NJR: A Normalized Java Resource

Jens Palsberg
University of California, Los Angeles (UCLA)
palsberg@ucla.edu

Cristina V. Lopes
University of California, Irvine
lopes@uci.edu

Abstract

We are on the cusp of a major opportunity: software tools that take advantage of Big Code. Specifically, Big Code will enable novel tools in areas such as security enhancers, bug finders, and code synthesizers. What do researchers need from Big Code to make progress on their tools? Our answer is an infrastructure that consists of 100,000 executable Java programs together with a set of working tools and an environment for building new tools. This Normalized Java Resource (NJR) will lower the barrier to implementation of new tools, speed up research, and ultimately help advance research frontiers.

Researchers get significant advantages from using NJR. They can write scripts that base their new tool on NJR's already-working tools, and they can search NJR for programs with desired characteristics. They will receive the search result as a container that they can run either locally or on a cloud service. Additionally, they benefit from NJR's normalized representation of each Java program, which enables scalable running of tools on the entire collection. Finally, they will find that NJR's collection of programs is diverse because of our efforts to run clone detection and near-duplicate removal. In this paper we describe our vision for NJR and our current prototype.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Keywords Software tools, 100,000 Java programs, static and dynamic analyses, plug-and-play environment, reproducible results

1 Introduction

Java is one of the most used programming languages. Among the massive evidence for that point, we will give three examples that all are based on data for 2016. First, pypl.github.io listed Java as by far the most sought-after target for language tutorials on Google Trends. Second, codingdojo.com ranked Java as second to only SQL in language skills sought in job postings on indeed.com. Third, tiobe.com ranked Java as the most popular programming language by a wide margin. As we can see, people in massive numbers want to learn Java, use Java, and deploy Java. How come?

Java enjoys a trifecta of advantages that few languages can match. First, it is well designed and has the meticulous Java Language Specification that spells how every corner of the language works. Second, it has massive attention from the academic community that has helped produce textbooks, suggest improvements, and weed out problems. Third, it has impressive tool support that help programmers with programming, debugging, understanding, etc. The tool support has evolved over the years to become more sophisticated, meet new demands, and enable new opportunities.

We are on the cusp of a major opportunity: tools that take advantage of *Big Code* [15]. Specifically, the huge Java libraries and massive number of Java applications are in themselves an enabler for novel tools. *Big Code* has a close cousin in *Big Data*, which has enabled novel analytics, insights, and business processes. Similarly, we see signs that *Big Code* will enable novel tools in areas such as security enhancers, bug finders, and code synthesizers [19]. The *Big Code* is a repository of ideas, techniques, and best practices that, once harnessed into easy-to-use tools, can make programming easier. Thus we arrive at a key question: what do researchers need from *Big Code* to make progress on their tools?

We can get insights from *Big Code* at several levels of detail and sophistication. First, we can think of code as text and use it for text-based search of interesting code snippets that can inspire our own programming effort. Second, we may go further and request that the search returns code that is syntactically correct, which makes it easier to try out. Third, we might want to use types in the search to help narrow down the functionality of code that we find. Those ideas

are embodied in existing tools, so time has come to eye the ultimate prize: code that *runs* and thus is immediately useful. Here the research community hits a road block: existing collections of Java code are either small, without ability to build and run, or both. We need a *large* collection of *executable* Java programs that *enable* building new tools. How do we get that?

We envision a collection of 100,000 Java programs that is searchable, a catalogued set of popular tools that all work on the collection, and an environment that enables scriptable interaction. In particular, a script that runs a tool on the entire collection should be a few lines of code. Our preliminary results suggest that we can achieve such great convenience by normalizing the representation of each Java program. The normalization enables searchability, scriptability, and reproducibility. In short, we believe that the time has come to establish a *Normalized Java Resource (NJR)*.

