Hermes: Assessment and Creation of Effective Test Corpora

Michael Reif Michael Eichberg Ben Hermann Mira Mezini
Technische Universität Darmstadt, Germany
{lastname}@cs.tu-darmstadt.de

Abstract

An integral part of developing a new analysis is to validate the correctness of its implementation and to demonstrate its usefulness when applied to real-world code. As a foundation for addressing both challenges developers typically use custom or well-established collections of Java projects. The hope is that the collected projects are *representative* for the analysis' target domain and therefore ensure a sound evaluation. But, without proper means to understand how and to which degree the features relevant to an analysis are found in the projects, the evaluation necessarily remains inconclusive. Additionally, it is likely that the collection contains many projects which are – w.r.t. the developed analysis – basically identical and therefore do not help the overall evaluation/testing of the analysis, but still cost evaluation time.

To overcome these limitations we propose *Hermes*, a framework that enables the systematic assessment of given corpora and the creation of new corpora of Java projects. To show the usefulness of *Hermes*, we used it to comprehend the nature of the projects belonging to the Qualitas Corpus (QC) and then used it to compute a minimal subset of all QC projects useful for generic data- and control-flow analyses. This subset enables effective and efficient integration test suites.

CCS Concepts **• Software and its engineering → General programming languages**; **• Theory of computation** → *Program analysis*

Keywords Benchmark Suites, Test Corpora, Program Analysis, Java

1. Introduction

Whenever a new static or dynamic analysis is developed, it is necessary to test its implementation and to evaluate its usefulness. For both tasks, researchers typically use large(r) collections of projects. When a well-established corpus of projects is used [4, 11, 16] the respective results are often easier to compare with the state-of-the-art. But, given the breadth of proposed code analyses, e.g., data-flow analyses [3, 10], concurrency analyses [2], call-graph algorithms [1, 14], or whole static analysis frameworks [7, 18], it becomes evident that most authors either have to customize existing corpora or have to create their own collection which targets their specific needs and goals.

Another major drawback of using an established suite is that it may have been build for another purpose and/or is outdated; e.g., Terra et al. [17] published their own version of the Qualitas Courpus [16] where they replaced outdated projects by newer versions. Other frequently used benchmarks suites like SecuriBench [11] or DaCapo [4] also did not receive updates for quite a while. This reduces their attractiveness since recent features may not be covered at all and outdated features may be over represented. The latter may skew evaluation results. For instance, when the new `invokedynamic` bytecode instruction was introduced in Java 7, it became necessary to explicitly support it and this only happened with Java 8 and, e.g., Scala 2.12. The latter was released 6 years after the release of Java 7.

When researchers create their own test corpora they often do so using a combination of synthetic test projects and open-source projects. Most often the authors take programs from different domains, developed by different people. The hope is that those programs vary in terms of language features, programming styles, and size and are therefore *representative*. At a closer look, it is unclear to which degree the constructed corpus of projects supports the evaluation goal and to which degree all relevant properties, like occurrences of programming language features, the usage of certain APIs, and problematic design patterns, can be found in the projects. This lack of knowledge of the properties of the used projects generally leads to questionable evaluation results.

To facilitate comprehension of existing test corpora and to ease the construction of new evaluation or integration test corpora, we propose *Hermes*. *Hermes* provides a generic framework for the assessment and construction of evaluation corpora. Based on an extensible set of queries, *Hermes* provides a comprehensive overview of features of Java bytecode projects whose understanding is critical for many analysis projects. Based on the results of evaluating the queries, *Hermes* can then compute the minimal set of those projects that are necessary to cover all relevant features. Using this set it is possible to efficiently test and evaluate analyses.

This paper represents our initial work, with the following specific contributions:

- *Hermes*, a framework for the assessment of a given corpus of Java projects and for the computation of a minimal corpus regarding the evaluated features.
- An initial set of feature queries to collect, provide, and comprehend information about a project.
- A first evaluation that shows the usefulness of the approach for both: the assessment and creation of test corpora.

To download Hermes go to: `www.opal-project.de`

Next, we will discuss the construction and the state of currently available benchmark suites and test corpora. After that, we will discuss the proposed approach and its realization. An evaluation and case study is presented thereafter. The paper ends with a conclusion and future work.

2. State-of-the-art

Comparing approaches, such as static or dynamic analyses which belong to the same research area can be performed with standardized test or evaluation corpora that suit the respective target domain. Therefore, most research papers try to rely on established copora. However, the creation of an unbiased, representative, and long-lived corpus is difficult. The lack of such corpora in various research areas has led authors to build their own corpora, which differ in particular in two dimensions: 1) criteria for project selection and 2) evaluation goals.

