
12   System.out.println("Failed to get timeline: " + te.getMessage());
13   System.exit(–1);
14 }
```

Fig. 2. The goal: An API usage template that generalizes the examples in Fig. 1.

$$\sigma_A = \{V_1 \mapsto .\text{getHomeTimeline}, V_2 \mapsto \text{System.out}, V_3 \mapsto \text{println}\}$$
$$\sigma_B = \{V_1 \mapsto \text{getRetweetsOfMe}, V_2 \mapsto \text{file}, V_3 \mapsto \text{write}\}$$
$$\sigma_C = \{V_1 \mapsto \text{getMentionsTimeline}, V_2 \mapsto \text{output}, V_3 \mapsto \text{print}\}$$

Fig. 3. The substitutions permitting recovery of examples (the subscript of the substitution corresponds to the example's label) from the API usage template in Fig. 2.

2.1. Frequently occurring API elements

Much research has focused on mining unordered method calls, finding calls that occur frequently together in code repositories (Livshits and Zimmermann, 2005; Bruch et al., 2006, 2009; Monperrus et al., 2010; Uddin et al.; Lindig, 2015; Saied et al., 2015a,b; Azad et al., 2017; Eyal Salman, 2017). For example, CodeWeb (Michail, 1999, 2000) attempts to mine API usage patterns by using association rule mining, resulting in relationships such as "if a class subclasses class C, it should also call methods of class D" (Robillard et al., 2013), i.e., without ordering. Such approaches are useful to understand the elements that are used together but fails to show how they are used and what variability there is in the usage.

2.2. Sequences of API elements

Some approaches mine association rules, adding sequencing afterwards (Ramanathan et al., 2007a,b; Thummalapenta and Xie, 2009a; Gruska et al., 2010; Wasylkowski and Zeller, 2011). JADET (Wasylkowski et al., 2007) mines API usage patterns in the form of method call sequences by building a directed graph based on ordered relations between method calls and call receivers where nodes represent method calls and edges represent control flow. TIKANGA (Wasylkowski and Zeller, 2011) is built atop JADET, introducing operational preconditions to specify a function's requirements; it uses pre-defined templates to learn constraints on sequences such as whether a method call must occur. DMMC (Monperrus et al., 2010) encodes usages as sets of methods called on the same receiver type. APIREC uses statistical learning from fine-grained changes in order to recommend the API method to be used at a request location in code (Nguyen et al., 2016).

Date et al. (2012) investigate the stability of coding patterns across versions. Their approach normalizes source code as a single sequence of call elements and control elements, then uses a sequential pattern mining algorithm to extract subsequences from the set of sequences.

Other work uses frequent sequence mining, taking as an input a set of sequences in order to identify those that occur as subsequences in the data (Xie and Pei, 2006; Acharya et al., 2007; Kagdi et al., 2007; Lo et al., 2008; Acharya and Xie, 2009; Thummalapenta and Xie, 2009b; Zhong et al., 2009; Date et al., 2012; Wang et al., 2013). MAPO (Xie and Pei, 2006; Zhong et al., 2009) mines method call sequences from code snippets, clustering them according to a heuristic distance metric; from each cluster, it mines the most frequent method calls. UPMiner (Wang et al., 2013) extends MAPO to reduce the redundancy in the mined sequences, using the BIDE closed frequent sequence miner that returns only subsequences with the same frequency.

```
1 .TwitterFactory.<init>;
2 .Twitter.verifyCredentials;
3 .TwitterException.printStackTrace;
4 .TwitterException.getMessage;
```

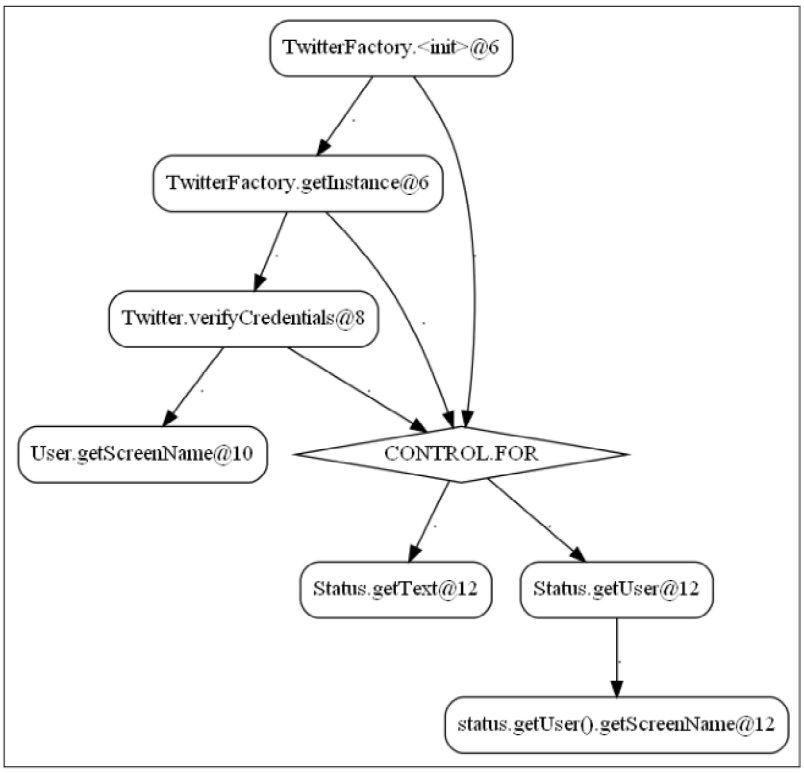
Fig. 4. A sample pattern from PAM.

PAM (Fowkes and Sutton, 2016) provides a near parameter-free probabilistic algorithm for mining "the most interesting" API call patterns. It defines API patterns as lists of API methods that are frequently used together. It uses probabilistic modelling to mine the API patterns representing them as sequences of API method calls. The results demonstrate superiority to the other sequence mining approaches including MAPO. Fig. 4 shows the output from the PAM approach as an output pattern for the examples shown in Fig. 1. We can see that only the method calls are preserved (probabilities of single method calls occurring are also output, not shown here). MARBLE (Nam et al., 2019) is based on PAM and aims to extract boilerplate code that is indicated as a usability issue for API design. Since PAM demonstrates superiority to other sequence mining approaches, we use it as a point of comparison in our empirical studies later in the paper.

2.3. Graph mining

Some work proposes graph-based representation of usage examples to mine API usage patterns, retaining more information from the originals (see Fig. 4).

GROUMINER (Nguyen et al., 2009) represents API usage patterns as object usage graphs, where the nodes represent things like constructor and method calls, field accesses, and control structures, while edges



(a) The pattern.

```
1 new TwitterFactory();
2 TwitterFactory.getInstance();
3 Twitter.verifyCredentials();
4 User.getScreenName();
5 for(;;) {
6   Status.getText();
7   Status.getUser().getScreenName();
8 }
```

(b) Corresponding pseudocode.

Fig. 5. A sample pattern from GROUMINER.

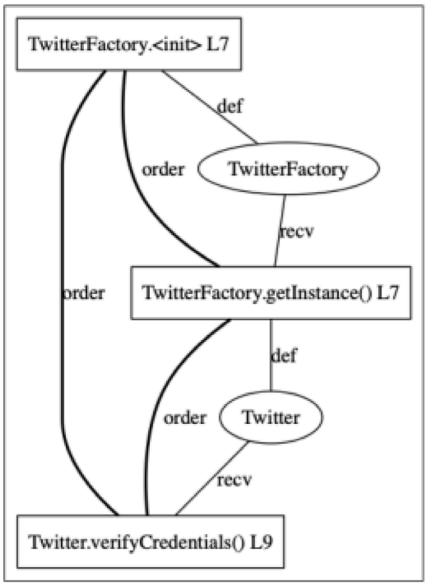
represent order and data dependencies. It uses sub-graph mining to detect frequent usage patterns. Fig. 5 shows sample output produced by GROUMINER as an API usage pattern for the examples of Fig. 1. While the key method calls are given, with some control dependence and control structure information, it lacks much information including receivers, arguments, return values, exception handling, and all details of the code *using* the API calls and their results. BigGROUM (Mover et al., 2018) is based on GROUMINER but uses frequent itemset mining and semantic analysis to scale the approach and to address the subgraph isomorphism problem that GROUMINER suffers from.

MUDetect (Amann et al., 2019) attempts to mine more information regarding the API usages through a graph-based representation of the examples called API usage graphs (AUGs). AUGs distinguish between different kinds of control/data dependencies in labelled edges as opposed to GROUMINER that uses unlabelled edges to indicate both dependency kinds. Amann et al. report 33% precision and 42.2% recall in detecting API misuses via MUDetect; they also investigate the performance of the previous API misuse detection approaches, reporting that those suffer from even lower precision and recall (e.g., GROUMINER having 2.6% precision and 3.1% recall, and JADET having 8.9% precision with 6.7% recall). For the examples of Fig. 1, MUDetect outputs five patterns: one related to the `Twitter` class, one for the exception handling, one related to the `Iterable` type, and two involving the input/output. The `Twitter`-relevant pattern is shown in Fig. 6; we automatically translate this to an equivalent pseudocode representation. The pattern shows the initialization of the `TwitterFactory` object and the `Twitter` object as well as the call to the method `verifyCredentials()` without any information about the return object from the call. While this approach retains more information than just the sequence of method calls, the subdivision of the examples into disjointed API usage patterns eliminates the utility of the variability and commonality in the examples' concrete contexts. However, since MUDetect demonstrates superiority to other graph mining approaches, we use it as a point of comparison in our empirical studies later in the paper.

The majority of such work focuses on finding frequently occurring sub-models of the model they use to represent abstracted information from the concrete examples; they all fail to represent points of variability. Nielebock et al. (2021) consider the problem of the collection of the usage examples to increase the likelihood of finding relevant patterns, showing that filtering on method level is better than file-level filtering.

2.4. Example selection or generation

Some research considers API usage pattern mining for example selection or generation. Buse and Weimer (2012) generate API usage documentation by extracting and synthesizing API usage examples. They aim to generate "minimal working examples" (in the sense expected by question & answer internet sites like Stack Overflow), actively avoiding lengthy ones that could arise in real code examples. Moreno et al. (2013) select the best examples from a set; they fail to include information that do not exist in the selected examples. MUSE (Moreno et al., 2015) uses static slicing to extract, rank, and cluster code examples that show how to use a specific API method, selecting a representative example from each cluster for presentation. FOCUS (Nguyen et al., 2019) attempts to recommend API method invocations and code snippets for the developer by considering the API usage in projects similar to the project under investigation. ExampleCheck (Zhang et al., 2018) investigates the prevalence of API misuse on Stack Overflow; they provide a visualization tool atop the miner, for code examples using a certain API method, providing summarization of the examples with statistical information based on a pre-defined code skeleton (Glassman et al., 2018). ExPort visualizes the API usage examples as relational topic models, where the API calls are modelled as a document network, where the programmer can select an API method in order to be presented with examples for it (Moritz et al., 2013). Exempla Gratis (Barnaby et al., 2020) uses a simplified parse tree of the examples to find patterns of usage for an API method; they



(a) The pattern.

```
1 TwitterFactory var1 = new TwitterFactory();
2 Twitter var2 = var1.getInstance();
3 var2.verifyCredentials();
```

(b) Corresponding pseudocode.

Fig. 6. A sample pattern from MUDetect.

focus on method usage rather than the API class and do not aim to infer "the most informative" usage example, rather a succinct example that represents a common usage. APISonar focuses on ways to extract relevant usage examples, rather than summarizing them or mining patterns (Hora, 2021).