NJR will consist of 100,000 executable Java programs, a set of working tools, and an environment for building new tools.
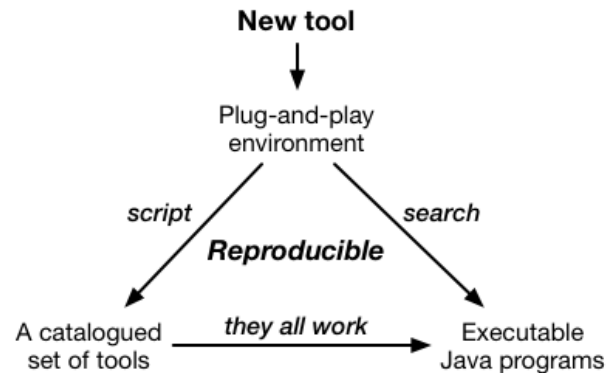
The Normalized Java Resource will lower the barrier to implementation of new tools, speed up research, and ultimately help advance research frontiers.

2   Our Vision for NJR

The Problem. The task of building NJR faces a problem that is large yet surmountable: *scale*. The problem of scale shows its face in three different guises. First, we need large-scale processing of Java programs that treats the programs uniformly rather than treating each one as a special case. Second, we need a large number of existing *tools* to succeed on the Java programs such that they can become building blocks for new tools. Third, we need a large scale of project *diversity* such that tools ingest a wide variety of coding ideas.

We have downloaded 1,481,468 Java projects from GitHub [13] (which excludes forks and invalid URLs). We found that few of those Java projects have build scripts, let alone specified inputs that enable the projects to run. Moreover, our experience is that popular tools often fail to work out of the box on the projects that we have tried. This is the reality that researchers face every day: great Java projects that fit their needs are hard to come by, and when they do, they come in a trickle rather than on a large scale. We have found that researchers spend much time, too much time we think, on looking for benchmarks and massaging benchmarks such that they fit their needs.

Until now, the benchmark suites of the world tend to be due to heroic efforts by single individuals or small groups of people. In most cases where a researcher determines that the prevailing benchmark suite is a poor fit for a new class of tools, the researcher will shy away from assembling a new benchmark suite. One reason is time: a benchmarking effort would take so much time that it would outweigh the

Figure 1. Our vision: the Normalized Java Resource (NJR).

potential gain. Another reason is academic reward: assembling a benchmark suite is often a thankless task that researchers rarely view as a research effort. NJR will enable researchers to build their own benchmark suite via search on our 100,000 Java programs.

> *Inside every large code corpus is a small code corpus struggling to get out.*

> "Inside every large language is a small language struggling to get out." — Tony Hoare.

Objectives. Figure 1 illustrates our vision for NJR. A tool builder gets a plug-and-play environment that provides 1) scriptable interaction with our catalogued set of tools and 2) the ability to search our collection of 100,000 executable Java programs. Those tools *all* work on *all* the Java programs and are set up to run in a way that leads to reproducible results.

We believe that the scale of 100,000 Java projects is sufficient to support a new direction of research and development. Moreover, our experiments suggest that 100,000 Java projects is realistic with the current raw material.

Part of our effort is to get from those 1,481,468 Java projects to 100,000 highly useful Java projects. This cannot be done *by hand* in a reasonable amount of time; blood, sweat, and tears won't get us there. We need to automate the process, particularly the tasks of creating build scripts, running tools, and enabling search. Our experiments show that crashes are lurking at every step of the way. This is sobering for us yet good news for researchers who will use NJR; we will do the heavy lifting and they will benefit. Ultimately we want to have so much automation that the next time somebody shows up with 1.5 million Java projects, we will be able to distill 100,000 useful Java projects fairly automatically.