Blackburn et al. [4] created the DaCapo benchmark suite which primarily targets Java performance evaluation. They also discussed how to develop and test such corpora. They determined that their benchmark should consist of diverse and easy to use real-world applications. Beside these criteria, they identified a set of dynamic and static software metrics, to assess a project's performance behavior.

Tempero et al. [16] first identified size, content, representativeness, and permanence as key aspects for project selection. Based on these criteria, they created a curated code collection of 100 Java projects. These projects range from libraries over application frameworks to different kinds of applications. The focus of the Qualitas Corpus is on aiding researchers to carry out empirical studies of code.

In SecuriBench, Livshits et al. [11] selected large web applications, which have known security vulnerabilities. Consequently, SecuriBench can be used to evaluate static and dynamic security analyses. Other corpora like DroidBench [3], PointerBench [15], or the Darmstadt Library Corpus (DLC) [14] provide data sets with yet different goals as well as different criteria to assemble the corpus.

Hence, all of the previously introduced corpora were designed with one specific goal in mind, but their suitability w.r.t. their original goals often remains unknown. Especially the inclusion of (yet) another real-world project into a corpus is repeatedly justified based on its perceived difference, rather than based on qualitative measures. Additional measures like goal-relevant metrics, e.g., the degree to which a project uses Java reflection, or the occurrence of certain properties – already used by some corpora – could be used to assess the suitability regarding a given goal. Furthermore, most of the previously presented corpora are no longer maintained, which may indicate the difficulty in keeping them up-to-date. To address the latter short-coming, efforts have been made to (semi-)automate the process of corpora creation.

Dujmović et al. [6] presented a parameterized approach to automatically generate fully synthetic programs that already allow benchmarking and testing, but cannot be used to evaluate an approach on real-world applications.

Do et al. [5] present Automatic Benchmark Management (ABM), a methodology for mining software repositories to semi-automatically extract an up-to-date, updatable, and representative corpus that includes applications from various domains. However, no assessment is done including the projects and it may be the case that many projects do not have relevant differences.

These corpora are good starting points to build up-to-date, comprehensive evaluation- or test corpora and, once a large set of projects is available, *Hermes* can be used to assess those projects regarding relevant features. After executing all queries over all projects, *Hermes* can be used in a second step, to compute a minimal sub corpus that ensures complete feature coverage.

3. Hermes

Hermes is an extensible, configurable framework for the comprehensive assessment of a given set of projects w.r.t. a wide range of different features. The projects have to consists of Java bytecode and can be either standalone programs or Java libraries; the set of all projects forms the *base corpus*. The extension of a feature for a given project is then determined by a respective query. An example feature of a project could be the use of a specific Java API (e.g., Java Reflection API, Unsafe or JDBC), the occurrence of methods with non-reducible control-flow graphs (primarily in explicitly obfuscated code) or the usage of lambda expressions. In the latter case, the query would collect all respective instructions in the project's code.

Using the results of the evaluation of all queries, *Hermes* can then automatically compute the *optimal corpus* which ensures that all features are found in at least one project. This subset can then be used for effective and efficient testing and evaluation purposes.

```
1  {”org”: { ”opalj”: { ”hermes”: { ”projects”: [
2        {
3          ”id”: ”Apache ANT 1.7.1 − Javac 6”,
4          ”cp”: ”../../projects/Apache ANT 1.7.1.jar”,
5          ”libcp”: ”../../dependenciess/Apache ANT 1.7.1.jar”,
6          ”libcp_default”: ”JRE”
7        },
8        {
9          ”id”: ”argouml−excerpt”,
10         ”cp”: ”../../projects/argouml−excerpt.jar”
11       }]}}}}
```

Listing 1. Example configuration file (.json) that specifies the corpus projects.

3.1 Approach

Hermes is based upon the Java bytecode analysis framework OPAL [7]. OPAL is written in Scala and provides multiple representations of Java bytecode which enables lowest level queries but also queries at a high abstraction level. Additionally, OPAL provides useful abstractions such as a `Project` and also provides a wide range of standard functionalities like computing control-flow graphs and call graphs. This facilitates the implementation of feature queries which range from metrics to data- and control-flow dependent metrics. The computation of the optimal corpus is done using the constraint programming library Choco [13].

In the following, we describe the main components of *Hermes* along with the steps a user has to take to assess and optimize a *base corpus*.

