An Empirical Study on API Parameter Rules

Hao Zhong[1], Na Meng[2], Zexuan Li[1], and Li Jia[1]
[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
[2]Virginia Polytechnic Institute and State University, USA
zhonghao@sjtu.edu.cn,nm8247@cs.vt.edu,lizx_17@sjtu.edu.cn,insanelung@sjtu.edu.cn

ABSTRACT

Developers build programs based on software libraries to reduce coding effort. If a program inappropriately sets an API parameter, the program may exhibit unexpected runtime behaviors. To help developers correctly use library APIs, researchers built tools to mine API parameter rules. However, it is still unknown (1) what types of parameter rules there are, and (2) how these rules distribute inside documents and source files. In this paper, we conducted an empirical study to investigate the above-mentioned questions. To analyze as many parameter rules as possible, we took a hybrid approach that combines automatic localization of constrained parameters with manual inspection. Our automatic approach—PaRu—locates parameters that have constraints either documented in Javadoc (*i.e.*, document rules) or implied by source code (*i.e.*, code rules). Our manual inspection (1) identifies and categorizes rules for the located parameters, and (2) establishes mapping between document and code rules. By applying PaRu to 9 widely used libraries, we located 5,334 parameters with either document or code rules. Interestingly, there are only 187 parameters that have both types of rules, and 79 pairs of these parameter rules are unmatched. Additionally, PaRu extracted 1,688 rule sentences from Javadoc and code. We manually classified these sentences into six categories, two of which are overlooked by prior approaches. We found that 86.2% of parameters have only code rules; 10.3% of parameters have only document rules; and only 3.5% of parameters have both document and code rules. Our research reveals the challenges for automating parameter rule extraction. Based on our findings, we discuss the potentials of prior approaches and present our insights for future tool design.

ACM Reference Format:
Hao Zhong[1], Na Meng[2], Zexuan Li[1], and Li Jia[1]. 2020. An Empirical Study on API Parameter Rules. In *Proceedings of 42nd International Conference on Software Engineering (ICSE).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Software libraries (*e.g.*, J2SE [4]) are widely used, because they provide thousands of reusable APIs. Incorrectly using APIs can cause programming errors, slow down code development, or even introduce security vulnerabilities to software [20, 44]. Since correctly using APIs is important for programmer productivity and software quality, researchers have built various approaches that detect or check API usage rules by analyzing code or documentation [27, 46, 54]. For instance, Engler *et al.* [27] mined frequent calling sequences of method APIs from the code of operating systems, and revealed abnormal API usage. As another example, Zhong *et al.* [54] inferred API specifications from library documentation.

Although the above approaches mainly focus on API invocation sequences, the careful selection of legal parameter values is also important for developers to ensure the correctness of API usage. In the literature, researchers [28, 80] have proposed approaches to mine API parameter rules. For instance, Ernst *et al.* [28] built Daikon to infer invariants of variables' values from dynamic profiling of program executions. Zhou *et al.* [80] detected defects in API documents using techniques of program analysis and natural language processing. Both approaches extract rules based on predefined templates.

Although prior studies (*e.g.*, Polikarpova *et al.*[56]) show that the above approaches inferred useful parameter rules, many research questions in this research line are still open. For instance, what types of parameter rules are there, and how do those parameter rules distribute among documents and source files? These questions are important because without an overview of the API parameter rules existing in libraries, it is hard to tell how far we are from the fully automatic approaches that (i) detect constraints on API parameters, (ii) document the parameter rules reflected by code, and (iii) reveal any constraint violation in the client code of libraries.

To explore these questions, in this paper, we conducted an extensive empirical study on parameter rules. Specifically, to reveal as many parameter rules as possible, we took a hybrid approach by combining automatic fact revealing and manual inspection. In particular, given a library, it can be very time-consuming for us to manually read all code and Javadoc comments to identify and summarize the parameter rules. Therefore, we built an approach—PaRu (Parameter Rules)—to locate (1) rule descriptions in Javadoc, and (2) method APIs whose source code has parameter-related exception declarations or assert statements. Although PaRu cannot comprehend or interpret any described or implied rule, it can locate the parameters with candidate rules for further manual inspection. Here, a **candidate rule** is a rule sentence or a parameter-related exception/assertion located by PaRu.

In the second step, for each parameter located by PaRu, we manually examined the rule description in Javadoc or inspected the code with related exception or assertion. In this way, we can comprehend the meaning of each located candidate rule, and explore the following research questions:

- **RQ1:** *What is the categorization of API parameter rules?* Prior work shows that there are constraints on the values, value ranges, or data types of API parameters [80]. However, we were curious whether there is any parameter rule that does not fall into the known categories. This question is important

ICSE, 2020, Seoul, South Korea

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

because by revealing new types of rules, we may shed light on future rule extraction tools.

- **RQ2:** *How do rules distribute in Javadoc and code implementation?* Our investigation for this question serves multiple purposes. For instance, if most rules only exist in code, we need new approaches that generate Javadoc comments from code to automate rule documentation. If the rules in Javadoc and source code often conflict with each other, we need new tools to detect and resolve the contradiction.

By applying PaRu to 9 widely used software libraries that contain in total 14,392 source files, we located 5,334 parameters with candidate rules. Based on these parameters and their rules, we made the following major observations.

- *There are five major categories of parameter rules, with the sixth category (i.e., "other") covering miscellaneous rules.* We analyzed 1,688 rule-related sentences, which are located in either Javadoc comments or the exception messages of code. In addition to the known categories such as `null`-value, constant values, and value ranges, we found that 18.5% of the studied rules constrain parameters' formats (*e.g., "csvKeyValueDelimiter must be exactly 1 character")*; while 5.3% of rules describe the relation between different parameters (*e.g., "polyLats and polyLons must be equal length"*). The miscellaneous rules count for 7.0% of the inspected data. In total, we identified three new rule categories that were unknown.
- *The majority of studied rules are implicitly indicated by API code.* Specifically, 86.2% of parameters have rules defined in code, while 10.3% of parameters' rules are defined in Javadoc. The results imply that developers seldom describe parameter usage explicitly, which can cause significant confusion on users of the APIs. We only found 2.0% of the parameters to have consistent rules that are reflected in both Javadoc and code. Even fewer parameters (1.5%) have inconsistent rules, *i.e.*, mismatches between the document rules and code rules for the same parameters. Such inconsistencies are usually not bugs. Instead, the rules describe different and complementary constraints on the same parameters.

The rest of this paper is organized as follows. Section 2 introduces the background. Section 3 presents our support tool. Section 4 presents our empirical study. Section 5 interprets our findings. Section 6 discuss the potentials of related tools. Section 8 introduces the related work. Section 9 concludes this paper.

2 BACKGROUND

This section defines terms related to API parameter rules (Section 2.1), and overviews rule-mining techniques (Section 2.2).

2.1 Terminologies

API parameter rules describe or reflect the constraints on parameters of API methods. Such constraints are imposed by either software library implementation or application domains, and may limit the value or format of any parameter. Rule violations can cause coding errors and jeopardize developers' productivity. In our research, we focus on the parameter rules of public APIs, as these APIs are visible to library users and the rules can affect those users.

```java
/** ...
* @param searcher IndexSearcher to find nearest points
    from.
* @param field field name. must not be null.
* @param latitude latitude at the center: must be
    within standard +/-90 coordinate bounds.
* @param longitude longitude at the center: must be
    within standard +/-180 coordinate bounds.
* @param n the number of nearest neighbors to retrieve
*/
public static TopFieldDocs nearest(IndexSearcher
    searcher, String field, double latitude, double
    longitude, int n){
 GeoUtils.checkLatitude(latitude);
 GeoUtils.checkLongitude(longitude);
 if (n < 1) {
   throw new IllegalArgumentException("n must be at
      least 1; got " + n);
 }
 if (field == null) {
   throw new IllegalArgumentException("field must not
      be null");
 }
 if (searcher == null) {
   throw new IllegalArgumentException("searcher must
      not be null");
 }
} ...}
/** validates latitude value is within standard +/-90
    coordinate bounds */
public static void checkLatitude(double latitude) {
 if (Double.isNaN(latitude) || latitude < MIN_LAT_INCL
      || latitude > MAX_LAT_INCL) {
   throw new IllegalArgumentException("invalid
      latitude " + latitude + "; must be between "
      + MIN_LAT_INCL + " and " + MAX_LAT_INCL);
 }}
```

