SpecTackle - A Specification Mining Experimentation Platform

Robert Heumüller, Sebastian Nielebock, Frank Ortmeier
Chair of Software Engineering
Faculty of Computer Science
Otto-von-Guericke University
Magdeburg, Germany
{*robert.heumueller,sebastian.nielebock,frank.ortmeier*}*@ovgu.de*

***Abstract*—Nowadays, API Specification Mining is an important cornerstone of automated software engineering. In this paper, we introduce SpecTackle, an IDE-based experimentation platform aiming to facilitate experimentation and validation of specification mining algorithms and tools. SpecTackle strives toward (1) providing easy access to various specification mining tools, (2) simplifying configuration and usage through a shared interface, and (3) in-code visualization of pattern occurrences. The first version supports two heterogeneous mining tools, a third-party graph-based miner as well as a custom sequence mining tool. In the long term, SpecTackle envisions to also provide ground-truth benchmark projects, a unified pattern meta-model and parameter optimization for mining tools.**

***Keywords*-Tool, Specification Mining, Experimentation, Validation, IDE**

I. Introduction

Specification mining has evolved to become a cornerstone of automated software engineering and researchers have proposed numerous data-mining based approaches for inferring different types of patterns from source code. As in any other field of research, understanding, experimenting with, and building on existing approaches is an essential step toward advancing the field of specification mining. With SpecTackle, we aim to facilitate these steps by providing an experimentation platform which supports easy, streamlined access to various existing specification mining tools.

The primary element of distinction between different types of algorithms is their specification meta-model, i.e. what type of patterns they can infer. A survey and categorization of different specification mining algorithms can be found in [1]. With SpecTackle, we focus on the largest category of algorithms which deals with finding temporal constraints on how APIs elements, typically methods, may correctly and safely be used [2]. In the following, we refer to this type of specification simply as API usage patterns.

The major challenge in specification mining has always been the low accuracy of the algorithms [3], in particular the prohibitively high false-positive rates. Comparing accuracy in terms of precision (and to a limited degree also recall) is therefore the standard way of comparing different approaches. However, in the absence of ground-truth specifications, evaluating these metrics often requires time-consuming manual classification, i.e. discriminating between true, valuable patterns and false-positives. This distinction is not always obvious at first glance, since false-positives can be artifacts of the underlying data, e.g. auto-generated code, or issues with the mining algorithm and respective parameterization. The analysis is further complicated by the fact that, each existing tool typically has a unique interface, configuration mechanism, and representation of results.

With SpecTackle we aim to assist researchers in the specification mining and automated software engineering field by providing a platform which unifies experimentation with different existing specification mining tools. In particular, SpecTackle should help in understanding the characteristics of different mining tools, understanding the influence of parameters, as well as comparing the results of different tools. In its current version, SpecTacke features

- easy, streamlined access to two different specification mining tools,
- simplified tool configuration and usage through a shared interface,
- in-code visualization of pattern occurrences.

Built as a plugin for the IntelliJ platform, SpecTackle can analyze target projects using different mining tools, provides uniform parameterization and in-code visualization of patterns. While SpecTackle is still in an early stage of maturity, some immediate benefits can already be anticipated. Therefore the first benefit of using SpecTackle is that, as a curated collection of specification mining tools, it frees researchers of the necessity of searching out tools online. While a large number of tools have been published, in our experience many of them are actually unavailable, outdated or poorly documented. We envision shipping SpecTackle with batteries-included support for multiple different mining tools. Therefore its architecture allows the integration of heterogeneous usage-pattern representations. Currently, we support one graph-based tool, GrouMiner [4], as well one sequence-based miner. The second benefit is that researchers do not have to be familiar with the usage of different tools due to the homogeneous integration in the IDE plugin: selecting, configuring and executing algorithms work the same for any mining tool. The third benefit is that the results are always presented in the same way, no matter which mining