Corpus configuration. Before running *Hermes*, all projects of the *base corpus* have to be specified. Listing 1 shows an example configuration for a small corpus consisting of two projects. Each project specification consists of a unique *id* (line 3 and 9) and a specification of its classpath (*cp* line 4 and 10). Additionally, the two optional attributes *libcp* (line 5) and *libcp_default* (line 6) can be used to specify the project's libraries. The first one specifies the paths to the libraries' jars and *libcp_default* is used to add a dependency to a predefined library to the project. The available default libraries are the *current* Java Runtime Environment (JRE) as a whole or just the `rt.jar`[1]. Library class paths need to be specified whenever some feature query requires information that cannot be extracted from the project alone, e.g., feature queries related to the inheritance hierarchy generally require a complete type hierarchy.

Feature queries. *Hermes* also requires that the queries, which should be evaluated, are configured. By default, all available queries will be evaluated, but this can be changed and new queries can also be specified. The set of queries, should – in general – be selected with a concrete analysis and test/evaluation goal in mind. For example, if a test corpus for integration testing of a static analysis should be created, it might be important to ensure that *all* language-specific features are found at least once in the given projects. If the evaluation goal is the scalability of the analysis, it may be more important to ensure that specific features occur with a certain frequency. Listing 2 shows a configuration that enables the queries in Line 2, 4, and 5 and which disables the query in Line 3. Each entry specifies the fully qualified name of the class that implements the query (*query*) and whether the given query should be executed or not (*activate*). New queries can simply be added to the configuration analogously.

[1] Here, *current* refers to the one used when running *Hermes*.

```
1  org.opalj.hermes.queries = [
2  { query = queries.Metrics, activate = true }
3  { query = queries.MethodsWithoutReturns, activate = true}
4  { query = queries.JDBCAPIUsage, activate = false }
5  { query = queries.MethodTypes, activate = true } ]
```

Listing 2. *Hermes*' configuration of enabled and disabled feature queries.

Corpus evaluation and visualization. Given a complete configuration, we can then start *Hermes*. *Hermes*' UI provides an overview of the current state of the evaluation, provides descriptions of the activated queries, and shows basic size metrics related to the projects.

Additionally, the evaluation of each activated feature query for each project belonging to the specified base corpus is directly started. As soon as a feature query was evaluated, *Hermes* shows the resulting number of feature occurrences and makes it possible to jump to concrete occurrences of the feature in the respective project's code base – if supported by the query. In general a query can report feature occurrences at the class, method, or instruction level. Being able to navigate to concrete feature occurrences is helpful when developing new feature queries, but also if a more detailed understanding of the feature in the context of a specific project is required. The amount of location information that is kept is configurable and managed by *Hermes* to ensure that very large test corpora such as the Qualitas Corpus can successfully be evaluated.

3.2 Feature Queries

A feature query is a static analysis that is given a project as input and then collects all feature extensions of one or multiple closely related features. For example, it is possible to write a query which collects all Java 7 class files found in a specific project and another one for Java 8 class files. Alternatively – and also more efficiently – it is possible to write a single feature query that analyzes every class file once and adds every class file to its respective feature category. To ensure that all features are uniquely identifiable across

Table 1. Available feature queries including their category, number of unique features and a short description.

feature query	category	# features	description
BasicMetrics	metrics, control flow	15	Extracts the following basic metrics: methods per class, fields per class, the number of children (NOC), and McCabe and groups them per complexity category (e.g., in case of McCabe: linear methods, simple methods (2 to 3 paths), complex methods (more than 3 paths).
BytecodeInstructions	JVM features	201	List of all Java bytecode instructions as defined in the Java Virtual Machine Specification (Java 1.1 up to Java 8).
ClassFileVersion	JVM features	6	Extracts the class file version (Java 1.1 up to Java 9) of each class file belonging to the project where each version is a single feature.
ClassLoaderAPIUsage	API usage, inheritance	5	Extracts the usage of methods of `java.lang.ClassLoader` in a project and also checks whether custom class loaders exist.
ClassTypes	language features	10	Extracts the information about the type of the specified class; e.g., how may concrete classes, annotations, interfaces, interfaces with default methods, or Java 9 modules are defined.
JavaCryptoArchitecture-Usage	API usage	8	Extracts information about the usage of core classes and interfaces, for instance ciphers, keys, or signatures, from the Java Crypto Architecture (JCA) according to the official reference guide.
JDBCAPIUsage	API usage	5	Extracts information about the usage of Java's JDBC API and SQL statement kinds.
MethodsWithoutReturns	control flow	2	Extracts whether a method either never returned normally, e.g., by throwing an exception, or has a real infinite loop without any possibility to return.
MethodTypes	language features	9	Extracts the information about the type of the specified methods; e.g., whether a method is native, synchronized, or is a varargs method.
ReflectionAPIUsage	API usage	12	Derives which methods/functionality of Java's classical Reflection API is used within a project
SystemAPIUsage	API usage, capabilities	8	Extracts the usage of API methods that are related to the state of the JVM, capabilities [9], or used to access the underlying operating system; e.g., spawning an external process, playing sound, or working with the `java.lang.SecurityManager`.
TrivialReflectionUsage	API usage, data flow	1	Counts the number of cases where `Class.forName` calls can be trivially resolved, because the respective String(s) are directly available.
UnsafeAPIUsage	API usage	19	Derives usage information about `sun.misc.Unsafe` according to the classification of Mastrangelo et al. [12].

```
1  trait FeatureQuery {
2
3    def featureIDs: Seq[String]
4
5    def apply[S](
6       projectConfiguration: ProjectConfiguration, String]
7       project: Project[S], String]
8       rawClassFiles: Traversable[(da.ClassFile, S)] String]
9    ) : TraversableOnce[Feature[S]]
10 }
```