We envision a diverse collection of 100,000 normalized Java projects that are executable, scriptable, and searchable. We get them from GitHub, we filter them, and we normalize their representation to enable large-scale processing with reproducible results. Such processing includes execution, static and dynamic analysis, and search for projects with specific

dynamic characteristics. For each search of the collection, NJR returns both a file with Java projects and a container (e.g., Docker) for a cloud service such as Amazon EC2. Thus, a researcher can run tools on those projects both locally and on a cloud service. Researchers will be both beneficiaries and contributors to NJR. They *benefit* from searching for Java projects that fit their need, and once their tools run on NJR, they *contribute* to an ever-increasing collection of measurements. Notice the powerful network effect: the more people run tools on NJR, the more data we get for search, and the more data we get for search, the more people will want to search and run on NJR.

Significance. NJR will speed up innovation in the area of Java-oriented tools. Those tools include security enhancers, bug finders, and code synthesizers. Such tools often build on other tools such as tools for call-graph construction, code coverage, and program instrumentation. NJR will enable a researcher to identify existing tools, search for Java programs, and be sure that those tools work on those Java programs. Thus, the researcher can leverage existing tools and focus on working with Java programs that are well suited to demonstrate the value of her tool. In total, NJR will enable faster implementation and evaluation of new tools.

NJR will make experiments easier that they typically are today. Many researchers build tools for Java and struggle with current benchmark suites such as DaCapo [1] when they want to evaluate their tools. DaCapo is a small benchmark suite that is particularly good for evaluation of performance-oriented tools, including virtual machines, but problematic for development and evaluation of many other kinds of tools. Specifically, security enhancers, bug finders, and code synthesizers can benefit from *Big Code* and have little use for DaCapo's convenient approach to repeatedly executing a program. Another example is Boa [7], which enables large-scale mining based on text processing of Java programs, but has no support for execution. Intuitively, while DaCapo is small and executable, Boa is large and nonexecutable, and we want NJR to be the best of both worlds: *large and executable*.

We will provide a web interface that enables easy search for Java programs with desired characteristics, and we will provide examples of how to write a script that runs a tool on the entire collection. A search can be based on both static measurements and dynamic measurements.

The 100,000 Executable Java Programs. We will create a set of Java programs that enable *execution*, *scalable processing* and *reproducible results*. Here are our plans.

First, we will enable *execution* of the 100,000 Java programs. Most of the programs from GitHub came without build scripts, while the rest have a mixture of ant, maven, and gradle scripts. We have found that the programs without build scripts tend to have problems that break straightforward attempts to build. Those problems include, starting

with the most frequent: dependencies on external libraries, non-standard file-encodings, duplicate versions of the same code that is not supposed to be compiled, wrong version of the libraries, and old versions of Java. The huge number of programs means that we must create build scripts *automatically*; we cannot treat each one as a special case. We will develop a *build-script synthesizer* that with a high rate of success will create working build scripts.

Second, we will enable *scalable processing* of the 100,000 programs. We will achieve scalability through uniformity. Our idea is to normalize both the structure of each Java program (in a semantics-preserving manner) and the build script. The resulting Java program conforms to a standard structure and it includes a cache of all dependencies. Each normalized Java program references the local copy of the dependencies explicitly, which freezes the environment in which the program operates. A key idea behind normalization is that the normalized build script is the *same* across all normalized Java programs. Thus, we will write the normalized build script once and for all, and for each use we will pass as arguments a few pieces of key information.

Third, we will enable *reproducible results* by caching dependencies and using Nix (http://nixos.org/nix) to write the normalized build script. Nix is a purely functional scripting language that forces us to make all dependencies explicit.

GitHub contains more duplication that meets the eye. Our preliminary work on SourcererCC [13] shows only 60 percent of Java files are distinct. We will use clone detection to ensure a high degree of diversity of our collection of Java programs. We may design and implement novel techniques for clone detection enable us to quantify the level of diversity of our collection. As three additional diversity measures, we will use program size, call-graph size, and API use.

NJR will for each Java program have a link to its GitHub repository. This will enable researchers to find out about the age of the project, the activity level, and other versions.