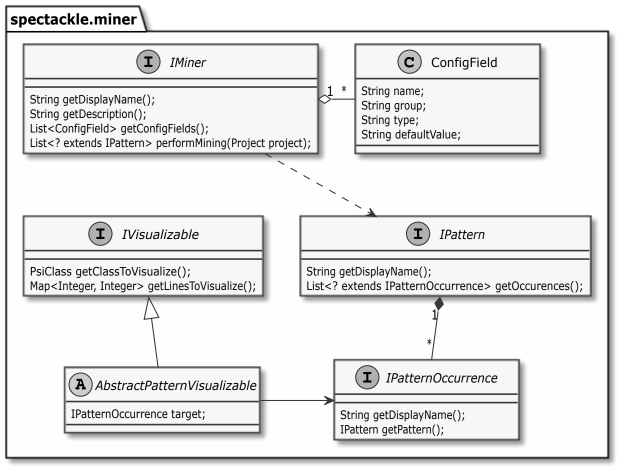
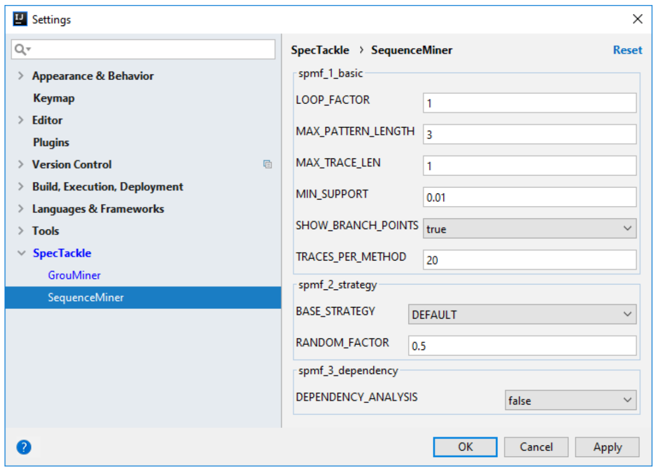

spectackle.miner
I IMiner
String getDisplayName();
String getDescription();
List<ConfigField> getConfigFields();
List<? extends IPattern> performMining(Project project);
1 *
C ConfigField
String name;
String group;
String type;
String defaultValue;
I IVisualizable
PsiClass getClassToVisualize();
Map<Integer, Integer> getLinesToVisualize();
I IPattern
String getDisplayName();
List<? extends IPatternOccurrence> getOccurences();
1
*
A AbstractPatternVisualizable
IPatternOccurrence target;
I IPatternOccurrence
String getDisplayName();
IPattern getPattern();

Figure 1. SpecTackle Interfaces



SpecTackle › SequenceMiner

spmf_1_basic

LOOP_FACTOR	1
MAX_PATTERN_LENGTH	3
MAX_TRACE_LEN	1
MIN_SUPPORT	0.01
SHOW_BRANCH_POINTS	true
TRACES_PER_METHOD	20

spmf_2_strategy

BASE_STRATEGY	DEFAULT
RANDOM_FACTOR	0.5

spmf_3_dependency

DEPENDENCY_ANALYSIS	false

Figure 2. SpecTackle Miner Settings Dialog

tool was used. The in-code visualization also facilitates understanding of patterns, as they are always shown directly in the relevant context. SpecTackle is available for download at *https://cse.cs.ovgu.de/cse/spectackle/*.

