AN ABSTRACT OF THE DISSERTATION OF

Arpit Christi for the degree of Doctor of Philosophy in Computer Science presented on September 20, 2019.

Title: Building Self Adaptive Software Systems via Test-based Modifications

Abstract approved: _____

Alex Groce

Building software systems that adapt to the changing environment is challenging. Developers cannot anticipate all the changes in advance, and even if they could, the effort required to handle such situations is too onerous for practical purposes. Self Adaptive Software (SAS) adapts itself as per changing environment. The area of building SAS has observed immense contribution from both researchers and practitioners.

This dissertation proposes a new technique to build SAS called Test-Based Software Minimization (TBSM) that relies on using tests to define functionality that can be sacrificed to achieve resource gain. TBSM combines Hierarchical Delta Debugger and statement deletion mutation to automatically build SAS using a labeled test suite where test labels define association of test with features or resource usage. We implement the technique using a tool called hddRASS. We demonstrate the easy applicability, usability, and effectiveness of TBSM using 2

real-world adaptation scenarios.

TBSM is applicable, effective, and usable but it is a very inefficient technique. Our research discovers the reasons for inefficiency and tries to mitigate the major reason; the search space. TBSM relies on applying the time-consuming reduction process to the whole program which removes program statements to achieve resource gain. Hence, the search space for TBSM is the whole program. We empirically demonstrate that the removed statements have predictable characteristics and therefore it is possible to use heuristics to select target statements for removals, reducing the search space. Our research proposes 3 heuristics for target selection and evaluates them for efficiency and accuracy.

Spectrum-Based Fault Localization ranks program statements by the likelihood of a statement being faulty using spectra of passing and failing tests. Our research proposes AdFL, a repurposing of Fault Localization that can shrink and prioritize the search space for TBSM. We empirically demonstrate that AdFL is better than previous heuristics in reducing the search space. We utilize 2 real-world adaptation scenarios to demonstrate the effectiveness of AdFL in reducing the search space while retaining accuracy. We also combine AdFL with previous heuristics for TBSM to propose an incremental, best-effort variant of TBSM.

Building Self Adaptive Software Systems via Test-based Modifications

by

Arpit Christi

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented September 20, 2019
Commencement June 2020

Doctor of Philosophy dissertation of Arpit Christi presented on September 20, 2019.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Arpit Christi, Author

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor Dr. Alex Groce for supporting me throughout the doctoral program. He entrusted me to work on the critical DARPA IMMORTALS project. He provided me a great degree of freedom in exploring research ideas while ensuring that we are making good progress on IMMORTALS project. As Self Adaptive Software was an unexplored topic for our research group, initially, I was out of ideas and even confused at times. He helped me to connect self adaptive research with other research ideas that our group already explored, making my research path clear and easier. He spent numerous hours reading my work, giving me feedback on my writing and many times correcting my writing to make my work more readable.

I would also like to thank my fellow graduate students at the Software Engineering Research Lab. I particularly want to thank Rahul Gopinath and Mohammad Amin Alipour. I almost always discussed my research ideas first with them before approaching Dr. Groce. They listened patiently and provided valuable and critical feedback. I also want to thank Chaoqiang Zhang, Ali Aburas, Hongyan Yi, Iftekhar Ahmed, Micheal Hilton, and Yuanli Pei for their friendship and collaboration.

This PhD would not be possible without the support and encouragement of my family. I want to thank my parents Pravina and Moses, for establishing the value of education in my life. My wife Angelina and daughter Aylin stood by my side patiently throughout the journey, believing in me and encouraging me all the time.

CONTRIBUTION OF AUTHORS

The following authors contributed to the manuscript: Arpit Christi (AC), Alex Groce (AG), Rahul Gopinath (RG), and Austin Wellman (AW). Their individual contributions are summarized in Table 1.