(a) A piece of API code with rules defined in Javadoc

```
Parameters:
searcher - IndexSearcher to find nearest points from.
field - field name. must not be null.
latitude - latitude at the center: must be within standard
           +/-90 coordinate bounds.
longitude - longitude at the center: must be within
            standard +/-180 coordinate bounds.
n - the number of nearest neighbors to retrieve.
```

(b) The Javadoc of the `nearest` method, which is generated from its code comments with the `@param` tags

Figure 1: Example parameter rules

As shown in Figure 1a, there is a method API `nearest(...)` defined in the Lucene library [1]. Among the five parameters defined for the API, one parameter is `field`. According to the API implementation, `field` must not be `null`, because the code throws an `IllegalArgumentException` if the parameter is null. Correspondingly, the library developers described this rule in the **Javadoc** comment enclosed by "/**" and "*/". In particular, when the tag `@param` is used in the comment to declare a parameter and describe the related rule(s) (see Figure 1a), a document on the parameter usage can be automatically generated when the method is publicized as a library method interface [9] (see Figure 1b).

Since parameter rules can be either explicitly mentioned in Javadoc comments or implicitly indicated by exceptions/assertions in code, we defined two terms to reflect the data sources of rules.

Definition 2.1. A **document rule** is an API parameter rule observed in API Javadoc, tagged with `@param`.

Definition 2.2. A **code rule** is an API parameter rule inferred from API source code.

In Figure 1a, `field` has both a document rule and a code rule. It is also possible that a parameter has only one kind of rule or no rule at all. For instance, the parameter `searcher` in Figure 1a has a code rule but no document rule.

Definition 2.3. A **rule sentence** is a sentence that explicitly describes constraints on a parameter.

In Javadoc, a document rule always corresponds to a rule sentence. In API implementation, a code rule may or may not correspond to a rule sentence. As shown in Figure 1a, an exception message explicitly mentions a parameter rule—"`field must not be null`", so we consider the message string as a rule sentence. There are also scenarios where an invalid parameter can trigger an exception in API implementation, but the exception message does not explicitly describe any rule. For such cases, there are code rules implied by the exceptions but there is no rule sentence in the code.

Definition 2.4. **Rule localization** is the process to identify rules (*i.e.*, document and code rules) in library implementation.

Definition 2.5. **Rule comprehension** is the process to interpret the meaning of a localized rule.

Definition 2.6. **Rule extraction/mining** involves both rule localization and rule comprehension.

In our research, we treat rule extraction as a two-step procedure. To extract a parameter rule, we first localize rules no matter whether they are in the format of rule sentences or exception-throwing/assertion code chunks. Next, for each localized rule, we summarize the meaning or semantics.

2.2 Existing Rule Extraction Techniques

Researchers explored various techniques to extract API parameter rules from client code, API documents, and/or API code.

Mining Client Code. Client code is the source code that invokes APIs. Given a software library, many approaches identify client code of the library in open source projects [23, 28, 49, 70]. Some of the approaches then compile and execute client projects [23, 28, 70]. They leverage dynamic analysis to collect the execution traces, gather run-time values of variables, and further infer invariants on the exact value or value ranges of parameters. Nguyen *et al.* use a light-weight, intra-procedural, static analysis technique to analyze the guard conditions in client code before an API is invoked [49]. This approach is limited by the API parameter rules sensed by developers of client code.

Mining Library Documents. Library documents describe the functionalities and usage of APIs in natural languages. Existing approaches typically analyze such documents with natural language processing techniques [54, 80]. These approaches usually define parsing semantic templates to locate specific natural language sentences, and convert those sentences to method specifications. For instance, one of the templates defined by Pandita *et al.* [54] is "*(subject) (verb) (object)*", which can locate rule sentences like "The path cannot be null".

Table 1: Subject projects.

Names	Files	Methods	Ex.	Para.	Doc.
commons-io	246	1,534	413	1,936	1,590
pdfbox	1,295	6,392	484	6,375	3,949
shiro	711	2,090	237	1,960	854
itext	1,503	8,930	1,110	11,089	5,784
poi	3,493	16,599	2,315	17,792	5,218
jfreechart	987	6,847	450	8,728	8,672
lucene	4,124	12,204	3,163	16,022	3,454
asm	269	1,925	274	2,614	1,015
jmonkey	1,764	10,867	1,312	14,470	4,679
total	14,392	67,388	9,758	80,986	35,215

Mining Library Code. Library or framework code is the implementation of class, method, or field APIs. Existing approaches use static analysis to infer parameter rules from API source code [17, 80]. Specifically, the state-of-the-art approach of parameter rule extraction was introduced by Zhou *et al.* [80], who combined document analysis with code analysis. For document analysis, Zhou *et al.* defined four parsing semantic templates (*e.g.*, "*(subject) equal to null*") to locate *document rules*. Meanwhile, for code analysis, they located exception throwing declarations in the body of any method API `m`. Then they related the declarations with any formal parameter defined by either `m` or other methods invoked by `m`. If a parameter `p` can trigger a thrown exception in any program execution path, they generated *code rules* by synthesizing all constraints on the path(s) for `p`'s value. By comparing the document rules and code rules of the same API parameters, they reported defective document rules. The approach's effectiveness is limited by (1) the representativeness of defined rule templates, and (2) the precision of static analysis.

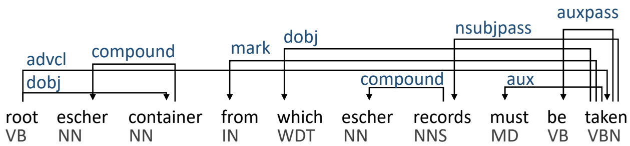
All above-mentioned techniques can automatically localize and comprehend certain rules. For this paper, we intended to identify as many parameter rules as possible in popular libraries, and assess (1) what types of parameter rules there are, and (2) how parameter rules distribute among documents and source files.

3 PARU

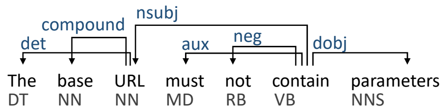
In this section, we first present our dataset (Section 3.1), and then introduce how PaRu extracts document rules (Section 3.2) and code rules (Section 3.3) from source files. Section 3.4 shows the f-scores of PaRu. PaRu focuses on rule localization instead of rule comprehension. PaRu borrows ideas from current rule mining tools, but can locate more diverse parameter rules in a scalable way.

3.1 Dataset

Table 1 shows the nine subject libraries. Column "**Names**" lists the names of libraries. In particular, `asm` [2] is an analysis library for Java bytecode, and `jmonkey` [5] is a game engine framework. Except `jfreechart`, all the other libraries were collected from the Apache foundation [8]; these libraries were designed for purposes like assisting IO functionalities, manipulating different types of files, and performing security managements. We selected these subjects because they are widely used in various programming contexts. For instance, a search of the keyword, `lucene`, returns more than 3,000 projects. Some of these projects (*e.g.*, `itext`), have been used in the evaluations of the prior rule-mining approaches [79]. Column



(a) A sentence without rules



(b) A rule sentence

Figure 2: The dependency tree

"**Files**" lists the number of Java source files. Column "**Methods**" lists the number of suspicious methods. A suspicious method is a public method that has either an `assert`/`throw` statement or a parameter document. Column "**Ex.**" lists the number of `assert`/`throw` statements inside the suspicious methods. Column "**Para.**" lists the number of parameters of the suspicious methods. Column "**Doc.**" lists the number of parameters that have documents.

