Target Selection for Test-Based Resource Adaptation

Arpit Christi
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, Oregon, USA
christia@oregonstate.edu

Alex Groce
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, Arizona, USA
agroce@gmail.com

***Abstract*—Building software systems that adapt to changing resources is challenging: developers cannot anticipate all future situations that a software system may face, and even if they could, the effort required would be onerous. A conceptually simple, yet practically applicable, way to build resource adaptive software is to use test-based software minimization, where tests define functionality. One drawback of the approach is that it requires a time-consuming reduction process that removes program statements in order to reduce resource usage, making it impractical for use in deployed systems. We show that statements removed have predictable characteristics, making it possible to use heuristics to choose statements to analyze. We demonstrate the utility of our heuristics via a case study of the NetBeans IDE: using our best heuristic, we were able to compute an effective resource adaptation almost 3 times faster than without heuristic guidance.**

I. Introduction

Modern day software systems are complex and use numerous resources, both explicit (e.g. memory, CPU, network, and storage space) and implicit (e.g. libraries and protocols). While developing complex software systems, assumptions are always made about the availability and usage of resources. For example, a navigation app is written with the assumption that some kind of location provider is available. An application that needs logging makes an assumption that disk space is always available to log data. An application making a service call to grab the latest stock prices makes an assumption about availability and accuracy of this service. For mission-critical systems that operate under extreme field conditions, it is difficult to construct and enumerate these assumptions and often the assumptions will not hold in all real-world environments or situations. For example, to consider an often overlooked example of a "resource," a change in one of the system libraries used by the software, due to an operating system update to address a security vulnerability, may force the development team to refactor some part of the application, triggering a cycle of design-development-testing-deployment: obviously, such a cycle is far too slow for fielded, critical applications, but rejecting operating system updates may leave software vulnerable in the field. Such inability to handle a changing environment is a key limitation on the reliability of many software systems. One way to handle these changes is to automatically adapt the system to handle resource availability changes.

Self-Adaptive Software Systems (SASS) or Self-Organizing Software Systems (SOSS) are designed to allow a system to adapt *itself* under varying conditions. Adaptation may manifest as (1) restricted functionality, (2) altered functionality, or (3) enhanced functionality [1]. Different engineering approaches to build adaptive software systems are summarized by Kruptizer et al. [2]. Resource adaptive software systems are a subset of self-adaptive systems where the reason for adaptation is unavailability or variability in one or more resource.

Based on the assumption that most mission-critical systems have an adequate (and in fact high quality) test suite, we proposed *test-based software minimization*, a conceptually simple but practically applicable approach [3] to resource adaptation that requires very little in the way of burdensome additional specification by developers. In our approach, adaptive software requirements are captured by annotating (usually simply labeling) tests. Tests, in the simplest instance, are simply given arbitrary but meaningful tags based on features present in the tests, usually including a special "top" label that indicates tests that must pass for any valid system. We introduced a tool called hddRASS, a program reducer tailored to the task of building Resource-Adaptive Software Systems, henceforth abbreviated as RASS. hddRASS takes as its input a Java program and annotated (labeled) tests, and automatically builds an adapted program that has the potential to work well even with degraded resource availability. In the simplest mode of operation (conceptually encompassing other uses), hddRASS takes a program plus a set of tests that must still pass, with the assumption being that other tests, in the full test suite but not in the provided set, check for behavior we are willing to sacrifice in order to use fewer resources. In our approach, tests themselves indicate *sacrificability of specification*. Given this input (program/class + subset of tests), hddRASS returns a program or class that has fewer statements (some of which will be resource-using statements, with high likelihood) that still passes the given tests. The resource adaptation arises from the fact that the original program is written to pass the *full* test suite, and if the reduced suite is properly selected, the missing tests concisely and with little effort represent resource-

using functionality that can be sacrificed. As a simple example, consider a system that produces detailed system logs, but needs to operate in an environment with reduced storage space. We remove (only) the tests labeled as `logging`, tests that check the existence and content of system logs, and run hddRASS, which (it is hoped) outputs a new version of the software with logging calls removed. The new system therefore uses less storage space, as it no longer produces detailed logs.

One major drawback of test-based software minimization as originally proposed is the time taken to build an adaptation of a program. In our original case study, it took about 3 hours to build an adapted NetBeans IDE that reduced memory usage by removing undo/redo functionality. For mission-critical systems deployed in the wild, there may not usually be time to spend hours on each adaptation. The system (or part of it) would need to be halted for a very long time, until adaptation could be applied and a new system made available. Instead of going through all possible targets (statements to be removed) for adaptations, predicting likely-removable targets and processing only those targets should significantly reduce the time required for adaptation. This paper proposes a heuristic approach to select a subset of statements to attempt to remove. The heuristic approach proposed here is in a sense independent of the precise original approach to test-based minimization. As long as a resource-adaptation approach relies on program modification by deletion, either at the source code level or bytecode level, processing only likely-removable targets will tend to improve performance.

The contributions of this paper are as follows:

- We applied hddRASS to a large set of classes (almost twice as many as in our previous work) from open-source Java projects and their test suites, computing 800 test-based reductions[1]. We found that the resulting reductions are 1) small in size: very few statements are removed and 2) simple in kind: complex blocks are rarely removed. We also show that certain program entities are most likely be removed, and the removed targets typically have a well-defined coverage relationship to tests that are retained in, and removed from, the suite.
- Based on these empirical findings, we propose three novel heuristics to speed up our original test-based software minimization approach to resource adaptation. These heuristics are a significant advance from our original proposal, in that delta-debugging approaches usually consider all components as potentially removable, which is expensive and un-necessary in our setting; to our knowledge, this is the first modification of a delta-debugging approach that removes some potential components from consideration on a heuristic basis, in order to improve performance.
- We verified the validity of our heuristics by applying them to a new set of open source Java projects, demonstrating that *processing only likely-removable targets* improves performance significantly while sacrificing very little in the way of accuracy of minimization. We also applied our heuristics to build an adapted NetBeans IDE without undo/redo, as in the original proposal's case study, in order to conserve memory. We show that the heuristic approach to selecting removal targets preserves memory-usage reductions, but speeds processing by a large factor (nearly 3x at best), moving us much closer to practical in-the-field adaptation.

[1]We refer to the general process of removing statements from a program or class as *minimization*, because that is indeed the goal; however, we refer to the outcome of the process (a new version of the program/class) as a *reduction*, not a minimization, to emphasize that the delta-debugging approach used is not optimal, and so we seldom obtain results that are truly guaranteed to be minimal.

II. Background

In a research road map to software engineering for self-adaptive software system, Cheng et al. mention the need to capture adaptive software requirements [4]. Adaptations manifest in 3 different ways: (1) restricted functionality, (2) altered functionality, or (3) enhanced functionality [1]. In restricted functionality, it is possible to meet the changing resource need by dropping some functionality, or by not exercising some features of the system. For the most part, the system will continue to work as it should, though it will no longer provide certain features, so that the resource under stress will be conserved. This "sacrificability of specification" is a subset of adaptive software requirements, one in which the *only* possible adaptation is sacrificing satisfaction of some specification(s): having the software do less.

