A Human-in-the-loop Approach to Generate Annotation Usage Rules

by

Mansur Gulami

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Frameworks and libraries provide functionality through Application Programming Interfaces (APIs). Developers might misuse these APIs, because the library's usage rules are often implicit, undocumented, or not readily available in the form of checkable rules. At the same time, manually writing usage rules for each API is time consuming. Therefore, researchers have proposed various techniques to automatically mine API usage rules. However, mined rules are not always accurate, resulting in false positives when used for misuse detection.

To overcome these trade-offs, in this thesis, we combine rule mining and manual rule authoring approaches by creating a human-in-the-loop API usage rule generation pipeline. Based on our industrial collaborator's needs, our work focuses on generating annotation-based API usage rules for MicroProfile, a framework designed for building microservices using Enterprise Java. We use an existing frequent itemset-based pattern-mining technique to mine MicroProfile annotation usage rules. We contribute a GUI-based rule validation tool (RVT) that represents rules in an easy-to-read English-like syntax and allows experts to browse through the mined rules to validate (accept, edit, discard) them. Our pipeline then automatically generates checkable API usage rules from the confirmed rules, which can then be used to detect misuses or to enhance documentation.

We verify the correctness of the automatically generated static analysis checks by evaluating our misuse detector against 517 projects. Our results

show that our misuse detector can find misuses with 100% recall and 84% precision.

To assess the usefulness of having mined rules as a starting point for rule authoring and to assess the usability of RVT in validating rules, we perform a user study with four MicroProfile API experts where they validate 18 mined rules. The results show that the API experts find having starting points useful when it comes to rule authoring. Additionally, the experts unanimously agree on the usefulness of having a dedicated tool for authoring rules.

Preface

Chapters 3-9 of this thesis were published at the proceedings of the CASCONxEVOKE 2022 conference.

Paper: Mansur Gulami, Ajay Kumar Jha, Sarah Nadi, Karim Ali and Yee-Kang Chang, Emily Jiang. **A Human-in-the-loop Approach to Generate Annotation Usage Rules: A Case Study with MicroProfile**. *In Proceedings of Proceedings of the 32nd Annual International Conference on Computer Science and Software Engineering (CASCON'22).*

This research was partially funded by IBM CAS Canada.

Acknowledgements

First, I have to thank my supervisor, Dr. Sarah Nadi, for her guidance and support throughout my graduate studies. Her impeccable attention to detail in her feedback and mentorship skills have helped me to reach new heights. I would also like to thank our close collaborator Dr. Karim Ali for his valuable feedback.

This thesis would not be possible without our collaborators Emily Jiang, Yee-Kang Chang and Vijay Sundaresan from IBM. Finally, I would like to thank my colleague and collaborator, Ajay Kumar Jha, for offering tons of constructive feedback and overall, always being ready to help whenever I needed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Java annotations provide meta-information about the program elements that they annotate. For example, the `@Deprecated` annotation indicates that the annotated program element is no longer supported [20]. Annotations are widely used in various types of Java applications including enterprise Java applications [34, 92]. Major Java enterprise frameworks, such as Spring [78] and MicroProfile [51], facilitate the development of enterprise applications mainly through annotations.

MicroProfile is a collection of specifications that provides Application Programming Interfaces (APIs) for client developers to create applications with a microservice architecture (*"small, autonomous services that work together"*) [51, 59]. For example, the MicroProfile Health specification provides mechanisms to check whether a service is started, ready to accept requests, or live [53]. MicroProfile provides these functionalities mainly through annotation-based APIs. For example, in Figure 1.1, the `@Liveness` annotation is used to check whether the authorization service is live [55]. There are different runtimes that support MicroProfile specifications such as Open Liberty [63], Helidon [26], and Payara [65].

Similar to how library API calls have usage rules that determine correct behaviour (e.g., `hasNext()` must return `true` before invoking `next()` on an `Iterator` object to avoid throwing a `NoSuchElementException` [82]), annotations also have usage rules. For example, as shown in Figure 1.1, the target class of the `@Liveness` annotation must implement the `HealthCheck` interface

```
1 @Liveness
2 public class AuthServiceHealthCheck implements HealthCheck {
3     @Override
4     public HealthCheckResponse call() { /*omitted*/ }
5 }
```

Figure 1.1: An example illustrating the usage of the MicroProfile `@Liveness` annotation.

to register for a liveness check [55]. Container management systems such as Kubernetes use liveness checks to see if a particular container needs to be restarted [16]. Violation of this usage rule will cause the liveness check to not function properly, without showing any explicit error message. We refer to such violations of annotation usage rules as *API misuses*, or *misuses* for short.

To prevent annotation misuses, we would ideally have access to checkable usage rules and tools that allow automated checking of client code against these rules. There are existing tools that enable writing annotation usage rules and scanning a target codebase for misuses [18, 48, 95]. However, such tools assume that the usage rules are already known and readily available to encode, which is typically not the case [73, 85]. Additionally, manually creating API usage rules from scratch requires human effort, which can be difficult and time-consuming [45, 81].

To address the issues of writing API usage rules from scratch, researchers have utilized pattern mining techniques [23, 47, 60, 82, 88]. Pattern mining discovers usage rules in an automated fashion. The general idea is that if the frequency of an API usage is more than a user-specified threshold, we consider this usage as a pattern, and patterns are considered as usage rules [71]. Accordingly, deviations from a pattern are treated as misuses. Researchers have mined different types of code artifacts, such as source code [60, 82, 88], API change history [47], and execution traces [23], to extract rules automatically. However, misuse detectors that directly use mined rules to detect misuses suffer from low precision, with the state-of-the-art detector having only 33% precision [82], which limits practical use. Reasons for the low precision include the fact that mined patterns may represent common usages (i.e., idioms) in-

stead of rules. Mined patterns may also represent rules that are not entirely correct (i.e., partially correct rule). A partially correct rule might have some missing or extra elements.

Overall, manually writing API usage rules from scratch is time-consuming but leads to more accurate rules; on the other hand, automatically mining patterns relieves the manual burden but can lead to inaccurate rules. In this thesis, our goal is to generate annotation usage rules by combining the advantages of these two approaches, while mitigating their disadvantages. Our main idea is to introduce a human into the loop, but without the full burden of authoring rules from scratch. We use mined, unverified annotation usage rules (i.e., candidate rules) as starting points for human experts to create usage rules so that the process of creating rules will be less difficult and tedious. At the end of this process, we have human-validated annotation usage rules that can be directly used for detecting misuses or enriching documentation.

In this thesis, we describe our industrial collaboration with IBM to create a human-in-the-loop approach for generating MicroProfile annotation usage rules. Figure 1.2 shows an overview of our approach. We first mine candidate rules from MicroProfile client projects (Step 1). We then present the mined candidate rules to human experts for validation (Step 2). Finally, we automatically generate static analysis checks from the confirmed rules; these checks are used by our misuse detector to find annotation misuses in MicroProfile client projects (Step 3). The previous work from our research group explores mining candidate MicroProfile annotation rules using frequent itemset mining [62]. This thesis thus focuses on Steps 2 and 3 of the process, which are essential to combine the two worlds of automated pattern mining and manual rule authoring.

Specifically, we focus on validating mined candidate rules and developing a misuse detector encoded with the validated rules. To validate mined candidate rules, we develop a web-based tool, Rule Validation Tool (RVT). RVT automatically encodes mined candidate rules in a domain-specific language (DSL) and presents them to experts for validation. RVT allows experts to not only validate the presented candidate rules, but also modify the ones that

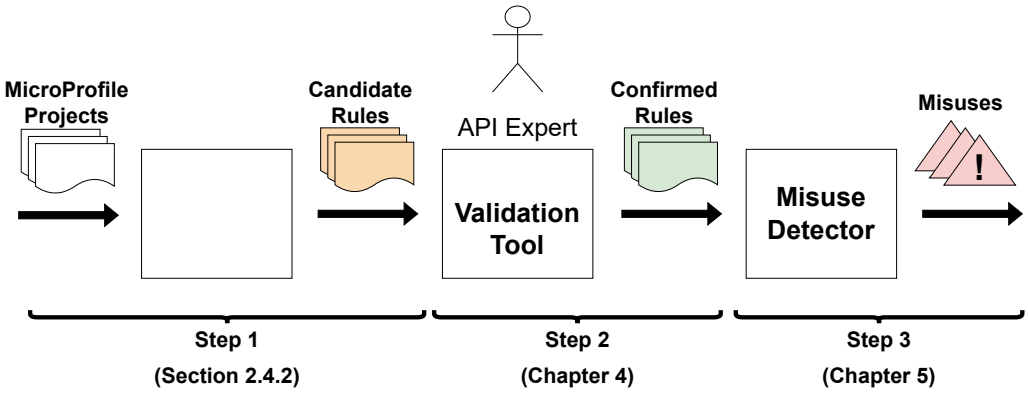

Figure 1.2: An overview of our human-in-the-loop rule generation approach.

are partially correct. After experts validate or modify the candidate rules, RVT generates a final set of static analysis checks for the validated rules. We also develop a misuse detector in the form of a Maven plugin that uses the generated static analysis checks to detect annotation misuse in client projects. We evaluate our misuse detector on 517 MicroProfile client projects and find 106 misuses, 6 of which are previously unknown misuses. Finally, we perform a user study with four MicroProfile API experts from our industry partner to assess the usability of RVT for validating and modifying mined candidate rules. Our results show that API experts find that having a starting point for rule authoring is useful, and also find our rule format to be easily understandable. Our participants also provide us with additional feedback on how to improve RVT and its DSL for authoring rules, which we discuss in this thesis. We publicly share the GitHub repository of our project [49] which includes the source code of the tools and the user study artifacts.

1.1 Thesis Contributions

The following are the contributions we make in this thesis:

- A human-in-the-loop approach that combines pattern mining and specification writing approaches to generate MicroProfile annotation usage rules.
- A web-based tool, RVT, that automatically encodes the mined candidate rules and presents them to human experts for modification and validation.
- A Maven plugin for misuse detection of the validated rules.

- An experiment for verifying the correctness of the misuse detector.
- A user study involving four MicroProfile API experts to evaluate the usefulness of RVT.

1.2 Thesis Organization

We organize the thesis as follows. Chapter 2 introduces the necessary background information. In Chapter 3, we provide the literature review. We introduce our human-in-the-loop approach in Chapter 4. Chapter 5 introduces our misuse detector and its evaluation. Chapter 6 discusses the user study of RVT. Chapter 7 and 8 present the threats to validity of our experiments and the discussion of the implications of our results, respectively. Finally, Chapter 9 concludes the thesis.

Chapter 2

Background

This chapter introduces topics that are essential to understand the problem that we discuss in this thesis. We start by briefly introducing microservices and what MicroProfile is and how it facilitates writing microservices in Java. We then introduce Java annotations and how MicroProfile makes use of them. Finally, we discuss the existing pattern mining approach that we use in Step 1 of our pipeline (see Figure 1.2) to discover MicroProfile annotation usage rules [62].

2.1 Microservices

Using a monolithic architecture is the traditional way of building applications where components responsible for different functionalities (such as authentication, processing payments, and data input/output) are packaged into a single executable [19]. A monolithic architecture has certain benefits such as easier log tracing, since all the processing takes place in the same process. Deployment is also easy with monolithic applications since it requires only a single executable to be deployed. However, there are certain disadvantages of developing applications in a monolithic style. Any minor change in the application requires the re-deployment of the whole application. Scalability is another key issue. Scaling individual components of the application becomes impossible since everything is included in a single executable (e.g, a JAR or a WAR file for Java applications). The application also becomes vulnerable to a single point of failure; if any module has an error, the availability of the entire sys-

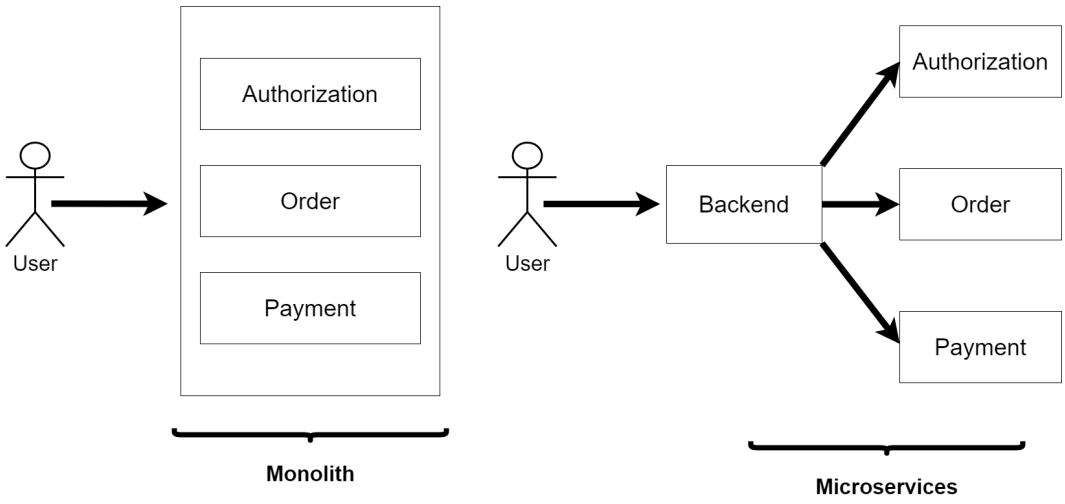

Figure 2.1: A comparison of monolithic and microservices architectures

tem is at risk. With very large monoliths, the development process can also be challenging since all developers have to work with the same codebase.

A microservices architecture, on the other hand, is a way of building applications that consists of multiple smaller and independent services [59]. Each service can have its own separate technology stack (i.e. the collection of tools, frameworks, and databases an application uses) and can be deployed independently of one another. The idea behind microservices is to split the monolithic applications into smaller, manageable applications that can be handled individually. Figure 2.1 illustrates the difference between monolithic and microservices architectures. As we can see, in the monolithic style, all the components are inside of a single application, while in microservices, the components represent independent services. Since the services are independent entities, microservices address the re-deployment, scalability, and a single point of failure issues of the monolithic architecture.

There are several tools and frameworks that facilitate building microservices in all major programming languages. For example, Spring Boot [78], Micronaut [50], and MicroProfile [51] are popular Java frameworks that facilitate building microservices.

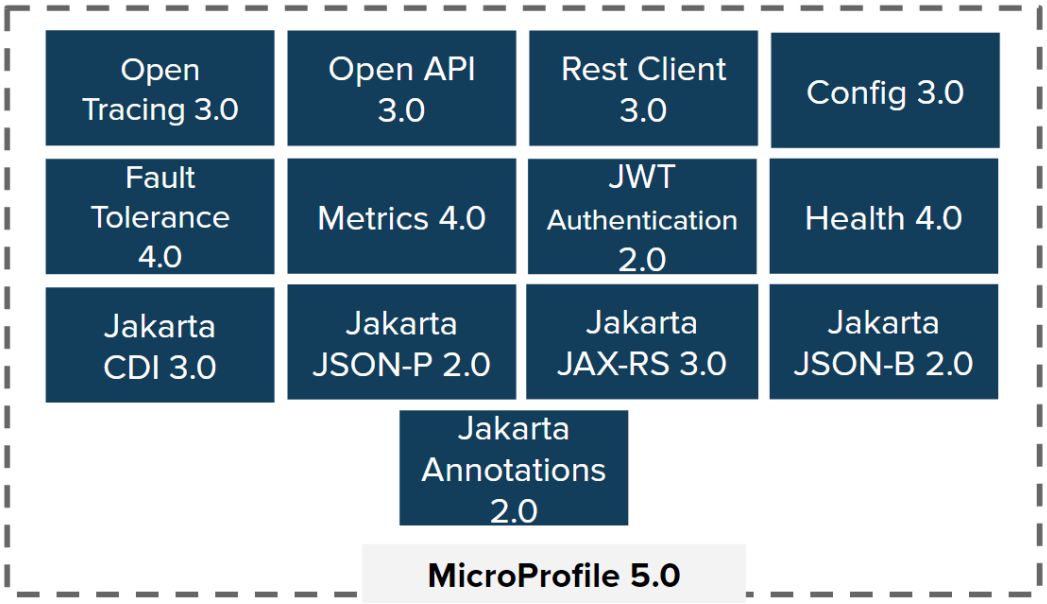

Figure 2.2: MicroProfile specifications as of version 5.0 [52]

2.2 MicroProfile

MicroProfile is a collection of specifications that provide client developers with all the necessary APIs to facilitate the development of microservices in Java [51]. MicroProfile extends Jakarta EE (formerly known as Java EE) which is another collection of specifications for building monolith enterprise applications in Java [27]. Jakarta EE provides specifications that enable client developers to accomplish tasks such as building web services (JAX-RS) [32], injecting dependencies (CDI) [8], binding XML documents or JSON input into Java objects (JAXB and JSON-B) [33, 37], and persistence (JPA) [35]. While Jakarta EE has specifications to provide all the functionalities above, it is not an implementation. To build real applications, developers need to use a server runtime that supports Jakarta EE specifications such as Open Liberty [63], GlassFish [24], WildFly [89], or TomEE [6]. As powerful as Jakarta EE is, modern enterprise applications require features such as distributed tracing, resiliency, and monitoring, which Jakarta EE does not provide. MicroProfile extends Jakarta EE to provide such specifications that are necessary for the development of microservices applications.