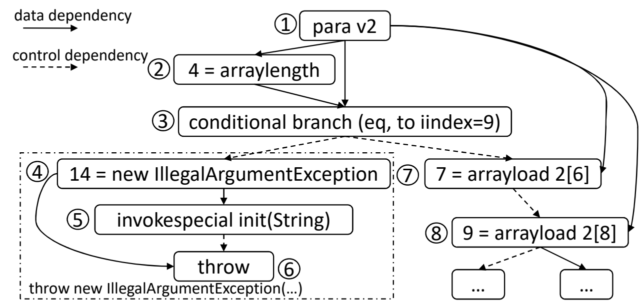
3.2 Step 1. Identifying Document Rules

Our extraction focuses on the parameter documentation labeled with `@param` tags. PaRu uses the Stanford parser [66] to build part-of-speech (POS) tags and dependencies among words of sentences. Figure 2 shows the parsing results of two sentences. The grey annotations under words denote their POS tags (*e.g.*, NN for noun). The arrows between words denote their dependencies. For example, the `dobj` arrow in Figure 2 implies that the direct object of *contain* is *parameters*. The `nsubj` arrow shows that the subject of *contain* is *URL*. More definitions of such dependencies are available in the Stanford parser manual [7]. Although the sentence in Figure 2a has a modal verb (*i.e.*, must), it does not define any rule. This sentence describes what a root escher container is and its relation to escher records, but the sentence does not define any constraint on the container usage. PaRu determines that a sentence is a **document rule**, only if (1) the sentence uses at least one modal verb, and (2) the modal verb does not appear in sub-clauses. PaRu relies on the tag `MD` to identify any modal verb within *{must, shall, should, can, may}*, because according to our observation, document rules usually contain such words.

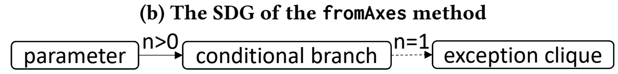
Some rule-mining approaches [54, 80] define NLP templates to mine rules, while some other approaches (*e.g.*, a variable can be `null` as defined in Zhou *et al.* [80]) include *can* and *may* as keywords when mining parameter rules. The goal of our study is not to reveal the implementation flaws in existing approaches, but to provide insights for follow-up researchers. To achieve this goal, we tried to reveal as many parameter rules as possible. Thus, we had to consider what existing approaches have done when designing PaRu. When the NLP-based approaches [54, 80] rely on parsing semantic templates to mine rules, they may miss rule sentences that do not match any predefined template. Thus, we designed PaRu to use modal verbs instead of templates to locate rules. Although *can* and *may* are less compulsory than the other modal verbs we use, because the two words were mentioned by prior work [80], we simply included them in our modal verb set for completeness.

```
1 public Quaternion fromAxes(Vector3f[] axis) {
2   if (axis.length != 3) {
3     throw new IllegalArgumentException("Axis array
          must have three elements");
4   }
5   return fromAxes(axis[0], axis[1], axis[2]);
6 }
```

(a) The code of the `fromAxes` method



(b) The SDG of the `fromAxes` method



(c) A path template that indicates parameter rules

Figure 3: Our analysis on SDGs

3.3 Step 2. Extracting Code Rules

The basic process of identifying parameter code rules. PaRu is built upon WALA [12]. PaRu first scans the Abstract Syntax Trees (AST) of source code to locate `throw` and `assert` statements. If a method API implementation includes such a statement, PaRu further builds a system dependency graph (SDG) for the API:

Definition 3.1. An SDG is a graph $g = \langle V, E \rangle$, where V is a set of nodes corresponding to code instructions, and $E \subseteq V \times V$ is a set of directed edges. Any edge, *e.g.*, $\langle s_1, s_2 \rangle \in E$, denotes a data or control dependency from s_1 to s_2.

Definition 3.2. An **exception clique** is an SDG subgraph, that corresponds to an exception-throwing statement or assertion.

To construct an SDG that visualizes any control or data dependencies within Java code, WALA first translates source code into its intermediate representation called IR [10] by converting each source line to one or more IR instructions. Next, WALA creates a node for each instruction, and connects nodes with directed edges based on the control or data dependencies between instructions.

For the example shown in Figure 3, an exception-throwing statement (see Line 3 in Figure 3a) is converted to three instructions, corresponding to three nodes in an SDG (see nodes ④, ⑤, and ⑥ in Figure 3b). We use **exception clique** to refer to the subgraph that consists of these three nodes and any edges in between (see dashed region in Figure 3b). Similarly, WALA translates each assert statement into three IR instructions, whose nodes compose an exception clique similar to the one shown in Figure 3b. The only difference is that an assert statement replaces ④ with a node for the `AssertionError` creation instruction. To detect code rules, PaRu locates both types of exception cliques in SDGs.

Given the method implementation of a public API, PaRu first identifies the declared parameters in the method header and locates all exception cliques in the body. For each located exception clique

Algorithm 1: `findAllPaths` Algorithm

```
Require:
    sn is a source node
    tn is a target node
Ensure:
    paths denotes all the paths from sn to tn
1:  nextNodes ← sn.successors
2:  for nextNode ∈ nextNodes do
3:    if nextNode.equal(tn) and path.isValid() then
4:      stack ← new Stack
5:      for node ∈ path do
6:        stack.add(node)
7:      end for
8:      paths.add(stack)
9:    else if nextNode ∉ path and path.isPartialValid() then
10:     path.push(nextNode)
11:     findAllPaths(nextNode, to)
12:     path.pop()
13:   end if
14: end for
```

in SDG, PaRu checks whether the clique is reachable from any parameter, i.e., whether there is any path that starts from a parameter declaration and goes through the exception clique. When such a path is found, PaRu concludes that the exception clique depends on the parameter and there is an implicit constraint on the parameter value. Figure 3c shows an exemplar path that PaRu can find. The path starts with the declaration of parameter `n`, goes through an `if`-condition that checks the parameter value range, and ends with an exception clique that prints "`n=1`" in the error message.

Algorithm 1 shows the details of searching for all the valid paths from a given source node to a target node. Before adding a path to the set of valid paths, PaRu checks the path at Line 3. However, if it only checks the path at Line 3, it has to search many invalid paths between Line 10 and Line 12. As each statement is split into multiple nodes in an SDG, SDGs can become quite large if a method is long. To reduce the search effort, we add another check to Line 9. At this line, it is infeasible to fully determine whether a path is valid, but we can remove many invalid paths. For example, if we find that an `if`-condition has no data dependency on any parameter, we can stop the exploration of its successors. As a path is incomplete at Line 9, at this point, PaRu concludes that a path is invalid if the incomplete path is not a prefix of a valid path. As shown in Figure 1a, code rules can have rule sentences. After a valid path is extracted, PaRu further extracts rule sentences from the thrown message, and such sentences are later manually analyzed (Sections 4.1.1 and 4.2.1).

Slicing. Algorithm 1 is less effective to find valid paths, if a graph is quite large. For example, when searching for all the valid paths from ① to $\boxed{4, 5, 6}$ in Figure 3b, Algorithm 1 will explore the paths such as ① ⟶ ⑦ and ① ⟶ ⑧. When an SDG is large, the exploration is time-consuming. Weiser [72] proposed the concept of program slicing. Given a program location l and a variable v, the backward slicing intends to find all the statements of the program that can affect the value of v at l. For each exception clique, PaRu locates the backward slice before it searches for all valid paths, in order to save the search effort. In particular, WALA has a program slicer [11]. Given a statement and an SDG, the slicer finds all the statements that appear in the backward slice of the statement. For each slice, PaRu builds a smaller SDG that contains only nodes of the slice. After SDGs are sliced, for Figure 3b, Algorithm 1 does not explore ⑦ or ⑧, since they do not appear in the sliced SDG.

Table 2: The precision, recall, and f-score of PaRu.

Name	Precision	Recall	F-score
commons-io	98.0%	94.2%	96.1%
pdfbox	92.9%	88.1%	90.4%
shiro	95.5%	96.9%	96.2%
itext	89.9%	94.7%	92.2%
poi	98.8%	96.4%	95.2%
jfreechart	95.3%	83.6%	89.1%
lucene	85.2%	100.0%	92.0%
asm	98.9%	96.9%	97.9%
jmonkey	93.8%	88.4%	91.0%

3.4 The F-scores of PaRu

We were curious how effectively PaRu can locate parameter rules, so we constructed a ground truth data set of parameter rules for some Java files, and applied PaRu to those files to automatically locate rules. By comparing PaRu's reports against our ground truth, we assessed the precision, recall, and f-score of PaRu.