Such approaches aim at a different problem than we do: they seek small examples, requiring complete programs to support their semantic analyses, and fail to preserve all commonality as well as variability from the originating examples: we seek to generalize the original examples, providing all details in common plus points of variability.

2.5. Code summarization

Source code summarization is concerned with the problem of producing a readable textual summary of source code, describing its functionality (McBurney and McMillan, 2016; Ahmad et al., 2020). It facilitates program comprehension through the production of code comments and documentation (LeClair and McMillan, 2019). Work in the literature mainly uses information retrieval or machine learning techniques to produce the summaries. Haiduc et al. (2010) propose the use of automatic text summarization techniques to produce such textual descriptions; they use statistical text retrieval techniques to produce extractive text summaries to generate source code summaries based on a corpus of the identifiers and comments found in the source code entities.

McBurney and McMillan (2014, 2016) propose an approach that produces descriptions of Java methods that summarize how they are invoked, including the context in which the method is found. This approach collects information about method calls and uses keywords from the context of the method to describe how it is used. Given a method to describe, the approach uses PageRank (Langville and Meyer, 2006) to identify the most important methods in the given method's context; it then uses the data generated from the Software Word Usage Model (Hill et al., 2009), which represents program statements as a set of nouns, to extract keywords about the action performed by the identified important methods. Finally, they use a custom, natural language generation system to generate English sentences describing the given methods. The summaries produce a high-level overview of the method and its context. For example, for the method `StdXMLReader.read()`, the code summary might be: "This method reads a character. That character is used in methods that add child XML elements and attributes of XML elements. Calls a method that skips whitespace". (Abid et al., 2015) considers summaries for C++ methods.

The problem considered by these approaches is different from the one our work tackles: we are concerned with generalizing the API usage examples into code templates that can be edited and utilized by developers in solving programming tasks; in addition, we include the commonality and indicate points of variability in the usage found in the example, a perspective not considered in the code summarization problem.

2.6. Applied anti-unification

Anti-unification (AU) is a set of formal problems for generalizing symbolic, tree-based expressions into a common form (an anti-unifier), utilizing structural variables to represent points of variation; generally, a most specific anti-unifier is desired, which preserves more commonality than others. Different variants of AU display different levels of expressive power, but at wildly varying computational cost. E-generalization (also called higher-order anti-unification modulo theory) (Burghardt, 2005) allows variation of inner nodes of trees while retaining commonality in the subtrees rooted there; it permits generalization of certain subtrees that are not syntactically related, when the appropriate equational theories are utilized. We approximate E-generalization.

AU has been applied to detect widespread patterns of formulae in scientific articles (Oancea et al., 2005) and for clone detection (Bulychev and Minea, 2008, 2009; Bulychev et al., 2010; Li and Thompson, 2010). Other works investigate the use of anti-unification to compute the least general type of two types in the Haskell language (Tabareau et al., 2013). Kostylev and Zakharov (2008) propose an algorithm for anti-unifying logic terms represented by acyclic directed graphs. Cottrell et al. (2008) propose a framework aiming to support small-scale source code reuse using higher-order AU; it does not store or represent the generalized AST. The approach of Cossette et al. (2014) recommends replacement functionality from a new API version, based on its structural similarity to functionality removed from the old API. Getafix (Bader et al., 2019) uses AU to learn recurring edit patterns for "automatically fixing instances of common bugs by learning from past fixes". Patternika mines migration patterns from historical data

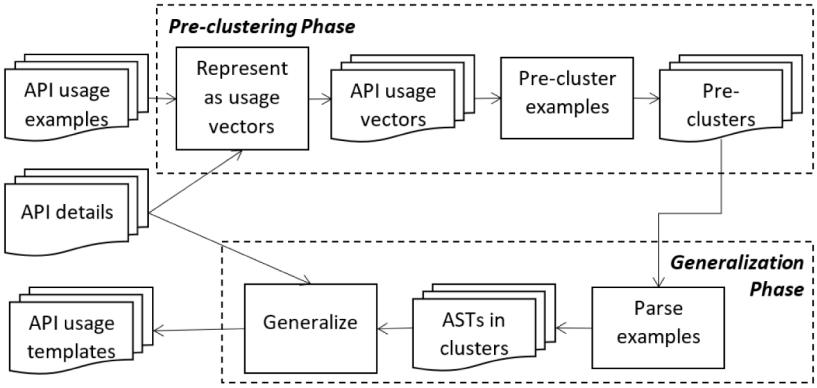


Fig. 7. Extracting API usage templates from usage examples via E-generalization.

using static code differencing techniques and anti-unification; it then applies such patterns to help automatically migrate libraries (Blech et al., 2021).

E-generalization defines key ideas like valid anti-unifiers (i.e., generalizations), most specific anti-unifiers, and equational theories, but it does not define how to practically compute them. We build atop the ideas of this formal problem to define novel algorithms for constructing potential correspondences and to select the best correspondences in order to infer API usage templates from detailed, originating examples of API usages.

3. Approach

ASGard takes as input a set of API usage examples (source code) and a specification of the API of interest (type name). ASGard constructs API usage templates through a two-phase process: (1) *pre-clustering* the examples based on their similarity with respect to the API elements' usage; and (2) *generalizing* each pre-cluster for templates. Fig. 7 visualizes the overall process. The first phase starts by representing the API usage examples with vectors of the frequencies of usage for each API element and each kind of control structure. The API entities are specified by the input API details (in our prototype implementation, this is simply the name of a class, the public methods and fields of which are then found). The usage vectors are used to pre-cluster the examples based on the similarity between them. These pre-clusters are fed to the second phase, which parses the examples from each pre-cluster into ASTs. The generalization algorithm takes these pre-clustered ASTs plus the specification of the API of interest; it determines the pairwise similarity between the nodes of the ASTs being compared, and it selects the best set of correspondences between those ASTs' nodes, based on the node similarity. The correspondences are used to calculate AST similarity to determine which ASTs to generalize. Then the approach generalizes the corresponding nodes between corresponding ASTs, producing a *generalized AST* (GAST) that generalizes the starting examples. The approach iterates through the pre-clusters, repeating these steps in order to obtain the final template(s). We explain these processes in more detail below.

3.1. Phase 1: Pre-clustering the examples

In principle, all the examples could be generalized without pre-clustering; but two examples with little in common can yield an excessively abstract template. Our initial investigations indicated that time spent on generalization in such circumstances had negative performance implications (discussed further in Section 6). Pre-clustering is performed in order to collect together examples that use the API in a similar fashion, so that useful templates can be inferred within each pre-cluster.

Each API usage example is represented by a usage vector U where each cell u_i represents the count of calls to a specific public method in the API, the count of references to a specific public field in the API, or the count of occurrences of a specific kind of control statement — **if**, loops, or **try**. We cluster the methods based on the similarity between their usage vectors, which is the Euclidean distance between them. We use the k-means algorithm (Lloyd, 1982) as the clustering algorithm, with k set to 15 (determined informally during development). The output of this optimization step is a set of pre-clusters of similar usage examples of the API.

We use the k-means algorithm as it is simple to implement, it can scale to large data sets, and it guarantees convergence (albeit not to an optimum). However, it requires the specification of the k value as an input, and it depends on the initial values of the centroid as well as being affected by outliers. Nonetheless, upon empirically investigating the output of our approach, we decided that for the purpose of our application, k-means works well enough as it gives us a starting point for processing the usage examples. If an example is added to a cluster where it is not really similar to the other examples, the processing of this pre-cluster will filter it out and place this new example in a new pre-cluster; in such cases, the approach will end with two or more templates for the pre-cluster.

3.2. Phase 2: Generalizing into templates

The second phase constitutes the computational core of ASGard. It operates on ASTs, trees of typed nodes. A GAST is just an AST that *may* contain structural variable nodes, which are ordinary AST nodes that are annotated to indicate their variable nature. Algorithms that process ASTs also work with GASTs if they treat structural variables specially, as needed; when operating on any combination of ASTs and GASTs, we use the shorthand "(G)AST".

This phase consists of six activities: (1) parsing the examples into ASTs (as this is done via standard technology, we do not describe it further — our prototype uses the parser and AST framework provided by the Eclipse integrated development environment); (2) calculation of the similarity between nodes in different (G)ASTs; (3) selection of corresponding nodes; (4) calculation of the similarity between (G)ASTs; (5) generalization of pairs of (G)ASTs; and (6) the iterative construction of the API usage templates.

(Note that an API usage template can be unparsed to obtain its pseudocode equivalent, useful for developers. This is also done via standard technology, so we do not describe it further. The one non-standard detail is the embedding of structural variables inside fake method calls "ID(·)" so that the resulting pseudocode will be syntactically valid, causing fewer problems with IDEs.)

Table 1
Grammar rules and equational theory for variable declarations/assignments.

⟨*VarDecl*⟩ := ⟨*Type*⟩ ⟨*VarName*⟩ ("=" ⟨*Expr*⟩)?
⟨*Assign*⟩ := ⟨*LHS*⟩ "=" ⟨*Expr*⟩
$varDecl_1(type(\cdot), varName(\cdot)) =_{assign} varDecl(type(\cdot), varName(\cdot), \text{NOOP})$
$varDecl(type(\cdot), varName(\cdot), expr(\cdot)) =_{assign} assign(varName(\cdot), expr(\cdot))$

3.2.1. Calculating (G)AST node similarity

The second activity computes the similarity between two nodes of different (G)ASTs – $\text{sim}_{\text{node}}(node_i, node_j)$ – which is a value in [0, 1], with 0 indicating that the nodes are unrelated and 1 that they are identical. This similarity is based on the nodes' type: types must be assignment compatible or equivalent under an equational theory.

For example, **if**-statements and variable declaration statements are not semantically equivalent, while an assignment and a variable declaration can be semantically equivalent if they are concerned with the value assignment of the same variable. We can represent an assignment as "*assign*(*varName*(·), *expr*(·))" and the corresponding variable declaration as "*varDecl*(*type*(·), *varName*(·), *expr*(·))". We can then construct "*varDecl*(V_1, *varName*(·), *expr*(·))" to generalize these constructs, where V_1 is substituted by "NOOP" for the assignment expression and by "*type*(·)" for the variable declaration. Table 1 shows an abridged grammar for variable declarations and assignments. Equating between a variable declaration and an assignment is to equate between the ⟨*LHS*⟩, which is the lefthand-side of the assignment, and the ⟨*VarDeclId*⟩, plus equating between the ⟨*VarInit*⟩ and the ⟨*AssignExpr*⟩. We can represent the two statements as a generalized variable declaration, having the type in the variable declaration replaced with a structural variable. For example, consider the corresponding snippets "**int** i = 0" and "i = 0". Their generalization would be "V_1 i = 0", where V_1 would be substituted by "NOOP" or "**int**" for the assignment and variable declaration, respectively. Note that, in practice, we would immediately translate "NOOP i = 0" to "i = 0". The full set of equational theories we use is too voluminous to describe here; the reader can find them elsewhere (Mahmoud, 2023, Section 4.4.2).

For our specific application, if a node is found to use the API of interest and another does not, the similarity between the two nodes is set to zero.

Node similarity is calculated differently depending on whether the nodes are leaf nodes or not. (Note that the similarity of nodes is calculated in the same way whether structural variables are involved or not.) For leaf nodes of string type, their similarity is the length of the least common subsequence normalized by the maximum length of the strings. Other leaf nodes' similarity is 1 if their values are identical and otherwise 0.

To calculate the similarity of two non-leaf nodes, we calculate all pairwise similarities between their child nodes. Greedy selection determines the best correspondences between the two sets of child nodes for the purposes of this calculation, where no node may have more than one correspondent. The similarities of the best correspondences are summed and normalized by the maximum of the two counts of child nodes. A leaf node and a non-leaf node will have 0 similarity by default.