Figure 2.2 shows MicroProfile specifications, including specifications from

```
1 @Counted(name="rate")
2 public int getCurrentRate() { return rate; }
```

Figure 2.3: An example illustrating the usage of the MicroProfile `@Counted` annotation [17].

```
1 @SimplyTimed
2 public void run() { /* some processing */ }
```

Figure 2.4: An example illustrating the usage of the MicroProfile `@SimplyTimed` annotation [75].

Jakarta EE. Each MicroProfile specification targets a specific functionality such as Configuration, Health, and Metrics. MicroProfile provides these functionalities mainly through annotations. Java annotations provide meta-information about program elements such as classes, fields, and methods they annotate [3]. For example, developers use the `@Override` annotation on a method to indicate that a class is overriding the annotated method from a parent class [64]. MicroProfile specifications contain annotations that client developers can use to perform various tasks. For example, the MicroProfile Health specification provides the `@Liveness` annotation to allow client developers to create and expose custom health checks [53]. Another example is the MicroProfile Metrics specification that provides annotations such as `@Counted` and `@SimplyTimed` [56]. The `@Counted` annotation tracks the number of invocations of the annotated methods or constructors (see Figure 2.3), whereas the `@SimplyTimed` annotation provides how long it takes for the invocations to complete (see Figure 2.4).

2.3 Annotation Usage Rules

Frameworks or libraries provide annotations for different purposes such as dependency injection, data binding, and code generation [92]. There are certain usage rules that client developers must follow when using annotations. We divide these rules into two categories: explicit and implicit usage rules. *Explicit usage rules* are those explicitly defined by the framework or library developers

```
1 @Target(ElementType.METHOD)
2 public @interface Override { }
```

Figure 2.5: The definition of the `@Override` annotation [64].

when they declare an annotation. A compiler can automatically check explicit usage rules. For example, framework developers can restrict the types of program elements an annotation can be used on (e.g., fields or methods) using the `@Target` annotation [84]. Figure 2.5 shows the definition of the `@Override` annotation. We can see that the `@Override` annotation can only be used to annotate methods. The compiler would complain if client developers use the annotation on a different program element, such as a field.

On the other hand, *implicit usage rules* are associated with the way an annotation is expected to be used in combination with other program elements including other annotations. Such implicit usage rules are not checked by the compiler. For example, to create a liveness check using MicroProfile Health, a client developer needs to (1) create a class and annotate it with `@Liveness`, and (2) the same class must implement the `HealthCheck` interface as shown in Figure 1.1. If client developers do not implement `HealthCheck`, it will cause the `@Liveness` annotation to be ignored, without showing any explicit error message. In this thesis, we are interested in generating implicit usage rules for MicroProfile annotations to prevent their misuse.

2.4 Pattern mining

Pattern mining seeks to automatically extract usage patterns or rules [71]. In our case, *a pattern* refers to a way an annotation is used in combination with other program elements, including other annotations. The main premise behind pattern mining is that a frequent usage represents a usage rule. We refer to mined rules that are not validated by experts as *candidate rules*. Most of the available pattern mining techniques to extract usage rules focus on control and data-flow relationships between different method calls and the order in which methods are invoked on objects [60, 83, 87, 88]. In contrast, annotation

usage rules do not require the annotations to be placed in a certain order. To the best of our knowledge, the previous work from our research group is the only pattern mining technique available for mining annotation usage rules [62], which we use in this thesis to mine candidate annotation usage rules for MicroProfile. We first provide background about frequent itemset mining, which is the mining technique Nuryyev *et al.* use. We then explain the details of their mining technique and the output it produces.

2.4.1 Frequent itemset mining

Frequent itemset mining is a data mining technique that is mainly used for association rule learning to discover interesting relationships between items in large databases [67]. Let $I = \{i_1, i_2, ..., i_n\}$ be a set of n items, $T \subseteq I$ be a non-empty subset of I called a *transaction*, and $D = \{T_1, T_2, ..., T_m\}$ be a set of all transactions called a *database*. An *itemset* X refers to any subset of transaction T. Each itemset has a support value $supp(X)$, which shows how often an itemset appears in the database, and is calculated with the following formula:

$$supp(X) = \frac{number\ of\ transactions\ containing\ X}{total\ number\ of\ transactions}$$

An itemset is considered a frequent itemset if the frequency of the itemset in a database is at least the same as a user-specified support value referred to as *minimum support threshold* ($supp_{min}$). For example, given I ={*milk*, *bread*, *butter*} items in a store, we may have observed the following purchases (a.k.a transactions) from five different customers: D ={{*bread*}, {*milk, bread*}, {*milk, bread*}, {*milk, bread, butter*}, {*bread, butter*}}. In this example, if we consider a purchase that occurs at least three times ($supp_{min} = 3$) as a frequent purchase, we have {*bread*} and {*milk, bread*} as frequent purchases or *frequent itemsets* with $supp(\{bread\}) = 5$ and $supp(\{milk,\ bread\}) = 3$. These frequent itemsets can be then used to generate association rules.

Association rules are relational rules of the form "If X, then Y", or more precisely $X \Longrightarrow Y$ where $X, Y \subseteq F$ and F is a frequent itemset. The "if" part is called *antecedent*, and the "then" part is called *consequent*. Each association

rule comes with a confidence value. The *confidence* of the association $X \Longrightarrow Y$ indicates the likelihood of Y being true given X is true. The confidence is calculated with the following formula:

$$conf(X \Longrightarrow Y) = \frac{supp(X \cup Y)}{supp(X)}$$

Minimum confidence threshold ($conf_{min}$) refers to a user-specified confidence value. The confidence value of an association rule needs to be at least the same as the minimum confidence threshold to be considered as a candidate rule. For the above frequent itemset $\{milk,\ bread\}$, we have $conf(milk \Longrightarrow bread) = 100\%$ and $conf(bread \Longrightarrow milk) = 60\%$. If we consider $conf_{min} = 80\%$, we will have only $milk \Longrightarrow bread$ as a candidate rule.

2.4.2 Candidate annotation usage rule mining

In this section, we explain the frequent itemset-based pattern mining approach proposed by Nuryyev *et al.* [62] to mine candidate annotation usage rules for MicroProfile. This work is the precedent to our work in this thesis and focuses on mining candidate rules, but does not automate the rule validation and misuse detection processes. The approach has three key steps: (1) extract transactions from client projects, (2) mine frequent itemsets, and (3) generate candidate rules. We now explain each step in detail.

Extract transactions from client projects. Since the semantics of using annotations differ from using method calls, Nuryyev *et al.* [62] first investigate what information or code facts should be tracked to mine annotation usage rules. They first manually search for MicroProfile annotation usage rules in the official documentation and various online forums and then extract code facts from these rules. They identify eight different code facts or relationships from their manual investigation:

- `annotatedWith` represents what annotation a program element has been annotated with. A program element can be a class, field, method, constructor, and method and constructor parameters.
- `hasType` represents the data type of a field.

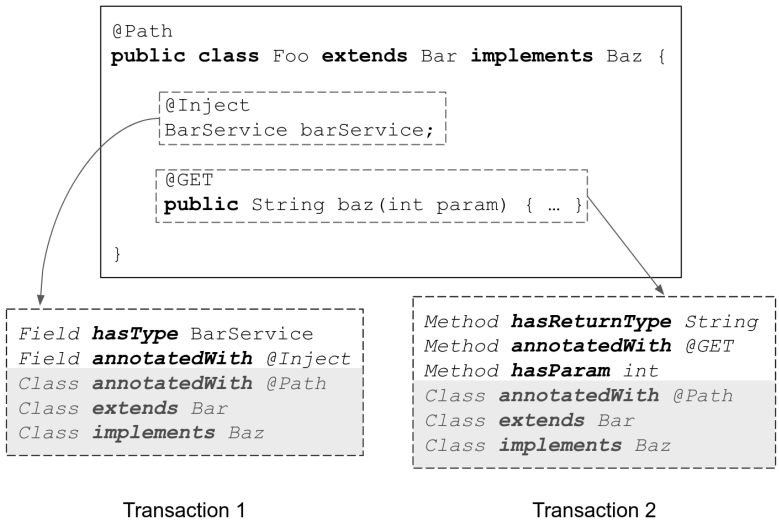

Figure 2.6: An example demonstrating what transactions are extracted from source code. As we can see from the greyed out parts, itemsets related to the class are present in both transactions

- `hasParam` represents a parameter of a method, constructor, or annotation.
- `hasReturnType` represents the return type of a method.
- `extends` represents a class extension.
- `implements` represents an interface implementation.
- `definedIn` represents an annotation parameter value that has been defined in `microprofile-config.properties`.
- `declaredInBeans` refers to the bean declaration of a class in the `beans.xml` file.

Once they establish the relationships that need to be tracked, they use JavaParser [30] to extract transactions containing such relationships or items from client projects. The granularity of a transaction is the existence of an annotation on a program element. The approach generates a separate transaction for each program element within a class (i.e. fields, methods and constructors). For example, given the source code example shown in Figure 2.6, the approach produces two transactions, one for the field and one for the method. Note that the class information is copied into all the transactions generated for all the program elements of the class to enable discovering relationships between classes and their members.

It is possible that a client project uses a certain API extensively while other client projects use the same API rarely. This can result in a skewed distribution of API usage. Therefore, to normalize the API usage distribution, the approach uses two different heuristics. First, the approach only takes a set of unique transactions from each client project. For example, if a project has three transactions `[T1, T2, T3]` where `T1` and `T2` have the same items, the approach only selects `T1` and `T3` transactions. Second, the approach groups transactions based on the annotations present in them. This results in smaller, separate lists of transactions for each annotation. If a transaction contains more than one annotation, the transaction will appear in more than one list. For example, for the transactions `{A, B, C}`, `{A}`, `{B, C}`, and `{A, C}` containing annotations `A`, `B`, and `C`, the approach would group transactions into three separate lists as follows:

A: `[{A, B, C}, {A}, {A, C}]`
B: `[{A, B, C}, {B, C}]`
C: `[{A, B, C}, {B, C}, {A, C}]`

After the approach groups the transactions by annotations, the approach mines frequent itemsets from each list separately.

Mine frequent itemsets. Nuryyev *et al.* [62] use the *FP-Growth* algorithm [94] to mine frequent itemsets. They found that the FP-Growth algorithm produces frequent itemsets in large quantities and most of these itemsets are either not useful or very similar to each other. Thus, to eliminate the redundant and not useful frequent itemsets, they perform three types of optimizations.

- *Remove redundant itemsets.* They identify and keep all the frequent itemsets without a proper superset (i.e., maximal itemsets) while filtering out the other frequent itemsets. For example, given two frequent itemsets $A = \{a, b\}$ and $B = \{a, b, c\}$, the approach filters out A because B already contains all the items of A. They only keep the maximal frequent itemsets to not lose any extracted information while reducing the number

```
{
    "antecedent": [
        "Class annotatedWith @ApplicationScoped",
        "Class annotatedWith @Readiness"
    ],
    "consequent": [
        "Class implements HealthCheck"
    ]
}
```

Figure 2.7: A candidate rule mined by Nuryyev *et al.* [62]

of redundant frequent itemsets.

- *Remove itemsets with no target API usage.* Client projects generally use APIs from various frameworks and libraries. Since the authors were interested in mining usage rules only for the target framework that is MicroProfile, they remove all the frequent itemsets that do not contain at least one MicroProfile API.
- *Remove semantically incorrect itemsets.* The generated frequent itemsets can contain any combination of the identified relationships. However, a combination of the relationships that form a semantically incorrect frequent itemset will generate meaningless usage rules. Therefore, the authors identify and remove semantically incorrect itemsets. The authors identify two cases where a combination of relationships would form a semantically incorrect itemset. The first one is having a ‘‘`@A hasParam B`’’ without having a ‘‘`...annotatedWith @A`’’. Annotation parameters require the presence of the annotation, having the parameter without the annotation is syntactically impossible. The second one is having a ‘‘`Param_value definedIn Configuration`’’ without having ‘‘`...hasParam Param_value`’’. An annotation parameter's value cannot be referenced in the configuration file if the parameter itself is not present.

Generate candidate rules. From the remaining frequent itemsets, the authors generate association rules, which they refer as *candidate rules* because they have not been validated yet. These rules consist of an antecedent and

a consequent (also known as "if-then" rules). Figure 2.7 shows an example candidate rule that this approach discovers [62]. Overall, the authors mine 23 candidate rules from 533 MicroProfile projects. Listing 2.1 shows all the mined candidate rules. Chapter 4 discusses how we use these candidate rules in our work.