Domain-specific languages (DSLs) and formal specifications [5], [6] have been proposed as ways to express sacrificability of specification; however, using a DSL forces developers to learn a new language, purely for the purpose of expressing adaptations, and very few existing systems have a formal specification on which to base a sacrificability specification. In practice, for most real-world software systems, even mission-critical ones, specifications are documented and verified only using *tests*. Our previous approach to expressing sacrificability of specification, therefore, aimed to "meet developers where they are" and base resource adaptation on the idea that most tests concern only some of the system's functionality. By no longer requiring a system to pass a restricted set of tests, a reduced functionality can be defined. Test annotations can define a multi-dimensional space in features such as functionality tested, priority, and resource usage. It is then possible to group tests and label those groups, such that each group represents a sacrificable or variable unit. The approach, as described in our original work, is summarized in Figure 1 [3].

In the figure, tests that are not surrounded by an oval represent non-sacrificable tests. They are the core tests: if any of them fail, the system is not usable. The tests within an oval are sacrificable, and the labels (A, B, C, D) group these by tested functionality. All tests with group A mark a sacrificable unit. Now, let us say that a resource R is unavailable and group A is chosen for adaptation, based on its connection to resource R. The test suite (without tests marked A) and existing program are fed into hddRASS and

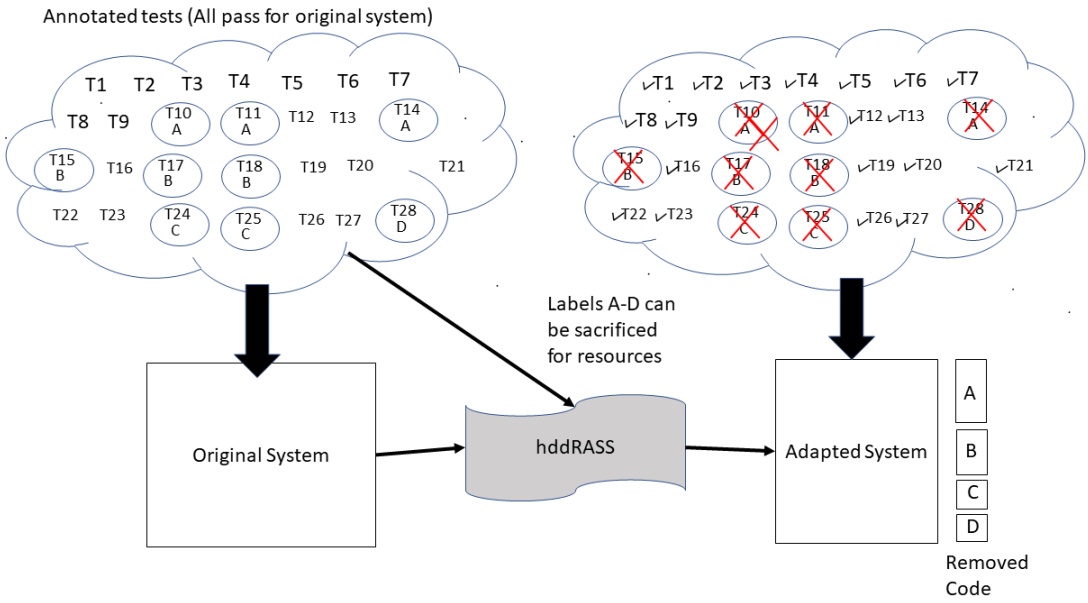

Fig. 1: The test-based approach to resource adaptation by program minimization [3]

it will automatically build the system consisting of the core system and components marked by labels B, C and D. As the figure shows, we could also build a minimal functional core, where all labels are considered non-essential. The approach is conceptually simple, as it requires only the grouping of tests and marking some tests as sacrificable. In our previous work, we built an adapted NetBeans IDE that used much less memory completely automatically simply by labeling 3 tests, those checking undo/redo functionality, as sacrificable. As part of the DARPA BRASS project [1], we work with a group of developers building a Resource Adaptive Software System (RASS) version of a Tactical Situational Awareness System (TSAS). Our approach is currently under consideration by the TSAS development team as a way to build adaptive system components automatically.

hddRASS, the reducer we developed to support our approach, implements a modified form of Hierarchical Delta Debugging (HDD) [7], optimized for the RASS workflow [3]. hddRASS reduces a program by removing one or more statements at a time, so long as such removals do not cause a test suite to fail. The intuition behind our changes to HDD is discussed in our previous work [3]. Like HDD, hddRASS is still a greedy search-based algorithm, with similar worst-case time complexity $O(n^3)$ [3][2]. This cost is even more significant with hddRASS, as every attempted reduction step consists of compiling a new version of the program and running potentially most of the program's test suite, a very time-consuming operation. For example, computing the NetBeans IDE reduction featured in our previous work took 3 hours and 35 minutes. Simply marking some statements as not likely to be removable could considerably reduce this time-to-minimize.

[2]hddRASS is based on HDD* with known worst-case runtime $O(n^3)$

Some possible restrictions on removals are obvious: if a removal is, by the syntax of the language of the system, guaranteed to cause a compilation failure, it should not be considered, for example: e.g., the sole return statement for a non-void method in Java simply cannot be removed (without removing the whole method); some statements in the data-flow of the return value of a method also may not be removable (there must be at least one def of each value used in the return); if a method is ever called, its definition cannot be removed. If a statement is not covered by the retained tests (is, with respect to those inputs, dead code), intuitively, it should be removable since it is certainly not needed to pass the retained tests. However, because of the restrictions on removal, and the fact that we only consider statement removals (not entire classes or methods) it is not always actually removed (or removable, in our context). E.g., if 10% of program statements are not covered by the complete test suite, this does not mean that at least 10% of the program statements are removable in every reduction. In our original experiments, we also noted that reductions are usually small (in terms of change, not absolute size: the absolute size is similar to the input program), leaving the software system mostly unchanged. Moreover, reductions are simple, not touching scattered and varied parts of a system, and also seldom removing complex blocks of code. However, these results were based on analyzing only a small set of classes. In this paper, we first conduct experiments on a much larger set of classes and test suites, and use the results to derive heuristics for determining likely-to-be-removed statements.

A. Terminology

- *Labeled test/annotated test/removed test:* A test that is marked as sacrificable. The minimization process uses a test suite that does not contain the labeled tests; hence they are also referred to as *removed* tests.
- *Unlabeled test/retained test:* A test that is *not* marked as sacrificable. The minimization process uses a test suite that contains all unlabeled tests; thus they are also refereed to as *retained* tests.