As an example calculation, let us assume we have two variable declarations "String node" and "String method"; the two child nodes of both these declarations are of types Type and Name. The similarity calculation between the two nodes of type Type, each with value String, will be 1. The similarity between the two nodes of type Name, with values "node" and "method" respectively, will be $\frac{1}{3}$, as the longest common substring "od" is of length 2 and the maximum length of the two strings is 6. The similarity of the nodes with respective type Type and Name will be 0, so these will not correspond. The two Type nodes will correspond best and the two Name nodes will correspond best. The sum of best correspondences is $1 + \frac{1}{3}$, the maximum count of the child nodes is 2, and thus the similarity of the two variable declarations is $\frac{2}{3}$.

Algorithm 1: Calculate node correspondences between AST_1 and AST_2

```
1  Correspondences (AST_1, AST_2)
       // If the correspondences are cached, return them (not shown)
2      PrQueue ← {}
3      Dict ← {}
4      Corrs ← {}
5      foreach node_i ∈ AST_1 do
6          foreach node_j ∈ AST_2 do
7              sim ← sim_node(node_i, node_j) ;        // see Sect. 3.2.1
8              if sim ≥ threshold_nodeSimilarity then
9                  corr ← (node_i, node_j)
10                 PrQueue.enqueue(corr, sim)
11                 Dict[node_i].append(corr)
12                 Dict[node_j].append(corr)
13     while !PrQueue.isEmpty() do
14         sim ← PrQueue.topPriority()
15         corr ← PrQueue.pop()
16         Corrs[corr.node_1 ← (corr.node_2, sim)]
17         foreach potenCorr ∈ Dict[corr.node_1] do
18             PrQueue.remove(potenCorr)
19         foreach potenCorr ∈ Dict[corr.node_2] do
20             PrQueue.remove(potenCorr)
21     eliminateUnmatchedParents(Corrs, AST_1, AST_2)
22     eliminateOutOfOrderMatches(Corrs, AST_1, AST_2)
23     return Corrs
```

Equational theories can override the default similarity calculation by defining how the children of the nodes correspond; in such cases, other aspects of the nodes are ignored and only the similarities arising from these correspondences are used, being summed and normalized by their count. The set of equational theories we use includes[4]: (A) between **for**-, **while**-, and **do**-loop statements; (B) between variable declarations and assignments; (C) between method invocations and cast expressions; (D) between method invocations and infix expressions; (E) between invocations of methods with the same name regardless of the number of arguments[5]; and (F) between a sequence of nodes and the same sequence of nodes where a "NOOP" node is inserted at any place. Theory F could be used, for example, to generalize a sequence of statements with another, longer sequence by padding the first with "NOOP" instances until they are the same length. To compute similarity of a pair of nodes for which an equational theory is defined, we use additional heuristics. For example, to calculate the similarity of a **while**- and a **do**-loop, we use the similarity of their condition.

3.2.2. Calculating (G)AST node correspondences

The third activity computes correspondences between nodes in the two input (G)ASTs. First, all pairwise node similarities are determined as a *potential* correspondence list; any pairs whose similarity is zero are not recorded. Second, a good-fit set of node correspondences are selected (Algorithm 1).

We use a priority queue for the potential correspondences with their associated node similarity, and a dictionary that maps each node to a list of potential correspondences involving that node (lines 1–4). The algorithm proceeds to create potential correspondences between the

[4] Since ASTs are language-specific, so must be equational theories involving them; but, one can imagine similar choices for other languages.

[5] In the formal problem, method overloading is not supported. This equational theory encodes the fact that overloaded methods (or operators) should be conceptually related.

(G)ASTs' nodes, storing them in the priority queue and the dictionary (lines 5–12). Correspondence of nodes is based on their similarity (line 7). All possible combinations of nodes are calculated (many are trivially zero), but only those whose similarity is above a threshold are stored (lines 8–12). (We use a threshold of 25%, informally determined during development to arrive at reasonable results.) Iteratively, the potential correspondence with the highest similarity is selected to become a true correspondence, being stored in the correspondences dictionary along with this similarity; all other potential correspondences involving the nodes in that correspondence are removed (lines 13–20). Finally, we check for two situations not easily handled: (1) we require that, for each node with a correspondence, its parent also have a correspondence; and, (2) for corresponding nodes, the order of those nodes has to be the same for both (G)ASTs; offending correspondences are eliminated (lines 21–22). These constraints are necessary to guarantee that the result remains a tree. The list of node correspondences is returned (line 23).

3.2.3. Calculating (G)AST similarity

The fourth activity calculates the similarity between pairs of (G)ASTs as:

$$\mathrm{sim}_{\mathrm{AST}}(AST_i, AST_j) = \sum_{Correspondences} \frac{\mathrm{sim}_{\mathrm{node}}(node_i, node_j)}{\max(|AST_i|, |AST_j|)} \quad (1)$$

where $|AST_k|$ is the node count in AST_k. This similarity is based on the similarity of nodes that correspond between these (G)ASTs; $Correspondences$ represents the set of correspondences between the nodes of the two (G)ASTs.

3.2.4. Generalizing two (G)ASTs

To generalize two (G)ASTs (the fifth activity), we generalize their corresponding nodes based on statement types and their elements; any nodes that do not correspond to a node in the other (G)AST are generalized as structural variables (substitution with "NOOP" is used to recover the other (G)AST).

First, the appropriate generalizer object is found for the types of the two nodes to be generalized. If the two nodes' types are identical or there exists an equational theory that allows them to generalize while retaining information, then a generalizer object will already be defined. When no appropriate generalizer exists, the two nodes are generalized by a structural variable. Generalizer objects operate by comparing the elements of the nodes to be generalized, keeping the common elements and abstracting away the differences with structural variables. The generalized node is of the same type of the nodes if they are identical or of the type defined by the implementation of the equational theory; this type will generally be either one of the two node types or a common supertype of these, but the specifics are dictated by implementation-level constraints.

This activity continues with the generalization of the whole list of corresponding nodes for the two input (G)ASTs. Algorithm 2 is used for constructing the resulting GAST, from the generalization of each corresponding node pair, where containment of nodes is preserved. For example, consider two (G)ASTs. Say that AST_1 has the containment hierarchy $n_1^1 \rightarrow n_2^1 \rightarrow n_3^1$ (where "$a \rightarrow b$" means "a contains b") and that AST_2 has $n_1^2 \rightarrow n_3^2$, where the n_1^i nodes correspond and the n_3^i nodes correspond, but n_2^1 has no correspondence. We can generalize these subtrees by Equational Theory F, allowing us to implicitly rewrite the containment hierarchy for AST_2 as $n_1^2 \rightarrow \mathrm{NOOP} \rightarrow n_3^2$, and we will compute the generalization $n_1^* \rightarrow V \rightarrow n_3^*$, where V is a structural variable with substitutions $V \mapsto n_2^1$ and $V \mapsto \mathrm{NOOP}$ to recover the two original (G)ASTs, respectively, and n_1^* and n_3^* are the generalizations of the original node pairs.

After some simple initialization (lines 1–5) and computation of the node correspondences (line 6), the algorithm proceeds down the (G)ASTs synchronously (lines 7–23); when a node is found with a correspondence, processing of that (G)AST is paused until the other can catch up. There are four cases to deal with: (1) the two nodes have no correspondences, so a structural variable is added that can be substituted with either node (lines 8–11); (2) the first node has a correspondence but the second does not, so a structural variable is added that can be substituted with either the first node or NOOP (lines 12–15); (3) the first node has no correspondence but the second does, so a structural variable is added that can be substituted with either NOOP or the second node (lines 16–20); and (4) the two nodes correspond, so their generalization is added (lines 21–23). The Update utility deals with updating/adding as needed: it sets the root of the resulting (G)AST if it is not already defined; it inserts the new node or structural variable as a child of *curr*; it updates *curr* to the next position in the (G)AST (not the new node); and it advances $node_1$ and/or $node_2$ if passed.

Algorithm 2: Generalize AST_1 and AST_2.

```
1  Generalize (AST_1, AST_2)
2      root ← {}
3      curr ← {}
4      node_1 ← AST_1.root
5      node_2 ← AST_2.root
6      correspondences ← Correspondences(AST_1, AST_2)
7      while node_1 ≠ {} ∧ node_2 ≠ {} do
8          if !correspondences.contains(node_1) then
9              if !correspondences.contains(node_2) then
10                 structVar ← NewStructVar(node_1, node_2)
11                 update(root, curr, structVar, node_1, node_2)
12             else
13                 while node_2.corr ≠ node_1 do
14                     structVar ← NewStructVar(node_1, NOOP)
15                     Update(root, curr, structVar, node_1)
16         else
17             if !correspondences.contains(node_2) then
18                 while node_1.corr ≠ node_2 do
19                     structVar ← NewStructVar(NOOP, node_2)
20                     Update(root, curr, structVar, node_2)
21             else
22                 newNode ← Generalize(node_1, node_2)
23                 Update(root, curr, newNode, node_1, node_2)
24     return root
```

3.2.5. Construction of the API usage templates

To construct the API usage templates, we iteratively apply the generalization algorithm (Algorithm 3) within each pre-cluster. After some initialization (lines 1–4), we find the most similar pair of ASTs in the pre-cluster (lines 5–10), as long as it is above the AST similarity threshold. We tested various threshold values: we found that 40% included more relevant examples that are not identical than did higher values, so we use it. The pair is generalized into a GAST (line 11), each is removed from the pre-cluster (lines 12–13), and a new pre-cluster is initialized for later use (line 14).

The algorithm proceeds iteratively, finding the remaining AST that is most similar to the current GAST (lines 15–23): any ASTs that are less similar to the current GAST than the threshold are removed from the pre-cluster and placed in another one (initialized in line 14) for later processing (lines 24–26). Otherwise, the most similar AST and the current GAST are generalized, becoming the new current GAST; the most similar AST is removed from the pre-cluster (lines 27–28). At the end of this process, we have a single GAST which is placed in a list (line 29). We recurse over any new pre-cluster, adding the GASTs that it returns to the GAST list (lines 30–31). The GAST list is returned (line 32), representing the API usage templates.

Algorithm 3: Constructing the API usage templates from the API usage examples in a pre-cluster.

```
1  ConstructGAST (PreCluster)
2      gastList ← {}
3      pair ← {}
4      maxSim ← 0
5      foreach AST_i ∈ PreCluster do
6          foreach AST_j ∈ PreCluster \ AST_i do
7              sim ← sim_AST(AST_i, AST_j) ;          // see Eq. (1)
8              if sim ≥ threshold_astSimilarity ∧ sim > maxSim then
9                  maxSim ← sim
10                 pair ← (AST_i, AST_j)
11     CurrGAST ← Generalize(pair.first, pair.second)
12     PreCluster.remove(pair.first)
13     PreCluster.remove(pair.second)
14     NewPreCluster ← {}
15     while PreCluster ≠ {} do
16         closest ← {}
17         maxSim ← 0
18         foreach AST_i ∈ PreCluster do
19             sim ← sim_AST(CurrGAST, AST_i)
20             if sim ≥ threshold_astSimilarity then
21                 if sim > maxSim then
22                     closest ← AST_i
23                     maxSim ← sim
24             else
25                 NewPreCluster.append(AST_i)
26                 PreCluster.remove(AST_i)
27         CurrGAST ← Generalize(CurrGAST, closest)
28         PreCluster.remove(closest)
29     gastList.append(CurrGAST)
30     if NewPreCluster ≠ {} then
31         gastList.addAll(ConstructGAST(NewPreCluster))
32     return gastList
```

4. Simulation-based evaluation

We performed a simulation-based evaluation on the ASGard templates and the best two alternative approaches. We address these research questions:

- RQ1: How complete is the information presented in the templates relative to the examples?