Listing 2.1: The candidate rules mined by Nuryyev *et al.* [62] in the original JSON format. We remove fully-qualified names for better readability

```
[{
    "id": 1,
    "antecedent": [
        "@APIResponse hasParam Param_responseCode:String",
        "Method annotatedWith @APIResponse",
        "Class annotatedWith @Path",
        "@APIResponse hasParam Param_description:String",
        "Method annotatedWith @Operation"
    ],
    "consequent": ["@Path hasParam Param_value:String"],
},{
    "id": 2,
    "antecedent": [
        "Method annotatedWith @SimplyTimed",
        "Method hasParam Param_String"
    ],
    "consequent": ["@SimplyTimed hasParam Param_name:String"]
},{
    "id": 3,
    "antecedent": [
        "Class annotatedWith @Entity",
        "Class annotatedWith @Type",
        "@Schema hasParam Param_name:String",
        "@Schema hasParam Param_title:String",
        "Class annotatedWith @Schema"
    ],
    "consequent": ["@Entity hasParam Param_name:String"]
},{
    "id": 4,
    "antecedent": [
        "Class annotatedWith @ApplicationPath",
        "@LoginConfig hasParam Param_authMethod:String",
        "@Path hasParam Param_value:String",
        "Class annotatedWith @Path",
        "@ApplicationPath hasParam Param_value:String",
        "@LoginConfig hasParam Param_realmName:String",
        "Class annotatedWith @LoginConfig",
        "Class annotatedWith @RequestScoped"
```

```
    ],
    "consequent": ["Class extends Class_Application"]
},{
    "id": 5,
    "antecedent": [
        "@Path hasParam Param_value:String",
        "Method annotatedWith @APIResponses",
        "Method annotatedWith @Operation"
    ],
    "consequent": ["Class annotatedWith @Path"]
},{
    "id": 6,
    "antecedent": ["Method annotatedWith @Outgoing"],
    "consequent": ["Class annotatedWith @ApplicationScoped"]
},{
    "id": 7,
    "antecedent": ["Method annotatedWith @Mutation"],
    "consequent": ["Class annotatedWith @GraphQLApi"]
},{
    "id": 8,
    "antecedent": [
        "@Gauge hasParam Param_unit:String",
        "Method annotatedWith @Gauge"
    ],
    "consequent": ["@Gauge hasParam Param_name:String"]
},{
    "id": 9,
    "antecedent": [
        "Field annotatedWith @RegistryType",
        "@RegistryType hasParam Param_type:Type",
        "Field annotatedWith @Inject"
    ],
    "consequent": ["Field hasType MetricRegistry"]
},{
    "id": 10,
    "antecedent": [
        "Class annotatedWith @ApplicationScoped",
        "Class annotatedWith @Health"
    ],
    "consequent": ["Class implements Interface_HealthCheck"]
},{
    "id": 11,
    "antecedent": [
        "Field annotatedWith @ConfigProperty",
        "Field annotatedWith @Inject"
    ],
    "consequent": ["@ConfigProperty hasParam Param_name:String"]
```

```
},{
    "id": 12,
    "antecedent": ["Method annotatedWith @Query"],
    "consequent": ["Class annotatedWith @GraphQLApi"]
},{
    "id": 13,
    "antecedent": [
        "@APIResponse hasParam Param_responseCode:String",
        "Method annotatedWith @APIResponse",
        "@Path hasParam Param_value:String",
        "Class annotatedWith @Path",
        "Method annotatedWith @Operation"
    ],
    "consequent": ["Method hasReturnType Response"]
},{
    "id": 14,
    "antecedent": [
        "Class annotatedWith @RegisterRestClient",
        "@Path hasParam Param_value:String",
        "Method annotatedWith @Path"
    ],
    "consequent": ["Class annotatedWith @RegisterClientHeaders"]
},{
    "id": 15,
    "antecedent": [
        "Class annotatedWith @ApplicationScoped",
        "Class annotatedWith @Readiness"
    ],
    "consequent": ["Class implements Interface_HealthCheck"]
},{
    "id": 16,
    "antecedent": [
        "Class annotatedWith @Path",
        "Class annotatedWith @OpenAPIDefinition",
        "@OpenAPIDefinition hasParam Param_info:"
    ],
    "consequent": ["@Path hasParam Param_value:String"]
},{
    "id": 17,
    "antecedent": ["Class annotatedWith @Liveness"],
    "consequent": ["Class implements Interface_HealthCheck"]
},{
    "id": 18,
    "antecedent": [
        "Class annotatedWith @Tag",
        "Class annotatedWith @Path",
        "@Tag hasParam Param_name:String"
```

```
    ],
    "consequent": ["@Path hasParam Param_value:String"]
},{
    "id": 19,
    "antecedent": ["Method annotatedWith @Incoming"],
    "consequent": ["Class annotatedWith @ApplicationScoped"]
},{
    "id": 20,
    "antecedent": ["Field annotatedWith @Inject"],
    "consequent": ["Field annotatedWith @RestClient"]
},{
    "id": 21,
    "antecedent": [
        "Class annotatedWith @RegisterRestClient",
        "Class annotatedWith @RegisterProvider"
    ],
    "consequent": ["@RegisterProvider hasParam Param_value:Class"
        ]
},{
    "id": 22,
    "antecedent": ["Field annotatedWith @Inject"],
    "consequent": ["Field annotatedWith @Metric"]
},{
    "id": 23,
    "antecedent": [
        "Field annotatedWith @Inject",
        "Field annotatedWith @Claim"
    ],
    "consequent": ["Class annotatedWith @Path"]
}]
```

Chapter 3

Related Work

In this thesis, we propose a complete pipeline for generating API usage rules that notably includes a human in the loop. Our approach combines two previously known techniques, namely, pattern mining and manual rule authoring. In this chapter, we perform a literature review categorized by these two techniques.

3.1 Mining API Usage Rules

Researchers have proposed various pattern mining techniques to automatically extract API usage rules [40, 60, 71, 83, 87, 88, 96]. JADET uses frequent itemset mining to produce *two-letter patterns* in the form of $a \prec b$, meaning that method `a` precedes method `b` [88]. Tikanga [87] mines temporal patterns on method parameters in the form of Computation Tree Logic (CTL) formulas [13]. These formulas describe what conditions a particular object should satisfy before being passed to a method.

There are graph-based solutions to mine API usage patterns such as GrouMiner and MuDetect [60, 83]. MuDetect builds upon the GrouMiner and introduces a graph representation called *API Usage Graph* that can capture more details about the method body (such as exceptions and synchronizations) [83]. MuDetect uses an apriori-based frequent subgraph mining algorithm to mine frequent patterns. A pattern is a subgraph that occurs more than a predefined minimum threshold. Any deviation from a pattern is considered a misuse. However, Amann *et al.* [2] mention that such an assumption is naive

and results in many false positives, which is the reason why we employ human expert validation to validate the mined rules before they are used for misuse detection. Kang and Lo [39] introduce Actively Learned Patterns (ALP) that incorporates active learning to identify discriminative subgraphs. ALP presents human annotators with some code examples for an API, and asks them to label them either as "correct" or "misuse". A *discriminative subgraph* is a subgraph that appears more frequently in one label than the other. Once the discriminative subgraphs are obtained, a machine learning classifier is trained to determine if a certain API usage is a misuse. While this method outperforms MuDetect in the MuBench benchmark [1], it would be impossible to produce rules based on the received output. It is impossible because ALP takes the source code as input and checks whether it contains a misuse without explaining the reason behind why a particular misuse, if any, is found. In other words, there are no explicit/separate rules mined in the process.

The aforementioned approaches, whether implicitly or explicitly, mine API usage rules from client code. There are solutions that combine both client code and the library source code to produce more accurate usage rules [72, 93]. Zeng *et al.* [93] propose an approach where they mine constraints from both client and library code. These constraints are represented as AUGs. The authors use MuDetect for extracting constraints from the client code. Their library constraint extraction algorithm can find constraints such as what parameters cannot be null, what conditions trigger exceptions, or in what order methods are invoked. The concept of combining constraints from both source and client code certainly adds more context to the mined rules involving library method calls. However, when it comes to mining annotation usage rules, it would be ineffective since annotation definitions do not contain any business logic. Most of the existing constraints such as the target program elements, data types of annotation parameters are checked by the compiler.

Most of the pattern mining solutions focus on analyzing method bodies and the order of the function calls on a specific type [60, 83, 87, 88] with no focus on annotation usage. In general, annotation usage rules are simplistic compared to other types of API usage rules, since an annotation is either

applied to a program element or not.

To the best of our knowledge, with the exception of the work done by Nuryyev *et al.* [62] , which we directly use to mine candidate rules in this thesis, there is no existing work on mining annotation usage rules. However, Liu *et al.* [46] build a deep learning solution, DeepAnna, that can recommend annotations and detect annotation misuses. The authors approach the annotation recommendation and misuse detection as a multi-label classification task. To train the classifier, the authors first extract code snippets and their annotations from open-source Java projects. A code snippet in this context refers to a class or a method declaration. DeepAnna then extracts the structural and textual contexts from the code snippets. For the structural context, DeepAnna extracts the abstract syntax tree (AST) of a code snippet and removes all the annotations from the AST. For the textual context, DeepAnna extracts identifiers in the AST and produces a token sequence. DeepAnna then uses the structural and textual contexts to predict and recommend annotation usage. In the context of annotation usage rule extraction, one main disadvantage of DeepAnna is that it cannot specify reasons behind its recommendations. For example, DeepAnna might suggest to use the `@Liveness` annotation on a class (consequent), but it cannot specify the reason behind that suggestion (antecedent). We need to have both an antecedent and a consequent to create an annotation usage rule that is informative during misuse detection. Moreover, DeepAnna recommends only class and method-level annotations, while annotations can also be used on other program elements such as fields and parameters.

While pattern mining reduces the human effort for finding API usage rules, it can produce rules that are insecure, overly simplistic, or missing context [43]. Pattern mining techniques may also mine usage rules for deprecated APIs, which are not useful. Thus, it is critical to validate or correct the mined rules to transform them into valid API usage rules.

3.2 Formats for Encoding API Usage Rules

In the previous section, we discussed various pattern mining approaches and the approach we will be employing for finding annotation usage rules. The mining approach we use (described in Section 2.4.2) produces candidate rules in the IF/THEN form. In this section, we discuss various API usage rule encoding formats that would allow us to encode the mined candidate rules for a human expert to validate. Our goal is to present these rules in a format (1) that is intuitive and easy-to-learn, and (2) that can support most of the mined relationships.

There is an abundance of rule authoring tools to write API usage rules [18, 43, 48, 95]. They typically use an internal or external domain-specific language (DSL) [22]. As opposed to general-purpose languages, *domain-specific languages* focus on a particular domain. An internal DSL is written in a general-purpose language (such as Java), while an external DSL is a separate language. Often, these rule authoring tools target specific libraries or domains. For example, CrySL is designed for writing usage rules of Java cryptography APIs [43]. It focuses on control and data-flow relationships between different method calls. CrySL rules are processed by CogniCrypt [43] (a static analysis tool that detects crypto misuses). However, CrySL does not support annotations.

There are also tools that support writing usage rules for annotations. AnnaBot [18] and RSL [95] come equipped with external DSLs and misuse checkers that utilize the rules written in those DSLs. In AnnaBot, users can write rules (or claims) in an IF/THEN format, similar to the rules produced by our mining process. In RSL, users write rules in a declarative way using `for` and `where` statements to only keep the target program elements, and `assert` statement to apply constraints. These tools support logical and aggregate operations such as "AND", "OR", "NOT" and "at most one". Additionally, RSL provides code inspection methods such as `callExists`, `pathExists`, and `getAttr`. However, they do not support most of the relationships that our mining process produces. For example, AnnaBot does not support writing rules

that specify a relationship between an annotation and a method return type. RSL, on the other hand, only allows writing rules that check the existence of an annotation on a program element.

Similar to previous tools, RulePad [48] allows software developers to create design rules and checks for misuses of these rules. An example for a design rule can be the following: *if method name starts with "set", it must return "void"*. To create such design rules, RulePad offers two rule authoring modes: snippet-based and using its semi-natural DSL. Figure 3.1 shows the graphical user interface (GUI) of RulePad. For snippet-based authoring, RulePad provides a GUI where users only need to fill in the appropriate fields to author a rule ①. This step generates a rule written in RulePad's semi-natural DSL ②. The idea is that users can create rules using the GUI and modify the produced rule using the semi-natural DSL if needed. Users can also write rules completely in the semi-natural DSL without using the GUI. Unlike previous tools we have mentioned so far, RulePad focuses on creating checkable and up-to-date documentation. Since the documentation is meant to be read by developers, to make the design rules easy to understand, RulePad provides an English-like DSL to encode rules in. For example, the above design rule can be expressed in RulePad's DSL as follows: `function with name "set..." must have type "void"`. RulePad's DSL has an IF/THEN structure which coincides with our mined rules. It also supports most of the mined relationships. Therefore, we use RulePad's DSL in this work with some customization to validate annotation usage rules.

Unlike focusing on the usage of APIs from specific libraries or domains, there are tools, such as PMD [68], SpotBugs [77] and CheckStyle [11], that focus on general-purpose static analysis. These tools can be utilized to encode annotation usage rules. For writing custom rules in these tools, developers need to use either a general-purpose language such as Java or a querying language such as XPath (in the case of PMD)[91]. When using Java, usually developers have to create custom visitors by extending visitor classes provided by the tool. In the implemented visitor methods, certain types of nodes of the abstract syntax tree (AST) can be searched. The main drawback of this

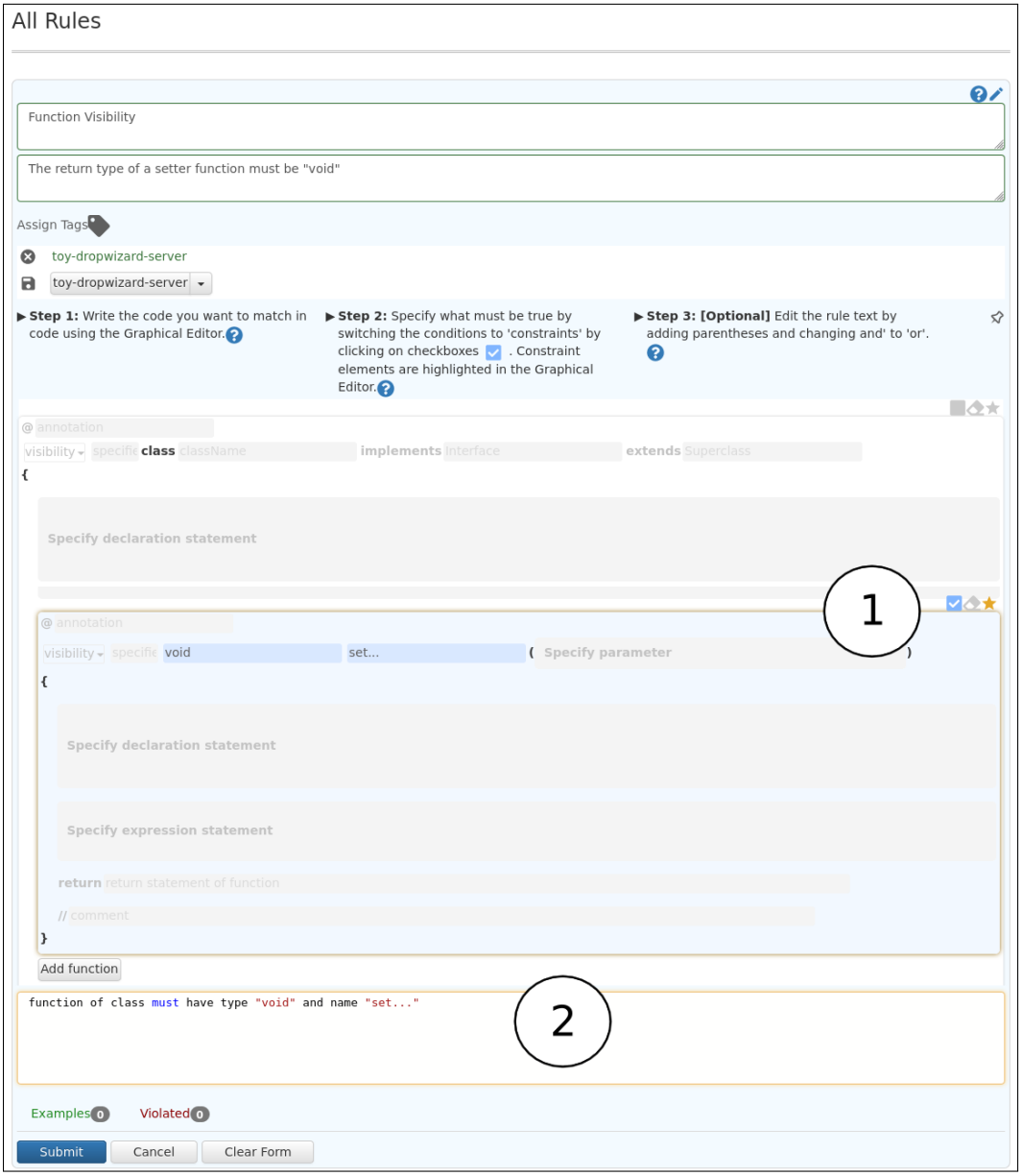

Figure 3.1: The original GUI of RulePad. Note: we removed blank sections from the screenshot for better visibility [48]

approach is that rather than declaring the rule they want to impose (declarative), developers now have to implement what kind of bugs they are trying to catch (imperative). The imperative nature of visitors may cause API experts to spend too much time trying to understand the rule, rather than validating it. Compared to implementing visitors, PMD's XPath queries provide a declarative way of creating usage rules in PMD. However, for sufficiently large rules, writing and reading XPath becomes cumbersome. Prior work shows that DSLs are easier to learn, read, and write than general-purpose languages [42]. Since we want to make rule validation as intuitive as possible, we did not choose a general-purpose static analysis tool for rule encoding.