II. Tool Overview

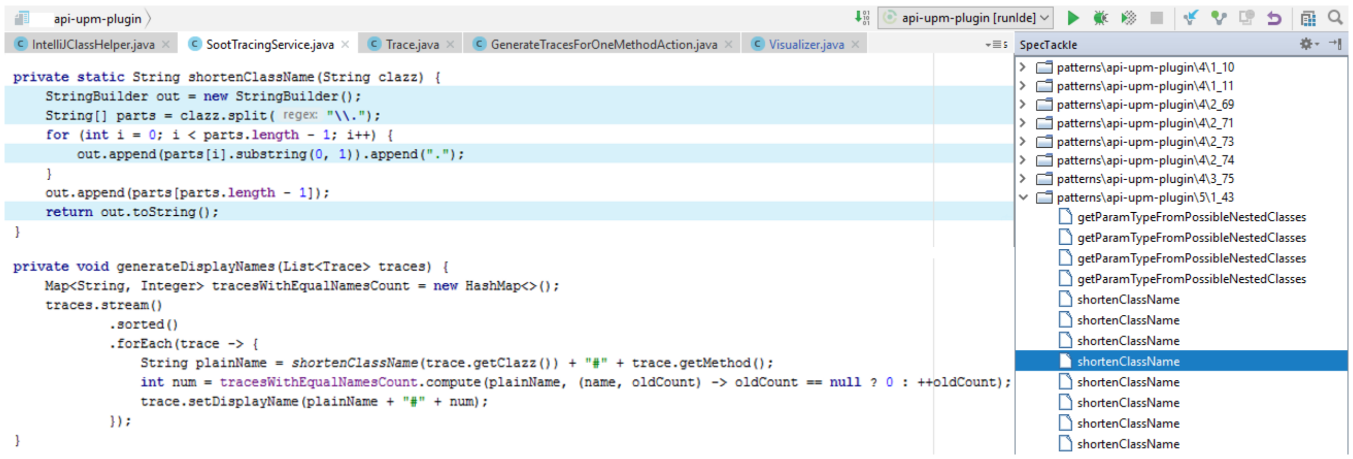
In this section, we introduce SpecTackle's achitecture for integrating heterogeneous specification miners and give an impression of the plugins currently supported features and usage.

A. *SpecTackle Architecture*

SpecTackle's architecture is primarily designed to make integrating new mining backends as simple as possible. Developers who want to provide a plugin for a new mining backend only need to implement four interfaces shown in Figure 1. The `IMiner` interface wraps the actual mining algorithm. Besides the descriptive functions, it needs to be able to retrieve a list of the available configuration parameters, which are used to populate e.g. the settings dialog. The `performMining` method bootstraps the actual mining process and therefore requires a reference to the target project as its input. Any preprocessing required by the miner must also be encapsulated in this method and upon completion, it must return a list of `IPattern` instances. The generic definition of `IPattern` does not enforce any particular pattern meta-model such as sequences or graphs. Instead, we initially define patterns as collections of pattern occurrences (`IPatternOccurrence`), which represent actual instances of API usage patterns in source code. In combination with the `AbstractPatternVisualizable` decorator, any type of API usage pattern can be visualized directly in source code. For this purpose, visualizable implementations must define a method to locate the source file to open, as well as a method which returns the line numbers and color depths for the highlighting. The plugins for `GrouMiner` and the `SequenceMiner` follow exactly this schema. This way, the SpecTackle framework can take care of populating the configuration menu, starting the mining process, displaying the patterns and visualizing the pattern occurrences in source code.

B. *Using SpecTackle*

SpecTackle seamlessly integrates into JetBrains' IntelliJ Idea IDE. Before starting an analysis, the first step is to select the desired mining tool and set the required parameters. For this purpose, we extended IntelliJ's default settings dialog with a section for configuring the SpecTackle plugin. Besides selecting the active miner, there is a subpage for the settings of each mining tool. With regard to the particular mining tools, these pages show appropriate controls for specifying the minimum support, maximum pattern sizes or auxiliary tool paths. Figure 2 shows the configuration page for the sequence miner. To start the mining process for the selected tool and respective parameterization, only a single click in the analysis menu is needed. Once completed, the results pop up in the SpecTackle view, as shown for GrouMiner on the right-hand side of Figure 3. In this view, the folder nodes correspond to patterns, whereas each leaf node represents a single occurrence of a pattern in the analyzed source code. When an occurrence node is selected, the corresponding source file is opened in the default editor. In the editor, the lines corresponding to the selected pattern occurrence are highlighted by a blue background. If a line features more than once in an occurrence, it is emphasized using a darker color. Currently, two mining backends are supported: (1) GrouMiner [4] performs frequent subgraph mining on *Graph Based Object Usage Models*, a program representation related to control flow graphs and (2) our own sequential pattern miner which uses CFGs to statically enumerate execution traces for frequent



```java
private static String shortenClassName(String clazz) {
    StringBuilder out = new StringBuilder();
    String[] parts = clazz.split( regex: "\\.");
    for (int i = 0; i < parts.length - 1; i++) {
        out.append(parts[i].substring(0, 1)).append(".");
    }
    out.append(parts[parts.length - 1]);
    return out.toString();
}

private void generateDisplayNames(List<Trace> traces) {
    Map<String, Integer> tracesWithEqualNamesCount = new HashMap<>();
    traces.stream()
            .sorted()
            .forEach(trace -> {
                String plainName = shortenClassName(trace.getClazz()) + "#" + trace.getMethod();
                int num = tracesWithEqualNamesCount.compute(plainName, (name, oldCount) -> oldCount == null ? 0 : ++oldCount);
                trace.setDisplayName(plainName + "#" + num);
            });
}
```

```
> patterns\api-upm-plugin\4\1_10
> patterns\api-upm-plugin\4\1_11
> patterns\api-upm-plugin\4\2_69
> patterns\api-upm-plugin\4\2_71
> patterns\api-upm-plugin\4\2_73
> patterns\api-upm-plugin\4\2_74
> patterns\api-upm-plugin\4\3_75
v patterns\api-upm-plugin\5\1_43
      getParamTypeFromPossibleNestedClasses
      getParamTypeFromPossibleNestedClasses
      getParamTypeFromPossibleNestedClasses
      getParamTypeFromPossibleNestedClasses
      shortenClassName
      shortenClassName
      shortenClassName
      shortenClassName
      shortenClassName
      shortenClassName
      shortenClassName
      shortenClassName
```