A Catalogued Set of Tools. Which tools do researchers build on? In an online survey and at three NJR workshops in 2017, we found that the most popular tools are for call-graph construction (Doop [2], Soot [18], WALA [8]), for code coverage (EMMA [5], EvoSuite [9], JaCoCo [10]), and for program instrumentation (ASM [3], Javassist [4]). Thus, those tools are prime candidates for the NJR catalogue of tools. We want every tool in the catalogue to work on all the 100,000 Java programs, which we will achieve by brute force: try every combination. Thus, if even one tool fails on a program, we will remove it from the collection. In practice, we will engage with the authors of each tool, report a bug, and hope the bug can be fixed fairly quickly. Thus, part of the NJR project may serve as a massive bug-finding expedition, focused on popular tools rather than general software.

We will also run tools that collect static data about each Java program. Such data includes size (LOC, number of methods, etc), whether the program uses assertions, and so on. We plan to run the Understand tool (https://scitools.com) to collect such measurements.

The Plug-and-Play Environment. A researcher (we will call her Alice) who wants to build a new tool gets a plug-and-play environment that provides 1) scriptable interaction with our catalogued set of tools and 2) the ability to search our collection of 100,000 executable Java programs. Ultimately, if she wants to run her tool on every Java program, she will write a single script:

```
for each Java program p in the collection:
    run the tool on p
```

Similarly, if she wants to call one of the tools in the NJR catalogue, this should be equally easy. We will provide examples of scripts that will be easy to modify.

We will store scripts rather than the results produced by scripts. However, we do want to store *some* information, namely information that will be useful for future searches. We will create a database with 100,000 entries, one for each program, and with data about each program that was collected by tools. Thus, each tool that runs on our entire collection has the potential to produce useful information that we will store in the database. This is a major step towards automatic classification of the Java programs. We will provide a web interface to query the database, and the search result will be a container that contains the retrieved programs plus scripts. This will make it easy to download the desired Java programs and then run them, either locally or on a cloud service such as Amazon EC2.

Overall, we picture an interaction where a researcher 1) receives a container with a few programs, 2) gets her tool to work on those program and sends us her tool plus scripts, and 3) we run her tool on the entire collection and eventually add her tool to the NJR catalogue. Thus, she is both a beneficiary and a contributor to NJR.

Limitations. While NJR will go beyond today's capabilities, it won't do everything for everybody. Here we list the three main limitations.

The first limitation is that NJR won't include multiple *versions* of programs. Some benchmark suites such as Qualitas [17] do include multiple versions, which enables studies of software evolution. NJR focuses on diversity over versions so we will pick the most recent version of a program.

The second limitation is that we won't spend time on turning a *popular* system into a great benchmark, which can take months (say the DaCapo authors). Thus, NJR may exclude popular distributed systems written in Java such as Cassandra, Hadoop, and Zookeeper. NJR focuses on automation over popularity so we will go with what our tools can
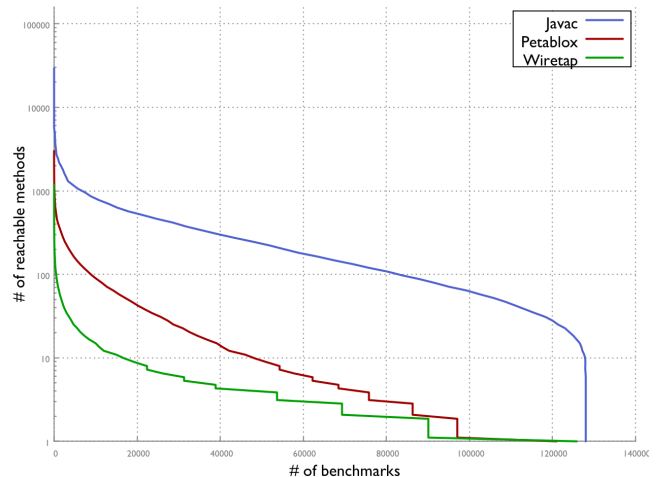

Figure 2. Running tools on 125,846 Java programs.

give us. However, we may make exceptions if the effort-to-benefit is in our favor. For example, we may include Pointerbench, which consists of 34 challenging small programs.