The setting. The third and the fourth authors are two PhD students in Computer Science, who have more than three years of Java coding experience. To construct the ground truth data set for PaRu evaluation, the two students read source files in Table 1, and tried their best to manually recognize parameter rules in those files. As these students did not read or write any source code for PaRu, so they have no bias towards PaRu when building the data set. Such setting ensures the objectiveness of PaRu evaluation.

Although some prior approaches [28, 54] mine parameter rules from data sources other than source files (*e.g.*, execution profiles), we believe our ground truth of parameter rules is reasonably good for two reasons. First, as illustrated in Figure 1, API documents are automatically generated from the Javadoc comments in code, so the rules described or implied by source files can always cover those rules in API documentation. Second, in high-quality software, the source files usually define or validate constraints on parameters before using those parameters. It is meaningful to rely on software implementation to distill parameter rules. Thus, we decided to manually inspect source code only, instead of also examining other information resources simultaneously.

Specifically, the two students manually analyzed 20 randomly selected source files in each project. For each parameter p of method API m, the students read the document and implementation of m to decide whether p has any document or code rule. Since the value p may be tested by code inside m or any method called by or calling m, the students examined m together with methods that have any caller-callee relationship with m to infer code rules. After the manual inspection, in our group meeting, the students discussed their results to reach a consensus. In total, the two students manually identified 135 documented rules and 539 code rules, which were used as the gold standard. For the 180=20×9 source files, we applied PaRu to locate any document or code rule. We then compared

Table 3: Top ten verbs.

commons-io	pdfbox	shiro	itext	poi	jfreechart	lucene	asm	jmonkey
be (153)	**be (35)**	**be (82)**	**be (95)**	**be (157)**	**be (68)**	**be (368)**	**be (102)**	**be (218)**
compared (6)	include (10)	**contain (2)**	retrieved (4)	exceed (5)	used(36)	>= (55)	**have (6)**	**have (29)**
>= (5)	enforced (6)	**used/use (3)**	**have (3)**	react (4)	>= (22)	**have (21)**	**used (3)**	loaded (9)
match (2)	**have (5)**	represented (1)	reuse (3)	supplied (4)	> (8)	<= (14)	delegate (3)	> (5)
contain (2)	**use/used (5)**	create (1)	registered (2)	aligned (4)	**contain (5)**	> (11)	updated (1)	add (4)
called (1)	compressed (4)	retained (1)	**contain (2)**	belong (3)	**have (3)**	change (10)	store (1)	filled (4)
failed (1)	defined (3)	null (1)	submitted (2)	>= (3)	<= (2)	**contain (9)**	create (1)	match (4)
write (1)	point (2)	examined (1)	fit (2)	**used (3)**	supplied (2)	**use (6)**	**contain (1)**	fall (3)
-	support (1)	queried (1)	opened (1)	**havel (3)**	add (2)	process (6)	updated (1)	assigned (3)
-	**contain (1)**	retained (1)	**use (1)**	override (2)	match(1)	match (3)	sorted(1)	**contain (2)**

the located rules against the gold standard to calculate precisions, recalls, and f-scores for PaRu.

Results. Table 2 shows the evaluation results. For 8 out of the 9 projects, PaRu acquired f-scores higher than 90%. Both precision and recall rates are generally high (*i.e.*, 85.2%-98.9% precision and 83.6%-100% recall). The recalls of PaRu are not 100%, because some rules can only be manually identified in nonstandard ways, but are very challenging to be located by any automated tool. For instance, a parameter rule is sometimes described by comments of the whole method, but not by the Javadoc comment of that parameter. As Apache projects follow strict regulations, based on our evaluation results shown in Table 3, the nonstandard scenarios are rare, and PaRu has detected most parameter rules. Our results imply that the rules reported by PaRu are very likely to be representative, and we can rely on these rules to build a taxonomy of parameter rules.

4 EMPIRICAL STUDY

With PaRu, we conducted an empirical study to explore our research questions listed in Section 1. We used PaRu to extract parameter rules from the dataset in Section 3.1. In the 14,392 files from 9 real projects, PaRu identified in total 5,334 parameters to have rules. From these parameters with rules, PaRu extracted 1,688 rule sentences that are described in either Javadoc comments or exception/assertion messages. There are only 187 parameters that have both document rules and code rules.

We manually examined the 1,688 rule sentences and rules related to the above-mentioned 187 parameters. The manual inspection procedure took several weeks. This section presents our manual analysis protocols and investigation findings. More details of our results and the gold standards are listed on our project website: https://github.com/drzhonghao/parameterstudy.

4.1 RQ1. Rule Categorization

4.1.1 Protocol. To explore RQ1, we manually classified all the 1,688 rule sentences. Here, if a parameter rule is extracted with no rule sentence identified (*e.g.*, an exception thrown with the empty message body), we do not inspect the rule, because it is too expensive to understand a parameter solely based on source code. We first classified rule sentences by the verbs which follow the extracted modal verbs. Although the result reveals how programmers write rule sentences, we realized that the verbs do not present an accurate classification. To handle the problem, we manually read all rule sentences, and classified them based on the semantics. During the manual inspection, the first and the third authors prepared the initial inspection results. The other authors checked the results, until they came to an agreement on all the results.

4.1.2 Result. Table 3 shows the top ten verbs. Our result shows that developers use limited verbs to define parameter rules. The `commons-io` project even does not have ten verbs. In this table, we highlight verbs that appear in more than half of the projects. According to this definition, only four verbs are commonly used: "be", "contain", "have", and "use". It seems that library developers exploited certain verbs much more frequently than other verbs when defining parameter rules, so it is infeasible to distinguish parameter rules only based on verbs. Instead, Table 4 shows the results of manual inspection. In total, we identified six types of rule sentences:

C1. Null. *This category is about whether a parameter is allowed or disallowed to be* `null`*.* For instance, the code in Figure 1a shows that an exception is thrown if the `field` parameter is `null`; the related rule sentence is "must not be null." Additionally, the `formatCellValue` method of `poi` has a document rule: "The cell (can be null)". C1 corresponds to two categories defined by Zhou *et al.* [80], including "Nullness allowed" and "Nullness not allowed".

C2. Range*. This category focuses on the legal ranges of parameter values.* When defining ranges, a rule sentence can define both maximum and minimum values, such as "`latitude` value: must be within standard +/-90 coordinate bounds." Meanwhile, a rule sentence can also define only the minimum or maximum, such as "`maxMergeCount` should be at least 1.". C2 corresponds to the "range limitation" category of Zhou *et al.* [80].

C3. Value. *This category is about legal values of parameters.* For example, a rule sentence goes "For STRING type, `missing` value must be either STRING_FIRST or STRING_LAST." The existing approach Daikon [28] monitors program execution status at runtime, collects values of variables, and infers value invariants accordingly. Therefore, it is likely that Daikon can identify rules of C1-C3.

C4. Format. *This category focuses on the formats of parameters.* An exemplar rule sentence is "`csvKeyValueDelimiter` must be exactly 1 character." Zhou *et al.* [80] defined a "type restriction" category to describe type requirements on parameters, such as "value of `key` must be an boolean." Generally speaking, type restrictions are about formats of parameters, so we map this "type restriction"

Table 4: Rule sentences

Name	Null	Range	Value	Format	Relation	Others
commons-io	141	22	0	3	4	0
pdfbox	13	6	8	36	0	3
shiro	78	2	0	11	0	1
itext	52	8	28	8	0	11
poi	54	52	13	62	13	13
jfreechart	6	77	5	16	1	4
lucene	87	274	29	55	30	45
asm	14	5	57	18	6	20
jmonkey	54	68	14	103	36	22
total	499	514	154	312	90	119
%	29.6%	30.5%	9.1%	18.5%	5.3%	7.0%

Table 5: Rule sentences from thrown messages.

Name	Null	Range	Value	Format	Relation	Other
commons-io	43	16	0	1	0	0
pdfbox	4	4	8	10	0	1
shiro	76	2	0	8	0	1
itext	8	8	0	0	0	7
poi	31	46	3	43	11	5
jfreechart	3	27	4	10	0	4
lucene	78	198	18	32	27	18
asm	11	5	2	5	1	2
jmonkey	51	53	3	69	34	21
total	305	359	38	178	73	59
%	30.1%	35.5%	3.8%	17.6%	7.2%	5.8%

category to C4. However, the scope of C4 is broader. More samples are as below.