Table 1: Author Contributions

Task	Conception	Data Collection	Empirical Analysis	Final Draft
Chapter 2	AC, AG	AC	AC, RG	AC, AG, RG
Chapter 3	AC, AG	AC	AC	AC, AG
Chapter 4	AC, AG	AC	AC,AG	AC, AG, RG
Chapter 5	AC, AG	AC, AW	—	AC, AG, AW

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF ALGORITHMS

LIST OF APPENDIX FIGURES

Chapter 1: Introduction

Modern software systems have varying and complex resource needs that change frequently based on changing environment. The inability of software systems to effectively adapt to these resource changes may lead to inferior and potentially vulnerable software [42]. When modern software systems like Internet of Thing (IoT) applications, Ultra Large Scale Systems, and cyber-physical systems fail to adapt to resource variations effectively, the consequences can be fatal or critical. Resource adaptation is such a serious and costly issue for mission-critical software that practitioners and researches are devoting significant effort to solve the problem. To this end, Defense Advanced Research Project Agency (DARPA) started a new program called BRASS (Building Resource Adaptive Software Systems) [42].

Self Adaptive Software (SAS) modifies its behavior in response to changes in operating conditions[1] [61]. Resource Adaptive Software (RAS) is a subset of SAS where the reason for adaptation is unavailability or variability in one or more resources. As RAS is a subset of SAS, it can change its behavior with respect to resource variations. As this dissertation mostly concentrates on RAS, we will use RAS and SAS interchangeably throughout the dissertation.

As part of DARPA's BRASS project, we work with a software development team trying to build a real-world mission-critical RAS called Tactical Situational

[1]We use one of the many definitions of SAS.

Awareness System (TSAS) mainly used for military purposes. TSAS is sometimes deployed at locations where resource availability is scarce and varying.

1.1 Limits of existing techniques to build SAS

Researchers and practitioners have proposed many tools, methods, guidelines, frameworks, and techniques to build SAS. Cheng et al. and de Lemos et al. detailed the possible modification needed to the existing Software Development Life Cycle (SDLC) to incorporate the development of SAS [13, 22]. Krupitzer et al. summarize and categorize different engineering approaches to build SAS [47] that includes but not limited to modeling based approach, architectural specification based approach, control theory based approach, and learning based approach etc. Salehi et al. summarize recent projects that attempt to develop SAS [70]. Some notable projects are Quo, Oceano, MADAM, CASA, DEAS, J3 and ML-IDS [70, 47, 8, 7, 30, 59, 48, 74]. Most of these tools, techniques, and projects target prototype systems, specific applications, or a particular scenario. While proposing and developing RAINBOW framework to build SAS, Garlan et al. noted the non-reusability and limited applicability of previous approaches [33]. Later, researchers proposed many frameworks like MUSIC, FESAS, FUSION, SASSY, MOSES, etc. that attempt to tackle reusability issue in SAS development [39, 46, 27, 55, 11]. Not all the techniques or frameworks were incorporated with a tool for easy applicability. FESAS-IDE and RAINBOW are a few techniques with built in tool support [45, 33]. Most of these tools, technique, and projects require developers to

learn domain-specific languages (DSL), formal specifications, modeling techniques, architectural specifications etc. [52, 26, 69] Also, developers need to map their applications as per the constraints of these techniques, obviously, a time-consuming and error-prone task. All these present a high entry barrier to developers. Hence, in spite of recent advances, building SAS remains a challenging problem.

1.2 Limits of existing techniques to capture adaptive software requirements

Self adaptive software requirements specification is an open research problem with very little contribution [13]. Formal specifications, modeling languages, DSL have been proposed to capture adaptability and uncertainty in SAS requirement specification[53, 29, 75]. Whittle et al. develop a DSL known as RELAX that provides expressions to capture uncertainty in requirement specification by allowing analyst to identify certain non-critical requirements as sacrificial [75, 68]. Several Goal-based modeling approaches were proposed to capture adaptive software requirements, they all require additional work to be performed by analysts [58, 68, 34, 14]. Again, all these approaches force developers to learn formal specification, DSL or other techniques.