III. Experiments

A. Subjects and Tests

Test-based minimization is essentially based on the idea of taking a program and a test suite and minimizing the program (by removing statements) until it is (approximately) as small as possible while still satisfying a (modified, by removing some tests) test suite. At a certain level of abstraction, this approach is similar to generate-and-validate automated program repair [8], [9]; however, the end goal (reduced-size program that passes a given test suite) it quite different than the goal of passing previously failing tests. Where program repair uses the concept of a test-adequate patch or test-suite-adequate patch [10], [11], we can consider a test-(suite)-adequate adaptation: a minimized program such that all tests in a modified test suite continue to pass.

Computing this reduced program version is computationally expensive, suggesting that more information about the minimization process is required to tune its performance. In order to study minimization, we used hddRASS to produce 800 distinct reductions at the class level. In prior work, we discussed how method and class level reductions can be combined to build a whole-program reduction. We focused our empirical examination on the class level because for many methods, no statements can be removed; the class level is the first level at which reduction is usually meaningful and useful. The 800 reductions are applied to 40 distinct classes across 10 open source Java projects. For each project, we randomly chose 4 classes. Details of the subjects are provided in Table I. We measured Java statements as defined by `java.parser.ast.statement`. Subjects have an average of 387 LOC (for the whole class) and 132 statements in non-constructor methods.

In addition to a program or class, minimization requires a test suite (and the ability to build the system under reduction and run the suite). For these projects, the build and test system consists of simple `ant` or `maven` commands. In our previous work on test-based minimization, we noted that arbitrary subsets of tests are not interesting (reduction will usually target some functionality), and some tests are not relevant to a particular class. Based on this, we used direct coverage as a criteria for selection, and we adopted the same criteria for this larger examination of reductions. This yielded 24 tests per subject class, on average (numbers per class are shown in Table I). For 32 out of 40 subjects, statement coverage of the tests is more than 80%, and it is more than 70% for 37

TABLE I: Subject Class Information. Stmt column counts Java statements inside class methods as defined by `java.parser.ast.Statement`. Mthd is number of class methods. If multiple classes are contained within a single Java file, LOC counts all lines. Statements counts only statements within the class under consideration.

Project	Subject	LOC	Mthd	Tests	Stmt
CruiseControl	AntBuilder	499	33	22	143
CruiseControl	Schedule	383	30	18	127
CruiseControl	Project	685	70	35	291
CruiseControl	AntScript	318	8	34	121
Ant	Available	289	21	28	133
Ant	Copy	679	48	24	179
Ant	FixCRLF	385	17	34	23
Ant	Checksum	431	24	15	142
Validator	UrlValidator	218	11	21	82
Validator	RegexValidator	93	4	7	40
Validator	DomainValdiator	1302	15	20	74
Validator	EmailValdiator	109	6	18	26
Jexl3	Engine	296	30	38	103
Jexl3	JexlArithmetic	781	54	35	289
Jexl3	JexlEvalContext	102	17	37	29
Jexl3	Script	207	20	14	50
Cli	Option	404	48	9	85
Cli	GnuParser	64	1	58	23
Cli	PosixParser	141	6	58	37
Cli	OptionGroup	86	8	13	30
Jena	OntTools	289	29	4	65
Jena	LocationMapper	292	21	10	138
Jena	OntlClassImpl	464	60	27	133
Jena	OntModelImpl	1174	162	65	686
Text	ExtendedMessageFormat	301	17	14	137
Text	LevenshteinDetailedDistance	220	6	12	142
Text	AlphabetConverter	277	13	10	82
Text	StringBuilder	1257	146	91	597
digester	BinderClassLoader	64	4	11	9
digester	CallMethodRule	255	9	15	58
digester	Digester	1456	149	19	432
digester	NodeCrerateRule	39	2	15	15
httpCore	URIBuilder	304	36	26	143
httpCore	HttpService	166	4	12	36
httpCore	HeaderGroup	168	17	11	72
httpCore	EofSensorInputStream	130	12	10	34
jfreechart	DefaultIntervalCategoryDataset	151	20	20	150
jfreechart	Hour	180	18	22	60
jfreechart	GridArrangement	215	13	18	119
jfreechart	StatasticalBarRenderer	281	11	11	159
MEAN		387	30	24	132
MEDIAN		285	17	18	94

TABLE II: Coverage distribution: #subjects is the number of subject classes within each category. Coverage here is statement coverage.

Coverage	<70%	70-79%	80-89%	>90%
#Subjects	3	5	15	17

subjects, as shown in Table II. The mean coverage over all 40 subjects was 84.7%. We can therefore say that we generally have subjects with *good test coverage*. For 17 subjects, the test coverage is excellent (>90%).

B. Procedure

For each class, we first randomly labeled 10% of the tests as sacrificable, and computed a minimization. We repeated this procedure 10 times for each class, randomly labeling tests each time, yielding 10 results per class, for 400 results. During each run, if the removed tests overlap with tests not removed, or concern only very minor functionality, there may be almost no reduction. On the other hand, if a very important test is selected for removal (one that covers a lot of functionality and has no overlap with other tests) the reduction will be significant. Labeling tests randomly and repeating the process 10 times provide us with an idea of typical results. Developers label tests based on some feature the test targets. Such labeling is highly context-specific, and lacking from current test suites. Because we are using random labels, rather than developer-provided labels, we lack a solid basis for guessing the size of a typical set of removed tests representing a feature. We therefore repeated the same procedure, but with 20% of the tests labeled as removable, in order to produce another 400 reductions. In the remainder of the paper, we identify the labeling scheme used (i.e., portion of tests removed) by percentage: *Label 10%* means the scheme where we labeled (and thus removed) 10% of tests (at random) as representing sacrificable functionality, and *Label 20%* means the same scheme with 20% of tests labeled and removed.

C. Measurements

Our interest in computing these reductions is to understand the type (and number) of entities removed, and how they relate to the tests in the suite. Our results, therefore, consist of 4 measurements:

1) **Reduction size:** this is the number of statements removed, a simple measure of the amount of change to the class.
2) **Reduction height:** this measures the maximum distance of a removed statement node from a leaf statement node in the class AST. Reduction height can be seen as measuring the complexity of changes to a class. If height is one, only leaf nodes (hence simple statements) were removed.
3) **Reduction type:** this describes the type of statement removed, as defined by `java.parser.ast.Statement`. We are interested in determining whether certain kinds of statement are more likely to be removed.
4) **Reduction relation:** this is the relationship of removed statements with labeled and unlabeled test coverage. For this measure only, the measurement is over 50 points (chosen via stratified sampling) rather than the full 800 minimizations.

We measured coverage of removed (labeled) and retained (unlabeled) tests separately for each subject. Unfortunately, this step required some manual interventions due to idiosyncrasies of build environments and coverage tools. Therefore, *for coverage data only* our results are based on a sample of 50 runs out of the 800, selecting one Label 10% data point and one Label 20% data point for each of 25 randomly selected classes, to yield a good sample of coverage data.