(1) `xmp` should end with a processing instruction.
(2) `Sheet` names must not begin or end with (').
(3) `uid` must be byte[16].
(4) The `moveFrom` must be a valid array index.
(5) Value data `source` must be numeric.
(6) The length of the data for a `ExObjListAtom` must be at least 4 bytes.

Since format-related rule sentences have diversified descriptions, it seems to be very challenging to define one or more templates to match all those sentences for automatic rule comprehension.

C5. Relation. *This category is about relations between parameters.* For example, a rule sentence defines the lengths of two parameters as "`polyLats` and `polyLons` must be equal length. " More samples are shown below:

(1) `origin` and `region` must both be arrays of length 3.
(2) `UserEditAtom` and `PersistPtrHolder` must exist and their offset need to match.
(3) Index of last row (inclusive), must be equal to or larger than firstRow.
(4) `maxItemsInBlock` must be at least 2*(`minItemsInBlock-1`).
(5) upper value's type should be the same as numericConfig type.
(6) the number of values of a block, must be equal to the block size of the BlockPackedWriter which has been used to write the stream.

C6. Other. *This category includes miscellaneous rules that do not belong to any of the above-mentioned categories.* For example, a rule sentence defines the synonyms of parameters: "Synonyms must be across the same field." More samples are as follows:

(1) A `filename` to view must be supplied as the first argument, but none was given.
(2) `RC4` must not be used with agile encryption.
(3) SplineStart must be preceded by another type.
(4) features must be present for `TextLogitStream`.
(5) This must be well-formed unicode text, with no unpaired surrogates.
(6) input `automaton` must be deterministic.

Among the above categories, C5 and C6 are solely detected by PaRu. It is more challenging to define templates of C5 and C6 than those of other rules. The templates of the other categories typically define usages of single variables, but the templates of C5 define usages of multiple variables. The prior approaches (*e.g.*, Ernst *et al.* [28]) reply on frequencies to mine parameter rules, but the supports of C6 are too low to be mined.

Table 4 presents the distribution of rule sentences among our six categories. These sentences are from either Javadoc or code. The rule categories in our taxonomy are mutually exclusive. If a parameter has multiple rule sentences, each sentence is analyzed and classified independently. According to this table, C1 (Null) is the dominant rule category in projects `commons-io`, `shiro`, and `itext`. C3 (Value) is the dominant category in project `asm`. C4 (Format) dominates the sentences in projects `pdfbox`, `poi`, and `jmonkey`. In total, the three categories "Null", "Range", and "Value" account for 69.1% of rule sentences.

Finding 1. In total, 69.1% of rule sentences define simple rules such as `null` values, range limits, and legal values.

We were also curious what types of parameter rules are usually enforced in code, so we reorganized our manual analysis results and constructed Table 5 to illustrate the rule distribution among code of different projects. Overall, the rule distributions shown in Table 5 are similar to those shown in Table 4. For instance, the top three categories in Table 5 include C1, C2, and C4, which categories separately count for 30.1%, 35.5%, and 17.6% of the rules in thrown messages. Meanwhile, the top three categories in Table 4 are also C1, C2, and C4, but their percentages are slightly different: 29.6%, 30.5%, and 18.5%. Interestingly, C3 takes up only 3.8% of the sentences in thrown messages, but counts for 9.1% of all inspected rule sentences. This discrepancy indicates that developers usually document more Value rules but enforce fewer Value rules in code, probably because it is tedious and error-prone for developers to enumerate all (il)legal values of a parameter for checking.

In both tables, C3, C5, and C6 have much fewer rules than the other three categories. It is tedious and time-consuming for developers to write code and enforce certain rules (*e.g.*, C6). For instance, as shown in Figure 6a, "The Strings must be ordered as they appear in the directory hierarchy of the document ...". This rule sentence belongs to C6 and it specifies a particular ordering of strings in the parameter array `components`. Although the description makes sense, it is difficult to implement the parameter validation logic.

```
/**...
* @param firstSheetIndex the scope, must be −2 for add−in references...
*/
public int addRef(int extBookIndex, int firstSheetIndex, int lastSheetIndex) {
  _list.add(new RefSubRecord(extBookIndex, firstSheetIndex, lastSheetIndex));
  return _list.size() − 1;
}
/** a Constructor for making new sub record*/
public RefSubRecord(int extBookIndex, int firstSheetIndex, int lastSheetIndex) {
  _extBookIndex = extBookIndex;
  _firstSheetIndex = firstSheetIndex;
  _lastSheetIndex = lastSheetIndex;
}
```

(a) Only document rules

```
/**
* replaces the internal child list with the contents of the supplied <tt>childRecords</tt>
*/
public void setChildRecords(List<EscherRecord> childRecords) {
  if (childRecords == _childRecords) {
    throw new IllegalStateException("Child␣records␣private␣data␣member␣has␣escaped");
  }...}
```

(b) Only code rules

Figure 4: Unmatched parameters rules

Finding 2. The top three categories of parameter rules include "Null", "Range", and "Format"; the other three categories contain a lot fewer rules, probably because those rules are harder to document and implement.

4.2 RQ2. Rule Distribution

4.2.1 Protocol. To explore RQ2, we first investigated how the 5,334 parameter rules localized by PaRu distribute among the subject projects. Next, for the 187 parameters with both document and code rules, we further examined how well the two kinds of rules for each parameter match each other. As we did in RQ1, the first and the third authors prepared the initial inspection results. The other authors checked the results, until they came to an agreement on all results.

4.2.2 Result. Among the 5,334 parameter rules located by PaRu, 550 parameters only have document rules; 4,597 parameters only have code rules; 108 parameters have document rules matching code rules; and 79 parameters have unmatched rules. Figure 5 illustrates the rule distributions among projects. In the figure, the horizontal axis represents the breakdown of parameters in each project, depending on (1) whether a parameter has one or two kinds of rules and (2) whether the two kinds of rules match if a parameter has both. With more details, "only doc" denotes parameters with only document rules; "only code" denotes parameters with only code rules; "matched" denotes parameters with matched rules; and "unmatched" denotes parameters with unmatched rules.

We found that most parameters have only one kind of rules. Figure 4 shows method samples from `poi`. Specifically, Figure 4a contains document rules only, which rules are not enforced in code. The `addRef` method has three parameters, and its document defines a rule for the `firstSheetIndex` parameter. The rules define that the parameters must be -2, for add-in references. However, the code of the method does not check the two parameter rules. The method calls the `RefSubRecord` method. In Figure 4a, we present the called code. This method does not check its parameters either. In contrast, Figure 4b shows a parameter rule that is not documented but implemented in the code. The `setChildRecords` method throws an exception, when an input is identical to its stored record, but its document does not define the parameter rule.

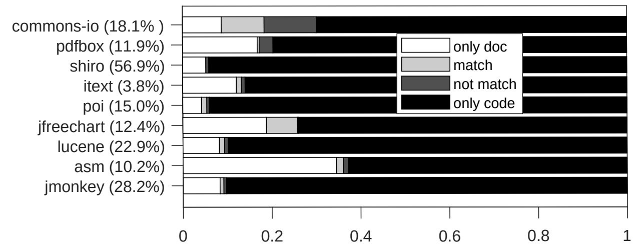


Figure 5: The distribution of detected conflicts

Finding 3. In total, 86.2% of parameters have only code rules, and 10.3% of parameters have only document rules.

Researchers have proposed various approaches to recommend API documents [24] and to mine specifications from documents [79]. Our results reveal a practical problem for these approaches, which is that API parameter rules are usually not documented. Novick and Ward [52] complained that programmers are reluctant to read documents or manuals. Probably due to this fact, instead of writing documents, API developers often implement parameter-checking logic in code, and warn client developers of any invalid API parameter via exceptions or assertions.