Figure 3. Pattern Visualization: GrouMiner Patterns

subsequence mining with SPMF [5]. The example in Figure 3 shows a typical usage pattern occurrence for Java's `StringBuilder` class: splitting a target string, modifying the resulting segments, and recombining them by appending to a `StringBuilder`.

III. Related Work

To the best of our knowledge, SpecTackle is the first framework which aims to assist researchers by providing homogenized access to different and potentially heterogeneous specification mining tools. Therefore, the relevant related work encompasses primarily papers which give ideas on how cross-tool evaluations can be performed. These can provide valuable input to the SpecTackle project but do not have the aspiration to be an extensible, re-usable experimentation platform.

Validation Frameworks.: Pradel et al. developed a framework for analyzing three different miners which infer specifications in the form of finite-state automata (FSM). Their benchmark consists of 32 Java JDK classes and 32 corresponding, manually-formalized specifications. Then they analyzed the miners' accuracy (precision, recall, and F1-score) in inferring these specifications from twelve open-source projects. In contrast to this validation framework, SpecTackle also strives to achieve comparability between heterogeneous pattern types, not just FSMs.

In a recent study on the effectiveness of detecting API misuses, the framework MuBenchPipe is used together with the MuBench dataset [6], [7]. They analyzed the ability of multiple miners (1) to find the known API misuses and (2) to assess whether further detected misuses represented actual bugs. MuBenchPipe is very useful in assessing misuse detection and also supports manual labeling of false-positives. However, it does not help in finding the causes of false-positives. In contrast, SpecTackle strives to find and mitigate weaknesses in mining approaches and, in this respect, is not restricted to any particular use case.

Benchmarks.: For SpecTackle, the manually extracted JDK specification by Legunson et al. [8] could be particularly beneficial to provide some ground-truth for experimenting with different miners. Other approaches used the DaCapo benchmark [9], the MuBench benchmark [6], or the benchmark from the Boa project [10]. However, as these are not specifically tailored towards API specification mining, they are probably not as immediately useful for SpecTackle.

Another interesting source for benchmark projects are Android Apps, since they are structurally very similar and exhibit a natural repetitiveness in their code. Popular resources for such applications are Android Drawer[1] and F-Droid[2].

IV. Conclusion

In this paper, we presented SpecTackle, an experimentation workbench for researchers in the specification mining domain. Our goal is to help researchers to improve on existing specification mining tools by providing them with a framework which helps its users to better understand how existing approaches work, how they compare to each other, and what the influences of different parameterizations are. In the previous sections, we described SpecTackle's extensible architecture which allows the integration of heterogeneous mining tools, as well as its current capabilities of configuration and pattern visualization.

In addition to this primary goal, we also want to use SpecTackle to increase the awareness of how important it is to publish not only algorithm descriptions but also the practical tools and datasets. Both our own experiences and those of fellow-researchers indicate, that there is significant potential for improvement in this point.