- RQ2: How well do the templates compress the examples?

Our analysis in Section 2 determined that MUDetect (Amann et al., 2019) and PAM (Fowkes and Sutton, 2016) are the most appropriate approaches for comparison, dominating other alternatives. We convert the MUDetect patterns to a pseudocode representation for comparison. We consider the explicit method calls as the pseudocode representation for PAM's patterns.

4.1. Data set

The PAM dataset[6] consists of the source code of projects that use 17 popular Java libraries on GitHub (Fowkes and Sutton, 2016); this original dataset contained 3852 examples. We wanted to reuse the dataset from PAM as it was readily available and examined by the PAM work before and had already been used before in API usage pattern mining.

We compiled a list of all APIs in the import list of such client code. We then extracted the client method calls that use any of the APIs in the list, looking for calls to the API public methods or use of the API type in order to organize the relevant examples relative to each API. We filtered the dataset to remove APIs with fewer than the average number of examples, which was 12. This resulted in our dataset of 1954 originating examples across 59 different APIs present in the PAM dataset, for an average of 34 examples per API, which we ran through PAM, MUDetect, and ASGard.

4.2. Pattern conversion from AUGs

Since MUDetect produces API usage graphs (AUGs), we need to convert these to code templates for comparison with ASGard. An API usage graph (Amann et al., 2019) is a directed graph, whose nodes represent entities like variables, method calls, and return values, and whose edges represent control- and data-flow between the nodes. AUGs represent (pseudo-)method calls and other actions like **return** as rectangular nodes. Elliptical nodes represent typed data that is passed between the rectangular nodes. Many kinds of directed arcs are used to represent dataflow ("`para`" for parameters, "`def`" for object definition, "`recv`" for the use of an object as a method call receiver, "`throw`" for thrown exceptions), for some dependency information ("`sel`" represents a control dependence), and for lexical ordering constraints. Special actions like initialization and **return** are represented as "`<init>`" and "`<return>`", respectively, to indicate the action represented by the node.

In order to compare ASGard and MUDetect, we automatically convert MUDetect's AUGs into ASTs and thus into code snippets that represent the usage patterns. The conversion starts by topologically traversing the AUG, considering the order of edges to constrain statement order. The action nodes in AUGs have different types: we consider those types in order to construct the corresponding statement. For example, nodes that initialize a type correspond to a new class instance creation statement, while nodes that represent a method call correspond to a method invocation expression statement. We investigate incoming and outgoing edges in order to interpret the different data nodes like parameters to or returned variable from an action node, constructing the corresponding statement. Fig. 6 shows an AUG and its code representation. We interpret the condition and handle edges in order to construct loops, **if**-statements, and **try**-statements. Some information is not present in the AUG, so we must insert an arbitrary value; for example, AUGs represent all relational operators with the generic label "`<r>`", so we choose an arbitrary relational value in its place (i.e., "`!=`", not equal). The same goes for arithmetic operations: AUGs represent all operations with a generic label of "`<a>`", so we choose an arbitrary arithmetic operation value in its place (i.e., "`+`", plus). The parameter data nodes do not indicate an explicit ordering, so we consider the order equivalent to that in the graph embedding as provided by MUDetect. Fig. 6 shows the AUG for a pattern that starts with creating a TwitterFactory object followed by a call to TwitterFactory .getInstance() and a call to Twitter.verifyCredentials(), as well as the pseudocode representation resulting from our AST conversion of the AUG that shows the equivalent three statements.

We ran the evaluation dataset through the MUDetect and converted the AUG automatically into ASTs that are saved as code snippets in order to use those code snippets in the evaluation.

4.3. Evaluation measures

We consider two aspects of the quality of API usage templates: (1) the preservation of the original information when generalized into a template set; and (2) compression of the template set over the original examples.

[6] https://github.com/mast-group/api-mining

Completeness: Our central goal is to generalize the usage examples, preserving as much information as possible. As such, we measure the distance from each template to an originating example in terms of the Levenshtein edit distance (Levenshtein, 1966) between the textual representations of the template and example; this measures single character edits (insertion, deletion, or substitution) needed in order to transform one string into another. We determine the relevant template for an originating example to be the one that minimizes the edit distance:

$$t_{\text{relevant}}(e, T) = \underset{t \in T}{\operatorname{argmin}} \left(\text{Levenshtein}(e, t) \right) \tag{2}$$

where e is an originating example, T is the set of templates for the API, and $\text{Levenshtein}(\cdot, \cdot)$ is the Levenshtein distance from the first argument to the second. The closer the pattern is to the desired, concrete implementation, the less work the developer needs to do and the more efficient is the pattern in depicting the API usage example, i.e., the more complete it is. We then normalize the Levenshtein distance to the unit interval and we take its complement so that a bigger value is more desirable. Thus, we measure the example-relative completeness as:

$$\tau(e, T) = 1 - \frac{\text{Levenshtein}\left(t_{\text{relevant}}(e, T), e\right)}{\max\left(\text{len}\left(t_{\text{relevant}}(e, T)\right), \text{len}(e)\right)} \tag{3}$$

where $\text{len}(\cdot)$ is the length of its argument string.

To determine the overall completeness of a set of API usage templates, we average the example-relative completeness over all examples:

$$\tau(E, T) = \sum_{e \in E} \frac{\tau(e, T)}{|E|}. \tag{4}$$

For the motivational example, the template in Fig. 2 has an overall completeness of 80% as it retains most details of the examples.

Compression:

The approach should reduce the number of examples that a developer or a tool need consider to understand the API usage. If an approach that were supposedly producing templates simply returned all the input API usage examples, the completeness would be maximized, and yet, this would clearly not be a satisfactory result, since it would still require manual inspection of examples whose differences might not be obvious. We thus consider the number of templates in the template set versus the count of examples as the compression:

$$\rho(T, E) = \begin{cases} 0 & \text{if } |T| = 0, \\ 1 - \frac{|T|}{|E|} & \text{otherwise.} \end{cases} \tag{5}$$

If no templates are produced, the compression is defined as 0. For the motivational example, the template in Fig. 2 has compression of 67%, indicating adequate compression as the developer needs to investigate one template as opposed to three examples.

4.4. Quantitative results

RQ1: How complete is the information presented in the templates relative to the examples?

Table 2 provides summary statistics for overall completeness. The maximum for ASGard is 92%: in that case, the examples are nearly identical, producing a template that is very close to the examples with little abstraction introduced. The minimum for ASGard is 16%; such examples are too different from each other to preserve many commonalities during generalization. The median is 51% showing that for most examples a template exists that provides a good picture of the API usage. In contrast, the maximum for PAM is 25%, showing that at best PAM provides overall completeness that is worse than our mean value. The minimum for PAM is 0%; this happens since, for some APIs, PAM produces no patterns. The median for PAM is 12%, equal to its mean value. MUDetect has a somewhat better overall completeness than PAM with mean and median of 25%; this is expected as MUDetect provides more information than PAM regarding the API usage, but this is still less than half the mean or median of ASGard. The maximum for MUDetect is 38% which is worse than our median/mean, while its minimum of 0% means MUDetect fails to identify a pattern there.

Table 2
Summary statistics for overall completeness.

Approach	Mean	SD	Median	Min	Max
ASGard	0.51	0.19	0.51	0.16	0.92
PAM	0.12	0.07	0.12	0.00	0.25
MUDetect	0.25	0.07	0.25	0.00	0.38

Table 3
Summary statistics for compression.

Approach	Mean	SD	Median	Min	Max
ASGard	0.81	0.08	0.82	0.58	0.97
PAM	0.54	0.27	0.56	−0.05	0.96
MUDetect	0.26	0.36	0.28	−1.00	0.86

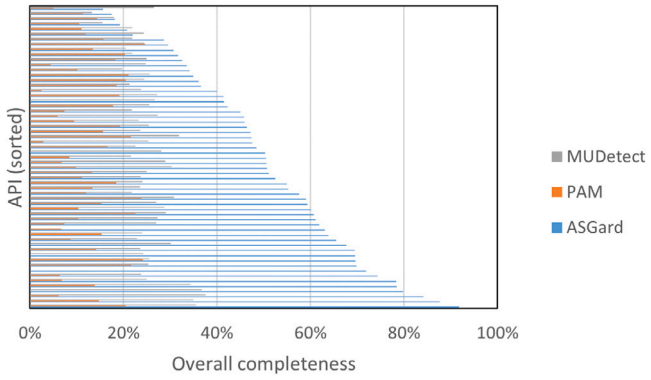


Fig. 8. Overall completeness versus sorted API.

Fig. 8 shows a clustered bar chart of the overall completeness measurements. The results for ASGard are shown in blue with those of PAM overlain in red while MUDetect is overlain in grey; the results for each API shown side-by-side; the APIs are sorted in terms of increasing completeness for ASGard. (We suppress the API names for legibility.) We observe that the APIs with the lowest overall completeness for ASGard are those that tend to fulfil a service role across different domains, such as java.util. InputStreamReader and java.util.Date with overall completeness of 16% and 17% respectively, where the concrete examples that utilize them frequently involve other, specific functionality with little in common with each other. In contrast, some APIs like org.w3c.dom. Element and org.restlet.data.Protocol have higher overall completeness – 92% and 84% respectively – as they are aimed at specialized domains for which examples tend to have functionality in common beyond the API method calls. PAM displays no such behaviour, remaining at low overall completeness levels for all the APIs (maximum 25%), and showing indifference to whether APIs fulfil service roles or not; for example, org.restlet.data.Protocol has a completeness of 6% from PAM and java.util.Date has 11%. Likewise, the completeness results of MUDetect tend to be very similar for most APIs regardless of the utility of the API. This could be because MUDetect mines for repeated patterns regardless of their relevancy to the API. ASGard tends to provide a more complete picture of how APIs are used in practice than do PAM and MUDetect.

RQ2: How well do the templates compress the examples?

Table 3 provides summary statistics on compression. ASGard dominates PAM and MUDetect for all compression statistics, with a better mean and median (81% and 82%) than PAM (54% and 56%) and MUDetect (26% and 28%). The minimum for ASGard is 58%, which is still an adequate reduction, as opposed to −5% reduction for PAM

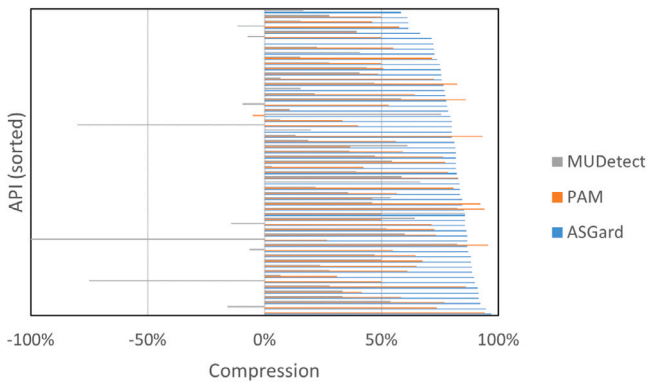


Fig. 9. Compression versus sorted API.

and −100% for MUDetect (a negative compression indicates that the number of patterns exceeds the number of originating examples). Fig. 9 shows the compression of ASGard, PAM, and MUDetect, again in a clustered bar chart. ASGard tends to have better compression than PAM and MUDetect, though in six cases PAM does slightly better.