The third limitation is that we won't include any *library* that has no main method. NJR focuses on execution over libraries so we insist on that every Java program has a main method and runs.

3   A Prototype of NJR

We have implemented a prototype of NJR that has rudimentary versions of much of the envisioned functionality.

First, we have implemented a build-script synthesizer that, using heuristics, has created build scripts for 190,000 Java projects. For example, for the missing dependencies, before any compilation happens, we first collect all libraries we can find (from Maven, etc.), and place them in our own central repository; we then compile the projects pointing the build script to this repository. For encoding and JDK version issues, we try several compiler configuration options. We will continue to improve our build-script synthesizer, and we estimate that we will be able to create build scripts for 800,000+ Java projects. The result of our preliminary work is a set of 125,846 normalized Java programs that all build and run. Some of them are identical except for their main methods. A typical example is a GitHub repository that contains code plus three main methods: one for the main functionality and two for unit tests. For those near-clones, we hope that dynamic measurements can help us sort out which one to keep in the collection and which ones to toss out. Another aspect we must consider is that some of the executions give "surprises", such as requests for input or spawning of GUIs. We will work on detecting and classifying such surprises and work on how to handle them. Thus, while 125,846 executable Java programs is a good start, the reality is that we have far fewer that eventually we will want to keep in the

collection. We will continue to add to our collection, while improving the techniques that we described above.

Second, we have run the static analyses Petablox [14], the static analysis Doop [2], and our own dynamic analysis Wiretap on all our 125,846 Java programs. Our Nix scripts required a major time investment up front to get all the details right, yet that investment has payed off handsomely in scalability and reproducibility.

Figure 2 shows results of our experiments with Petablox and Wiretap on the 125,846 Java programs. Specifically, Figure 2 shows, for each Java program, the number of application methods compiled by javac, the number of application methods found to be reachable by Petablox, and the number of actually executed application methods as determined by Wiretap. Notice that Figure 2 orders the programs according to the number of methods, from most to fewest. This ordering has the pleasant effect that if we ask, for example, "how big are the largest 20,000 programs?", then Figure 2 shows that largest 20,000 programs execute *at least* 9 application methods. Thus, Figure 2 is good for answering questions that have "at least" in the answer.

In summary, our prototype of NJR has a large collection of "nondiverse", executable programs, a small set of tools, and some support for scriptable interaction. Our goal is to get from here to the vision in Figure 1.

4   Research Opportunities enabled by NJR

In an online survey and at three NJR workshops in 2017, researchers said that for their research, the most important and useful characteristic of a collection of Java programs is that they all compile and run. Most of the respondents also called for a large, diverse, categorized set of Java programs.

Cross-Cutting Research Opportunities. Researchers have intersecting needs that could be met by research supported by NJR. The three biggest such cross-cutting research opportunities are: *ground truth*, *standard call graphs*, and *hybrids of static and dynamic analysis*. We will detail each opportunity in turn.

First, most researchers who develop static analyses have a need for *ground truth*. The interesting case is any static analysis that produces approximate information about the run-time behavior of a program. For such a static analysis, researchers want to use the ground truth to compute such metrics as precision, recall, and accuracy for their static analysis. The ground truth involves *running* a program on all inputs, for all possible execution paths, and for as long as it takes (which can be indefinitely, in case the computation never terminates). While the ground truth is hard to come by, NJR provides hope because the Java programs compile and run. Specifically, the more inputs we can try and the more execution paths we can try, the closer we can come to knowing the ground truth about a program. NJR will be

a solid basis for pursuing the impossible dream of knowing the ground truth. We believe that many approaches to finding ground truth are possible and that NJR can support innovation in this area.