The previous tools all provide external DSLs to encode the rules. There has also been *meta-annotation* solutions that encode rules in the source code directly [41, 61, 76]. These solutions provide a set of meta-annotations (i.e. an annotation that can be applied to other annotations) to embed the usage rules in annotation source code. *Smart Annotations* by Kellens *et al.* [41] supports two types of constraints using the `@Necessary` and `@Sufficient` meta-annotations. `@Necessary` is used to specify that if a program element is annotated with an annotation, then it must obey the rules defined for the annotation. `@Sufficient` is used to specify that if a program element obeys all the rules defined for an annotation, then it needs to be annotated with the annotation. Similarly, Noguera and Duchien [61] provide annotations such as `@Association` and `@Associations` to specify relationships between different annotations. *Esfinge*$_{METADATA}$ by Siqueira *et al.* [76] also provides a set of meta-annotations that can be used to encode annotation usage rules. While these techniques are powerful, they require source code changes which would force us to modify the library/framework source code every time a new rule is required, thus these solutions are not useful for our use case.

Apart from DSLs, there are logic programming languages that can be used to encode the mined rules [10, 14, 90]. One such programming language is Datalog [10]. A Datalog program is a set of facts and rules. A *fact* is something that is always true. Given a Java class `Foo`, a fact could be that the class is being annotated with `@X` or that the class extends another class `Z`. In

```
class with annotation "X" must have
annotation "Y" or extension of "Z"
```

Listing 3.1: RulePad version

```
antecedent(Class) :- has_annotation(Class, "X").
consequent(Class) :- has_annotation(Class, "Y").
consequent(Class) :- class_extends(Class, "Z").
rule(Class) :- antecedent(Class), consequent(Class).
```

Listing 3.2: Datalog version

Figure 3.2: A comparison of a rule written in both RulePad and Datalog

Datalog, we could represent these facts as follows: `has_annotation("Foo", "X")`, `class_extends("Foo", "Z")`. Datalog *rules* define how new facts can be derived from existing facts. A rule in Datalog is written in the form of `consequent(...) :- antecedent(...)`, which means that the consequent is true if the antecedent holds. This IF/THEN structure of Datalog rules fits nicely with our candidate rules. However, compared to RulePad's English-like syntax, we find Datalog to be less-intuitive. For example, Figure 3.2 shows the same rule written using Datalog and RulePad. For a very simple rule, we can see that RulePad version is much shorter and concise.

All the above tools assume that the rules are readily available. However, someone needs to find the rules and encode them from scratch. For example, Krüger *et al.* [43] went through all the Java Cryptography API documentation, and manually authored all the found rules in CrySL. There are two issues with this approach: (1) it is time-consuming, and (2) authors will miss undocumented rules. Our hybrid approach leverages pattern mining to reduce the time spent on authoring rules from scratch. It can also discover undocumented patterns.

Chapter 4

Rule Validation Tool

In this chapter, we describe our approach to generate annotation usage rules for MicroProfile APIs. Figure 1.2 illustrates the overview of our approach. We first use the existing pattern-mining approach that we described in Section 2.4) to mine candidate usage rules from MicroProfile client projects. We then present the mined candidate rules to experts for validation. To facilitate rule validation, we develop a web-based Rule Validation Tool (RVT). Rule validation is a twofold process. Given mined candidate rules, RVT automatically encodes them in the RulePad format [48]. RVT then presents the encoded rules to experts for validation who can confirm a rule as is, confirm a rule after modifications, or reject a rule as invalid. Finally, we develop a misuse detector that uses the validated rules to find misuses (Chapter 5). In this section, we describe the rule validation process and RVT.

4.1 RVT DSL

4.1.1 Rule Encoding

The mined candidate annotation usage rules are in a JSON [36] format, as shown in Figure 2.7 and in Listing 2.1. Since experts will read, modify, and validate the candidate rules, we need to present them in a format that is easy to comprehend. Overall, we do not want experts to spend too much time trying to learn and understand the rule format, because it defeats the purpose of reducing human effort in generating rules.

To present the mined candidate rules in a specific format, we considered

multiple existing domain-specific languages (DSLs) developed for encoding various types of API usage rules. Specifically, we considered AnnaBot [18], RSL [95], CrySL [43], RulePad [48], ModelAn [61], and Smart Annotations [41]. We also considered the original JSON format of the mined candidate rules. We discuss these options with our industry partner, IBM, to understand how they perceive their pros and cons.

Among the DSLs, we exclude ModelAn [61] and Smart Annotations [41], because they require modifications to the MicroProfile source code. CrySL [43], on the other hand, is designed for the specification of correct cryptography API uses in Java, and it heavily focuses on control and data-flow relationships, whereas annotation usage does not require control and data-flow relationships. Therefore, we also exclude CrySL from potential formats for presenting candidate rules. AnnaBot [18] and RSL [95] are specifically designed for writing annotation usage rules in a declarative way. However, they only support annotation usage rules between two annotations. For example, if `@X`, then `@Y`. They do not support usage rules between annotation and other program elements (e.g., field and method). For example, if `@X`, then a method must return `Z`. However, most of our mined candidate rules specify relationships between annotation and other program elements [62]. Therefore, we also exclude AnnaBot and RSL from potential formats for presenting candidate rules. In the end, we narrow down our potential formats to the following three options: (1) use the original JSON format, (2) create a DSL from scratch specifically designed for our needs, and (3) use an existing DSL that we have not ruled out yet.

The original JSON format (option 1) is concise and simple to understand. However, the structure of the mined candidate rules, specifically items in both the antecedent and consequent, is ad-hoc at best. More importantly, the JSON rule format does not have a grammar, which means any post-processing of candidate rules will involve lots of string manipulations and regular expressions.

Creating a DSL from scratch (option 2) gives us the advantage of customizing it according to our requirements. However, creating a DSL from scratch is time-consuming, when compared to using an existing DSL such as RulePad.

```
class with annotation "ApplicationScoped" and
           annotation "Readiness"
must have
     implementation of "HealthCheck"
```

Figure 4.1: An example illustrating the RulePad rule for the mined candidate rule shown in Figure 2.7.

RulePad [48] is a tool that allows users to write code design rules (e.g., `function with name "set..." must have type "void"`) and check the source code for any violations of those rules. Developers can create rules using a semi-natural DSL created by the RulePad authors. For example, Figure 4.1 shows how the mined rule from Figure 2.7 can be expressed in RulePad's DSL. Based on the feedback we received from our industry collaborators, we decide to use RulePad (option 3), because: (1) it has an intuitive English-like syntax, (2) rules have IF/THEN format that fits nicely with the mined rules, and (3) it has a grammar, making later extensions easier to implement. However, there were some RulePad shortcomings that we needed to address. We next discuss the extensions and changes we have made to the original RulePad DSL.

4.1.2 Our RulePad Extensions

In the original RulePad, there are three shortcomings we need to address. First, RulePad does not support writing rules for the following relationships that appear in MicroProfile candidate rules: (1) method/constructor parameters having annotations, (2) annotations having parameters, and (3) configuration files. Therefore, we extend RulePad's grammar to support those relationships. For specifying configuration files, we design the syntax to be suitable for properties files, specifically. A *properties file* is a plain text file where configuration properties are stored as key-value pairs, separated by an equal sign ("="). The following is an example configuration property: `‘‘password=some-value’’`. In the extended RulePad, the same property can be expressed as `configuration file with property with name "password"`.

Second, some RulePad keywords, specifically `declaration statement`, `function`, and `type` (to represent the return type of a method) do not reflect Java language terminology. This might affect the readability or comprehensibility of rules. Therefore, we change the keywords to make them better align with Java language terminology. We change `declaration statement` to `field`, `function` to `method`, and `type` to `return type`. Please note that, there are other RulePad constructs (such as `field` and `parameter`) that refer to the `type` keyword, as well. We only make the mentioned change for representing the return type of a method.

Third, our industry collaborators perceived some of RulePad's syntax as too verbose. To address this, we introduce shortcuts into the DSL. We create a shortcut to express method, constructor or annotation parameters, and configuration properties. For example, a `String` parameter with the name `foo` is expressed as `parameter with type "String" and name "foo"` in RulePad's original DSL. We shorten it to `parameter "String foo"`, mirroring Java-style parameter declaration. In extended RulePad DSL, it is also possible to provide a single string token for a parameter such as `parameter "foo"`. This expression is interpreted differently for annotation parameters, and method parameters. In the case of annotation parameters, the extended RulePad rule parser treats the expression as a parameter with the name "foo", while for method parameters it treats the expression as a parameter with the type "foo". The reason behind this distinction is that for annotation parameters, a rule can impose which parameter should be used by referring to the parameter's name. On the other hand, for methods, a rule cannot impose a certain name for a parameter since a developer is allowed to choose any name. Additionally, we add the capability of specifying constant values for annotation parameters (e.g., `parameter "foo" with value "bar"`). All the shortcuts introduced for annotation parameters in extended RulePad are also available for configuration properties (e.g., `property "enabled" with value "true"`).

We also create a shortcut that allows grouping of annotations from the same package. For example, to require one of JAX-RS HTTP method annotations (GET, POST, PUT, DELETE) [31], the corresponding RulePad expression is

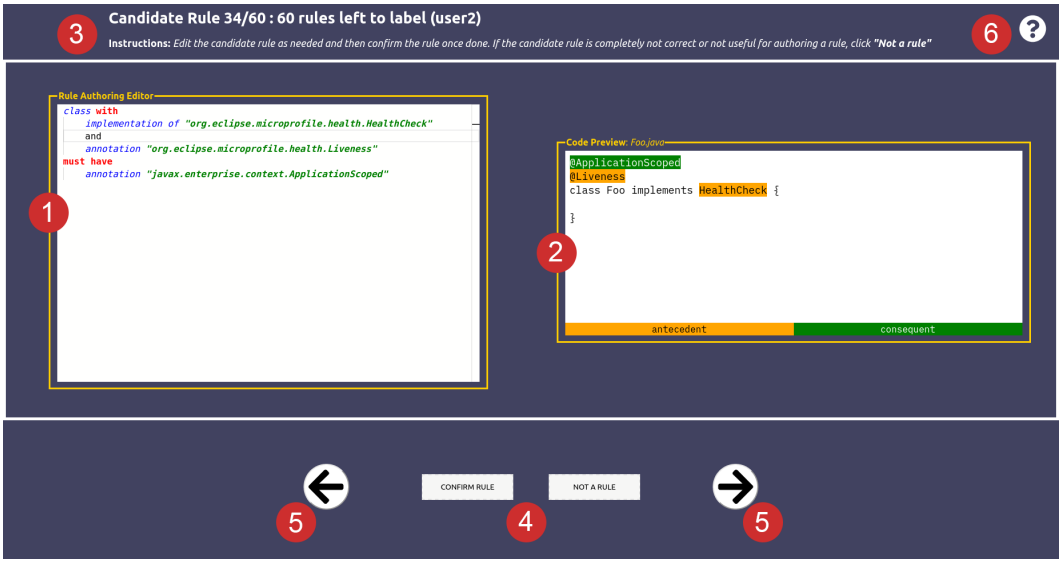

Figure 4.2: The main GUI elements of our Rule Validation Tool (RVT). Features 1–6 are detailed in Section 4.1.

`annotation "javax.ws.rs.GET" or annotation "javax.ws.rs.POST" or annotation "javax.ws.rs.PUT" or annotation "javax.ws.rs.DELETE"`. We condense that expression into `annotation "javax.ws.rs.[GET|POST|PUT|DELETE]"`.

RVT takes the mined candidate rules in JSON format and converts them into RulePad rules, which we present to experts for validation through a Graphical User Interface (GUI) described next.

4.2 User Interaction with RVT

The main purpose of RVT is to make the rule validation process intuitive and straightforward for API experts. In this section, we describe the rule validation workflow, and how we have structured the RVT GUI to facilitate this process.

4.2.1 RVT GUI

RVT provides a GUI for experts to go through and validate the presented candidate rules. Figure 4.2 shows the main elements of RVT's GUI. The *Rule Authoring Editor* ① presents the candidate rule, encoded in RulePad format, that needs to be validated. To improve the readability of the presented rules, we equip the editor with syntax highlighting and formatting features. The

Code Preview ② provides a minimal Java code example to show what the presented candidate rule corresponds to in actual Java code. That preview also highlights the code representing the antecedent (orange) and the consequent (green), enabling visual separation between the two parts of a candidate rule. The goal of the Code Preview is helping experts further understand the candidate rule presented in the Rule Authoring Editor. The *progress indicator* ③ shows the total number of candidate rules that need to be validated, the position of the currently presented rule among all the candidate rules, and the number of candidate rules left to validate. RVT has two *labeling buttons* ④ ("Confirm rule" and "Not a rule") to label the presented candidate rule as a correct or incorrect rule. RVT also has two *rule navigator* buttons ⑤ to navigate through candidate rules. Finally, the question mark ⑥ indicates *help* and opens a tutorial page on using RVT. The tutorial page contains information about the RulePad grammar and all possible actions that experts can perform using RVT. Overall, RVT's GUI not only presents the encoded rules but also provides the necessary features to help experts understand the presented rules.

4.2.2 Rule Validation Process

Once RVT presents a candidate rule, experts can take one of the following three actions to validate and label the rule:

- *Confirm the rule as is*: If the presented candidate rule is correct, experts can label the rule as correct by clicking "Confirm rule".
- *Confirm the rule with changes*: If the presented candidate rule is a partially correct rule, e.g., the rule has some missing or extra items, experts can edit the presented rule by adding, removing, or modifying items of the rule using the Rule Authoring Editor. After finishing their editing, they can confirm the edited rule by clicking "Confirm rule".
- *Label the rule as incorrect*: If the presented candidate rule does not represent any annotation usage rule, experts can discard this rule by clicking "Not a rule".

For each presented candidate rule, RVT stores the validated form of the rule and the label in a database. Once experts validate all candidate rules, we use only confirmed rules for misuse detection.

4.3 RVT: Implementation Details

RVT is a web-based application. We choose the web platform because it does not require users to install additional tools/libraries to get started, as opposed to other platforms such as desktop. RVT consists of two main components: a frontend (or GUI) and a backend. For the frontend, we build a React [70] application that communicates with the backend to retrieve validation related data (such as candidate rules). Figure 4.2 shows the main components of the frontend. The frontend provides the code highlighting, formatting and code preview generation. For the Rule Authoring Editor, we use the highly extensible Monaco code editor [57]. We use ANTLR4 [4] to process the RulePad grammar for providing rule formatting and visualization (code preview generation). For the visualization, we parse the RulePad rule and create an object that represents the Java class which the rule describes. We then generate the corresponding Java code from that object. For the formatting, we again parse the RulePad rule, however, we generate a string that is the same as the original rule, but with the added tab and newline characters to make it more readable. RVT backend is a FastAPI [21] application written in Python. The backend is responsible for transforming the candidate rules into the extended RulePad format, persisting the changes that API experts make to the database, and exporting the confirmed rules. We use MySQL [58] as our database vendor.

RVT provides a way to upload mined candidate rules for validation through the frontend. These rules are in JSON format (shown in Figure 2.7). After uploading the target JSON file that contains the mined rules, the backend converts each mined rule into an extended RulePad rule. Once this process is done, the API experts can start validating the mined candidate rules.

Chapter 5

Misuse Detection

Since our industry partner is interested in ensuring the correct usage of MicroProfile annotations by client developers, we next focus on developing a misuse detector that uses the correct rules from the previous step to detect annotation misuses.