4.5. Statistical analysis

We performed a statistical analysis of the completeness and compression measurements per each example for each of the three approaches. The data does not follow a normal distribution for either measure, according to the Shapiro–Wilk test and the Kolmogorov–Smirnov test ($p < 0.001$ in all cases); thus, we performed non-parametric tests. For the Kruskal–Wallis tests, our null hypothesis is that the samples are from the same distribution of completeness (resp. compression) across all approaches; our alternative hypothesis is that the completeness (resp. compression) for the ASGard results stochastically dominates that for the PAM and MUDetect results. For the Median tests, our null hypothesis is that the medians are the same across all approaches; our alternative hypothesis is that the median for the ASGard results is greater than that of either of the other approaches. All tests yielded $p < 0.001$; thus we reject the null hypotheses in favour of the alternatives: the completeness and compression results for ASGard stochastically dominate those for the other approaches, and the medians of the completeness and compression results for ASGard were greater than those for either of the other approaches.

5. User study

We consider the usefulness of the ASGard API usage templates in helping developers understand how to use an API. Based on our related work analysis and simulation results, we decided that MUDetect is the best alternative to compare against. Our research question for this study is:

- RQ3. Does ASGard help developers solve programming tasks in less time than MUDetect?

We designed and developed the *TemplateView* tool to allow developers to investigate the output of MUDetect or ASGard (as appropriate to the treatment being followed); we aimed for a standardized tool to minimize the potential to bias the results. The tool allows the user to browse through the available APIs and select a specific API to view its patterns/templates.

5.1. Design of the user study

The null and alternative hypotheses for our study are:

- H_0: Time to complete tasks is equal using ASGard and MUDetect.
- H_A: Time to complete tasks is less using ASGard than MUDetect.

We designed two tasks that involve two common programming activities. The first task requires the developer to complete the implementation of a simple function to use a certain API; this represents a coding task. The second task asks the developer to alter the implementation of a certain function to fix an error; this represents a debugging task. For the criteria of success: in the first task the goal was to cause the failing test cases to pass (they need to get the code to tokenize the string into an array of Strings using the java.util.StringTokenizer API to pass the test); and in the second task the goal was to eliminate the exception that was being thrown, getting the code to read the content of the given URL using the java.net.HttpURLConnection. Both tasks were done using the Eclipse IDE.

We selected the two APIs StringTokenizer and HttpURLConnection as they tend to have a defined usage that interacts with other APIs; a developer needs to understand such usages in order to use the API. Developers were given a maximum of 30 min to finish each task and the time taken for them to finish the task was recorded. We asked the developers to use the think-aloud method and we recorded our observations of them interacting with the code and TemplateView, as well as any interesting comments they made. The experiment was designed as within-subjects where each participant did the two tasks in the order and with the treatment specified by the experimental block to which they were assigned. Our study had two treatments for each task: in one treatment they use TemplateView with ASGard usage templates; in the other, they use TemplateView with the pseudocode representation of the MUDetect patterns. We had four experimental blocks for all combinations of task order and treatment. After each task and at the end of the experiment, participants were asked to complete a survey asking questions with 5-point Likert scales plus open-ended comments.

5.2. Independent and dependent variables

Our purpose is to investigate the usefulness of the usage templates or patterns in helping developers solve programming tasks efficiently. We have three independent variables: (1) the *Treatment*, with two levels: ASGard and MUDetect; (2) the *Task*, with two levels: Task 1 (coding) and Task 2 (debugging); and (3) the task *Order*, with two levels: ASGard first or MUDetect first. The dependent variable in the study is the *Completion Time*.

5.3. Participants

We recruited 12 participants, 3 participants per experiment block. We assigned participants to experiment blocks randomly with the constraint of balanced block sizes. We had 8 graduate students and 4 industrial developers. Among the graduate students, 5 had been previously employed as a developer (for 1–3 years), while 3 had not been. The industrial developers had 3–16 years of experience in industry. All participants had at least 1 year of experience with Java. All participants had some experience with Eclipse with the majority being slightly or moderately familiar with it.

5.4. Task descriptions

Participants were asked to complete two tasks, in the order defined by the experiment block. One task asks participants to use the StringTokenizer API to complete a method that takes two strings and returns a string array of the tokens in the first string split on the given delimiter represented by second string. The basic method declaration for the code was given to them, in addition to three failing test cases. They were asked to complete the task by implementing the method, getting the test cases to pass without modification. Participants needed to understand how to instantiate a StringTokenizer, how to use the methods hasMoreTokens() and nextToken() in a loop to get the tokens of a String, and how to either create an array

Table 4
Descriptive statistics for completion time data.

Treatment	Task	Mean	Min	Max	StDev
ASGard	1	9.17	5.0	13.0	3.19
	2	7.17	3.0	15.0	4.26
	Total	8.16	3.0	15.0	3.73
MUDetect	1	14.5	10.0	19.0	3.83
	2	9.83	4.0	16.0	4.75
	Total	12.16	4.0	19.0	4.78

of Strings using the countToken() method of the StringTokenizer API, or create a list, add the tokens to it, and then convert the list to an array. The other task asks participants to debug an implementation to correct a null pointer exception. They were asked to adjust the code to fix the bug using the HttpURLConnection API. The code is attempting to read from the input stream of a defined URL and print the input to the console. We explained that the task would be finished once the exception was no longer thrown and certain required output would go to the console. Participants needed to understand how to create a HttpURLConnection object from a URL object by calling the method URL.openConnection() and casting the result to HttpURLConnection. Then, they needed to create a InputStreamReader by passing it the return of the HttpURLConnection.getInputStream() and then to create a BufferedReader, passing the InputStreamReader object to it.

5.5. Results

5.5.1. Quantitative analysis

RQ3. Does ASGard help developers solve programming tasks in less time than MUDetect? In Table 4, we provide descriptive statistics about the time to complete teach task by treatment. For Task 1, ASGard required a mean time of 9.17 min versus 12.5 min for MUDetect; ASGard required a minimum of 5 min and a maximum of 13 min (10 min and 19 min for MUDetect). For Task 2, ASGard required a mean time of 7.17 min versus 9.83 min for MUDetect; ASGard required a minimum of 3 min and a maximum of 25 min (4 min and 16 min for MUDetect).

A visual inspection of the box plot for the data (Fig. 10) gives the impression that the ASGard treatment was noticeably faster than the MUDetect treatment for both tasks. We ran an N-way analysis of variance (ANOVA) to test the main effect of the different independent variables – *Treatment*, *Task*, and *Order* – on completion time. There is a significant main effect of the *Treatment* at the 0.05 level ($F(1, 20) = 6.028$, $p = 0.023$); thus, we reject the null hypothesis in favour of the alternative. The main effects of the *Task* and of the *Order* were not significant at the 0.05 level ($F(1, 2) = 4.186$, $p = 0.054$; $F(1, 20) = 1.266$, $p = 0.274$).

The results of a post hoc pairwise comparison of completion time is shown in Table 5: the difference between the average completion time is significant at the 0.05 level with $p = 0.012$, with ASGard requiring 4 ± 1.4 min less time than MUDetect. For Task 1, the average is 9.2 min for the ASGard treatment versus 14.5 min for the MUPattern treatment, a reduction by 48%. For Task 2, the average is 7.2 min for the ASGard treatment versus 9.8 min for the MUDetect treatment, a reduction by 31%.

We conclude that ASGard helps developers solve programming tasks in significantly less time compared to MUDetect.

5.5.2. Analysis of questionnaire results

After each task we asked the participants seven questions regarding their perceptions. Fig. 11 provides a histogram for the responses to each question, relative to the treatment in use. When using ASGard, 7/12 of participants found their tasks easy or moderately easy, as compared to 5/12 when using MUDetect (Fig. 11(a)). When using ASGard, 9/12 of participants thought they did extremely or very well, as opposed to 5/12 for MUDetect (Fig. 11(b)). When using ASGard, 11/12 of participants found that the task took less time than they anticipated or just about the right amount of time; with MUDetect, this dropped to 6/12 (Fig. 11(c)). We interpret these results to suggest that users feel greater help from ASGard than MUDetect.

Regarding whether participants thought that the overhead for using an approach was too great, 9/12 somewhat or strongly disagreed for ASGard, while only 5/12 somewhat or strongly disagreed for MUDetect (Fig. 11(d)). Regarding whether participants felt more likely to solve the task and solve it well when using an approach, 10/12 strongly or somewhat agreed for ASGard as opposed to 9/12 for MUDetect (Fig. 11(e)). Regarding whether participants felt better able to understand the API usage with an approach, 12/12 strongly or somewhat agreed for ASGard as opposed to 7/12 for MUDetect (Fig. 11(f)). Regarding whether participants would use an approach again with the current functionality, 11/12 strongly or somewhat agreed for ASGard as opposed to 5/12 for MUDetect (Fig. 11(g)). We interpret these results as indicating preference for ASGard on average.

At the end of the experimental session, we asked the participants which approach made the task easier to solve: 7 participants preferred ASGard, 3 preferred MUDetect, and 2 had no preference.

We asked participants their suggestions for improvements to ASGard. Some participants desired integrating the tool in the IDE. One participant suggested adding a textual description and short titles for the templates for easier navigability. Some suggestions focused on improvement to the tool interface like providing a feature to search for a certain API element in the template selection menu or a pop-up dialogue to see the substitutions for a certain structural variable by clicking or hovering on it.

6. Discussion

We discuss some remaining issues here.

6.1. Fairness

The fact that the other tools perform badly in our evaluations does not imply a lack of fairness. Fairness can only be judged by considering the details of what was done and how it was measured.

There are no other tools that aim for our goals. Since PAM and MUDetect are not designed to meet our goals, but they aim to preserve only certain information, it should not be surprising that they do not work well for our purposes. However, the study was necessary because the possibility could not be excluded beforehand that they might also do a reasonable job at preserving other information as well, through serendipity or other factors.

We must emphasize that this comparison does *not* imply that ASGard would work well as a data miner for API usage patterns (as per PAM) or as a misuse detector (as per MUDetect): our comparison was strictly on the basis of the preservation of commonality and reduction of numbers of cases to be examined.

6.2. Optimization

Various of the parameters within ASGard have been determined through an informal process of trial-and-error on some simple examples (not the ones used in our formal studies) during its development. As a result, there is no guarantee as to their optimality in any sense. However, we see this as largely irrelevant; the tool is not optimized and yet performs well. In a concrete, real-world context, the cost and value of optimization could be warranted; currently, we consider it premature.

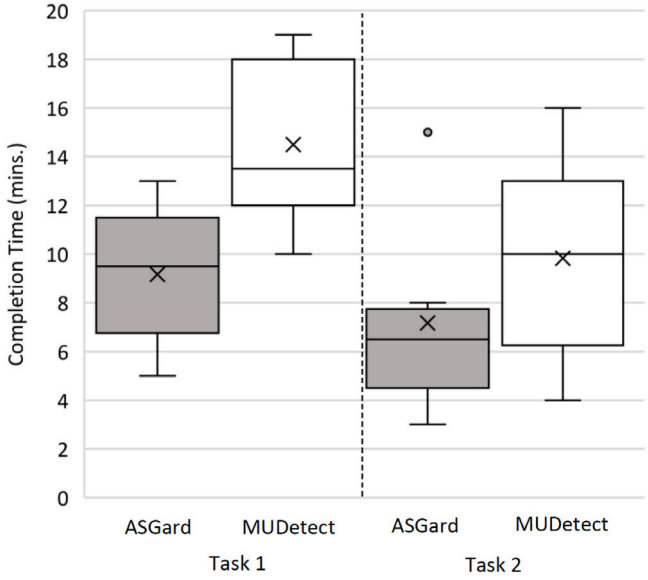


Fig. 10. Box plot of completion time versus task and treatment.