[1]https://www.androiddrawer.com/
[2]https://f-droid.org/

Future Work

While the current features already constitute significant practical value, they are only the first steps in our vision for the SpecTackle project.

Toward better efficiency in manually assessing API usage patterns: Starting from the current features, we envision further extensions. First, we want to create a command-line distribution for use in headless environments such as high-performance servers. This distribution will export the mined patterns into an archive, which can later be imported into the IDE for visualization. Second, we want to allow users to annotate patterns as true- or false-positives, which would be convenient for creating the ground-truth benchmarks.

Toward heterogeneous specification miners in a homogeneous environment: Today, SpecTackle features two mining backends and a simple architecture for integrating additional source-code based mining tools. In future we aim to support many additional backends, which could also be created by other members of the research community. Next, we see the potential for an automated parameter optimization system. This would greatly help in determining appropriate parameters for different mining tools, which otherwise requires a deep understanding of the data and the underlying algorithms. However, optimization requires an appropriate loss-function, for example, based on precision, recall, or the F1-score. In any case, the computation requires either a benchmark with ground truth or some approximation thereof. Further, dynamic specification mining tools require not only the source-code of the benchmark projects but also appropriate input data. Sometimes, automatic test cases can provide a valid source of program input. If none exist, or if the existing tests are insufficient, we could consider automatic test-generation techniques, e.g. coverage-driven tools like EvoSuite [11].

Toward qualitative and quantitative comparability of specification miners: An important challenge in automatically comparing specification mining tools is matching equivalent usage patterns found by multiple miners. If this is not possible, one is restricted to quantitatively comparing accuracy statistics, essentially *how many patterns* were discovered by the candidate tools, without gaining insight into *which sets of patterns* were discovered by which tool.
be found in [12]. A common meta-model would, in turn,

Matching equivalent patterns across heterogeneous pattern types requires the semantics-preserving transformation to a common meta-model. Related work proposes several pattern types including frequent item-sets, frequent sequences, association rules, regular expressions, finite-state automata, temporal logic formulas, frequent sub-trees, and frequent sub-graphs [1]. We believe that a graph depicting temporal constraints between API elements could serve as the unifying meta-model because all of the aforementioned models can be transformed into a semantically equivalent graph representation. Ideas on how to match the semantically equivalent graphs expressed in a common meta-model can increase the value of standard benchmarks. We aim to outfit SpecTackle with benchmarks for different applications, such as the aforementioned work on JDK specifications by Pradel et al. [13]. This way, SpecTackle could be used to perform fully automated cross-tool evaluations.

References

[1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API Property Inference Techniques," *TSE*, vol. 39, no. 5, 2013.

[2] C. Le Goues and W. Weimer, "Measuring code quality to improve specification mining," *IEEE Transactions on Software Engineering*, vol. 38, pp. 175–190, 2012.

[3] M. Gabel and Z. Su, "Testing mined specifications," in *20th FSE*. ACM, 2012.

[4] 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 *7th ESEC/FSE*. New York, NY, USA: ACM, 2009.

[5] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, and V. S. Tseng, "SPMF: A Java Open-Source Pattern Mining Library," *JMLR*, vol. 15, no. 1, 2014.

[6] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A Benchmark for API-Misuse Detectors," in *13th MSR*, 2016.

[7] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A Systematic Evaluation of Static API-Misuse Detectors," *TSE*, vol. Early Access, 2018.

[8] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How Good Are the Specs? A Study of the Bug-finding Effectiveness of Existing Java API Specifications," in *31st ASE*. New York, NY, USA: ACM, 2016.

[9] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *21st OOPSLA*. New York, NY, USA: ACM, 2006.

[10] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: Ultra-Large-Scale Software Repository and Source-Code Mining," *TOSEM*, vol. 25, no. 1, Dec. 2015.

[11] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," in *19th ESEC/FSE*. New York, NY, USA: ACM, 2011.

[12] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh, "Synergizing Specification Miners Through Model Fissions and Fusions (t)," in *ASE*. IEEE, 2015.

[13] M. Pradel, P. Bichsel, and T. R. Gross, "A Framework for the Evaluation of Specification Miners Based on Finite State Machines," in *26th ICSM*, Sept 2010.