5.1 Misuse Detector Tool

We first consider using RulePad for misuse detection as it comes with a misuse detector out of the box. However, we find that RulePad's misuse detector is not suitable for our needs. First, RulePad's misuse detector works in a browser; however, we are interested in using the detector as a standalone tool. RulePad has an IDE plugin that sends client code to the web UI for misuse detection. To use RulePad's misuse detector, both the web UI and the IDE plugin need to run simultaneously. Instead, we reimplement RulePad's grammar parsing and misuse detection logic as Python scripts to enable running the misuse detector independently of a browser and the IDE. Second, RulePad's misuse detector uses srcML [79] to transform Java code into XML representations. However, the generated XML output does not contain resolved types. We could not use srcType [80], a srcML-based type resolution system, to resolve types as it does not support Java. To address this issue, we create a preprocessor using JavaParser [30] that resolves all the types used in MicroProfile APIs before generating XML files. Third, the XML representation does not represent some of the Java code constructs such as the data type of annotation

parameters. We could have continued resolving issues with the XML representation by adding more preprocessing layers. However, the preprocessing step and misuse detection involved the generation of lots of intermediate text files and performing text manipulations which would affect the performance negatively. Considering all these issues, we decided to build our own misuse detector as follows.

A misuse detector may be implemented as a build tool (Maven [5] or Gradle [25]) plugin or an IDE plugin. Client developers can use a build plugin as a part of their continuous integration (CI) pipeline. However, locally, client developers have to explicitly run a build plugin to detect misuses. Unlike build plugins, an IDE plugin would detect misuses instantly as developers are working on their code. However, an IDE plugin cannot be used in the CI pipeline. After a discussion with our industry collaborator, we decided to build our misuse detector as a Maven plugin to enable CI integration. Even though we build the detector as a Maven plugin, we separate the misuse detection logic into its separate module. The Maven plugin simply provides the files and the rules to check for misuses, and reports the found misuses. This way, in the future, creating detectors for IDEs or other build plugins will be relatively easy.

Figure 5.1 shows a high-level overview of our misuse detection process. Our misuse detector uses the confirmed rules to generate static analysis checks. The grammar of RulePad's DSL is in ANTLR4 [4] format which allows us to easily generate parsers and visitors for converting the rules from text format into static analysis checks (which are simply Plain Old Java Objects). We use JavaParser [30] to parse the Java files of the target project and to resolve types. The detector scans for the misuses of all available rules in the parsed and type-resolved file. Depending on the checked rule, an analyzer is selected and the processing is delegated to that analyzer. Since the mining process produces rules concerning classes, fields and methods, we create an analyzer for each element. In each analyzer, we automatically extract the antecedent part of a RulePad rule and search for occurrences of it in the Abstract Syntax Tree (AST) of a given parsed Java file. If we find an occurrence of the antecedent, we then check if the consequent holds. If not, then we have detected a misuse.

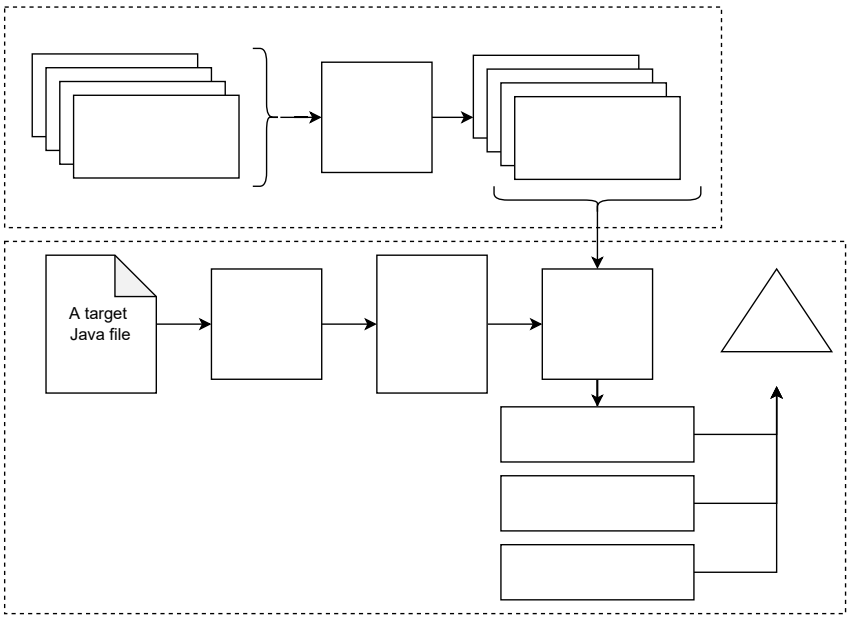

Figure 5.1: An overview of our Misuse Detector

```
$ mvn violation-detector:scan
[WARN] For rule: QueryGraphQLApiRule
[WARN]    class with function with annotation "Query" \
[WARN]          must have annotation "GraphQLApi"
[WARN] -----------------------------------------------
[WARN] In: .../membership/graphql/MembershipGraphQLApi.java
[WARN] 1. at: (22, 1) =>
[WARN]     Class MembershipGraphQLApi is missing @GraphQLApi
```

Figure 5.2: A sample report generated by our misuse detector.

The detector scans one Java file from the target project at a time for any misuse of the available rules. Once the detector scans all the Java files of the target project, the Maven plugin collects the found misuses, and prints a detailed report for each misuse. The report contains the misused rule, the misuse location, and the missing element. Figure 5.2 shows an example of a report generated by our Maven plugin. The English-like structure of RulePad's DSL allows us to use the rules themselves as error descriptions.

5.2 Misuse Detector Evaluation

Since we automatically create the static analysis checks, we want to make sure that our generation process is accurate and the generated checks correctly

detect misuses. In Nuryyev *et al.*'s work [62] on mining annotation usage rules (and the work we use in this thesis), they mined 23 candidate rules from 533 MicroProfile projects. They manually validated these 23 rules, producing 12 confirmed rules. To measure the usefulness of those rules, they carried out an experiment where they manually create checkers for 12 confirmed rules and searched for misuses of those rules in the same 533 projects. As a result, they found 100 misuses of five different rules. To verify that our generation of static analysis checks that our misuse detector uses works as expected, we evaluate it against the same set of projects, given the same 12 rules. The goal is to see whether the static analysis checks generated by our misuse detector find the same misuses as the manually-defined misuse checkers.

5.2.1 Evaluation Setup

First, we collect all the target repositories (both open-source and IBM proprietary) from Nuryyev *et al.*'s experiment which totals to 533 projects. However, since our detector is a Maven plugin, we remove all the non-Maven projects and end up with 517 MicroProfile projects. It is worth noting that Nuryyev *et al.* did not find any misuses from the projects we exclude.

Next, we encode the aforementioned 12 rules in the extended RulePad format (see Table 5.1). However, unlike Nuryyev *et al.*'s work, the encoding process involves no manual work. We simply import the mined rules (which are in the JSON format) into RVT, confirm all the rules as "correct" and export the final list of RulePad rules from RVT. We then import this list into our misuse detector.

5.2.2 Results

For each project, we scan the entire commit history to find misuses, which results in the scanning of 52,369 individual commits. We now present our findings.

In total, our detector finds 215 unique misuses of eight unique rules across all 517 projects (see Table 5.2 for the complete breakdown). After cross-checking our results with the results from Nuryyev *et al.*'s work, we find out

that our detector finds all 100 misuses that the Nuryyev *et al.* found. As stated earlier, the main difference between the two approaches is that our tool generates static analysis checks automatically while in the previous study, the authors encoded these checks manually. From these results, we can conclude that the recall of our detector is 100%, since it finds all known misuses.

To measure the precision of the detector, we analyze the rest of the found misuses. We find that the remaining 115 of the 215 misuses are previously unknown misuses. Interestingly, we notice that 89/115 are false positives. However, this phenomenon is not because of a flaw in our detector, but in the rules that we are using for finding misuses. There are three rules out of eight (ConfigPropertyInject, RegisterRestClientPath, RestClientInject from Table 5.1) that involve the annotations `@javax.inject.Inject` and `@javax.ws.rs.Path`. When observing the misuse location flagged by the detector, we notice that the required annotations are present in the code, but from a different package, namely `jakarta.inject` and `jakarta.ws.rs`. This difference happens because after Java EE 8, the specification was renamed as Jakarta EE, and with the version 9, the top-level package name `javax` was changed to `jakarta` [28]. However, there is no real difference between `javax.inject.Inject` or `jakarta.inject.Inject`, thus using the latter is not a misuse. This update means that all those 89 misuses are false positives. Given the complete rules (with Jakarta EE annotations added), theoretically the detector should not report the previously reported "false positives". Therefore, we add all the necessary Jakarta EE annotations to the 12 rules, and rerun the misuse detection on projects where the 89 misuses happen. With the complete rules, our detector indeed reports zero misuses, thus leaving us with the total of 126 misuses instead of 215. We find that 26 of these 126 misuses are previously unknown misuses. The distribution of these 26 misuses is as follows: ConfigPropertyInject - 20, RegisterRestClientPath - 4, QueryGraphQLApi - 1 and MutationGraphQLApi - 1. We analyze these misuses to understand why developers make such mistakes or if they are even actual misuses. We now discuss our findings.

MicroProfile specifications specify what functionalities are provided, but

Table 5.1: The 12 rules that we use in the evaluation of the misuse detector. Note: "o.e.m" is the abbreviation of "org.eclipse.microprofile"

Rule name	**Rule definition**
ClaimInject	field with annotation "o.e.m.jwt.Claim" must have annotation "javax.inject.Inject"
ConfigPropertyInject	field with annotation "o.e.m.config.inject.ConfigProperty" must have annotation "javax.inject.Inject"
HealthHealthCheck	class with annotation "o.e.m.health.Health" must have implementation of "o.e.m.health.HealthCheck"
IncomingBean	class with method with annotation "o.e.m.reactive.messaging.Incoming" must have annotation "javax.enterprise.context.ApplicationScoped"
LivenessHealthCheck	class with annotation "o.e.m.health.Liveness" must have implementation of "o.e.m.health.HealthCheck"
MetricRegistryInject	field with (annotation "o.e.m.metrics.annotation.RegistryType" with parameter "o.e.m.metrics.MetricRegistry.Type type" and annotation "javax.inject.Inject") must have type "o.e.m.metrics.MetricRegistry"
MutationGraphQLApi	class with method with annotation "o.e.m.graphql.Mutation" must have annotation "o.e.m.graphql.GraphQLApi"
OutgoingBean	class with method with annotation "o.e.m.reactive.messaging.Outgoing" must have annotation "javax.enterprise.context.ApplicationScoped"
QueryGraphQLApi	class with method with annotation "o.e.m.graphql.Query" must have annotation "o.e.m.graphql.GraphQLApi"
ReadinessHealthCheck	class with annotation "o.e.m.health.Readiness" must have implementation of "o.e.m.health.HealthCheck"
RegisterRestClientPath	class with annotation "o.e.m.rest.client.inject.RegisterRestClient" must have (method with annotation "javax.ws.rs.Path") and annotation "javax.ws.rs.Path"
RestClientInject	field with annotation "o.e.m.rest.client.inject.RestClient" must have annotation "javax.inject.Inject"

Table 5.2: A complete breakdown of the misuses found after our evaluation process. Table 5.1 shows the definitions for each rule. TP - true positive, FP - false positive

Rule	**Previously known** [62]	**New**	**TP**	**FP**	**Total**
ConfigPropertyInject	87	85	91	81	172
RegisterRestClientPath	0	16	0	16	16
RestClientInject	0	12	0	12	12
ClaimInject	6	0	6	0	6
QueryGraphQLApi	2	1	3	0	3
OutgoingBean	3	0	3	0	3
IncomingBean	2	0	2	0	2
MutationGraphQLApi	0	1	1	0	1
Total	**100**	**115**	**106**	**209**	**215**

a runtime (such as Open Liberty [63] or Quarkus [15]) that provides the implementation of a specification can be lenient towards some requirements to improve the developer experience. One such case happens with the ConfigPropertyInject rule (see Table 5.1). If we inspect the MicroProfile Config specification, we notice that `@Inject` is always used with `@ConfigProperty`, and the specification clearly states that *"MicroProfile Config also provides ways to inject configured values into your beans using the `@Inject` and the `@ConfigProperty` qualifier"* [74]. However, Quarkus, a MicroProfile runtime, makes the `@Inject` annotation optional when the injection is made to a class field. Out of these 26 misuses, 20 misuses happen because of the absence of the `@Inject` annotation. Upon checking what runtime the projects are using, we find that the projects where 16/20 misuses happen are, in fact, using Quarkus. For the remaining four misuses of the ConfigPropertyInject rule, we find that three of them are using Open Liberty, while one is using Payara [65]. To verify the legitimacy of these misuses, we check the documentation provided by these vendors [7, 66]. Unlike the Quarkus documentation, the guides provided by Open Liberty and Payara do not explicitly mention the optionality of `@Inject` annotation. To empirically verify the requirement of the `@Inject` annotation, we create projects using both Open Liberty and Payara runtimes

where we try to inject a configuration property into a field annotated with the `@ConfigProperty` annotation without using the `@Inject` annotation. In both servers, using `@ConfigProperty` without `@Inject` results in the field having a `null` value. This shows that the remaining four issues are true positives. Therefore, we conclude that out of 20 misuses of the ConfigPropertyInject rule, only four are actual true positives.

The next four misuses we examine violate the RegisterRestClientPath rule. In all these four misuses, the `@Path` annotation is missing. After examining the Java documentation for `RegisterRestClient` and the MicroProfile Rest Client specification, we learn that there is a way to set a base path by using either `baseUri` or `configKey` parameters. When using `baseUri`, client developers can set the base endpoint directly in the Java code. `configKey` on the other hand requires the specified key to be present in the `microprofile-config.properties` file. If none of them are present, the MicroProfile runtime uses a default `configKey` value, and requires the presence of that key in the `microprofile-config.properties` file. `@Path` annotation specifies which URI path a particular method handles. A common thing among these four misuses is that all the interfaces contain a single method with no `@Path` annotation to represent an endpoint. It is possible that a client developer simply wants to create a method that handles requests to the base path. It is also possible that the URI path is already included in the base path, removing the need for using `@Path` annotation. In all four misuses for the RegisterRestClientPath rule, we observe these cases. Thus, we can conclude that none of the misuses for the RegisterRestClientPath rule are true positives.

We can summarize that out of 26 misuses, only six are true positives, while the remaining 20 are false positives, which makes the precision of the detector 84% (106/126).

5.2.3 Summary

Our results show that given any confirmed extended RulePad rule, our misuse detector successfully finds the deviations from that rule if there are any. However, after analyzing the misuses detected during our evaluation, we identify

three factors that need to be considered while scanning for misuses in Java files. First, confirmed rules need to provide information about which API versions they are valid for. As we discovered, the version change from Java EE to Jakarta EE affected the validity of 89 reported misuses. Second, confirmed rules need to provide information about which MicroProfile runtime they are valid for. Our detector reported 16 false positives due to runtime related issues. Third, the detector should scan the configuration sources. We discuss these limitations in more detail in Section 8.3.

While the misuse detector has limitations, these limitations do not affect the premise of our work to proceed with a user study since the goal of the detector evaluation in Section 5.2 is only to verify the correctness of the automatically-generated static analysis checks. However, the user study we discuss in the next chapter evaluates the usefulness of having mined rules as starting points as well as the usefulness of the human-in-the-loop concept.

Chapter 6

User study

To evaluate the usefulness of RVT in modifying and validating the mined candidate rules, as well as the usefulness of the whole idea of using mined rules as starting points, we conduct a user study with MicroProfile API experts. Our goal is to answer the following three research questions:

RQ1. Is the rule specification DSL in RVT expressive enough for specifying rules? We adopt RulePad with some extensions as our DSL of choice for encoding rules. There is a possibility that an API expert might want to specify a constraint that cannot be expressed using RulePad. Thus, we want to know how expressive our extended RulePad DSL is for authoring annotation-based API usage rules.

RQ2. Is RVT useful for the modification and validation of mined rules? The key concept in our proposed pipeline is having a human in the loop. Thus, we want to know if RVT makes it easy for experts to author and validate the mined rules.

RQ3. Are candidate rules effective in alleviating the difficulties of writing API usage rules from scratch? We want to understand if the mined candidate rules provide good starting points for API experts when authoring rules. Overall, we want to determine if the idea of having mined rules as a starting point is useful to API experts.