Although exceptions and assertions can potentially assist client developers, they may fail to warn programmers due to various issues. First, programmers cannot see any thrown message or assertion failure, if exceptions/assertions are hidden or screened. For example, DERBY-5396 [3] reports that an exception is swallowed. Second, client developers may find it difficult to understand why exceptions are thrown. Among the examined code rules shown in Figure 5, only 19% of rules have rule sentences to explicitly explain why exceptions are thrown. Finally, developers need to have high-quality test cases with good test coverage, in order to trigger exceptions/assertions related to API parameter usage. However, it is very unlikely that client developers can always develop good test suites to satisfy the need.

Figure 5 shows that except `commons-io`, less than 4% of parameters have both document rules and code rules (*i.e.*, either matched or unmatched). In addition, we found that unmatched rules do not necessarily imply bugs; they were introduced when API developers specified one set of rules in comments but implemented a distinct set of rules in code. For example, Figure 6a shows a method from `poi`. The method description defines three document rules (*e.g.*, the input list must be ordered), but the code checks none of these rules. Instead, the code checks whether the input list contains `null` values, which rule is not mentioned in Javadoc. Figure 6b shows another example, which is from the project `commons-io`. The method description defines only one document rule, but the code checks three other rules.

```java
/**...
*@param components the Strings making up the path to a
*document. The Strings must be ordered as they appear
*in the directory hierarchy of the the document -- the
*first string must be the name of a directory in the
*root of the POIFSFileSystem, and every Nth (for N>1)
*string thereafter must be the name of a directory in
*the directory identified by the (N-1)th string...
*/
public POIFSDocumentPath(final String [] components)
      throws IllegalArgumentException{ ...
 for(int j = 0; j < components.length; j++){
    if((components[j]==null)||(components[j].length()
          ==0)){
      throw new IllegalArgumentException("components 
           cannot contain null or empty strings");
    }...}}
```

(a) Document rules are more detailed

```java
/**...
* @param file ..., must not be null
*/
public ... openInputStream(final File file)...{
 if(file.exists()){
    if (file.isDirectory()){
      throw new IOException("File ... is a directory");
    }
    if(file.canRead()==false){
      throw new IOException("File ... cannot be read");
    }
 }else{
    throw new FileNotFoundException("File ... does not 
         exist");
 }...}
```

(b) Code rules are more detailed

Figure 6: Conflicts are not always bugs

> **Finding 4.** In total, 3.5% of parameters have both document rules and code rules. Only 1.5% of parameters have inconsistent rules, and such inconsistencies are often not bugs.

Zhou *et al.* [80] complained that it is often infeasible to extract accurate method calls when they appear in the branches of condition statements. As a result, they skip all constraints that are related to such method calls, and thus ignore the conflicts between documents and code implementations of corresponding parameters. The distribution of parameter rules reveals that even if an approach can infer all correct rule conditions, the approach still cannot detect many rule conflicts because the two types of rules have little overlap. Meanwhile, our results also highlight the importance of conflict detection tools, such as the one built by Zhou *et al.* [80]. Programmers seem reluctant to have consistent rules between Javadoc comments and source code, probably because it is challenging for them to maintain the rule consistency. Conflict detection tools can assist developers to maintain the consistency. Therefore, such tools are likely to (1) encourage programmers to document more parameter rules, and (2) reduce the technical barriers for library API adoption.

We found that some methods have document rules but no code rules, mainly because there are flaws in source code. Namely, programmers describe those rules in Javadoc, and wait for the flaws to be fixed before implementing those rules in code. Such scenarios indeed introduce technical debts. It will be interesting to further explore these scenarios in the future.

5 THE INTERPRETATION OF OUR FINDINGS

In this section, we interpret our findings:

Data sources. Researchers mined API rules from various data sources such as client code [28], documents [79], and API code [80]. Our empirical study focuses on a single data source—source files, because we believe this source to be sufficient to cover most API parameter rules extractable from other data sources. There are two reasons to explain our insight. First, lots of API documents about parameter usage are automatically generated from source files (i.e., from Javadoc comments). Second, when client developers invoke APIs, they usually refer to library documentation and/or API code for correct API usage. Additionally, Finding 4 shows that the extracted document rules and code rules have little overlap. This observation justifies our study approach that analyzes both code and comments of API methods, instead of only inspecting one type of data in source files.

Mining techniques. As introduced in Section 2.2, existing approaches typically use predefined parsing semantic templates to mine parameter rules, while we took a hybrid approach (i.e., refined keyword-based search + manual inspection) to mine rules. Finding 1 shows that the templates of existing tools can handle at most 69.1% of parameter rules. Unfortunately, adding more templates does not necessarily help current tools to retrieve more rules, because the remaining rules seldom present common sentence structures. If tool builders would like to define specialized templates to capture remaining rules, it is quite like that (1) many complicated templates have to be defined, and (2) many irrelevant sentences may be wrongly extracted. As mentioned by Legunsen *et al.* [36], rules mined based on templates can be superficial or even false.

Hidden and changing rules. For more than half of the studied source files, PaRu did not localize any parameter rule. However, it is unsafe to claim that all these source files have no rule at all. Shi *et al.* [64] show that even API developers may be unaware of parameter rules sometimes; once developers realize any missing rules, they have to rewrite the documentation and/or code to append rules. In such scenarios, we may miss parameter rules by mining source files.

6 DISCUSSION ON RELATED TOOLS

Motivation. To assess the effectiveness of existing rule mining tools, we chose not to apply tools to our dataset, because direct comparisons reveal problems of specific tools but such problems may be not worth further investigation by future research. Instead of determining the effectiveness of a specific tool, researchers [76, 77] have estimated the potential of the tool by comparing its technical assumptions with the nature of data. For instance, Zhong and Su [77] compared manual fixes with the methodology design of automatic program repair [29] to estimate the potentials of the state-of-the-art tools. In our research, we also conducted a similar theoretical comparison between existing parameter rules and the potentials of current rule mining tools. As long as the method design of a tool can reveal some rules in one category, we considered the tool to be able to handle the whole category given comprehensive extensions. The theoretical comparison puts higher bars for us to claim our research novelties, but can effectively reveal new research directions and inspire new tool design.

Comparison between PaRu and current rule mining tools. Although PaRu is similar to current tools in certain aspects, it is different in terms of the research objectives, methodologies, and some approach design choices.

As for research objectives, prior work reveals parameter rules for (1) dynamic rule checking [23, 28, 49, 70], (2) consistency checking [80], or (3) automatic document comprehension [54]. Researchers focused on certain types of rules, but never explored the gap between the rules in the wild and those extractable by current tools. We designed PaRu to localize as many candidate rules as possible, in order to identify any rule category overlooked by prior research.

As for methodologies, existing tools automate both rule localization and rule comprehension, while PaRu automates rule localization only. Because PaRu does not need to automatically comprehend rules, its approach based on modal verbs is more flexible than prior work [49, 80]. Consequently, PaRu can locate more candidate rules than prior work, many of which rules may not match any parsing semantic template defined before.

As for design choices, Nguyen *et al.* extracted the `if`-conditions before API method invocations in client code, and then leveraged those frequently checked conditions to infer parameter rules [49]. We designed PaRu to scan library implementation instead of the client code of library APIs, because there can be APIs that have not been invoked by any client but still have parameter rules. Additionally, Zhou *et al.* [80] analyzed code statically to reveal the intra-procedural control/data dependency relationship, while PaRu conducts inter-procedural program depenency analysis to gather more context information and ensure higher analysis accuracy.

Theoretical assessment of the effectiveness by current rule mining tools. JML [19] includes written parameter rules such as pre- and postconditions. To calculate how many rules can be mined, the prior approaches (*e.g.*, Nguyen *et al.* [49]) typically consider JML as the golden standard of their evaluations. Due to the heavy manual efforts, JML defines parameter rules of only limited J2SE classes. In addition, as writing JML specifications is too time-consuming and error-prone, the authors of JML [19] mentioned that they wrote JML specifications based on what were inferred by Daikon. As a result, JML can be biased and incomplete. Although our identified rules are not fully correct, Table 2 shows that their f-scores are reasonably high. We have released our identified parameter rules on our website. If researchers remove all wrong parameter rules, the remaining rules can enrich the gold standard of JML, and researchers can evaluate their tools on the enriched gold standard to explore the limitations of such tools.