1.3 Limited reset resource budget while performing adaptations for systems in the wild

De Lemos et al. categorize self-adaptive systems into (1) self-managing systems which rely on explicit pre-computed adaptations (2) self-organizing systems that rely on implicit run time adaptations [22]. For SAS in the wild, when adaptations are performed, the system (or part of the system) is in reset mode and cannot be utilized. For mission critical systems a very long reset time is not preferred. If only a component of the system is being adapted, we can continue to use rest of the system. Even in that case, the adaptation process competes for resources (CPU cycles, RAM, bandwidth) with functioning system creating performance issues for existing system. So, in both cases, either the complete system is in reset mode or only a component is in reset mode, long reset time should be avoided.

1.4 Research Goals

Philosophically, at a high level, this dissertation has the following research goals.

1. A simpler way to capture adaptive software requirements that developers can apply with little training and without learning any new tools or technology.
2. A highly applicable, usable, and effective technique to build RAS that can be applied to a large set of applications.
3. A readily available tool implementing the technique that developers can utilize without much effort.

4. The technique (and tool) should be able to efficiently build resource adaptation in a limited time budget, if we want to use it for the systems in the wild.

1.5 Structure

This dissertation is based on the following papers:

Chapter 2 presents our paper "Resource Adaptation via Test-Based Software Minimization" which was published at SASO 2017. This paper captures first, second and third research goals by proposing (1) test labeling as a way to capture adaptive software requirement (2) Test-Based Software Minimization (TBSM) as a new technique to build RAS (3) a new tool called hddRASS that implements the technique.

Chapter 3 presents our paper "Target Selection for Test-Based Resource Adaptation" which was published at QRS 2018. This paper was our first attempt to realize the fourth research goal. It does so by proposing 3 heuristics. Two of them are based on empirical findings and one is based on coverage relationship and empirical findings.

Chapter 4 presents our paper "Evaluating Fault Localization for Resource Adaptation via Test-Based Software Modification" which was published in QRS 2019. This paper also captures the fourth research goal by proposing Adaptation Fault Localization or AdFL as a way to reduce and prioritize the search space for TBSM.

Chapter 5 presents our paper "Building Resource Adaptations via Test-based Software Minimization: Application, Challenges and Opportunities" which is selected for publication in ISSRE 2019 - Industry Track. It encompasses all the research goals by evaluating TBSM for effectiveness, usability, applicability, and scalability using 2 case study scenarios.

1.6 Contributions

This dissertation makes the following contributions:

1. Proposing a way to capture adaptive software requirements via test labeling (Chapter 2).

2. Proposing a technique - TBSM to automatically build RAS and implementing it in a tool called hddRASS (Chapter 2).

3. Identifying reasons for inefficiency in TBSM. Proposing three heuristics to improve the efficiency and evaluating them. (Chapter 3 and Chapter 5).

4. Proposing AdFL, a repurposing of Fault Localization (FL) for TBSM that can shrink and prioritize the search space more effectively than previous heuristics. Also proposing an incremental, best-effort variant of TBSM that uses AdFL to prioritize the search (Chapter 4).

Manuscript 1

Resource Adaptations via Test-Based Software Minimization

Arpit Christi, Alex Groce, Rahul Gopinath

2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Tuscon, Arizona.