Table 5
Pairwise comparison of completion time.

Treatm. A	Treatm. B	A–B	Std. Err.	Sig.	95% Conf. Int.	
					Lower	Upper
ASGard	MUDetect	−4.000	1.402	0.012	−6.972	−1.028
MUDetect	ASGard	4.000	1.402	0.012	1.028	6.972

6.3. Threats to validity

For the simulation-based study, while our approach should apply to any programming language, we have implemented and evaluated it only for Java. Furthermore, we mined the examples and APIs to evaluate from the PAM dataset; any threats to the external validity of that dataset will also threaten our results. We see no impediments for the approach to work for other contexts since it is not tailored for Java and the PAM dataset, but its generalizability remains speculation.

We used two measures to consider the quality of API usage templates. While we argue the value of those measures, construct validity is threatened by a lack of external support to justify our choices: we cannot be certain that those measures represent what is most important about API usage templates. The rationale of the completeness measure assumes that the developer will start from the usage template closest to their mental target. While this is likely to often be the case, it is also likely to not always be the case, and the developer could choose a less-than-optimal starting template; how often that happens in practice will need to be assessed empirically.

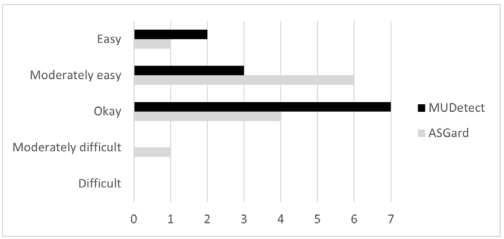
Our conversion of MUDetect's patterns to pseudocode represents a threat to construct validity, as the pseudocode may not be a fair representation of the patterns. First, it should be clear that a conversion of some form is necessary, as MUDetect's general graphs are not meaningfully comparable to (G)ASTs. We automated the conversion process to minimize the likelihood of making errors in the process. The most serious issue was the need to make fairly arbitrary decisions to introduce missing information needed in the pseudocode for a reasonable comparison with ASGard's templates: MUDetect fails to preserve various details from the original source code examples, often replacing them with something more generic, so all loops are labelled simply as "loop" without differentiation, for example. However, such choices would at best have led to a positive bias towards MUDetect: we treat "loop" as "for" which will often be the right choice and which will result in a *smaller* edit distance in our simulation than what MUDetect explicitly provides.

In the user study, we defined two tasks that represent the common programming activities that developers engage in when writing code using API which are coding and debugging. However, these two tasks are not similar in complexity as one required participants to write code from scratch and the other required them to modify some already existing code. This threatens the construct validity of the measurement where the time taken for the tasks may not be comparable. However, we wished to gain a broader understanding of the usage of the API templates in different contexts of programming activities. We defined specific criteria for the success of the task so that completion could be interpreted without bias. Further study will be needed to focus on the quality of the resulting code.

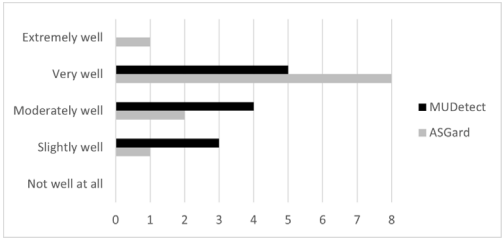
Since the interface of the tool for both treatments is similar, learning effects were possible as the participants became familiar with the TemplateView tool or the nature of these tasks, which could have affected the validity of the second task measurement. We created different experimental blocks to account for the different order of treatment and task to try and limit such effects. We also had different tasks in nature so as not to have any connection between them. Statistical analysis indicates that the learning effects were not significant. In addition, the personal experience and knowledge of participants could confound measurements. We recruited participants with diverse knowledge and background to try and mitigate such effects. We only have two tasks; one for coding and the other for debugging and we ran the study with only twelve participants. The limited number of tasks and participants threaten the generalizability of our findings. Additional studies for the tool in the field may be conducted in the future to better understand the use of our approach in a real setting. We used only Java code in our study: this can threaten the generalizability to other programming languages.

6.4. Performance

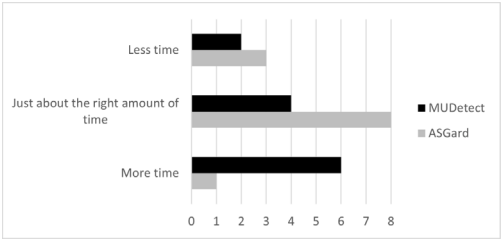
The full 1954 examples required 20 min and 5 s to process on a 2.4 GHz Dual-Core Intel Core i5 processor, using 8 GB DDR3 RAM,



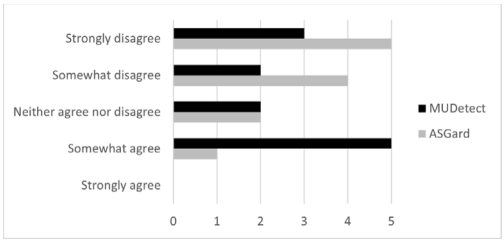
(a) "How did you find the task?"



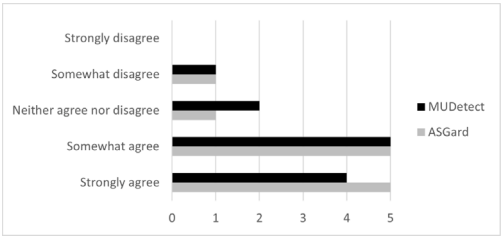
(b) "How well did you think you did?"



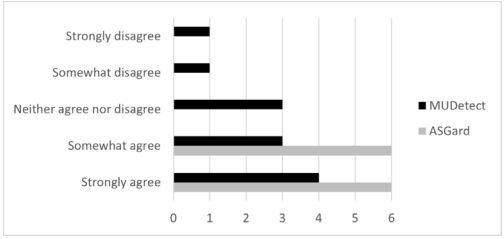
(c) "Did the task take more time, or less than you anticipated?"



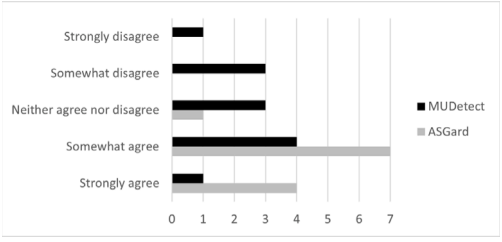
(d) "The overhead for using the approach was too much."



(e) "Using the approach, I was more likely to solve the programming task and solve it well."



(f) "I feel I could better understand the API usage with the approach than without it."



(g) "I would use the approach tool with its existing functionality again."

Fig. 11. Responses to questionnaire.

running on MacOS (Catalina). To consider the scalability of ASGard, we measured the processing time for example sets of varying sizes (Fig. 12). The datapoints fit a logarithmic distribution ($t = 244.79 \ln(|\text{examples}|) - 786.42$, $R^2 = 0.9$); we suspect that increasing benefit from pre-clustering, avoiding generalization of unrelated examples, leads to sublinear growth.

Pre-clustering is a useful optimization: we used the dataset of one API with 79 examples to test this. Without pre-clustering, ASGard took 9 min and 6 s to run versus 2 min and 59 s with it. The results were only slightly altered by this optimization since the pre-clustering does not perfectly reflect the detailed similarity calculations used otherwise. In future, we will more precisely characterize the costs of individual steps in the process (pre-clustering, AST similarity calculation, pairwise anti-unification) and variation due to the properties of the examples themselves.

6.5. Effects of example quantity and quality

The performance of ASGard and the quality of its results has a complex relationship with the quantity and quality of the examples input to it. If the examples are highly similar, the resulting templates will tend to be fewer in number and more concrete in nature. Greater diversity amongst the input examples will tend towards more resulting templates and/or more abstract templates. Reducing the various threshold values will permit evermore dissimilar examples to be generalized, resulting in more abstract templates, failing to preserve much information. Although the quantity of examples can increase the amount of work that ASGard must do, the quantity does not directly affect the abstractness nor quantity of the resulting API template set. Ultimately, the question of how many examples are needed for ASGard to become useful in a real development context beyond manual inspection or

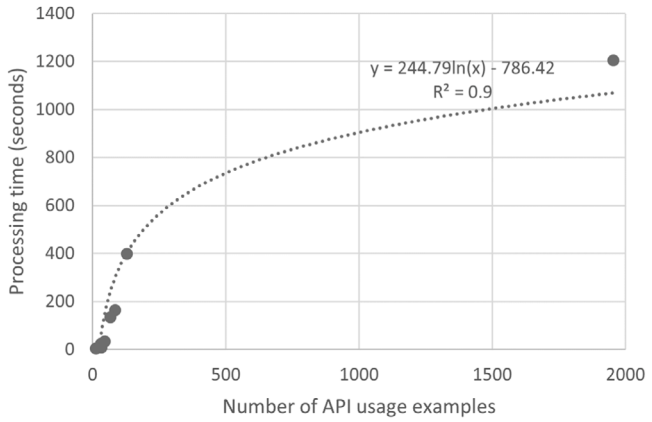


Fig. 12. Number of examples for an API versus processing time (in seconds).

simpler tools (like diff) is a complex one involving factors not easily controlled concerning the developer, like stress, fatigue, familiarity, and willingness to take risks.

6.6. Future work

Although our application focuses on API usage templates, similar approaches could preserve commonality and points of variation between examples without the constraints that were appropriate here. We see this as being useful for the discovery of novel design patterns, for design conformance, and for clone detection, going beyond first-order antiunification, as applied previously.

We will investigate several ways to improve our approach, starting with improving the pre-clustering step. Our approach relies on the k-means approach which requires the specification of the k value which can be challenging to determine especially for users with limited information regarding the data. We will investigate automating the calculation of the clusters' quality, using different measures proposed in the literature (Kirkland and De La Iglesia, 2013), under different k values using the underlying dataset so the user can determine the k value suitable for the data. We will also investigate using other clustering algorithms that do not require the specification of k, like DBSCAN (Saied and Sahraoui, 2016; Schubert et al., 2017). We will also consider doing a complete analysis of the dataset used and its coverage in terms of the API methods and their usages and investigate the implications and possible limitations of the dataset size on the output template sets.

We will study the hierarchical presentation of API usage templates, allowing developers to iteratively refine their search for a pertinent template by including evermore concrete details. In addition, we will study the elimination of the need to specify the API of interest, instead inferring the APIs present in the input examples in order to construct the feature vectors; we see this as most useful in combination with the hierarchical presentation idea. We wish to combine the work of Cottrell et al. (2008), allowing for a given template to be integrated with a concrete implementation. To address participants' concerns about the abstract names of structural variables, previewing the integration of each template would allow a context-specific presentation; otherwise, context-appropriate, generic names could be used.

Cross-validating the templates would be an interesting study to undertake: constructing API templates on one set of examples and evaluating their quality on a disjoint set of examples. This would indicate how much can be learned from existing examples in order to work in a novel context. Both simulation- and user-based studies on this would be useful.

To deal with examples arising from multiple API versions, the designated API could be specified as the union of individual versions, extending the usage vector accordingly; examples using different versions would be less likely to be generalized together within ASGard since that would tend to result in the suppression of API usage details. To allow for methods from different versions to be treated as equivalent, we would need to support equational theories that equate them, presumably allowing the specification of a mapping from the older version to a newer one. As is well known in research on API evolution (e.g., Cossette and Walker, 2012), such mappings are not always feasible when the API in question has undergone conceptual-level changes; regardless, such an extension would require additional research.