6.1 Experiment Setup

The experiment is an online, 90-minute Zoom session where experts use RVT to validate the presented candidate rules. We audio and video record the session, with participants' consent and after our university's ethics clearance, for post-analysis purposes. For video recording purposes, the experts have to share their computer screen. The experiment is divided into three parts that we describe below.

6.1.1 Tutorial and setup.

At the beginning of the experiment (up to 30 minutes), experts go through a tutorial that we prepared to get familiar with RVT and the DSL that we use to present rules (i.e., the extended RulePad).

6.1.2 Live experiment.

After participants get familiar with RVT, we proceed to the main experiment task, where API experts validate candidate rules encoded in the RulePad DSL. For each candidate rule, we first ask the participants to rate it in terms of understandability of the presented rule on a scale of 1 to 3 (1-hard to understand, 2-neither hard nor easy to understand, 3-easy to understand). This task enables us to quantify how easy it is for API experts to understand a given rule and contributes to the evaluation of RQ1. After getting familiar with the presented rule, participants proceed to validate it. Participants are allowed to use online resources such as documentation and online discussion forums, if needed. To validate a rule, participants can (1) confirm the rule as is, (2) confirm the rule with changes, or (3) reject the rule. During this validation process, we employ the think-aloud protocol [86] where we ask participants to verbally share the reasoning behind their decisions. For example, when a participant rejects a candidate rule, we ask them to share the reasons that led them to this decision. This feedback can help us improve the mining process.

6.1.3 Exit survey.

At the end of the session, we ask participants three rating-based (RB) and three open-ended (OE) questions. For the rating-based questions, participants can also provide verbal explanations for their ratings. We ask the following questions:

RB1: For rule authoring, having an existing candidate rule as a starting point is easier than writing a rule from scratch (strongly disagree, disagree, neither agree nor disagree, agree, strongly agree). This question addresses RQ3.

RB2: Having a dedicated tool for rule validation makes it easy to validate rules (from strongly disagree to strongly agree). This question addresses RQ2.

RB3: How do you rate the difficulty level of editing rules using RVT? (very hard to edit, hard to edit, neutral, easy to edit, very easy to edit). This question addresses RQ2.

OE1: Are there additional code constructs you think need to be a part of RulePad? This addresses RQ1.

OE2: What types of additional information could have assisted you in validating the rules? This indirectly addresses RQ2 and enables us to know what other information experts would find helpful.

OE3: Are there any additional rules you can think of that were not presented? This question does not address a specific RQ but enables us to understand what rules the mining process cannot discover and what other code relationships need to be tracked (which may require further RulePad extensions).

Our open-ended questions allow participants to share valuable feedback with us, which helps us further improve our approach.

Table 6.1: The candidate rules mined by Nuryyev *et al.* [62] , encoded in the extended RulePad format (explained in Section 4.1.1). We remove fully-qualified names for better readability. Note 1: Each rule given in this table is the RulePad encoding of a rule with the same id number given in the Listing 2.1. Note 2: The five rules that were not used during our user study due to expert unavailability are greyed out

Id	**Rule definition**
1	class with (method with (annotation "APIResponse" with parameter "String responseCode" and parameter "String description" and annotation "Operation"))and annotation "Path" must have annotation "Path" with parameter "String value"
2	method with annotation "SimplyTimed" and parameter "String" must have annotation "SimplyTimed" with parameter "String name"
3	class with (annotation "Entity" and annotation "Type" and annotation "media.Schema" with parameter "String name" and parameter "String title") must have annotation "Entity" with parameter "String name"
4	class with (annotation "ApplicationPath" with parameter "String value" and annotation "Path" with parameter "String value" and annotation "LoginConfig" with parameter "String authMethod" and parameter "String realmName" and annotation "RequestScoped") must have extension of "Application"
5	class with method with (annotation "APIResponses" and annotation "Operation") must have annotation "Path"
6	class with method with annotation "Outgoing" must have annotation "ApplicationScoped"
7	class with method with annotation "Mutation" must have annotation "GraphQLApi"
8	method with annotation "Gauge" with parameter "String unit" must have annotation "Gauge" with parameter "String name"
9	field with (annotation "RegistryType" with parameter "MetricRegistry.Type type" and annotation "Inject") must have type "MetricRegistry"
10	class with (annotation "ApplicationScoped" and annotation "Health") must have implementation of "HealthCheck"
11	field with (annotation "ConfigProperty" and annotation "Inject") must have annotation "ConfigProperty" with parameter "String name"
12	class with method with annotation "Query" must have annotation "GraphQLApi"

13	class with (method with (annotation “APIResponse” with parameter “String responseCode” and annotation “Operation”)) and annotation “Path” with parameter “String value” must have method with return type “Response”
14	class with (method with annotation “Path” with parameter “String value”) and annotation “RegisterRestClient” must have annotation “RegisterClientHeaders”
15	class with (annotation “ApplicationScoped” and annotation “Readiness”) must have implementation of “HealthCheck”
16	class with (annotation “Path” and annotation “OpenAPIDefinition” with parameter “info”) must have annotation “Path” with parameter “String value”
17	class with annotation “Liveness” must have implementation of “HealthCheck”
18	class with (annotation “Tag” with parameter “String name” and annotation “Path”) must have annotation “Path” with parameter “String value”
19	class with method with annotation “Incoming” must have annotation “ApplicationScoped”
20	field with annotation “Inject” must have annotation “RestClient”
21	class with (annotation “RegisterRestClient” and annotation “RegisterProvider”) must have annotation “RegisterProvider” with parameter “Class value”
22	field with annotation “Inject” must have annotation “Metric”
23	class with field with (annotation “Inject” and annotation “Claim”) must have annotation “Path”

6.1.4 Interviewer involvement

The author of this thesis is the only person conducting the interviews. During the experiments, we provide varying amount of guidance to the participants in each segment of the interview. We provide the most guidance during the tutorial segment, where we explain what the tool and the DSL are, what the objective of the user study is, and how the validation process works. During the live experiment, our guidance is minimal, unless the participant asks questions. During the exit survey, we explain each presented question to make sure the participant does not misinterpret the question. Additionally, based on the answers a participant provides, we may ask follow-up questions. It is also worth noting that the participants have complete control over the RVT and

Table 6.2: Number of candidate rules mined for each MicroProfile specification [62]. One rule belongs to both GraphQL and OpenAPI specifications, hence the total is 23, not 24.

MicroProfile Specification	**# mined rules**	**# confirmed rules**
Config	1	0
GraphQL	3	N/A
Health	3	3
JWT-Auth	2	1
Metrics	4	3
OpenAPI	6	2
REST Client	3	N/A
Reactive Messaging	2	2
Total	**23**	**11**

they perform the tasks with the tool themselves. At the beginning of the session, we share the web link to RVT with the participants.

6.2 Participant Recruitment

MicroProfile API experts (i.e., direct contributors to various MicroProfile specifications) are the target population of our study. We drafted a recruitment email that our industry collaborator sent to six IBM MicroProfile API developers. Our goal was to recruit at least one expert for each MicroProfile specification that we have candidate rules for. We consider validating all 23 rules from Listing 2.1. In Table 6.1, we also share these 23 rules encoded in the extended RulePad format, which is the format that the participants validate the candidate rules in. Table 6.2 groups those rules based on MicroProfile specifications. Four API experts (P1-P4) agreed to participate in the user study. Before the experiment, for each participant, we collected background information on which MicroProfile components they are familiar with and created a set of candidate rules that contain APIs from these components. Table 6.3 shows the summary of the details we collect about the participants of our study.

Table 6.3: Information about the participants (P1-P4) of our study. Please note that "Experience" column specifies the years of experience in their current team

Participant	Spec(s) responsible	*Experience	# of rules validated
P1	Config, Reactive Messaging, OpenAPI	4 years	9
P2	Health	6 years	3
P3	Metrics	5 years	4
P4	JWT-Auth	2 years	2

6.3 Results

While we wanted to validate all 23 rules, given expert availability, we could only validate 18 rules. We share all the confirmed rules in Table 6.4. There were no common rules shared between participants due to their different expertise. P1, P2, P3, and P4 confirm 4/9 rules, 3/3 rules, 3/4 rules and 1/2 rules, respectively. Overall, the four participants label 11/23 rules as correct, with all except one rule (from MicroProfile Metrics) requiring modifications. All presented rules for MicroProfile Health and Reactive Messaging are validated as partially correct. For MicroProfile Metrics, P3 considers the only rejected rule as "best practice" and not necessarily incorrect.

We now present the main results of our user study, where we focus on the whole pipeline rather than the accuracy of the mining process, which Nuryyev *et al.* evaluated already [62].

6.3.1 RQ1: Expressiveness of the extended RulePad DSL in RVT

Figure 6.1 shows how participants perceive the presented candidate rules in terms of their understandability. The graph shows that the majority of the presented candidate rules are easy to understand for participants. P2 mentions that the English-like syntax of RulePad makes it easy to learn in a short period of time. Recall that we introduced two constructs to RulePad to reduce

Table 6.4: The rules that the participants have confirmed during our User study. We remove fully-qualified names for better readability. Note: Each confirmed rule given in this table is the confirmed version of a candidate rule with the same id number given in the Table 6.1

Expert	Rule id	**Rule definition**
P1	5	method with annotation "APIResponses" or annotation "Operation" must have annotation "[GET|POST|HEAD|OPTIONS|DELETE|PATCH|PUT]"
P1	6	class with method with annotation "Outgoing" must have bean declaration
P1	16	class with annotation "OpenAPIDefinition" must have extension of "Application"
P1	19	class with method with annotation "Incoming" must have bean declaration
P2	10	class with (annotation "javax.enterprise.context.ApplicationScoped" or annotation "jakarta.enterprise.context.ApplicationScoped") and annotation "[Startup|Liveness|Readiness]") must have implementation of "HealthCheck"
P2	15	class with (annotation "javax.enterprise.context.ApplicationScoped" or annotation "jakarta.enterprise.context.ApplicationScoped") and annotation "Readiness" must have implementation of "HealthCheck"
P2	17	class with (annotation "javax.enterprise.context.ApplicationScoped" or annotation "jakarta.enterprise.context.ApplicationScoped") and annotation "Liveness" must have implementation of "HealthCheck"
P3	8	class with method with annotation "Gauge" must have annotation "ApplicationScoped" and method with annotation "Gauge" with parameter "String unit"
P3	9	field with annotation "RegistryType" with parameter "MetricRegistry.Type type" and annotation "Inject" must have type "MetricRegistry"
P3	22	field with annotation "Metric" must have annotation "Inject"
P4	4	class with annotation "ApplicationPath" and annotation "LoginConfig" with parameter "String authMethod" must have extension of "Application"

the verbosity of the DSL (Section 4.1.1). We observe that P1 and P2 use the shortcut that allows grouping of annotations from the same package, showing the usefulness of the shortcut. We find that while the extended RulePad is expressive enough to specify most of the code constructs needed to encode annotation usage rules, there is still room for improvement. Participants suggest that the extended RulePad can be further improved by including the following code constructs:

1. Specify mutual exclusivity. A rule might require usage of only one annotation from a set of annotations. Currently, the extended RulePad supports disjunctions (i.e., or) which does not guarantee mutual exclusivity (i.e., xor).

2. Invert a predicate. Our extension to RulePad does not support negations. For example, our extension cannot encode the following hypothetical rule: a field with annotation A and not with annotation B requires annotation C.

3. Require overriding a specific method. Method overriding is useful in two cases. First, it enables rule completeness. Currently, we can specify that a class needs to implement an interface, but a complete rule must indicate which method needs to be overridden/implemented from that interface. Second, given a predicate, an expert might want to require overriding specific methods (e.g., if class is annotated with X and extends Y, then it must override method Z).

4. Shortcuts for frequently used MicroProfile constructs such as CDI beans, JAX-RS resource methods (i.e. a method that is annotated with request method designators such as `@GET` or `@POST`) or classes (i.e., a class that either is annotated with `@Path` or contains at least one resource method). For example, instead of saying `method with annotation "Operation" must have annotation "[GET|POST|PUT|DELETE|..]"`, an expert can simplify the rule to `method with annotation "Operation" must be a JAXRS resource`.

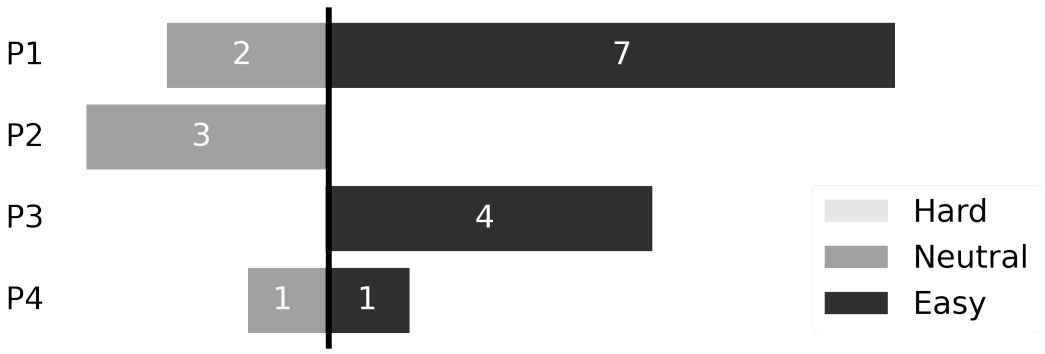

Figure 6.1: Understandability of candidate rules

All the potential RulePad extensions above represent common cases that happen in more than one instance. However, during the interview with P4, we encounter a specific constraint involving the annotation `@Claim` that RulePad is not capable of expressing [12]. `@Claim` annotation has two parameters to specify a claim name for injection, `String value` and `Claims standard`. The constraint is that a client developer should use either one of parameters, however, if both of them are present, then their values must align (e.g., `@Claim(value="exp", standard=Claims.exp)`), otherwise a `DeploymentException` is thrown (e.g., when `@Claim(value="exp", standard=Claims.iat)`). Currently, RulePad is not capable of specifying this requirement, and adding the capability of expressing such requirements might affect the overall readability of a rule.

RQ1: The extended RulePad DSL is capable enough to express most of the code constructs required for validating the candidate rules. The participants mention that the resemblance to the English language makes RulePad easy to learn in a short period of time. Additionally, they suggest four potential improvements to further improve the DSL.

6.3.2 RQ2: Usefulness of RVT in modifying and validating candidate rules

Figure 6.2-RB2 shows that participants unanimously agree that RVT is useful. Therefore, we conclude that having a dedicated tool for rule validation is useful for API experts. That said, Figure 6.2-RB3 shows that our participants have varying opinions on the level of difficulty of editing rules using RVT. P1

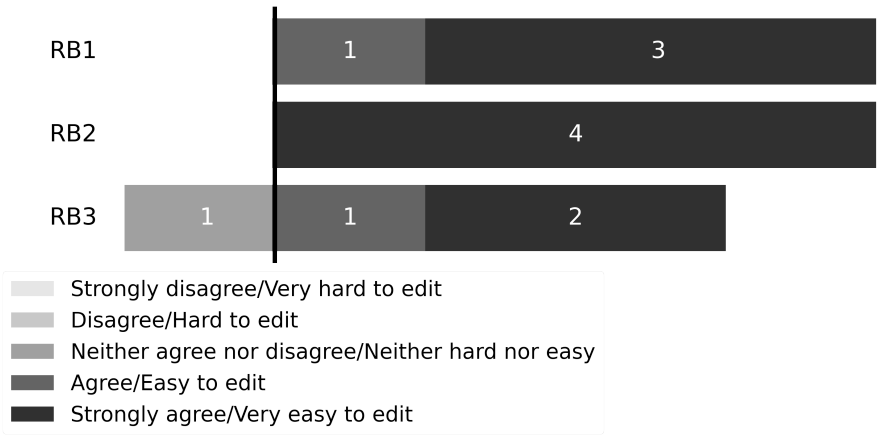

Figure 6.2: RB1 results regarding having a starting point for rule authoring. RB2 results regarding usefulness of having a dedicated rule validation tool. RB3 results for levels of difficulty of editing rules using RVT.

who rated "Neither hard nor easy" states that in some cases the rules written in the extended RulePad format are not how they would be written as a specification. For example, consider the following rule: "a field with the `@ConfigProperty` annotation should have the `@Inject` annotation unless the class has the `@ConfigProperties` annotation". In the extended RulePad, we encode the rule as "`class with field with annotation "ConfigProperty" must have annotation "ConfigProperties" or field with annotation "Inject"`". While this rule is correct, P1 argues that it is not natural to write the rule in that format. Instead, P1 mentions that an API expert would write the rule as "`field with annotation "ConfigProperty" must have annotation "Inject" unless enclosing class has annotation "ConfigProperties"`". This aligns with the previous suggestion of supporting predicate negation in RulePad.