For reduction type, we used JavaParser's [12] definition of statement types. The version of JavaParser we used classifies statements into 22 different types. In order to simplify discussion and presentation, based on the results obtained, we grouped the 22 types into (1) `If` statements (if, if-else, and if-else-if-else), (2) `Return` statements, (3) `Expression` statements (most often assignments, but also variable declarations within methods, and method calls that are not part of another statement type such as a `return`, etc.), and (4) `Other`, a catch-all class for the other, less common statement types.

IV. Empirical Results

A. Reduction Size

Table III shows the size of removals, in terms of both absolute numbers and % of statements removed. For Label 10%, mean statement removal is 17.31 statements or 14.06%. For Label 20%, the mean removal is 21.05 statements or 17.23%; doubling the number of removed tests does not proportionally increase reduction. The corresponding median numbers are 7.64 and 9.87% for Label 10% and 10.5 and 14.59% for Label 20%. Reduction size is less than 15 statements for 34 of 40 subjects for Label 10% and for 29 of 40 subjects for Label 20%. It is clear that reduction size is small, for randomly selected sets of tests; it is possible that larger reductions are more common when removed tests are grouped by functionality, but we expect most programs without very high quality tests to have fairly non-redundant test suites, so this effect should not be very large. One consequence is that even heuristics that rule our large numbers of statements as not being likely candidates for removal are potentially valid, so long as the a-priori rejected statements match a set of statements seldom, in practice, removed.

B. Reduction Height

Table IV shows mean and maximum reduction heights. It is clear that complex blocks are only rarely removed. For 10% test labeling, the average reduction height is greater than 2 for only 4 subjects out of 40. For 20% test labeling, the average reduction height is greater than 2 for only 8 subjects, and greater than 3 for only 1 subject out of 40. As more tests are labeled, complex removals only become modestly more common, and are never very frequent.

C. Reduction Type

Reduction type measures type of removals. As noted, the version of JavaParser that hddRASS uses defines 22 types of statement. We are primarily interested in knowing if certain statement types dominate removals. To measure this, we

TABLE III: Reduction size: how many statements are removed? Tests removed and reduction size are averaged across 10 runs. % Reduction is measured against total statements in the class as defined by Table I

.

	Label 10%			Label 20%		
Class	Tests removed	Reduction size	% Reduction	Tests removed	Reduction size	% Reduction
AntBuilder	3	6.45	4.54	4.4	9.53	6.73
Schedule	2.7	0.8	0.62	3.5	1	0.78
Project	3	11.27	3.87	6.1	43.27	14.86
AntScript	3	4.3	3.55	8.6	7	5.78
Available	34.1	26.05	24	7.14	23.2	17.44
Copy	2.6	79	44.1	5.1	88.1	49.2
FixCRLF	3.1	10.1	16.55	6.4	12.3	20.16
Checksum	2	31.5	22.18	2.5	39.2	27.60
UrlValidator	2.8	7.18	8.75	6.1	13.54	16.51
RegexValidator	1.4	2.4	6	2	3.6	9
DomainValidator	2.6	9.45	12.77	5.4	14.8	20
EmailValidator	1.6	3	16.66	2.9	3	16.66
Engine	5.3	46	56	7.9	46	46
JexlArithmetic	4.2	1.2	0.41	7	4.4	1.52
JexlEvalContext	3.9	7.1	24.48	8.4	7.4	25.51
Script	3.9	7.1	24.48	8.4	7.4	25.51
Option	1.3	6.3	7.41	1.6	8.3	9.76
GnuParser	4	2.2	9.56	4	2.2	9.56
PosixParser	6.3	0	0	10.7	0	0
OptionGroup	2.1	2.2	7.33	3.2	4.3	14.33
OntTools	3.2	6.5	10.76	5.7	7.5	11.53
LocationMapper	1.6	11.66	8.44	2.83	12.66	9.17
OntlClassImpl	2.6	2.5	1.87	2.25	3.5	2.61
OntlModelImpl	5.9	27	5.7	12.8	36	7.6
ExtendedMessageFormat	1.5	18.4	13.4	2.4	21.3	15.5
LevenshteinDetailedDistance	1.3	10.6	7.46	1.9	17.3	12.2
AlphabetConverter	1.4	8.1	9.87	2.3	10.5	12.8
StrBuilder	1.4	8.1	9.87	2.3	10.5	12.8
BinderClassLoader	2	0.7	7.77	2.1	1	11.11
CallMethodRule	2	22.4	39.48	3.1	22.44	38.69
Digester	2.1	207.4	48.0	3.5	235	54.39
NodeCreateRule	1.8	1.8	12	2.6	3	20
URIBuilder	2.4	0	0	4.6	0	0
HttpService	2	7.1	19.72	2.1	7.4	21.74
HeaderGroup	1.6	17.5	24.30	2.3	20.3	28.19
EofSensorInputStream	1.5	4.2	12.35	2.1	7.4	21.74
DefaultIntervalCategoryDataset	1.9	41.7	27.8	3.8	44.4	29.6
Hour	2.2	8.8	14.66	3.9	11.6	19.33
GridArrangement	1.5	10.33	8.6	3.4	13.11	11.0
StatisticalBarRenderer	1.3	14.30	9.08	2.2	19.62	12.34
MEAN	3.35	17.31	14.60	4.48	21.07	17.23
MEDIAN	2.15	7.64	9.87	3.5	10.5	14.59

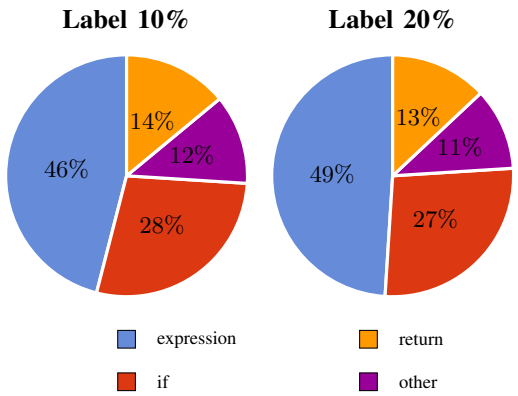

Fig. 2: Reduction type: what kind of statements are removed?

categorized removals into the four categories defined above (`If`, `Return`, `Expression`, and `Other`). Figure 2 shows the breakdown for removals across all 800 reductions. With 20% vs. 10% removed tests, the removal pattern did not change notably. `Expression`, `If` and `Return` statements were most common, by far (while `Other` has nearly as many total removals as `Return`, recall that it covers 19 different types, none of which are very frequent). Removal of `return` statements was somewhat surprising, as Java requires every non-void method to have a `return` statement, but a `return` inside an `if` block is frequently removable.