The approach underlying ASGard can be adapted so that an API designer can discover commonality and differences amongst a large number of example usages; this would serve to inform them as to recurring patterns of use (good ones or erroneous ones) by the community being supported. We are currently pursuing the research needed to achieve this.

ChatGPT (OpenAI, 2023) represents a recent, disruptive technology that many people perceive as relevant even in technical contexts. It identifies follow-up words to a set of given words, based on a deep neural network trained on trillions of examples. The end result simulates a conversational style with another person, but it is not magic. As a proprietary technology, many of its details are secret, but we can see various problems with it and related tools: being built atop a neural network, a human being cannot analyse its patterns to understand their basis, unlike with the usage templates generated by ASGard; it is prone to so-called "hallucinations", when the tool provides a confident result that cannot be justified by its training data (Ji et al., 2022); it displays stochastic behaviour, where the same query results in differing responses; and it needs massive quantities of data for its training, in combination with reinforcement learning from expert responses. One can envisage a natural language interface being supported for ASGard in order to help the developer to constrain aspects of the example selection, generalization, and filtering of templates. Our work can lay the foundation for further innovation in this area.

7. Conclusion

While mining APIs for key information can yield useful information for some applications, it enforces the conjecture that only a subset of the information contained in examples be pertinent. We have instead worked from the premise that we do not know a priori which information is useful, and instead seek all points of commonality and differences between examples; the commonalities are retained and the differences are replaced with structural variables. Our approach can produce useful templates from few examples, even just two, in principle. The approximate solution to the formal problem known as E-generalization allows us to consider both the syntactic aspects of source code examples and some limited semantic information, during the process of generalization. We utilize equational theories to represent knowledge of equivalence in semantics between relevant AST nodes without the need for a full semantic analysis. We conducted both a simulation-based evaluation and a user study for the sake of improved generality of our results, finding that the approach works better than the best alternative and that developers are able to use it in practice more effectively. While ASGard has focused on learning from API usages, the techniques underlying it are equally applicable to the generalization of any tree-based information.

CRediT authorship contribution statement

May Mahmoud: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Robert J. Walker:** Conceptualization, Methodology, Validation, Data curation, Writing – original draft, Writing – review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Jörg Denzinger:** Conceptualization, Methodology, Validation, Writing – original draft, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

DOI: 10.6084/m9.figshare.25040405.

Acknowledgements

This work was funded by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2024.111974.

References

Abid, N.J., Dragan, N., Collard, M.L., Maletic, J.I., 2015. Using stereotypes in the automatic generation of natural language summaries for C++ methods. In: Proceedings of the IEEE 31st International Conference on Software Maintenance and Evolution. In: ICSME, pp. 561–565. http://dx.doi.org/10.1109/ICSM.2015.7332514.

Acharya, M., Xie, T., 2009. Mining API error-handling specifications from source code. In: Proceedings of the 12th International Conference on Fundamental Approaches To Software Engineering. In: Lecture Notes in Computer Science, vol. 5503, pp. 370–384. http://dx.doi.org/10.1007/978-3-642-00593-0_25.

Acharya, M., Xie, T., Pei, J., Xu, J., 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 25–34. http://dx.doi.org/10.1145/1287624.1287630.

Ahmad, W.U., Chakraborty, S., Ray, B., Chang, K.W., 2020. A transformer-based approach for source code summarization. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 4998–5007. http://dx.doi.org/10.18653/v1/2020.acl-main.449.

Amann, S., Nguyen, H.A., Nadi, S., Nguyen, T.N., Mezini, M., 2019. Investigating next steps in static API-misuse detection. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories. pp. 265–275. http://dx.doi.org/10.1109/MSR.2019.00053.

Azad, S., Rigby, P.C., Guerrouj, L., 2017. Generating API call rules from version history and Stack Overflow posts. ACM Trans. Softw. Eng. Methodol. 25 (4), 29:1–29:22. http://dx.doi.org/10.1145/2990497.

Bader, J., Scott, A., Pradel, M., Chandra, S., 2019. Getafix: Learning to fix bugs automatically. Proc. ACM Program. Lang. 3 (OOPSLA), 159:1–159:27. http://dx.doi.org/10.1145/3360585.

Barnaby, C., Sen, K., Zhang, T., Glassman, E., Chandra, S., 2020. Exempla Gratis (E.G.): Code examples for free. In: Proceedings of the 28th Joint Meeting on European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 1353–1364. http://dx.doi.org/10.1145/3368089.3417052.

Blech, E., Grishchenko, A., Kniazkov, I., Liang, G., Serebrennikov, O., Tatarnikov, A., Volkhontseva, P., Yakimets, K., 2021. Patternika: A pattern-mining-based tool for automatic library migration. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops. pp. 333–338. http://dx.doi.org/10.1109/ISSREW53611.2021.00098.

Bruch, M., Monperrus, M., Mezini, M., 2009. Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering. pp. 213–222. http://dx.doi.org/10.1145/1595696.1595728.

Bruch, M., Schäfer, T., Mezini, M., 2006. FrUiT: IDE support for framework understanding. In: Proceedings of the OOPSLA Workshop on Eclipse Technology EXchange. ACM, pp. 55–59. http://dx.doi.org/10.1145/1188835.1188847.

Bulychev, P.E., Kostylev, E.V., Zakharov, V.A., 2010. Anti-unification algorithms and their applications in program analysis. In: Revised Papers of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics 2009. In: Lecture Notes in Computer Science, vol. 5947, Springer, pp. 413–423. http://dx.doi.org/10.1007/978-3-642-11486-1_35.

Bulychev, P., Minea, M., 2008. Duplicate code detection using anti-unification. In: Spring Young Researchers Colloquium on Software Engineering. pp. 51–54. http://dx.doi.org/10.15514/SYRCOSE-2008-2-22.

Bulychev, P., Minea, M., 2009. An evaluation of duplicate code detection using anti-unification. In: Proceedings of the 3rd International Workshop on Software Clones. URL http://www.informatik.uni-bremen.de/st/IWSC/bulychev.pdf.

Burghardt, J., 2005. E-generalization using grammars. Artificial Intelligence 165 (1), 1–35. http://dx.doi.org/10.1016/j.artint.2005.01.008.

Buse, R.P.L., Weimer, W., 2012. Synthesizing API usage examples. In: Proceedings of the 34th International Conference on Software Engineering. pp. 782–792. http://dx.doi.org/10.1109/ICSE.2012.6227140.

Chatterjee, P., Damevski, K., Pollock, L., Augustine, V., Kraft, N.A., 2019. Exploratory study of slack Q&A chats as a mining source for software engineering tools. In: Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories. pp. 490–501. http://dx.doi.org/10.1109/MSR.2019.00075.

Cossette, B.E., Walker, R.J., 2012. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 55:1–55:11. http://dx.doi.org/10.1145/2393596.2393661.

Cossette, B., Walker, R., Cottrell, R., 2014. Using structural generalization to discover replacement functionality for API evolution. Technical Report 2014-1058-09, Department of Computer Science, University of Calgary, URL http://hdl.handle.net/1880/49996.

Cottrell, R., Walker, R.J., Denzinger, J., 2008. Semi-automating small-scale source code reuse via structural correspondence. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 214–225. http://dx.doi.org/10.1145/1453101.1453130.

Date, H., Ishio, T., Inoue, K., 2012. Investigation of coding patterns over version history. In: Proceedings of the 4th International Workshop on Empirical Software Engineering in Practice. pp. 40–45. http://dx.doi.org/10.1109/IWESEP.2012.18.

Duala-Ekoko, E., Robillard, M.P., 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In: Proceedings of the IEEE/ACM 34th International Conference on Software Engineering. pp. 266–276. http://dx.doi.org/10.5555/2337223.2337255.

Eyal Salman, H., 2017. Identification multi-level frequent usage patterns from APIs. J. Syst. Softw. 130, 42–56. http://dx.doi.org/10.1016/j.jss.2017.05.039.

Fowkes, J., Sutton, C., 2016. Parameter-free probabilistic API mining across GitHub. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 254–265. http://dx.doi.org/10.1145/2950290.2950319.

Glassman, E.L., Zhang, T., Hartmann, B., Kim, M., 2018. Visualizing API usage examples at scale. In: Proceedings of the ACM Conference on Human Factors in Computing Systems. pp. 580:1–580:12. http://dx.doi.org/10.1145/3173574.3174154.

Gruska, N., Wasylkowski, A., Zeller, A., 2010. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In: Proceedings of the 19th International Symposium on Software Testing and Analysis. ACM, pp. 119–130. http://dx.doi.org/10.1145/1831708.1831723.

Haiduc, S., Aponte, J., Moreno, L., Marcus, A., 2010. On the use of automated text summarization techniques for summarizing source code. In: Proceedings of the 17th Working Conference on Reverse Engineering. IEEE, pp. 35–44. http://dx.doi.org/10.1109/WCRE.2010.13.

Hill, E., Pollock, L., Vijay-Shanker, K., 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In: Proceedings of the 31st International Conference on Software Engineering. pp. 232–242. http://dx.doi.org/10.1109/ICSE.2009.5070524.

Hora, A., 2021. APISonar: Mining API usage examples. Softw. - Pract. Exp. 51 (2), 319–352. http://dx.doi.org/10.1002/spe.2906.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y., Dai, W., Madotto, A., Fung, P., 2022. Survey of hallucination in natural language generation. ACM Comput. Surv. 55 (12), 248:1–248:38. http://dx.doi.org/10.1145/3571730.

Kagdi, H., Collard, M.L., Maletic, J.I., 2007. An approach to mining call-usage patterns with syntactic context. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering. pp. 457–460. http://dx.doi.org/10.1145/1321631.1321708.

Kirkland, O., De La Iglesia, B., 2013. Experimental evaluation of cluster quality measures. In: Proceedings of the 13th UK Workshop on Computational Intelligence. IEEE, pp. 236–243. http://dx.doi.org/10.1109/UKCI.2013.6651311.

Ko, A.J., Myers, B.A., Aung, H.H., 2004. Six learning barriers in end-user programming systems. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 199–206. http://dx.doi.org/10.1109/VLHCC.2004.47.

Kostylev, E.V., Zakharov, V.A., 2008. On complexity of the anti-unification problem. Discrete Math. Appl. 18 (1), 85–98. http://dx.doi.org/10.1515/DMA.2008.007.

Langville, A.N., Meyer, C.D., 2006. Google's PageRank and beyond: The Science of Search Engine Rankings. Princeton University Press, URL http://www.jstor.org/stable/j.ctt7t8z9.

LeClair, A., McMillan, C., 2019. Recommendations for datasets for source code summarization. In: Proceedings of the 17th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 1, pp. 3931–3937, URL https://aclanthology.org/N19-1000.

Levenshtein, V.I., 1966. Binary ccode capable of correcting deletions, insertions, and reversals. Sov. Phys. Doklady 10 (8), 707–710, URL https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf.

Li, H., Thompson, S., 2010. Similar code detection and elimination for Erlang programs. In: Proceedings of the International Symposium on Practical Aspects of Declarative Languages. In: Lecture Notes in Computer Science, vol. 5937, Springer, pp. 104–118. http://dx.doi.org/10.1007/978-3-642-11503-5_10.

Lindig, C., 2015. Mining patterns and violations using concept analysis. In: Bird, C., Menzies, T., Zimmermann, T. (Eds.), The Art and Science of Analyzing Software Data. pp. 17–38. http://dx.doi.org/10.1016/B978-0-12-411519-4.00002-1.