Second, most researchers who develop almost any kind of analysis needs a call graph as a first step, and they wish they could have *standard call graphs* to enable easy comparison of techniques. A call graph has the methods of a program as it nodes, and possible control-flow (method calls) between the nodes as its edges. The three most commonly used tools to construct a call graph are Doop, Soot, and WALA. How do they compare? If my paper uses Doop as the basis for doing a task and your paper uses WALA as the basis for doing the same task, how much does Doop-WALA influence the results and any comparison of our papers? NJR provides hope for improving the situation by enabling large-scale comparisons of call-graph-construction algorithms and enabling efforts to standardize them. The Soundiness Manifesto [12] talked about the choices that a static analysis has to make and that some of those choices may well be unsound. NJR can help demonstrate the effect of those choices for every corner of Java. Standardization has been sorely lacking in the area of static analysis; NJR may help change that.

Third, many researchers are excited about the possibility of developing *hybrids of static and dynamic analysis*. Currently, they face an uphill battle with getting tools to work together. NJR will have a catalogued set of both static analyses and dynamic analysis on which researcher can build. For example, researchers can use such analysis to determine quantitative aspects such as: is the execution long-running? is it data-intensive? which APIs does it use? This may enable researchers to classify programs into domains such as scientific computing, big data, and virtual reality.

Area-Specific Research Opportunities. Our Java resource ushers in a new age of picking and working with great benchmarks. Much tool development can benefit from access to the *Big Code* that NJR provides, particularly in the important areas of security enhancement, bug finding, and code synthesis. NJR will enable faster building and evaluation of a variety of tools. A tool developer can rely on that many popular tools already work for our 100,000 Java projects, and then build their own tools on top. Additionally, developers can search for Java projects that are particularly suited to demonstrate the value of their tool.

First, suppose a researcher (Alice) wants to build a Java deadlock detector that combines a variety of techniques into a single, novel approach. She wants to explore inputs that exercise different aspects of the Java project; use static analysis to narrow down the potential deadlocks, and use concolic execution to drive execution towards a deadlock. NJR will offer a large number of Java projects and enable easy search for concurrent projects of a desired size and complexity and for which the needed tools are known to work. Thus, Alice's

starting point can be tools for generating test inputs, building static analyses, and doing concolic execution that work out of the box on the benchmarks. Alice needs no familiarity with those tools and might use their cached outputs.

Second, suppose Alice has designed a novel approach to discover security vulnerabilities. She bases her implementation on some of the static and dynamic analyses that work out of the box on NJR, and she searches NJR for input programs that use specific libraries. Finally she writes a simple script that enables NJR to run her tool on those Java programs, deliver easily reproducible results, and compare with security tools that run on NJR already. Thus, Alice is both a beneficiary and a contributor to NJR.

Third, suppose Alice wants to build a Java code synthesizer that can help programmers complete a half-finished piece of code. The synthesizer has a pre-computed summary of a massive, existing body of code and can match the programmer's code against that summary. The result is that the tool makes a suggestion to the programmer for how to complete the code. The needed summaries may be computed by large-scale static and dynamic analyses. NJR will be a massive body of code that will enable new directions of research on synthesizers. Those new directions can go beyond the state of the art for Java that tends to focus on code as text, and instead work with the NJR that consists of projects that build and run.

5   Existing Related Resources

Many collections of Java programs exist; we will discuss six of them and why we still need NJR.

DaCapo. The 2009 version of DaCapo [1] consists of 14 Java programs. The main focus of DaCapo is *performance* and each of the Java programs has non-trivial memory loads. Thus, DaCapo is excellent for answering such questions as *how much speed-up does my optimization provide?* We share three key objectives with DaCapo: 1) each Java program compiles and runs, 2) the Java programs are diverse, and 3) the results are reproducible. However, in contrast to DaCapo, for NJR we have little interest in performance.

Boa. Boa [7] is a large collection of programs in several languages, including 380,000 Java programs (mainly from GitHub). The main focus of Boa is *repository mining* via static analysis. Thus, Boa is excellent for answering such questions as *how do programmers use assert statements?* and *what can we learn from applying source code metrics?* Additionally, Boa supports working with multiple revisions of Java programs and can help answer such questions as *how much has unit testing been adopted over time?* Boa has no support for compiling a Java project and no current support for detection similarities or clones across Java programs.