Daikon [28] is the state-of-the-art tool for mining invariants. Section 5.5 of its manual [6] lists the templates of its supported invariants. According to this manual, Daikon has the potential to mine the parameter rules in the "Range" category (*e.g.*, the `EltUpperBound` template), the "Value" category (*e.g.*, the `EltOneOf` template), and the "Relation" category (*e.g.*, the `Equality` template). For the "Null" category, Daikon has a related `EltNonZero` template to define "$x \neq 0$". It may be feasible to extend this template to detect parameter rules in the "Null" category. Based on the above templates, we estimate that Daikon has the potential to mine 74.5% of parameter rules. However, adding more templates may be sufficient to make only minor improvements, since the remaining rules are fractional. For example, we inspected rule sentences of the "Format" category, and we found that it is infeasible to summarize them into limited rule templates. Polikarpova *et al.* [56] found that Daikon inferred about half of their manually written rules. Their analyzed rules are loop invariants, preconditions, postconditions, and class invariants. Typically, these rules fall into the "Null", "Value", and "Range" categories. Considering their results, in practice, Daikon can mine about 30% of parameter rules, which leaves adequate space for improvement.

Zhou *et al.* [80] defined four templates to locate parameter rules, *i.e.*, nullness not allowed, nullness allowed, type restriction, and range limitation. In Table 4, "Null" and "Range" categories account for 60% of parameter rules. We consider type restrictions to belong to the "Format" category, and this category accounts for additional 18.5% of parameter rules. As shown in Section 4.1, "Format" contains more types of parameter rules than type restrictions. As a result, the approach by Zhou *et al.* has the potential of mining about 70% of parameter rules.

Nguyen *et al.* [49] extracted preconditions API method invocations in client code. Similar to PaRu, the technique can locate a parameter rule if the parameter is checked in client code. After a parameter rule is located, Nguyen *et al.* propose techniques to merge conditions and to infer non-strict inequality preconditions. These techniques are limited to "Null", "Range", and "Value" in Table 4, since other types of parameter rules (*e.g.*, formats) are difficult to be merged. In total, the approach by Nguyen *et al.* can potentially identify 69.1% of parameter rules.

7 THREATS TO VALIDITY

Threats to internal validity. Our manual inspection of parameter rules may be subject to human bias. As introduced in Section 4.2.1, if a parameter has both document rules and code rules, we have to manually determine whether they are identical, which can introduce errors. As Finding 4 shows that less than 3.5% of parameters can have unmatched rules, although we need more advanced techniques to eliminate the threat, the impact of this threat is low. Additionally, some identified document rules and code rules may be incorrect due to random errors. To reduce the threat, we released all found parameter rules on our website. Researchers can inspect the results and help us further reduce the threat.

Threats to external validity. Although we analyzed thousands of files of nine popular libraries, the subjects are limited and all in Java. In addition, eight out of the nine projects are from Apache, which has a more strict coding convention than other open source communities. We can mitigate the threat by including more subject projects [48], and exploiting more sources to extract parameter rules [55]. However, our major findings may not change much, since we select different types of projects. Another threat is concerning code rules without any rule sentence. In our study, we did not manually inspect such rules. Although the limitation has no impact on Finding 3, it can influence the generalizability of Findings 1 and 2. Zhou *et al.* [80] showed that even recent approaches cannot formalize accurate code rules from API code. We need more advanced techniques or nontrivial manual efforts to reduce the threat. We listed all the code rules without rule sentences on our website, so other researchers can help further reduce the threat.

8 RELATED WORK

Empirical studies on APIs. Researchers conducted empirical studies to understand various issues about API usages such as the knowledge on concurrency APIs [53] or deprecated APIs [60], rules in API documents [45], the evolution of APIs [34, 64, 73], the obstacles to learn APIs [62], the links between software quality and APIs [39], the impact of API changes on forum discussions [40], the practice on specific APIs [47], the mappings of APIs [78], the adoption of trivial APIs [13], and the impact of the type system and API documents on API usability [26]. Like ours, most of the above studies focus on the taxonomies of software engineering data. Usman *et al.* [68] and Ralph [58] presented guidelines for such studies. Amann *et al.* [15] and Legunsen *et al.* [36] compared the effectiveness of tools that detect API-related bugs. These studies explore other angles than our research questions. Zhong and Mei [75] conducted an empirical study to answer open questions in mining API call sequences, but our study focuses on parameter rules.

Mining parameter rules. Client code is a major data source for invariant mining. With test cases, Ernst *et al.* [28] and Hangal and Lammine [31] mined invariants from program execution traces. In particular, Henkel and Diwan [32] mined invariants in algebraic specifications, and proposed a tool [33] for writing such specifications. Csallner *et al.* [21] introduced dynamic symbolic execution to mine invariants. Wei *et al.* [70, 71] inferred postconditions based on Eiffel contracts. Smith *et al.* [65] inferred relations between inputs and outputs. API code is also a major data source of invariant mining. Dillig *et al.* [25] inferred invariants through abductive inference. Gulwani *et al.* [30] encoded programs into boolean formulae, and inferred preconditions. API document is another data source of mining invariants. Zhou *et al.* [80] inferred four types of invariants from documents. Pandita *et al.* [54] combined documents and API code to infer invariants. Zhou *et al.* [80] complained that it is challenging to extract accurate rule conditions from API code. Partially due to the challenges, researchers [74] conducted large-scale evaluations only on client code or documents. For API libraries, invariants typically define parameter rules. Although this research topic is intensively studied, researchers [56, 67] argued that some underlying questions are still open. Our study explores such questions, and our findings are useful to further improve the state of the art.

Mining sequential rules. Ammons *et al.* [16] mined automata for APIs. Follow-up researchers [41, 54] refined this approach, and others [50, 51] mined similar formats such as graphs. Robillard *et al.* [61] showed that automata and graphs are equivalent in the scenario of specification mining. The research in this line can be reduced to the grammar inference problem, and can be solved by corresponding techniques (*e.g.*, the k-tail algorithm [18]). Li and Zhou [38] mined method pairs, and other researchers [63] improved their approaches in more complicated contexts. Engler *et al.* [27] extracted frequent call sequences, and other researchers [59, 69] improved their approach with more advanced techniques. Furthermore, researchers [37, 42] encoded mined sequences as temporal logic. The research in this line can be reduced to sequence mining [14]. Furthermore, Le *et al.* [35] combined sequences and invariants for more informative specifications, and researchers [22, 57] used test cases to enrich mined specifications. Mei and Zhang [43] advocate applying big data analysis for software automation, and mining sequential rules is one of the key techniques to extract knowledge from software engineering data. Our empirical study focuses on parameter rules, but its findings may be useful to these approaches. For example, the distribution of document rules and code rules can apply to sequential rules. It is worthy exploring whether our results are still valid for sequential rules.

9 CONCLUSION AND FUTURE WORK

API libraries have been widely used, but are often poorly documented. When programmers do not fully understand API usage, they can introduce API-related bugs into their code. To handle this issue, researchers have proposed various approaches to facilitate better API usage. In particular, a popular research area is to mine parameter rules for APIs. Although some industrial tools are already implemented, we still do not know (1) how many categories of API parameter rules there are, and (2) what is the rule distribution among Javadoc comments and source code. The exploration of both questions is meaningful and important, because the acquired knowledge can guide our future tool design for rule mining and rule enforcement.

To explore both questions, we developed PaRu that localizes document rules and code rules in library source files. Based on the localized rules, our study identifies six categories of parameter rules, and reveals that most parameter rules are defined only in code, but not in documentation. Based on our results, we summarized four findings, and provided our insights on three topics such as data sources, mining techniques, and hidden rules. With our insights, in the future, we plan to work towards better mining and recommendation techniques for parameter rules.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is sponsored by the National Key R&D Program of China No. 2018YFC083050, NSF-1845446, and ONR N00014-17-1-2498.