pp. 61-70

Chapter 2: Resource Adaptation via Test-Based Software Minimization

2.1 Introduction

Modern software systems are often complex, and use numerous resources – obvious ones, such as memory, storage, and network bandwidth – and less obvious "resources" such as software libraries. While developing such software systems, implicit or explicit assumptions are often made about their resource usage or the availability of resources. For example, a mobile navigation app assumes availability of location provider services via network, GPS or other means. A problem occurs when the software is used in a situation it was not designed for, where the original resource assumptions made during development no longer hold. For example, the navigation app may no longer be able to access the network or GPS satellite when in the middle of a forest or under water. Moreover, the availability of such resources is often subject to change, with older resources obsoleted and discarded by newer technology. These advances in underlying technology can often force software developers to adapt or evolve their software. For instance, a change in a library that the software depends on can force its developers to refactor or rewrite part of the software. These rewrites often require multiple cycles of development and testing before the application can be fully adapted, which is costly, time con-

suming and error prone. One way to tackle these problems is to let the software evolve itself in response to change [47]. Adaptations may manifest as *restricted functionality*, *altered functionality*, or *enhanced functionality* [42].

We are working with a group of developers trying to build a real world resource adaptive software system for military applications called the *Tactical Situational Awareness System* (TSA). One of the components of TSA is its location provider. It continuously sends the location of the device to the server. The location information typically includes actual location coordinates, images, and streaming video of the location. The location provider of TSA needs to be available in extreme locations such as remote battlefields, forests, or under the sea where resource availability is drastically different from the expected environment. That is, devices that TSA is deployed on may have varying hardware depending upon the environment, and parts of available hardware may not function in some environments. Hence, the quantity and quality of resources is not known in advance and will vary drastically. Two primary strategies are employed by developers to make the software adapt to varying resource availability: **1) Reduction:** use reduced versions of original components that do not satisfy all aspects of the original specification, achieved by removing or avoiding executing part of the code. **2) Replacement:** use components that are altered from the original components – these new components use different libraries, data structures, hardware, and resources.

The *reduction* strategy is often used when it is possible to lower resource usage by relaxing invariants or specifications. It may be accomplished either by turning off some features or by not executing certain parts of the code such that a resource

under stress will be less utilized, or not utilized at all. The *replacement* strategy is used when it is possible to come up with completely different components with different resource profiles. Usage of these changed components should alter the resource usage so that the application can weather the resource degradation gracefully.

Currently, both strategies are executed manually. A developer has to carefully observe component resource needs and consider the impacts on system specifications in order to come up with correct adaptations. Further, adapted components have to be identified, associated with resources, and verified against their resource consumption. Changes have to be verified against the *new or reduced set of specifications* they are going to satisfy. All these steps are manual, error-prone and tedious, and trigger new cycles of development and testing. In this paper we focus on a method to allow systems (broadly considered) to perform *reduction* adaptations themselves, using only their own test suites.

Resource adaptive software systems[1] are designed and implemented mostly for mission critical software systems. Hence, they are often accompanied by a high quality test suite that captures and verifies the specification of the system adequately. When we perform adaptation by reduction, some of this specification may have to be sacrificed. For reduction based adaptations, the *sacrificability of specifications* can often be represented cleanly and simply by annotating the test suite. These annotations may be at the test case level, at the invariant/property level, or use a combinations of these methods. The annotations may take the form

[1]Often abbreviated as *RASS*.

of marking test cases that exercise a given feature or turning off specific asserts or invariant checks. Given such annotated test suites, we show that an automated program reducer can automatically adapt a system to use fewer resources.

We perform program reduction using a modified *hierarchical delta debugging* technique combined with *statement deletion mutation*. The program is reduced until a locally minimal version of the program is found that can still pass all tests corresponding to a certain level of preserved specification satisfaction. Delta-debugging [81] is a binary-search like algorithm intended to reduce the size of failing test cases. It removes components of a test that are not required in order for the test to fail. Hierarchical-delta-debugging (HDD) [56] modified the algorithm to produce better results for tree-structured test inputs, especially, e.g., programs input to compilers. *Cause reduction* [37, 38] further proposed that delta-debugging/HDD are not limited to reducing tests with respect to the property "the test fails" – they can also reduce a test so that it is shorter but covers the same code, uses the same amount of memory, and so forth. In this paper, we argue that HDD's aim of reducing inputs that are themselves source code means that delta-debugging-like methods can also be used to reduce actual programs, with respect to the property that the program still passes a set of tests representing required program behavior. The basic idea is mentioned in the cause reduction journal paper [37]; our insight is to exploit this concept for resource adaptation, by only requiring the program to pass some tests, rather than all tests. At heart, we are proposing labeling of tests (and using them to control reduction adaptations) as a response to the need for a requirements vocabulary that expresses flexibility and uncertainty

```
final static Logger log = Logger.getLogger(logname);
...
   log.info("Begin transmit ",b,"bytes");
...
   if (failed) {
      log.error("Transmit ",b," failed");
      handle_failure();
   } else
      log.info("End transmit ",b,"bytes");
```