Qualitas, XCorpus and Hermes. The Qualitas corpus [17] consists of 112 Java programs in a total of 754 versions, all fit for static analysis, yet not buildable and not executable. Qualitas has its main strength in enabling studies of software evolution based on a sequence of versions of each program. The XCorpus corpus [6] builds on Qualitas, consists executable Java programs, and provides many inputs for each program to ensure high coverage. The Hermes system [16] supports search, filtering, and selection of Java programs from the Qualitas corpus. Currently, search in Hermes is based on static measurements, while we plan the search in NJR to be based on dynamic measurements.

JaConTeBe. JaConTeBe [11] contains 8 benchmarks that all run and have known concurrency bugs. We share the goal to catalogue bugs in the programs in NJR.

6   Conclusion and Acknowledgments

Our prototype of NJR already has many capabilities and we plan to release NJR 1.0 in 2019.

References

[1] 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. L. Intel, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. v. Dincklage, and B. Wiedermann. 2006. The DaCapo benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA'06, ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications.* 169–190.

[2] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA'09, ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications.* 243–262.

[3] E. Bruneton, R. Lenglet, and T. Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems.*

[4] Shigeru Chiba. 2000. Load-time Structural Reflection in Java. In *ECOOP'00, European Conf. on Object-Oriented Programming.* Springer-Verlag (*LNCS* 1850), 313–336.

[5] EMMA Developers. 2018. EMMA, a free Java Code Coverage Tool. (2018). http://emma.sourceforge.net, accessed Jan 6, 2018.

[6] Jens Dietrich, Li Sui, Shawn Rasheed, and Amjed Tahir. 2017. On the Construction of Soundness Oracles. In *SOAP'17, 6th ACM SIGPLAN Int. Workshop on State Of the Art in Program Analysis.*

[7] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *35th Int. Conf. on Software Engineering (ICSE 2013).*

[8] T. J. Watson Libraries for Analysis. 2018. WALA. (2018). http://wala.sourceforge.net, accessed Jan 6, 2018.

[9] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[10] Marc R. Hoffmann, Evgeny Mandrikov, and Mirko Friedenhagen. 2018. JaCoCo: Java Code Coverage for Eclipse. (2018). http://www.eclemma.org/research/index.html, accessed Jan 6, 2018.

[11] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. A Benchmark Suite of Real-World Java Concurrency Bugs. In *ASE'15, IEEE Int. Conf. on Automated Software Engineering*. 178–189.

[12] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *CACM* 58, 2 (February 2015), 44–46.

[13] Cristina Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DejaVu: A Map of Code Duplicates on GitHub. In *OOPSLA'17, ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications*.

[14] Ravi Mangal, Xin Zhang, Aditya Nori, and Mayur Naik. 2015. A User-Guided Approach to Program Analysis. In *FSE'15, ACM SIGSOFT Int. Symposium on the Foundations of Software Engineering*.

[15] Veselin Raychev, Martin T. Vechev, and Andreas Krause. 2015. Predicting Program Properties from Big Code. In *POPL'15, ACM Annual Symposium on Principles of Programming Languages*. 111–124.

[16] M. Reif, M. Eichberg, B. Hermann, and M. Mezini. 2017. Hermes: assessment and creation of effective test corpora. In *SOAP'17, the 6th ACM SIGPLAN Int. Workshop on State Of the Art in Program Analysis*.

[17] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. 2010. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *APSEC'10, Asia Pacific Software Engineering Conf.*. 336–345.

[18] Raja Vallé-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode using the Soot Framework: Is it Feasible?. In *CC'00, Int. Conf. on Compiler Construction*. Springer-Verlag (*LNCS*).

[19] Eran Yahav. 2015. Programming with Big Code. In *13th Asian Symposium on Programming Languages and Systems (APLAS'15)*. 3–8.