We now discuss what other information participants think can help them in the rule validation process. From the existing assisting components of RVT, participants find the code preview section particularly helpful for visualizing how a rule might potentially look. P3 states that the rules being presented preformatted makes rules easier to understand. Participants share the following ideas for potential enhancements:

- Easier access to Java documentation. Finding the correct documentation page may take time, and providing quick access to it can be useful.

- Auto-completing fully qualified names (FQNs). When generating static analysis checks, we expect the validated rules to have FQNs. While candidate rules have FQNs, it might be hard for experts to know the FQN for every annotation/class they need to add or modify. Providing such assistance can reduce the time spent on looking for the correct package name.
- Syntax checking for the DSL. Participants believe that having a syntax error checker can assist them in writing rules properly.

RQ2: Our results show that having a dedicated tool for rule validation is indeed useful and the participants' experience with RVT is mostly positive. Participants find the preformatted rules and the generated code preview feature useful. Additionally, they suggest three potential features we can add to further enhance RVT.

6.3.3 RQ3: Effectiveness of the mined rules in alleviating the difficulties of writing usage rules

Figure 6.2-RB1 illustrates what participants think about having starting points for authoring rules. Our results show that having starting points is helpful. According to P1, given a rule, it is generally easier to point out the problems with the rule. In a similar fashion, P3 states that as with anything in tech, it becomes easier once you have something to work with. In response to OE3, P1 states that there are probably additional rules to encode; however, to find them, one needs to go through the documentation. Note that this statement strengthens our argument that having starting points reduces the effort of manually going through documentation to find rules, especially that not all API usage rules are documented.

API usage rule mining solutions are capable of finding patterns that are frequently used in client code. However, they can also miss usage patterns that are less frequent, or intricate. This is the reason why we ask OE3 in our exit questionnaire to understand what rules the our pattern mining approach has missed. In response to OE3, P4 shared two rules:

- a field with type `JsonWebToken` must be injected using `@Inject` annotation [38].

- `@Claim` can only be injected into a field with one of the following types:
  - ◇ "java.lang.String"
  - ◇ "java.lang.Long" and "long"
  - ◇ "java.lang.Boolean" and "boolean"
  - ◇ "java.util.Set<java.lang.String>"
  - ◇ "jakarta.json.JsonValue.TRUE/FALSE"
  - ◇ "jakarta.json.JsonString"
  - ◇ "jakarta.json.JsonNumber"
  - ◇ "jakarta.json.JsonArray"
  - ◇ "jakarta.json.JsonObject"
  - ◇ "java.util.Optional" wrapper of the above types
  - ◇ "org.eclipse.microprofile.jwt.ClaimValue" wrapper of the above types

One of the potential issues with finding API usage rules through pattern mining is that a mined pattern can contain deprecated APIs. There was one such rule that P2 had to validate. While converting the candidate rule into the correct rule, P2 explains that the candidate rule uses the `@Health` annotation which has been deprecated since MicroProfile Health version 2.0 and is no longer part of the API [54]. This case shows that having human validation is critical when it comes to mining rules.

RQ3: Our results show that the concept of using mined rules as starting points is useful. Having starting points reduces the difficulty of manually searching for rules in the documentation. Expert validation also eliminates the chance of using mined patterns that are incorrect/partially correct (e.g., patterns using deprecated annotations). Overall, we conclude that our human-in-the-loop concept is feasible and useful.

Chapter 7

Threats to Validity

7.1 Internal validity

It might be possible that we convert the mined rules into the extended RulePad format incorrectly. However, for all 23 rules we use in our user study, we manually compared each converted RulePad rule with its original JSON format to make sure that the conversion process does not misrepresent the mined rule.

There might be a flaw in the implementation of our extended RulePad parser, resulting in the generation of incorrect static analysis checks. In addition to conducting an experiment to validate the correctness of the generated checks, we write 73 unit tests, each checking different use cases. There may be some misuses that we could not catch due to our detector not scanning the class hierarchy and annotation definitions. In future, we intend to address this problem by scanning the entire project instead of each file in isolation. We discuss the limitations of our detector in detail in Section 8.3. However, it is worth noting that our misuse detector still found new misuses in addition to all the misuses found by Nuryyev *et al.* [62] .

During our user study, it is possible that we misrepresent what API experts mention, which might affect the results presented in RQ1 and RQ2 related to the open-ended questions. For this reason, we audio and video record each interview, and post-analyze them. It is also possible that there was an inconsistency in the treatment each participant received. The author of this thesis conducted each interview. While it is impossible to provide the exact same treatment to each participant, the interviewer followed a specific

procedure, and adhered to it unless a participant specifically asked for help.

7.2 Construct validity

To mine candidate rules, our pipeline uses a previously developed and evaluated pattern mining technique [62]. The quality of the mining process can affect the overall experience of participants in our user study. However, the user study does not focus on measuring the correctness of the rules but rather focuses on the editing and validation process.

Similar to the user study, our misuse detector evaluation process used the existing set of rules from Nuryyev *et al.* [62] . The goal was to see if our misuse detector, which generates static analysis checks automatically, could find all the previously known misuses detected by manually-written checkers. There is a likelihood that some of those rules might be incorrect. While we do not check for the validity of the rules used in the previous study, there were three rules that were incomplete due to version changes in the specification. We made necessary additions and rerun the experiment with complete rules to verify that our detector works as expected.

7.3 External validity

Our goal was to validate all the mined rules by at least one API expert. We reached out to six of the relevant MicroProfile API experts working for our industry partner, and four agreed to participate. These four participants validated 18 out of 23 mined rules (78%) from six different MicroProfile specifications. Although we could not validate all the mined rules, the validated rules cover most of the annotation relationships in our rules. Our work focuses on validating MicroProfile annotation usage rules. While, in principle, our approach can be applied to other annotation-based libraries, we present only a case study of MicroProfile and our findings may not generalize beyond that. Future work can reuse our pipeline to investigate its applicability to other libraries and frameworks.

Chapter 8

Discussion and Implications

In this thesis, we proposed a human-in-the-loop approach to generate annotation usage rules for MicroProfile. We developed a web-based tool, RVT, to facilitate rule validation and generation. To evaluate the usefulness of RVT and our approach, we performed a user study with MicroProfile API experts. We also created a misuse detector, and verified its correctness. We now discuss the implications of our findings.

8.1 Generating API usage rules

Our main objective in this thesis is to generate accurate annotation usage rules while reducing the burden of writing them from scratch. Therefore, we use a pattern mining technique to automatically mine candidate rules that provide starting points to experts for generating accurate rules. The results of our user study (RQ3) show that all the API experts agree or strongly agree that having starting points in the form of mined candidate rules reduces not only the difficulty but also the effort of writing rules. They state that it is easier and takes less effort to work with candidate rules and find problems in the rules rather than discovering a rule manually. Our participants also confirm 11 of the 18 presented candidate rules and modify 10 of these 11 confirmed rules, which indicates that most of the presented candidate rules are partially correct or incorrect rules. Thus, using the mined candidate rules directly for misuse detection could have produced a lot of false positives. Therefore, our approach that introduces experts for validating the mined rules is critical for generating

accurate rules. The results also show that experts go to extra lengths to produce accurate rules. For example, P2 checked whether the generated rules use any deprecated APIs.

Our results show that 10/11 confirmed rules are present in the documentation. This shows that our approach is effective in reducing the time spent on finding the rules in the documentation. While these rules are present in the documentation, they are not automatically checked for, and client developers might violate them. Additionally, some of these usage rules that are available in the documentation are more implicit than others, requiring some amount of domain knowledge. For instance, one of the confirmed rules (see the rule with id 5 in Table 6.4) requires the method to have one of the following annotations: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@HEAD`, `@OPTIONS`, `@PATCH`. The same rule that is available in the documentation refers to the same property as a "JAX-RS method". As opposed to the original documentation version, our confirmed rule is more explicit, indicating exactly what program element is required.

8.2 Facilitating rule validation

To facilitate rule validation, our approach uses an extended version of the RulePad DSL. Not only can experts use RVT to validate the presented rules, but they can also modify them. Therefore, it is critical that the DSL is able to express or construct annotation usage rules. While our results (RQ1 and RQ2) show that the DSL does not lack any grammar to express the confirmed rules, participants would like the ability to express API usage rules in a more natural form and with finer granularity. For example, P1 states that RulePad's DSL might not be the most natural way to express a rule in some cases. However, expressing a rule in a natural form can be challenging, because it might be hard to find a consistent DSL format or syntax that allows natural expression. Addressing this limitation requires another user study where the focus is on the way experts naturally author rules without adhering to a specific format. We can then analyze the produced rules and see if there are common patterns in the rules that can be incorporated into RulePad gram-

mar. Additionally, introducing more logical operators, such as XOR, NOR, and NOT, would allow experts to express rules with finer granularity. The extended RulePad also does not support referencing to the class from a field or method statement. In some cases, this limitation can cause a rule lose its focus. For example, consider the following rule: *if a method parameter is annotated with "PathParam" then either that method or the class should have the annotation "Path"*. In the extended RulePad, we can specify the rule as follows: `class with method with parameter "PathParam" must have annotation "Path" or method with annotation "Path"`. Let us now express the same rule while referring to the enclosing class: `method with parameter with annotation "PathParam" must have annotation "Path" or enclosing class with annotation "Path"`. While those two rules are equivalent, the latter immediately shows that the focus is on the method, the former references the class first, which is not the main element of this rule. Addressing these limitations simply require adding elements to the grammar of the DSL.

Another avenue for improvement is the assisting features in RVT. Currently, the feature that has been brought up the most is the auto-completion of fully qualified names. This feature will allow experts to easily specify fully qualified names without consulting documentation, which will improve the overall user experience. We can implement the auto-completion feature by extracting fully qualified names of MicroProfile APIs from the documentation and storing them in a database integrated with our Rule Authoring Editor. We can also provide easy access to Java documentation for all the program elements used in the rule. This feature can be a part of the Code Preview where clicking on a program element will open the corresponding Javadoc page. Finally, we can provide another labeling button to allow experts to label a rule as a "best practice". When used for misuse detection, the best practice rules can produce warnings instead of errors. API experts also suggest including a mechanism to check for syntax errors in the DSL.

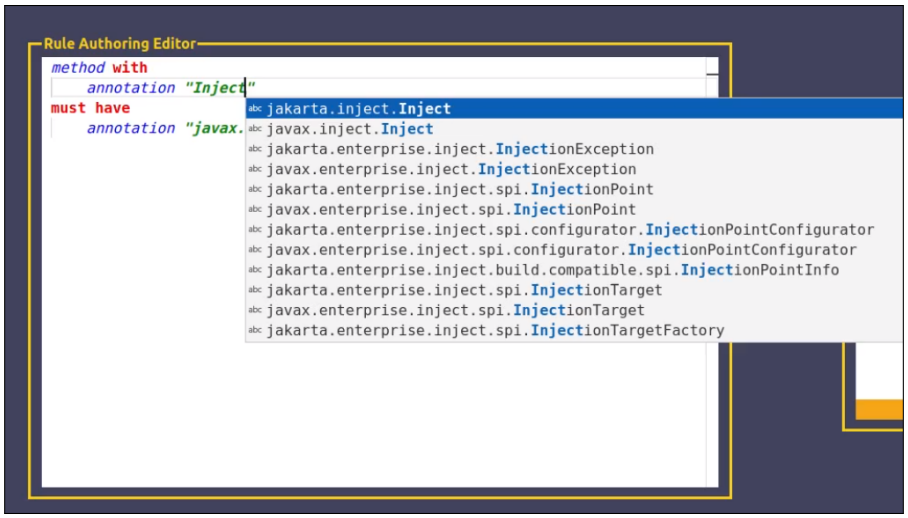

Figure 8.1: Autocompletion suggestions for the keyword "Inject" suggested by RVT

Figure 8.2: The "best practice" label in RVT

8.2.1 Current state of RVT

In this section, we discuss the current state of RVT, which includes the extensions we have made based on the feedback we received during the user study.

For the assisting features, in the improved RVT, we include the class name autocompletion and easy documentation access features. For the former, it suffices to provide a partial string, and the Rule Authoring Editor provides all possible completion options (see Figure 8.1). For the latter, in the Code Preview section, pressing on a particular program element (e.g., annotation `@ConfigProperty`) opens the latest documentation available for that element. We implement these features in a way that enables easy extension in future. We also add the possibility of labeling the rule as a "best practice" (see Figure 8.2).

We also improve the extended RulePad DSL, by incorporating the following capabilities into the DSL:

- We add more logical operators:

- ◇ "`one of`" for specifying mutual exclusivity. For example, "`...must have one of (annotation "Foo" or annotation "Bar")`"
- ◇ "`no`" for negating a predicate. For example, "`...must have no annotation "Foo"`"
- ◇ "`none of`" for negating multiple predicates. For example, "`...must have none of (annotation "Foo" or annotation "Bar")`"

- We allow specifying method overriding using "`overridden method`" keyword. For example, "`class with annotation "Foo" and extension of "Bar" must have overridden method "foobar(String)"`"
- Fields and methods can refer to the enclosing class using "`enclosing class`" keyword. For example, "`method with parameter with annotation "PathParam" must have annotation "Path" or enclosing class with annotation "Path"`"

We leave features such as creating shortcuts for frequently used MicroProfile constructs and syntax checking as future work. We publicly share the GitHub repository of our project [49].

8.3 Detecting misuses

To verify the correctness of our misuse detector, we conducted an experiment. We now discuss the implications of our findings and limitations of our misuse detector. We identify four aspects that should be improved to reduce the number of false positives.

API versioning. APIs change over time [44]. These changes include adding new features, deprecating existing features, or changing namespaces. These changes might alter the previously valid API usage rules. During our experiment, we noticed one such case in usage rules concerning Java EE/Jakarta EE APIs. A single namespace change resulted in our detector reporting 89 false positives. While we fixed this issue by making necessary additions to those rules, it is important to mention that our mining approach does not consider different versions of APIs. However, taking API versions into account during

the mining process might result in fewer rules being mined. The API versioning can be a part of rule validation process where API experts can set what versions a particular rule is valid for.

MicroProfile runtimes. MicroProfile is a collection of specifications, and on its own it does not provide functionality. There are vendors that implement these specifications. However, these vendors are also capable of being lenient at some cases to provide a better developer experience. One such case happens with Quarkus [69] where it makes the usage of `@Inject` annotation optional when used with `@ConfigProperty` on class fields. In turn, this resulted in 16 false positives during our misuse detector evaluation. Similarly to API versioning, rules could be supplied with runtime related information. The main issue related to providing runtime-related information is that we must recruit API experts that have expertise in different MicroProfile implementations, which is infeasible. Additionally, statically extracting which MicroProfile runtime a project uses can be challenging. One possible solution might be to create checkers for each known MicroProfile runtime. A checker can check whether a project uses dependencies, configuration files or configuration properties that are specific to a particular runtime. However, this approach is ad-hoc at best, and is not guaranteed to produce accurate results. It is also worth noting that it is possible to configure the MicroProfile runtime outside of the project, which renders obtaining the runtime information impossible.