Of course, percent of total removals is not all that matters. If some kinds of statement are more common, this will also matter (e.g., if 90% of removals are `Expressions` but `Expression` statements are 95% of the program, this is less meaningful). In practice, the combination of both raw numbers

TABLE IV: Reduction height: how far from a leaf in the AST are removed statements? Averaged across 10 runs.

	Label 10%		Label 20%	
Class	Mean	Max	Mean	Max
AntBuilder	1.2	3	1.2	3
Schedule	0.4	2	0.8	4
Project	0.6	6	2.4	6
AntScript	1.3	2	1.7	2
Available	0.8	2	2.6	3
Copy	2.5	8	3.6	8
FixCRLF	1.6	2	2	3
Checksum	0	0	0.222	2
UrlValidator	1.6	3	2.2	3
RegexValidator	1.6	3	1.9	3
DomainValidator	1.3	3	1.85	3
EmailValidator	1	1	1	1
Engine	3	3	3	3
JexlArithmetic	0.4	2	0.6	2
JexlEvalContext	1	1	1	1
Script	1	1	1	1
Option	2.3	3	2.6	3
GnuParser	1.4	6	1.8	6
PosixParser	0	0	0	0
OptionGroup	1.2	2	1.5	3
OntTools	2	2	2	2
LocationMapper	1.8	2	2	4
OntlClassImpl	2	3	2.5	3
OntlModelImpl	3	3	3	3
ExtendedMessageFormat	2	2	2.3	3
LevenshteinDetailedDistance	2	2	2	2
AlphabetConverter	2	2	2	2
StrBuilder	1.6	2	2	2
BinderClassLoader	1.4	2	2	2
CallMethodRule	1.8	2	1.7	2
Digester	2.7	3	3	3
NodeCreateRule	0.6	1	1	2
EofSensorInputStream	3	3	3	3
URIBuilder	0	0	0	0
HttpService	1.4	2	1.1	2
headerGroup	3	3	3	3
DefaultCategory	2.1	3	2.2	3
Hour	1.6	2	2.1	3
GridArrangement	2	2	2	2
StatisticalBarRenderer	1.1	2	1.4	2
MEAN	1.55	1.85	2.76	3.14
MEDIAN	1.6	2	2	3

and probability is important to determining likely reduction targets. As it turns out, `If` and `Expression` are also simply more *likely* to be removed than other types of statement, in addition to dominating the raw numbers (Figure 3).

D. Reduction Relation

Test based program minimization relies on labeled tests, where (in this paper's simplified version of the process), labeled tests are removed from the set of tests the program must pass, in order to compute a reduction. We measured code coverage for both labeled (removed) and unlabeled (retained) tests. Our original approach assumed the availability of a *high quality test suite* for a program. Intuitively, if we have a good test suite, all removed statements (or at least almost all removed statements) should be covered by some test. In reality, of course, a program may have a poor, highly inadequate test

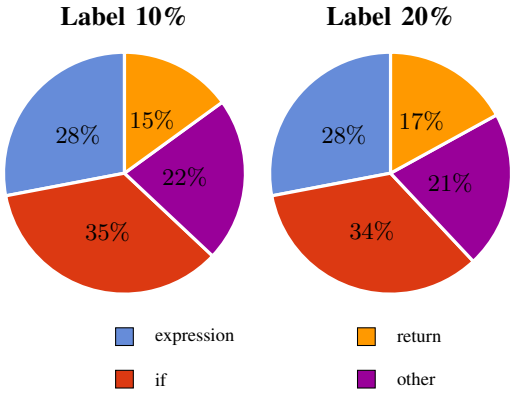

Fig. 3: Reduction probability: what is the breakdown of removal types, adjusting for statement type distribution differences?

suite, in which case removals will sometimes be due to code not tested at all (in which case test labels are not even relevant).

However, for the expected case, where code coverage is very high and test quality is good, we make predictions about statements to be analyzed according to their coverage. In particular, we can consider a few key subsets of statements:

$CL = \{ s \mid s \textit{ is covered by } \geq 1 \textit{ labeled (removed) tests} \}$.
$CU = \{ s \mid s \textit{ is covered by } \geq 1 \textit{ unlabeled (retained) tests} \}$.
$CU' = \{ s \mid s \textit{ is covered by 0 unlabeled (retained) tests} \}$

We consider these sets in the light of a few further claims: first, any statement not covered by a retained test (and not somehow required for successful compilation) will certainly be removed (this is basically a kind of dynamic dead-code removal). Second, any statement that *is* removed, despite being covered by retained tests, is likely *tested* by the labeled (removed) tests: that is, while the retained tests execute it, a removed test is actually checking its behavior. Together, these ideas suggest that $CL \cup CU'$ should contain most removed statements. We therefore define a set of *Coverage-Based Likely-removable Statements*, $CBLS = CL \cup CU'$. While only considering statements covered by the retained tests is somewhat obvious, not considering statements that are not also covered by the removed tests is an important and considerably more subtle point.

How does this prediction match reality? Tables V and Table VI show how the 50 sampled data points match up with the $CBLS$ hypothesis. For 39 of the 50 sampled coverage data points, *all* removed statements are in $CBLS$. Overall, 95.9% of removed statements were in $CBLS$. The relationship did not significantly change with change of labeled strategy, either. This is a fairly strong result, especially given that we expect test suites for real mission-critical systems built for adaptation in deployment to supply better test suites than typical open source Java projects.

TABLE V: Reduction Relationship: % of removed statements in *CBLS*

Relation	Avg	Median	SD	Mode	Min	Max
Label 10%	96.04	100	8.9	100	64	100
Label 20%	95.76	100	9.1	100	64	100
All	95.9	100	9.0	100	64	100

TABLE VI: Reduction Relationship distribution: distribution of % of removed statements in *CBLS*

Relation	<80%	80-89%	90-99%	100%
Label 10%	2	2	1	20
Label 20%	2	3	1	19
All	4	5	2	39

V. Heuristics

The high cost of computing a reduction derives from the fact that each attempt at removing a statement requires building a new version of the system and running (potentially) all the retained tests. Based on the results of our empirical study of reductions, therefore, we propose heuristics for reducing the number of statements to consider in computing a reduction. These heuristics are most important for deployed systems, where it is better to build a good candidate reduction quickly than to build the smallest possible system after an unacceptable long computation period, during which the system may be unable to perform critical functions. The proposed heuristics are all based on one of our empirical measurements discussed above.

A. Heuristic H1: Only Attempt to Remove Simple Statements

The results in Sections IV-A and IV-B show that relatively few statements are removed, and most of these are leaf nodes in the AST. We also expect that resource-uses will typically be via method calls, which are simple statements. We therefore first propose only trying to remove leaf nodes in the AST, and nodes one level above leaf nodes (to handle removal of simple conditionals and loops).