Listing 3. Scala trait that has to be implemented by all feature queries.

all feature queries, each query assigns a unique id to each derived feature.

All feature queries have to implement the `FeatureQuery` interface, which defines the two functions shown in Listing 3. The first function `featureIDs` (Line 3) defines a list of unique feature ids where each id represents the name of a derived feature. The second function (`apply` - Line 5) defines the query itself. The input for the static analysis is the project configuration (Line 6), OPAL's representation of a `Project` (Line 7), and a raw one-to-one representation of the project's Java class files (Line 8). The raw representation supports queries which need to work on unprocessed class files; e.g., those that want to analyze the constant pool in detail. The representation provided by the project enables higher level code analyses, such as control- and data-flow analyses or abstract code interpretation.

The currently available feature queries are listed in Table 1 together with the number of derived features and a short description of each feature group. The available queries demonstrate the variety of possible analyses: they reach from basic *API usage* queries, which can be used to select projects for API misuse detection, specification mining, or injection analyses, over *JVM* and *language features* based queries – e.g., to find suitable integration test corpora – up to *control*- and *data-flow* analyses. The latter can, e.g., be used to get some understanding how Java reflection is used.

3.3 Computing an Optimal Corpus

After all queries have been evaluated for all projects it is possible to let *Hermes* compute the subset of all projects which has the overall minimal number of methods (*optimization goal*) and which ensures that every feature occurs at least once in some project (*constraint*). I.e., *Hermes* would prefer two small projects with, e.g., 2 methods each over one project with 10 methods. Minimization of the overall number of methods is done because in most cases it better reflects the overall effort that is necessary when the corpus is eventually used for evaluation or test purposes.

For more elaborated used cases, it is possible to export the computed results using a CSV file and to perform some external post processing.

4. Evaluation

In the following, we describe the evaluation of *Hermes* for the two use cases: "*Comprehending a test corpus*" and "*Generation of an effective integration test suite*".

All measurements were done on a Mac Pro with a Xeon E5 CPU with 8 cores@3GHz. The Java Virtual Machine (Java 8 Update 121) was given 24GB of memory.

4.1 Comprehending Test Corpora

To understand the nature of the projects contained in the latest release of the Qualitas Corpus [16] (QC) from September 2013, we run *Hermes* using all queries against all projects and inspected the result. As expected – given the release date of QC – none of the projects used any Java 8 features. More surprisingly, none of the projects used the JavaFX framework already introduced in 2008. This indicates that even though the corpus already contains over 100 projects some domains are not well represented. Furthermore, only one (Hibernate) of the 100 included projects uses Java 7[2] features. Overall, this preliminary analysis suggests that using the Qualitas Corpus to evaluate or test analyses that support Java features released after 2011 is not meaningful/possible.

[2] Java 7 was released in 2011 and already two years old when the updated corpus was created.

4.2 Generating Integration Test Suites

For the second evaluation, we used *Hermes* to compute an optimal test corpus based on the Qualitas Corpus [16] (QC) for generic integration testing purposes; i.e., we used *Hermes* to compute the subset of all QC projects that should enable us to perform effective and efficient integration testing of general static and dynamic analyses. The concrete goal for the evaluation was to use the minimal set of projects for testing the analysis described in the paper "Hidden Truths in Dead Software Paths" [8]. The core part of that analysis is a very generic data- and control-flow analysis and it should be able to handle all valid Java bytecode. Using this minimal set of projects should give us basically the same level of confidence in our developed analysis as using all QC projects.