Livshits, B., Zimmermann, T., 2005. DynaMine: finding common error patterns by mining software revision histories. ACM SIGSOFT Softw. Eng. Notes 30 (5), 296–305. http://dx.doi.org/10.1145/1095430.1081754.

Lloyd, S.P., 1982. Least squares quantization in PCM. IEEE Trans. Inform. Theory 28 (2), 129–137. http://dx.doi.org/10.1109/TIT.1982.1056489.

Lo, D., Khoo, S.C., Liu, C., 2008. Mining temporal rules for software maintenance. J. Softw.: Evol. Process 20 (4), 227–247. http://dx.doi.org/10.1002/smr.375.

Mahmoud, M.A.S., 2023. API Usage Templates via Structural Generalization (Ph.D. thesis). University of Calgary, URL http://hdl.handle.net/1880/116188.

McBurney, P.W., McMillan, C., 2014. Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd International Conference on Program Comprehension. pp. 279–290. http://dx.doi.org/10.1145/2597008.2597149.

McBurney, P.W., McMillan, C., 2016. Automatic source code summarization of context for java methods. IEEE Trans. Softw. Eng. 42 (2), 103–119. http://dx.doi.org/10.1109/TSE.2015.2465386.

McLellan, S.G., Roesler, A.W., Tempest, J.T., Spinuzzi, C.I., 1998. Building more usable APIs. IEEE Softw. 15 (3), 78–86. http://dx.doi.org/10.1109/52.676963.

Michail, A., 1999. Data mining library reuse patterns in user-selected applications. In: Proceedings of the 14th IEEE International Conference on Automated Software Engineering. pp. 24–33. http://dx.doi.org/10.1109/ASE.1999.802089.

Michail, A., 2000. Data mining library reuse patterns using generalized association rules. In: Proceedings of the 22nd International Conference on Software Engineering. pp. 167–176. http://dx.doi.org/10.1145/337180.337200.

Monperrus, M., Bruch, M., Mezini, M., 2010. Detecting missing method calls in object-oriented software. In: Lecture Notes in Computer Science, vol. 6183, Springer, pp. 2–25. http://dx.doi.org/10.1007/978-3-642-14107-2_2,

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K., 2013. Automatic generation of natural language summaries for java classes. In: Proceedings of the 21st International Conference on Program Comprehension. pp. 23–32. http://dx.doi.org/10.1109/ICPC.2013.6613830.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., 2015. How can I use this method? In: Proceedings of the 37th International Conference on Software Engineering, Vol. 1. pp. 880–890. http://dx.doi.org/10.5555/2818754.2818860.

Moritz, E., Linares-Vásquez, M., Poshyvanyk, D., Grechanik, M., McMillan, C., Gethers, M., 2013. ExPort: Detecting and visualizing API usages in large source code repositories. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. pp. 646–651. http://dx.doi.org/10.1109/ASE.2013.6693127.

Mover, S., Sankaranarayanan, S., Olsen, R.B.P., Chang, B.Y.E., 2018. Mining framework usage graphs from app corpora. In: Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering. pp. 277–289. http://dx.doi.org/10.1109/SANER.2018.8330216.

Murphy, L., Kery, M.B., Alliyu, O., Macvean, A., Myers, B.A., 2018. API designers in the field: Design practices and challenges for creating usable APIs. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 249–258. http://dx.doi.org/10.1109/VLHCC.2018.8506523.

Myers, B.A., Stylos, J., 2016. Improving API usability. Commun. ACM 59 (6), 62–69. http://dx.doi.org/10.1145/2896587.

Nam, D., Horvath, A., Macvean, A., Myers, B., Vasilescu, B., 2019. MARBLE: Mining for boilerplate code to identify API usability problems. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering. pp. 615–627. http://dx.doi.org/10.1109/ASE.2019.00063.

Nasehi, S.M., Sillito, J., Maurer, F., Burns, C., 2012. What makes a good code example?: A study of programming Q&A in Stack Overflow. In: Proceedings of the 28th IEEE International Conference on Software Maintenance. pp. 25–34. http://dx.doi.org/10.1109/ICSM.2012.6405249.

Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., Di Penta, M., 2019. FOCUS: A recommender system for mining API function calls and usage patterns. In: Proceedings of the IEEE/ACM 41st International Conference on Software Engineering. pp. 1050–1060. http://dx.doi.org/10.1109/ICSE.2019.00109.

Nguyen, A.T., Hilton, M., Codoban, M., Nguyen, H.A., Mast, L., Rademacher, E., Nguyen, T.N., Dig, D., 2016. API code recommendation using statistical learning from fine-grained changes. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 511–522. http://dx.doi.org/10.1145/2950290.2950333.

Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N., 2009. Graph-based mining of multiple object usage patterns. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 383–392. http://dx.doi.org/10.1145/1595696.1595767.

Nielebock, S., Heumüller, R., Schott, K.M., Ortmeier, F., 2021. Guided pattern mining for API misuse detection by change-based code analysis. Autom. Softw. Eng. 28 (2), 15:1–15:48. http://dx.doi.org/10.1007/s10515-021-00294-x.

Oancea, C., So, C., Watt, S.M., 2005. Generalization in maple. In: Proceedings of Maple Conference. pp. 277–382, URL https://www.csd.uwo.ca/~watt/pub/reprints/2005-mc-gen.pdf.

OpenAI, 2023. Introducing ChatGPT. https://openai.com/blog/chatgpt.

Ramanathan, M.K., Grama, A., Jagannathan, S., 2007a. Path-sensitive inference of function precedence protocols. In: Proceedings of the 29th Conference on Software Engineering. pp. 240–250. http://dx.doi.org/10.1109/ICSE.2007.63.

Ramanathan, M.K., Grama, A., Jagannathan, S., 2007b. Static specification inference using predicate mining. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 123–134. http://dx.doi.org/10.1145/1250734.1250749.

Robillard, M.P., 2009. What makes APIs hard to learn?: Answers from developers. IEEE Softw. 26 (6), 27–34. http://dx.doi.org/10.1109/MS.2009.193.

Robillard, M.P., Bodden, E., Kawrykow, D., Mezini, M., Ratchford, T., 2013. Automated API property inference techniques. IEEE Trans. Softw. Eng. 39 (5), 613–637. http://dx.doi.org/10.1109/TSE.2012.63.

Robillard, M.P., Deline, R., 2011. A field study of API learning obstacles. Empir. Softw. Eng. 16 (6), 703–732. http://dx.doi.org/10.1007/s10664-010-9150-8.

Saied, M.A., Abdeen, H., Benomar, O., Sahraoui, H., 2015a. Could we infer unordered API usage patterns only using the library source code? In: Proceedings of the IEEE 23rd International Conference on Program Comprehension. pp. 71–81. http://dx.doi.org/10.1109/ICPC.2015.16.

Saied, M.A., Benomar, O., Abdeen, H., Sahraoui, H., 2015b. Mining multi-level API usage patterns. In: Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering. pp. 23–32. http://dx.doi.org/10.1109/SANER.2015.7081812.

Saied, M.A., Sahraoui, H., 2016. A cooperative approach for combining client-based and library-based api usage pattern mining. In: Proceedings of the IEEE 24th International Conference on Program Comprehension. pp. 1–10. http://dx.doi.org/10.1109/ICPC.2016.7503717.

Schubert, E., Sander, J., Ester, M., Kriegel, H.P., Xu, X., 2017. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. ACM Trans. Database Syst. 42 (3), 19:1–19:21. http://dx.doi.org/10.1145/3068335.

Sillito, J., Begel, A., 2013. App-directed learning: An exploratory study. In: Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering. pp. 81–84. http://dx.doi.org/10.1109/CHASE.2013.6614736.

Starke, J., Luce, C., Sillito, J., 2009. Working with search results. In: Proceedings of the ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools, and Evaluation. pp. 53–56. http://dx.doi.org/10.1109/SUITE.2009.5070023.

Stylos, J., Myers, B., 2007. Mapping the space of API design decisions. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 50–60. http://dx.doi.org/10.1109/VLHCC.2007.44.

Sushine, J., Herbsleb, J.D., Aldrich, J., 2015. Searching the state space: A qualitative study of API protocol usability. In: Proceedings of the IEEE 23rd International Conference on Program Comprehension. pp. 82–93. http://dx.doi.org/10.1109/ICPC.2015.17.

Tabareau, N., Tanter, É., Figueroa, I., 2013. Anti-unification with type classes. In: JournÉEs Francophones Des Langages Applicatifs. URL https://hal.inria.fr/hal-00765862.

Thayer, K., Chasins, S.E., Ko, A.J., 2021. A theory of robust API knowledge. ACM Trans. Comput. Educ. 21 (1), 8:1–8:32. http://dx.doi.org/10.1145/3444945.

Thummalapenta, S., Xie, T., 2009a. Alattin: Mining alternative patterns for detecting neglected conditions. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. pp. 283–294. http://dx.doi.org/10.1109/ASE.2009.72.

Thummalapenta, S., Xie, T., 2009b. Mining exception-handling rules as sequence association rules. In: Proceedings of the ACM/IEEE 31st International Conference on Software Engineering. pp. 496–506. http://dx.doi.org/10.1109/ICSE.2009.5070548.

Uddin, G., Dagenais, B., Robillard, M.P., Temporal analysis of API usage concepts. In: Proceedings of the ACM/IEEE 34th International Conference on Software Engineering. pp. 804–814. http://dx.doi.org/10.1109/ICSE.2012.6227138.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T., Zhang, D., 2013. Mining succinct and high-coverage API usage patterns from source code. In: Proceedings of the 10th Working Conference on Mining Software Repositories. pp. 319–328. http://dx.doi.org/10.1109/MSR.2013.6624045.

Wasylkowski, A., Zeller, A., 2011. Mining temporal specifications from object usage. Autom. Softw. Eng. 18 (3–4), 263–292. http://dx.doi.org/10.1007/s10515-011-0084-1.

Wasylkowski, A., Zeller, A., Lindig, C., 2007. Detecting object usage anomalies. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. pp. 35–44. http://dx.doi.org/10.1145/1287624.1287632.

Xie, T., Pei, J., 2006. MAPO: Mining API usages from open source repositories. In: Proceedings of the International Workshop on Mining Software Repositories. pp. 54–57. http://dx.doi.org/10.1145/1137983.1137997.

Zhang, T., Hartmann, B., Kim, M., Glassman, E.L., 2020. Enabling data-driven API design with community usage data: A need-finding study. In: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems. pp. 1–13. http://dx.doi.org/10.1145/3313831.3376382.

Zhang, T., Upadhyaya, G., Reinhardt, A., Rajan, H., Kim, M., 2018. Are code examples on an online Q&A forum reliable?: A study of API misuse on stack overflow. In: Proceedings of the ACM/IEEE 40th International Conference on Software Engineering. pp. 886–896. http://dx.doi.org/10.1145/3180155.3180260.

Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H., 2009. MAPO: Mining and recommending API usage patterns. In: Proceedings of the European Conference on Object-Oriented Programming. In: Lecture Notes in Computer Science, vol. 5653, pp. 318–343. http://dx.doi.org/10.1007/978-3-642-03013-0_15.

May Mahmoud is a Postdoctoral Fellow in the Department of Computer Science at the University of Calgary. Her research interests involve generalization and pattern detection in software.

Robert J. Walker is a Professor and Associate Head in the Department of Computer Science at the University of Calgary. His research interests involve software evolution, software reuse, and applied artificial intelligence.

Jörg Denzinger is an Associate Professor in the Department of Computer Science at the University of Calgary. His research interests involve multi-agent systems and artificial intelligence.