REFERENCES

[1] 2018. Apache Lucene. https://github.com/apache/lucene-solr.
[2] 2018. ASM. https://asm.ow2.io.
[3] 2018. DERBY-5396. https://issues.apache.org/jira/browse/DERBY-5396.
[4] 2018. The documentation of J2SE. https://docs.oracle.com/javase/8/docs/api/index.html.
[5] 2018. jMonkeyEngine. http://jmonkeyengine.org.
[6] 2018. The manual of Daikon. http://plse.cs.washington.edu/daikon/download/doc/daikon.pdf.
[7] 2018. The manual of the Stanford parser. https://nlp.stanford.edu/software/dependencies_manual.pdf.
[8] 2018. The Apache foundation. http://www.apache.org/.
[9] 2018. The documentation of Javadoc. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html.
[10] 2018. The WALA IR. http://wala.sourceforge.net/wiki/index.php/UserGuide:IR.
[11] 2018. The WALA Slicer. http://wala.sourceforge.net/wiki/index.php/UserGuide:Slicer.
[12] 2018. WALA. https://wala.sf.net.
[13] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? An empirical case study on Npm. In *Proc. ESEC/FSE*. 385–395.

[14] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *Proc. ICDE*. 3–14.

[15] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. A systematic evaluation of static API-misuse detectors. *IEEE Transactions on Software Engineering* (2019).

[16] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Proc. 29th POPL*. 4–16.

[17] Florence Benoy and Andy King. 1996. Inferring argument size relationships with CLP(R). In *Proc. LOPSTR*. 204–223.

[18] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.* 100, 6 (1972), 592–597.

[19] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. 2005. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer* 7, 3 (2005), 212–232.

[20] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags. 2019. How Reliable is the Crowdsourced Knowledge of Security Implementation?. In *Proc. ICSE*. 536–547.

[21] C. Csallner, N. Tillmann, and Y. Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proc. ICSE*. 281–290.

[22] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. 2010. Generating test cases for specification mining. In *Proc. ISSTA*. 85–96.

[23] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight defect localization for Java. In *Proc. 23rd ECOOP*. 528–550.

[24] Uri Dekel and James D. Herbsleb. 2009. Improving API documentation usability with knowledge pushing. In *Proc. ICSE*. 320–330.

[25] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. In *Proc. OOPSLA*. 443–456.

[26] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How do API documentation and static typing affect API usability?. In *Proc. 36th ICSE*. 632–642.

[27] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proc. 18th SOSP*. 57–72.

[28] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1-3 (2007), 35–45.

[29] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[30] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2009. Constraint-based invariant inference over predicate abstraction. In *Proc. VMCAI*. 120–135.

[31] S. Hangal and M.S. Lam. 2002. Tracking down software bugs using automatic anomaly detection. *Proc. ICSE* (2002), 291–301.

[32] Johannes Henkel and Amer Diwan. 2003. Discovering Algebraic Specifications from Java Classes. In *Proc. ECOOP*. 431–456.

[33] Johannes Henkel and Amer Diwan. 2004. A Tool for Writing and Debugging Algebraic Specifications. In *Proc. ICSE*. 449–458.

[34] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How Do Developers React to API Evolution? The Pharo Ecosystem Case. In *Proc. 31st ICSME*. 1–9.

[35] TienDuy Le, XuanBach Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions. In *Proc. ASE*. 115–125.

[36] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *Proc. ASE*. 602–613.

[37] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL specification mining. In *Proc. ASE*. 81–92.

[38] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. ESEC/FSE*. 306–315.

[39] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API change and fault proneness: a threat to the success of Android apps. In *Proc. FSE*. 477–487.

[40] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How do API changes trigger Stack Overflow discussions? a study on the Android SDK. In *Proc. 22nd ICPC*. 83–94.

[41] David Lo, Leonardo Mariani, and Mauro Pezzè. 2009. Automatic steering of behavioral model inference. In *Proc. ESEC/FSE*. 345–354.

[42] Shahar Maoz and Jan Oliver Ringert. 2015. GR(1) Synthesis for LTL Specification Patterns. In *Proc. ESEC/FSE*. 96–106.

[43] Hong Mei and Lu Zhang. 2018. Can big data bring a breakthrough for software automation? *Science China Information Sciences* 61, 5 (2018), 056101.

[44] Na Meng, Stefan Nagy, Daphne Yao, Wenjie Zhuang, and Gustavo Arango Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *ICSE*.

[45] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737.

[46] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method?. In *Proc. 37th ICSE*. 880–890.

[47] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. "Jumping Through Hoops": Why do Java developers struggle with cryptography APIs?. In *Proc. ICSE*. 935–946.

[48] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in software engineering research. In *Proc. ESEC/FSE*. 466–476.

[49] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. 2014. Mining preconditions of APIs in large-scale code corpus. In *Proc. 22nd FSE*. 166–177.

[50] Hung Viet Nguyen, Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2014. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proc. 36th ICSE*. 791–802.

[51] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proc. ESEC/FSE*. 383–392.

[52] David Novick and Karen Ward. 2006. Why don't people read the manual?. In *Proc. SIGDOC*. 11–18.

[53] Semih Okur and Danny Dig. 2012. How do developers use parallel libraries?. In *Proc. 20th FSE*. 54–65.

[54] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *Proc. ICSE*. 815–825.

[55] Gayane Petrosyan, Martin P Robillard, and Renato De Mori. 2015. Discovering information explaining API types using text classification. In *Proc. ICSE*. 869–879.

[56] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. 2009. A comparative study of programmer-written and automatically inferred contracts. In *Proc. ISSTA*. 93–104.

[57] Michael Pradel and Thomas R. Gross. 2012. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proc. ICSE*. 288–298.

[58] Paul Ralph. 2018. Toward methodological guidelines for process theories and taxonomies in software engineering. *IEEE Transactions on Software Engineering* (2018).

[59] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Path-sensitive inference of function precedence protocols. In *Proc. ICSE*. 240–250.

[60] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation?: the case of a smalltalk ecosystem. In *Proc. 20th FSE*. 76–87.

[61] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API property inference techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013), 613–637.

[62] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[63] Aymen Saied, Omar Benomar, Hani Abdeen, and Houari Sahraoui. 2015. Mining multi-level API usage patterns. In *Proc. SANER*. 23–32.

[64] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An empirical study on evolution of API documentation. In *Proc. FASE*. 416–431.

[65] Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. 2017. Discovering relational specifications. In *Proc. ESEC/FSE*. 616–626.

[66] Richard Socher, John Bauer, Christopher D Manning, et al. 2013. Parsing with compositional vector grammars. In *Proc. ACL*. 455–465.

[67] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. 2012. Understanding user understanding: determining correctness of generated program invariants. In *Proc. ISSTA*. 188–198.

[68] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. 2017. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology* 85 (2017), 43–59.

[69] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. 35–44.

[70] Y. Wei, C.A. Furia, N. Kazmin, and B. Meyer. 2011. Inferring better contracts. In *Proc. 33rd ICSE*. 191–200.

[71] Yi Wei, Hannes Roth, Carlo A Furia, Yu Pei, Alexander Horton, Michael Steindorfer, Martin Nordio, and Bertrand Meyer. 2011. Stateful testing: Finding more errors in code and contracts. In *Proc. ASE*. 440–443.

[72] Mark Weiser. 1981. Program slicing. In *Proce. ICSE*. 439–449.

[73] Wei Wu, Adrien Serveaux, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2014. The impact of imperfect change rules on framework API evolution identification: an empirical study. *Empirical Software Engineering* 20, 4 (2014), 1126–1158.

[74] Tao Xie and David Notkin. 2003. Tool-assisted unit test selection based on operational violations. In *Proc. ASE*. 40–48.

[75] Hao Zhong and Hong Mei. 2018. An empirical study on API usages. *IEEE Transaction on software engineering* (2018).

[76] Hao Zhong and Na Meng. 2018. Towards reusing hints from past fixes -An exploratory study on thousands of real samples. *Empirical Software Engineering*

23, 5 (2018), 2521–2549.

[77] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proc. ICSE*. 913–923.

[78] Hao Zhong, Suresh Thummalapenta, and Tao Xie. 2013. Exposing behavioral differences in cross-language API mapping relations. In *Proc. ETAPS/FASE*. 130–145.

[79] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring resource specifications from natural language API documentation. In *Proc. ASE*. 307–318.

[80] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *Proc. ICSE*. 27–37.