The first step of this evaluation was to run *Hermes* against all projects using all queries. After all queries were evaluated for all projects, we let *Hermes* compute the minimal set of projects which (a) has the minimal overall number of methods and (b) ensures that every feature at least occurs once in some project[3]. The set of projects computed by *Hermes* consists of the following five projects: `joggplayer`, `jchempaint`, `hibernate`, `quilt`, and `nakedobjects`.

The second step was to determine the overall coverage of the code of the paper's core control- and data-flow analysis. We measured the coverage twice: Once, running the analysis against all 100 projects of the Qualitas Corpus and once running it only against the automatically determined set of 5 projects. The time to run the analysis against all projects was $1006s(\approx 16.77min)$ while it took $169s(\approx 2.82min)$ for the selected projects. I.e., just using the selected projects is nearly 6 times faster. However, the code coverage is just 1.06% better in the latter case when we use all QC projects instead of the five selected ones. A closer analysis of the coverage data revealed that the difference is due to advanced exception handling and more elaborated array-based accesses found in projects which did not belong to the test corpus of five projects.

4.3 Discussion

Overall, we can conclude that it is already possible to use *Hermes* to get a better understanding of available test corpora and also to compute test corpora that enable effective testing. Furthermore, given the very primitive nature of the available queries and the achieved quality of the results, it is evident that we don't need complex queries to compute effective test corpora.

Additionally, by adding further queries related to exceptions/exception handling and to array accesses, it will be possible to compute a test corpus that is most likely still much smaller than the complete QC, but which will be as effective when testing general data- and control-flow analyses.

[3] Recall that features which are not found at all across all given projects, such as those related to Java 8 features in case of the QC, are simply ignored.

5. Future Work & Conclusion

Testing and evaluation are essential and generally very time consuming tasks that are part of the development of every new analysis. Both tasks are typically done using test corpora consisting of large(r) collections of projects. But as discussed, without explicit tool support it is impossible to know if the selected projects are actually having the desired/necessary features and it is also impossible to know which projects are actually useful or which just test/evaluate the same functionality over and over again.

To address these issues, we have proposed *Hermes*– a generic framework that facilitates the assessment of a given set of Java bytecode projects. *Hermes* contains several built-in feature queries which allow users to explore various properties of projects and which are a starting point for selecting projects for different static analyses; e.g., for SQL injections, cryptographic security flaws, or call graph construction. We demonstrated *Hermes*' usefulness by using it to better understand the Qualitas Corpus and by computing a minimal test corpus useful for integration testing of generic data- and control-flow analyses.

To broaden the scope of *Hermes*, we will implement further queries in future work and make the computation of the *optimal* corpus extensible and configurable.

Acknowledgments

This work was supported by the DFG as part of CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

References

[1] K. Ali and O. Lhoták. Application-only call graph construction. In *European Conference on Object-Oriented Programming*, pages 688–712. Springer, 2012.

[2] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 68–75. IEEE, 2001.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid. *ACM SIGPLAN Notices*, 49(6): 259–269, jun 2014. ISSN 03621340.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[5] L. N. Q. Do, M. Eichberg, and E. Bodden. Toward an automated benchmark management system. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 13–17. ACM, 2016.

[6] J. Dujmović. Automatic generation of benchmark and test workloads. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 263–274. ACM, 2010.

[7] M. Eichberg and B. Hermann. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[8] M. Eichberg, B. Hermann, M. Mezini, and L. Glanz. Hidden truths in dead software paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 474–484, New York, NY, USA, 2015. ACM.

[9] B. Hermann, M. Reif, M. Eichberg, and M. Mezini. Getting to know you: Towards a capability model for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 758–769, New York, NY, USA, 2015. ACM.

[10] J. Lerch, J. Spath, E. Bodden, and M. Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 619–629. IEEE, nov 2015. ISBN 978-1-5090-0025-8.

[11] B. Livshits. Defining a set of common benchmarks for web application security. 2005.

[12] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: the Java unsafe API in the wild. *ACM SIGPLAN Notices*, 50(10): 695–710, 2015. ISSN 03621340.

[13] C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL `http://www.choco-solver.org`.

[14] M. Reif, M. Eichberg, B. Hermann, J. Lerch, and M. Mezini. Call graph construction for java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 474–486. ACM, 2016.

[15] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. *DROPS-IDN/6116*, 56, 2016. ISSN 1868-8969.

[16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.

[17] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha. Qualitas. class corpus: A compiled version of the qualitas corpus. *ACM SIGSOFT Software Engineering Notes*, 38(5): 1–4, 2013.

[18] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.