B. Heuristic H2: Only Attempt to Remove `If`, `Return` and `Expression` Statements

Section IV-C shows that certain statement types are most frequently removed. Avoiding attempted to remove other statement types may not save a large amount of time (since such statements are also less common in programs), but is also unlikely to be costly in loss of opportunity to reduce or failure to remove resource-using statements. We therefore also evaluate restricting the *type* of statement to remove. Note that due to the structure of programs, this heuristic has some overlap with the first heuristic.

C. Heuristic H3: Only Attempt to Remove Statements Contained in CBLS

Section IV-D shows that the vast majority of removed statements are in a set easily computable from a single run of the full test suite on the original program. Most removed statements are covered by the removed tests, or not covered by the retained tests, or both. We therefore finally propose only attempting to remove statements that match this coverage behavior. This heuristic is conceptually quite different than the other heuristics, in that it can ignore large numbers of leaf statements, including assignment statements and method calls. We therefore expect it to have the largest potential for speeding up reduction.

D. Validity of Heuristics

We validate these heuristics by applying them to 6 randomly chosen classes, taken from 2 open source projects not part of the empirical study corpus used to arrive at the heuristics. For these classes, we produced baseline results by applying hddRASS as described in the empirical study. For each class, we then applied each of the three heuristics in isolation to produce three additional reductions for comparison. We measured accuracy and efficiency for these additional, heuristic-guided, reduction processes.

1) Accuracy: Accuracy measures the similarity of heuristic-based results to baseline results. We measure accuracy as follows: $BR = \{ x \mid x$ *is a statement removed by the original approach*$\}$ $HR = \{ x \mid x$ *is a statement removed by the heuristic-based approach*$\}$. As the heuristics all involve subsetting the set of statements considered for removal, $HR \subseteq BR$. Accuracy is thus defined as a percentage, $\frac{|HR|}{|BR|} * 100$. If the baseline removes 20 statements and a heuristic-based run removes 18 statements, accuracy is 90%.

2) Gain in Efficiency: Efficiency measures how many statements a measure allows us to avoid attempting to remove; it is not a simple wall-time measure, because a the cost of a removal attempt varies with build time and test execution time. . We measure efficiency as follows: $BS = \{ x \mid x$ *is a statement the baseline approach attempts to remove*$\}$ $HS = \{ x \mid x$ *is a statement the heuristic-based approach attempts to remove*$\}$. Clearly, $HS \subseteq BS$, as above. We measure gain in efficiency as another percentage: $\frac{|BS|-|HS|}{|BS|} * 100$. If the baseline run attempts to remove 100 statements and the heuristic-based approach attempts to remove 70 statements then gain in efficiency is 30%. A gain in efficiency of 100% is of course, actually undesirable, as it would imply the heuristic rejected all statements. For all the heuristics, by construction, this basically cannot happen[3].

Table VII shows that all heuristics produce quite accurate results. H3 is more accurate than H1 and H2. H3 is also, it turns out, more efficient than the other heuristics (Table VIII), a rare case where there does not seem to be any tradeoff between accuracy and efficiency. However, all three methods appear useful, and it is not possible without further experimentation to be certain which approach is best for use in real-world applications, with meaningfully labeled test suites

[3]Technically, there could exist Java programs with no `if`, `return`, or Expression statements, and with no statements covered by the removed tests or not covered by the retained tests, but these would clearly be pathological, unrealistic, situations.

TABLE VII: Heuristic Accuracy (as %)

	H1		H2		H3	
Subject	L10	L20	L10	L20	L10	L20
Array2DRowRealMatrix	100	100	83	85	100	100
EigenDecomposition	100	100	75	85	100	100
MillerUpdatingRegression	85	89	85	85	100	100
SevenZOutputFile	100	100	86	88	100	100
ZipFile	89	92	66	73	100	100
ZipArchiveEntry	100	75	75	90	100	100
MEAN	96.25	97.25	85.75	88.75	100	100

TABLE VIII: Heuristic Gains in Efficiency (as %); broken down by Labeling for H3

Subject	H1	H2	H3-L%10	H3-L%20
Array2DRowRealMatrix	16	22	63	65
EigenDecomposition	27	19	26	33
MillerUpdatingRegression	29	12	51	15
SevenZOutputFile	15	11	24	23
ZipFile	6	9	16	20
ZipArchiveEntry	11	10	47	36
MEAN	17.33	13.83	37.83	32

and real resource-usage reduction goals. It could be that while H3 is the most accurate method, it is also the most likely to fail to remove some resource-using statements, for some non-obvious reason. Because our validation does not distinguish between resource-using and non-resource-using statements, we must turn to a more complete case study to begin confirming the superiority of H3.

VI. Case Study

We use the NetBeans IDE [13] (a popular IDE among Java developers) as a subject for our case study. The NetBeans IDE version we used has 7,386,809 LOC. The code base is well tested, with a large number of unit, function, and performance tests for all modules. The IDE uses significant system resources, including memory and CPU time. Our previous work on resource adaptation via test based software minimization also used NetBeans IDE to demonstrate automatic resource adaptation for memory. Resource adaptation was achieved by removing the IDE's undo-redo functionality. Full details of the experiment can be found in our previous work [3].

The target for adaptation, the *UndoManager*, implements the undo/redo functionality and is part of the *openide.awt* module. This module consists of 11,284 lines of code, and has 146 tests. We continue to use same 3 labeled tests used in the original case study. The baseline run removed 130 statements, none of which were more than 5 levels above a leaf node. By providing a memory profile of NetBeans IDE, we demonstrated that the adapted IDE uses less memory. By carefully observing tests, module, and coverage information, we can identify the exact 19 statements that, when executed, fill up certain buffers, making undo-redo operations so memory intensive. Those statements were automatically removed by hddRASS in our original case study. Unfortunately, computing this reduction required 175 minutes – nearly 3 hours!

TABLE IX: NetBeans IDE case study: Accuracy and efficiency gain for heuristics. Accuracy-all measures accuracy for all removals. Accuracy-res measures accuracy for the 19 critical, resource-using statements.

NetBeans IDE	H1	H2	H3
Accuracy-res	100	100	100
Accuracy-all	80	88	86
Efficiency	25	20	56

TABLE X: NetBeans IDE case study: Time (in minutes) to build a memory-adaptive NetBenas IDE.

NetBeans IDE	baseline	H1	H2	H3
Time	175.45	106.39	135.51	61.01
Improvement	1	1.64	1.29	2.87

In order to confirm that our heuristics are accurate and provide efficiency gains in a realistic setting, we applied them to the same case study. We measured accuracy and gain in efficiency, as above, and also verified (1) that the 19 critical statements were all removed and (2) that the version of the IDE produced indeed did now allow undo/redo operations. The critical statements were, indeed, removed by all three heuristic approaches, undo/redo functionality disabled, and the memory footprint of the IDE significantly reduced.