Configuration sources. Currently, our misuse detector only scans Java files and does not take into account configuration sources. MicroProfile supports multiple configuration sources such as properties files or XML files. Analyzing configuration sources is hard to achieve with static analysis, because of the different ways a client developer can provide a configuration source. In MicroProfile, apart from well-known configuration sources (such as the `microprofile-config.properties` file), there are ways to create custom configuration sources that can pull the configuration data from sources such as the database, which cannot be accessed till runtime.

```
1 class Parent extends Bar {
2
3 }
4
5 @Foo
6 class Child extends Parent {
7 }
```

(a)

```
1 @NormalScope
2 // omitted the rest for brevity
3 @interface ApplicationScoped {}
4
5 @Foo
6 @ApplicationScoped
7 class Child {}
```

(b)

Figure 8.3: Code examples for demonstrating the limitations of the misuse detector

Improving the detection process. Currently, the detector scans one Java file at a time, in isolation. We do not scan the class hierarchy (parent classes, interfaces) or the annotation definitions. While we did not find any issues during the evaluation process caused by these limitations, we believe considering these two aspects can potentially reduce false positives. Scanning the class hierarchy is important because child classes might inherit annotations or other classes/interfaces through their parents. For example, consider the following rule: `class with annotation "Foo" must have extension of "Bar"`. For the code example given in Figure 8.3 (a), our detector would report a misuse since class `Child` does not extend `Bar`. Scanning annotation definitions is also important because some annotations can be annotated with other annotations that the misuse detector is searching for. Consider the annotations `@NormalScope` and `@ApplicationScoped` from Jakarta EE [29]. The `@ApplicationScoped` annotation has been annotated with `@NormalScope` in its definition [9]. Now, consider the following rule: `class with annotation "Foo" must have annotation "NormalScope"`. Without scanning the annotation definition, for the code example given in Figure 8.3 (b), the misuse detector will report a misuse since `@NormalScope` does not exist on the `Child` class. Scanning the annotation definition would enable the misuse detector to treat this instance as a correct usage.

Chapter 9

Conclusion

Similar to library API calls, Java annotations also have usage constraints associated with them. Violation of such constraints can lead to various issues ranging from silent faulty behavior to runtime exceptions. Violation of a constraint is formally called as an API misuse, or simply a misuse. To detect misuses, the ideal path is asking library authors to write accurate checkable usage rules, and then verifying the client developer's code against these checkable usage rules. However, writing rules from scratch is cumbersome. Because of the time-consuming nature of manual rule authoring, researchers have proposed pattern mining techniques to discover those usage rules automatically. The main premise behind pattern mining is that a frequent usage pattern is a rule. However, in practice, the mined rules are not always accurate, and they produce false positives during misuse detection.

In this thesis, we combine both techniques and introduce a human-in-the-loop approach for producing accurate annotation usage rules of MicroProfile APIs. We leverage pattern mining to produce starting points for writing API usage rules. We build a specialized tool, RVT, to facilitate the rule validation process, and a misuse detector to automatically generate static analysis checks from correct rules. To make rules easily understandable, RVT extends an English-like DSL called RulePad for encoding mined rules. We evaluate our approach in a user study with MicroProfile subject matter experts. The user study results show that having starting points makes writing rules easier and that our proposed pipeline can be used to automatically produce accurate

MicroProfile API usage rules. These usage rules can be integrated into static analysis tools to help MicroProfile client developers write less buggy code, or they can improve the documentation.

Additionally, we verify the correctness of our misuse detector by running it against the complete commit history of 517 projects. We discover 126 misuses in total. 100/126 misuses are previously known misuses, showing that given correct confirmed rules, our detector can accurately find the misuses. From the remaining 26 misuses, only six are true positives while the rest are false positives. We find that false positives are caused by reasons such API version changes, differences between MicroProfile runtimes and configuration sources. However, the confirmed rules do not contain any information related to the version of the API or the supported MicroProfile runtime. It is worth noting that the main premise of our evaluation was to check given a correct confirmed rule whether our detector can successfully find a misuse of that rule, which remains unaffected by the produced false positives.

Future work can explore developing a DSL that allows expressing rules in a natural form. A good starting point can be starting from the rules present in documentation/specifications. Additionally, conducting a user study and focusing on the way API experts author rules if they did not have to adhere to any format would also be useful. Another path for future work is adding version-related and runtime-related information to the confirmed rules. This can either be achieved during the mining process or during the rule validation.

References

[1] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A benchmark for API-misuse detectors," in *Proceedings of the 13th international conference on mining software repositories*, 2016, pp. 464–467.

[2] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of Static API-Misuse Detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2019. DOI: 10.1109/TSE.2018.2827384.

[3] *Annotations*, https://docs.oracle.com/javase/8/docs/technotes/guides/language/annotations.html, 2014.

[4] *ANTLR*, https://www.antlr.org/, 2022.

[5] *Apache Maven*, https://maven.apache.org, [Last accessed: May 31, 2022],

[6] *Apache TomEE*, https://tomee.apache.org/, 2022.

[7] J. Cass, *MicroProfile Config 2.0*, https://openliberty.io/blog/2021/03/31/microprofile-config-2.0.html, 2021.

[8] *CDI — Jakarta EE*, https://jakarta.ee/specifications/dependency-injection/, 2022.

[9] *cdi/ApplicationScoped.java — GitHub*, https://github.com/jakartaee/cdi/blob/74827d2808a4db73d1402206a8e116b5702d031a/api/src/main/java/jakarta/enterprise/context/ApplicationScoped.java, 2020.

[10] S. Ceri, G. Gottlob, L. Tanca, *et al.*, "What you always wanted to know about Datalog(and never dared to ask)," *IEEE transactions on knowledge and data engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[11] *CheckStyle*, https://checkstyle.org, 2022.

[12] *Claim (MicroProfile JWT-Auth API)*, https://download.eclipse.org/microprofile/microprofile-jwt-auth-2.0/apidocs/org/eclipse/microprofile/jwt/Claim.html, 2021.

[13] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*, Springer, 1981, pp. 52–71.

[14] A. Colmerauer, "An introduction to Prolog III," in *Computational Logic*, Springer, 1990, pp. 37–79.

[15] *Configuration Reference Guide — Quarkus*, https://quarkus.io/guides/config-reference#inject, 2021.

[16] *Configure Liveness, Readiness and Startup Probes*, https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes, [Last accessed: May 9, 2022], Apr. 2022.

[17] *Counted (MicroProfile 4.0)*, https://download.eclipse.org/microprofile/microprofile-metrics-4.0/apidocs/org/eclipse/microprofile/metrics/annotation/Counted.html, 2018.

[18] I. Darwin, "Annabot: A static verifier for java annotation usage," *Advances in Software Engineering*, vol. 2010, 2009.

[19] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, 2019, pp. 93–96.

[20] *Deprecated (Java SE 17 & JDK 17)*, https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Deprecated.html, 2021.

[21] *FastAPI*, https://fastapi.tiangolo.com/, 2022.

[22] M. Fowler, *Domain Specific Languages*, https://martinfowler.com/books/dsl.html, 2010.

[23] M. Gabel and Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008, pp. 339–349.

[24] *GlassFish*, https://javaee.github.io/glassfish/, 2022.

[25] *Gradle Build Tool*, https://gradle.org, 2022.

[26] *Helidon Project*, https://helidon.io, 2021.

[27] *Jakarta EE*, https://jakarta.ee/, 2022.

[28] *Jakarta EE*, https://jakarta.ee/about/faq/, 2022.

[29] *Jakarta Enterprise Context — Jakarta EE*, https://jakarta.ee/specifications/platform/9/apidocs/jakarta/enterprise/context/package-summary.html, 2022.

[30] *Java Parser*, https://javaparser.org, 2019.

[31] *JAX-RS - javax.ws.rs package*, https://javadoc.io/doc/javax.ws.rs/javax.ws.rs-api/2.1.1/javax/ws/rs/package-summary.html, 2018.

[32] *JAX-RS — Jakarta EE*, https://jakarta.ee/specifications/restful-ws/, 2022.

[33] *JAXB — Jakarta EE*, https://jakarta.ee/specifications/xml-binding/, 2022.

[34] A. K. Jha and S. Nadi, "Annotation practices in Android apps," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2020, pp. 132–142.

[35] *JPA — Jakarta EE*, https://jakarta.ee/specifications/persistence/, 2022.

[36] *JSON*, https://www.json.org, [Last accessed: May 31, 2022],

[37] *JSON-B — Jakarta EE*, https://jakarta.ee/specifications/jsonb/, 2022.

[38] *JsonWebToken (MicroProfile JWT-Auth API)*, https://download.eclipse.org/microprofile/microprofile-jwt-auth-2.0/apidocs/org/eclipse/microprofile/jwt/JsonWebToken.html, 2021.

[39] H. J. Kang and D. Lo, "Active Learning of Discriminative Subgraph Patterns for API Misuse Detection," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021. DOI: 10.1109/TSE.2021.3069978.

[40] H. J. Kang and D. Lo, "Active learning of discriminative subgraph patterns for API misuse detection," *IEEE Transactions on Software Engineering*, 2021.

[41] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt, "Co-evolving annotations and source code through smart annotations," in *2010 14th European Conference on Software Maintenance and Reengineering*, IEEE, 2010, pp. 117–126.

[42] T. Kosar *et al.*, "Comparing general-purpose and domain-specific languages: An empirical study," *Computer Science and Information Systems*, vol. 7, no. 2, pp. 247–264, 2010.

[43] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2382–2400, 2019.

[44] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, "A systematic review of API evolution literature," *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–36, 2021.

[45] T.-D. B. Le, L. Bao, and D. Lo, "DSM: a specification mining tool using recurrent neural network based language model," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 896–899.

[46] Y. Liu, Y. Yan, C. Sha, X. Peng, B. Chen, and C. Wang, "DeepAnna: Deep Learning based Java Annotation Recommendation and Misuse Detection," *29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022.

[47] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 296–305, 2005.

[48] S. Mehrpour, T. D. LaToza, and H. Sarvari, "RulePad: interactive authoring of checkable design rules," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 386–397.

[49] mensurowary, *generating-annotation-usage-rules — GitHub*, `https://github.com/ualberta-smr/generating-annotation-usage-rules`, 2022.

[50] *Micronaut*, `https://micronaut.io`, 2022.

[51] *MicroProfile*, `https://microprofile.io`, 2022.

[52] *MicroProfile 5.0*, `https://docs.google.com/presentation/d/1PfXEOJzy8v2kGRudvknPox5elRasewv3MBeV0P4Rk6M/edit#slide=id.p`, 2021.

[53] *MicroProfile Health*, `https://download.eclipse.org/microprofile/microprofile-health-4.0/microprofile-health-spec-4.0.html`, Nov. 2021.

[54] *MicroProfile Health Check 2.0 final*, `https://github.com/eclipse/microprofile-health/releases/tag/2.0`, 2019.

[55] *MicroProfile Health#Liveness check*, `https://download.eclipse.org/microprofile/microprofile-health-4.0/microprofile-health-spec-4.0.html#_liveness_check`, Nov. 2021.

[56] *MicroProfile Metrics*, `https://download.eclipse.org/microprofile/microprofile-metrics-4.0/microprofile-metrics-spec-4.0.html`, Oct. 2021.

[57] *Monaco Editor*, `https://microsoft.github.io/monaco-editor/`, 2022.

[58] *MySQL*, `https://www.mysql.com/`, 2022.

[59] S. Newman, *Building microservices.* " O'Reilly Media, Inc.", 2021.

[60] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "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*, 2009, pp. 383–392.

[61] C. Noguera and L. Duchien, "Annotation framework validation using domain models," in *European Conference on Model Driven Architecture-Foundations and Applications*, Springer, 2008, pp. 48–62.

[62] B. Nuryyev, A. K. Jha, S. Nadi, Y.-K. Chang, E. Jiang, and V. Sundaresan, "Mining Annotation Usage Rules: A Case Study with MicroProfile," in *2022 38th International Conference on Software Maintenance and Evolution*, IEEE, 2022.

[63] *Open Liberty*, `https://openliberty.io`, 2021.

[64] *Override (Java SE 17 & JDK 17)*, `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Override.html`, 2021.

[65] *Payara*, `https://www.payara.fish`, 2022.

[66] Payara, *MicroProfile Config Guide*, `https://info.payara.fish/hubfs/MicroProfile%20Guides/MicroProfile%20Config%20Guide.pdf`, 2021.

[67] G. Piatetsky-Shapiro, "Discovery, analysis, and presentation of strong rules," *Knowledge discovery in databases*, pp. 229–238, 1991.

[68] *PMD*, `pmd.github.io`, 2022.

[69] *Quarkus*, `https://quarkus.io`, 2021.

[70] *React*, `https://reactjs.org/`, 2022.

[71] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2012.

[72] M. A. Saied and H. Sahraoui, "A cooperative approach for combining client-based and library-based API usage pattern mining," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, IEEE, 2016, pp. 1–10.

[73] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2015, pp. 33–42.

[74] *Simple Dependency Injection Example — Configuration for MicroProfile*, `https://download.eclipse.org/microprofile/microprofile-config-3.0.1/microprofile-config-spec-3.0.1.html#_simple_dependency_injection_example`, 2021.

[75] *SimplyTimed (MicroProfile 4.0)*, `https://download.eclipse.org/microprofile/microprofile-metrics-4.0/apidocs/org/eclipse/microprofile/metrics/annotation/SimplyTimed.html`, 2018.

[76] J. L. d. Siqueira, F. F. Silveira, and E. M. Guerra, "An approach for code annotation validation with metadata location transparency," in *International Conference on Computational Science and Its Applications*, Springer, 2016, pp. 422–438.

[77] *SpotBugs*, `https://spotbugs.github.io`, 2021.

[78] *Spring*, `https://spring.io`, 2022.

[79] *srcML*, `https://www.srcml.org/`, 2022.

[80] *srcType*, `https://github.com/srcML/srcType`, 2022.

[81] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel, "Extracting taint specifications for javascript libraries," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 198–209.

[82] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating Next Steps in Static API-Misuse Detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 265–275. DOI: `10.1109/MSR.2019.00053`.

[83] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating Next Steps in Static API-Misuse Detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 265–275. DOI: `10.1109/MSR.2019.00053`.

[84] *Target (Java SE 17 & JDK 17)*, `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/annotation/Target.html`, 2021.

[85] G. Uddin, F. Khomh, and C. K. Roy, "Towards crowd-sourced API documentation," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, IEEE, 2019, pp. 310–311.

[86] M. Van Someren, Y. F. Barnard, and J. Sandberg, "The think aloud method: a practical approach to modelling cognitive," 1994.

[87] A. Wasylkowski and A. Zeller, "Mining temporal specifications from object usage," *Automated Software Engineering*, vol. 18, no. 3, pp. 263–292, 2011.

[88] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 35–44.

[89] *WildFly*, `https://www.wildfly.org/`, 2022.

[90] R. Wuyts *et al.*, "A logic meta-programming approach to support the co-evolution of object-oriented design and implementation," Ph.D. dissertation, Citeseer, 2001.

[91] *XPath*, `https://developer.mozilla.org/en-US/docs/Web/XPath`, Mar. 2022.

[92] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus, "Characterizing the Usage and Impact of Java Annotations Over 1000+ Projects," *arXiv preprint arXiv:1805.01965*, 2018.

[93] H. Zeng, J. Chen, B. Shen, and H. Zhong, "Mining API Constraints from Library and Client to Detect API Misuses," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2021, pp. 161–170.

[94] W. Zhang, H. Liao, and N. Zhao, "Research on the FP growth algorithm about association rule mining," in *2008 international seminar on business and information management*, IEEE, vol. 1, 2008, pp. 315–318.

[95] Y. Zhang, "Checking metadata usage for enterprise applications," Ph.D. dissertation, Virginia Tech, 2021.

[96] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *European Conference on Object-Oriented Programming*, Springer, 2009, pp. 318–343.