Table IX shows accuracy and gains in efficiency for all 3 heuristics. Even with a real, complex case study, and tests focused on a realistic functionality, the heuristics provide a large benefit. In particular, again, H3 performs well: it provides 86% accuracy, while analyzing 56% fewer statements.

We also computed the actual time required to build a memory-adaptive NetBeans IDE using the baseline approach and the heuristics-based approaches, as shown in Table X (results differ from the original experiment, due to using a different hardware configuration). Using heuristic H3, we can build a memory-adaptive NetBeans IDE that is just as useful (in terms of memory reduction) *2.87 times faster than without using a heuristic.*

VII. Discussion

Resource adaptations via test-based software minimization is a conceptually simple and practically useful, but, unfortunately, quite expensive way to build resource-adaptive software systems. The high cost of this kind of adaptation is due to three primary factors:

1) hddRASS uses a slow algorithm. hddRASS is a modification of HDD* [7], with worst case complexity of n^3.
2) hddRASS considers all program statements as potential targets for removal.
3) Every potential reduction involves builiding a new version of the system and running (potentially) the entire retained test suite.

Improvement in any of these areas will improve the performance of our approach. E.g., a better algorithm for computing a reduction (across the same possible removed statements, perhaps by removing more statements at once, successfully) would help with the first issue. Using partial builds and prioritizing tests effectively might address the third problem. This paper is a step in the direction of improving performance by addressing the second cause: there are too many possible reduction targets. We performed an empirical analysis of a large number of class reductions, and used the information obtained to predict statements likely to be removed, resulting in three heuristics:

- H1: Remove only statements close to the leaf nodes of the AST.
- H2: Remove only `If`, `Return`, and `Expression` statements.
- H3: Only remove statements covered by the removed tests or not covered by the retained tests.

The only additional step required, beyond the original algorithm, is to compute coverage for all tests in the full suite, and associate the coverage data with retained and removed tests. Given that each potential removal may run much of the test suite, this is a small price to pay for significant reduction of the number of statements to consider for removal.

It is no surprise that H3 is most effective. In a sense, H1 and H2 are fundamentally limited in two ways: first, they cannot (at least usually) provide a very large reduction in number of statements to consider, since most statements are close to leaf nodes of an AST in most programs, and most statements are `If`, `Return`, and `Expression` statements. H3, however, is not bounded by the basic structure of normal Java programs. It can throw out large numbers of leaf nodes and "normal" statements as unlikely to be removable. On the other hand, it will attempt to remove non-leaf nodes that coverage suggests may only be needed to pass the removed tests. In other words, it can be both more efficient (in terms of throwing out statements unlikely to be removable) and more conservative/accurate (in terms of keeping some statements that are likely to be removable).

Since our initial results suggest that the dynamic, coverage-based heuristic works best, they suggest that future attempts to improve the speed of reduction should, perhaps, also be based on information from runtime and the test suite. E.g., for the NetBeans reduction, there are only 19 statements that are critical to improve system resource usage. If we can identify resource-using statements at runtime on the full test suite, perhaps we can further focus on those statements. However, such a focus is complicated, since other statements may have to be removed to produce a "clean" removal of functionality. It is useful to have an NetBeans IDE that does not provide undo/redo; it is less useful to have one that crashes when undo/redo actions are attempted. Data-flow from resource-using statements in the H3-predicted removals based on a set of tests may help guide a further refinement of the method, based on H3.

A. Threats to validity

Our idea of adaptation assumes that test suites are not only adequate, but very high quality; in the absence of such tests, the validity of results is questionable. It is also questionable in the absence of meaningful test labels. This means that our empirical results, other than the case study, are useful for understanding basic statistics of removal, but may not resemble real-world application patterns as closely as we would like. However, our empirical results had a very strong resemblance to results for the NetBeans IDE case study which has a good test suite and meaningful labeling, and test coverage for most classes included in the empirical results was good; for 17 of the 40 classes, it was probably as good as can be expected even in many real-world adaptation efforts (>90%).

Our results are also only valid for Java programs, and we rely on JavaParser's notions of statement. Generalizing to other programming languages would not be warranted, at least for non C-like languages (we suspect C and C++ at least would have similar patterns). Whether Haskell, Ruby, or Python programs would have the same reduction statistics is more uncertain, though in a sense, the coverage patterns are likely to be fairly universal due to their grounding in the semantics of the program and test suite.

Our subjects in the empirical study were open source Java programs used in other software engineering research. The extent to which our results would apply to other classes and programs is not certain; however, the results across all classes and randomly computed test labelings were highly consistent, suggesting that the results might generalize to at least a large number of other open source Java Programs.

The hddRASS tool is available on GitHub at https://github.com/amchristi/hddRASS. The release includes both the complete, non-heuristic implementation and all three heuristics introduced in this paper. All subjects used for the empirical study and for heuristics validation are classes of popular open source projects and available online. The NetBeans IDE source code is also available online. All results should therefore be easily replicable.

VIII. Related work

This paper extends our previous work on using test-based software minimization to adapt programs to resource-poor

environments by automatically removing non-critical features that use resources [3], by empirically studying a much larger set of reductions and using the results to derive heuristics to speed adaptation.

The field of self-adaptive systems has enjoyed renewed interest in recent years, with the growth of both autonomy and ubiquitous computing resulting in more systems that are difficult to communicate with, and so must be able to "go it alone." Salehie et al. [14] summarize much of the earlier recent work in the field. Different engineering approaches to building adaptive software systems are categorized by Kruptizer et al. [2]. Cheng et al. [4] suggest *adaptive requirements engineering* to capture uncertainty in an adaptive software system, and envision a new requirements language that captures what a system *might* do instead of what a system *will* do. Our test-based version of sacrificability of specifications is a limited, but easy-to-apply, version of *adaptive requirements*, where the only possible adaptation of requirements is the possibility of removing certain requirements, as represented by labeled tests. Whittle et al. [15] propose a requirements language, RELAX, that encodes the ability to relax certain requirements at runtime if the environment changes. RELAX is designed to sacrifice requirements marked as non-critical in order to adapt to changing environment, while still continuing to satisfy all critical requirements: it is thus a language-based, rather than test-based, analogue to our approach. Delemos et al. [16] categorize self-adaptive systems as self-managing systems that rely on explicitly pre-computed adaptations, in contrast to self-organizing systems which rely on implicit runtime adaptations. The present work moves our test-based approach closer to self-organization. With slower adaptation, it is harder to go beyond a pure self-management approach. Fredericks et al. [17] suggest choosing only a subset of test cases to execute based on resource constraints for runtime adaptations, but do not use this to guide an adaptation.

Delta-debugging [18] is an algorithm for reducing the size of failing test cases or test inputs. *Hierarchical delta debugging* (HDD) [7] was proposed as way to efficiently reduce hierarchical inputs, such as computer programs. *Cause reduction* extended those ideas to a much more general applicability, including our use of reducing programs with respect to tests they pass [19], [20].

The idea that modifications of a program that are both useful and computationally tractable to identify are likely to be (statement) deletions was proposed by Qi et al. [9] in their criticism of much work in automatic program repair. Our approach is a (very specialized) instance of program repair, that aims to minimize the need for a complete specification and reduce computational needs by only using statement deletion. Because we only use deletion, modified programs are both easier to compute (there are fewer alternatives, and we can exploit an HDD-like algorithm). Deletions are also more likely to be valid even in the absence of a full specification. Changing expressions in code, rather than simply removing computation, is more likely to produce subtly incorrect results that escape even a good test suite. This is demonstrated by the cases in the work of Lin et al. where a test suite that kills all statement deletion mutants has a weaker mutation score on a set of mutants containing other operators [21], or even by the simple fact that the most obviously non-equivalent mutants when using mutants to drive test suite improvements are usually statement deletions [22].

An additional difference between our approach and program repair efforts is that program repair typically needs an actual fault to repair (and in adaptation, would thus start from a failure caused by resource usage), while our approach can compute, blindly, numerous potential reductions that sacrifice different functionality, any of which can be applied when needed. This means our approach can, in theory, "anticipate" unusual or ill-defined "resource" usage problems, even when these have not been thought of by a developer, or experienced in the field. In program repair, on the other hand, fault localization can be used to identify good candidate statements for modification [23]; because we lack a fault, we instead must make use of statistical characterization of deleted statements, and the coverage relationship of statements with removed and retained tests (the topic of this paper).

Conceptually, this relates to the statement-deletion mutation operator [24], a special instance of deletion mutation operators [25] known to achieve a good balance between the number of mutants generated and the value of the artificial faults produced Our hddRASS algorithm is a combination of the ideas of HDD and statement-deletion, with heuristic optimization for the case where the program is nearly minimal already (unlike a test input, which is often far from minimal), and where dependencies tend to flow forward in the source code. This paper extends our previous work on the topic [3] by using an empirical study of reductions to motivate heuristics to speed adaptation, achieving nearly a factor of 3 improvement in reduction time for a real-world case study. In the future, we are likely to examine other modifications of delta-debugging [26] to see if they promise further improvements in the runtime of hddRASS.

IX. Conclusions and Future Work

Building robust resource-adaptive systems is critical to the long-term goal of producing software systems that can effectively respond to their changing computational (and physical) environments. Because anticipating all possibilities for trading reduced functionality for lower resource usage is extremely difficult for developers, there is an ongoing need for methods that allow software to adapt without human intervention.

In previous work [3] we showed that by labeling tests, developers could easily indicate a basis for computing resource adaptations, without the burden of learning a specification language or performing extensive program annotation. The adaptation works by removing statements not required to pass a reduced test suite. Unfortunately, the process is also very slow, making it impractical for use in field adaptation in deployed systems with limited computational resources. This paper presents a first step towards practical in-the-field test-based software minimization. We show that, by examining

patterns in a large set of reductions of Java open source projects, using random subsets of their test suites, it is possible to identify promising heuristics for which statements will eventually be removed in a reduction. It is then simple to only try removing those statements. All three heuristics we derived maintain an acceptable accuracy of reduction, while significantly reducing the time taken to compute the reduction. One heuristic, in particular, based on test suite coverage, is both most accurate and provides the highest gain in efficiency.

As future work, we aim to extract more information from test suites, including effective ways to prioritize test execution to speed rejecting statements that cannot be removed, and ways to predict statements that can certainly be removed without the expense of running tests (e.g., using code coverage or data-flow to assertions, as in checked coverage [27]). We also plan to apply our approach to large-scale real-world systems operating in an Android environment, as part of the DARPA BRASS project.

Acknowledgments: this work was partly funded by the DARPA BRASS [1] program, and the authors would like to thank our collaborators at Oregon State University and Raytheon/BBN.

REFERENCES

[1] J. Hughes, C. Sparks, A. Stoughton, R. Parikh, A. Reuther, and S. Jagannathan, "Building resource adaptive software systems (BRASS): Objectives and system evaluation," *SIGSOFT Softw. Eng. Notes*, vol. 41, no. 1, pp. 1–2, Feb. 2016.

[2] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Pervasive Mob. Comput.*, vol. 17, no. PB, pp. 184–206, Feb. 2015.

[3] A. Christi, A. Groce, and R. Gopinath, "Resource adaptation via test-based software minimization," in *11th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2017, Tucson, AZ, USA, September 18-22, 2017*, 2017, pp. 61–70. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SASO.2017.15

[4] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, in *Software Engineering for Self-Adaptive Systems*, B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

[5] M. Luckey and G. Engels, "High-quality specification of self-adaptive software systems," in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 143–152. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487336.2487359

[6] F. Fleurey and A. Solberg, "A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 606–621. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04425-0_47

[7] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 142–151.

[8] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[9] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[10] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Test case generation for program repair: A study of feasibility and effectiveness," *CoRR*, vol. abs/1703.00198, 2017. [Online]. Available: http://arxiv.org/abs/1703.00198

[11] M. Martinez and M. Monperrus, "Astor: A program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 441–444. [Online]. Available: http://doi.acm.org/10.1145/2931037.2948705

[12] "Javaparser." [Online]. Available: http://javaparser.org/

[13] "NetBeans IDE." [Online]. Available: https://netbeans.org/

[14] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[15] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems," in *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE*, ser. RE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 79–88. [Online]. Available: http://dx.doi.org/10.1109/RE.2009.36

[16] R. De Lemos, H. Giese, H. A. Muller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezze, C. Prehofe, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. Dagstuhl Seminar Proceedings, R. De Lemos, H. Giese, H. Müller, and M. Shaw, Eds. Springer, 2013, vol. 7475, pp. 1–26.

[17] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng, "Towards run-time testing of dynamic adaptive systems," in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '13, 2013, pp. 169–174.

[18] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[19] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *IEEE International Conference on Software Testing, Verification and Validation*, 2014, pp. 243–252.

[20] ——, "Cause reduction: Delta-debugging, even without bugs," *Journal of Software Testing, Verification, and Reliability*, vol. 26, no. 1, pp. 40–68, 2016.

[21] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ser. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 84–93. [Online]. Available: http://dx.doi.org/10.1109/ICST.2013.20

[22] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney, "How verified is my code? falsification-driven verification (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2015, pp. 737–748.

[23] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013. [Online]. Available: https://doi.org/10.1007/s11219-013-9208-0

[24] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *International Conference on Software Testing, Verification and Validation*, March 2013, pp. 84–93.

[25] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *International Conference on Software Testing, Verification and Validation*, 2014, pp. 11–20.

[26] R. Hodován and A. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, ser. A-TEST 2016. New York, NY, USA: ACM, 2016, pp. 31–37.

[27] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, March 2011, pp. 90–99.