Figure 2.1: A fragment of a Java program with potentially over-aggressive logging.

proposed in a research roadmaps for software engineering for self-adaptive systems by Cheng et al. [13]. The advantage of our approach is that it is both conceptually simple and practically applicable to existing systems: simply by labeling some tests, a potential reduction can be defined. Unlike a novel requirements language, developers already tend to "speak" the language of tests.

2.1.1 A Simple Example: Removing Logging

Consider the code in Figure 2.1. Suppose that this is part of the code for a system that sometimes operates in an embedded environment with very limited flash file storage, and where core system functionality requires storing critical image files. These files require only constant sized space (they are held until they can be transmitted), but unfortunately the logging information on transmissions and other events fills the limited flash storage. While the logging stops when the space is filled, the system does not have an automatic way to reclaim the space taken by the logs. Of course, the developers could write such a behavior, explicitly,

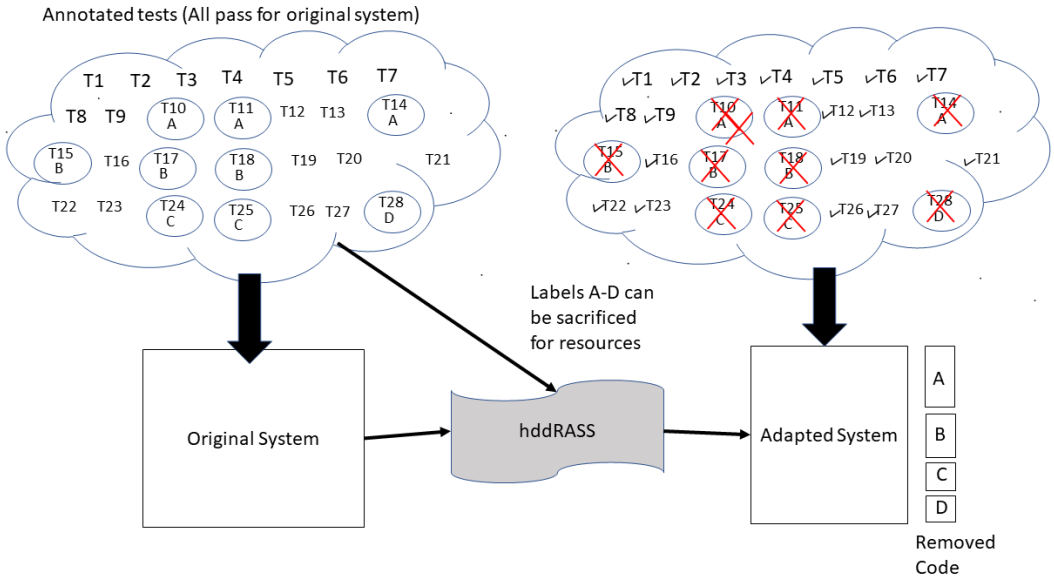

Figure 2.2: The test-based approach to resource adaptation by program minimization

and modern logging systems are usually relatively easy to configure on the fly. However, it is possible to automatically construct a version of the program with either less logging or no logging, depending upon our needs, that conforms not to a programmer's possibly erroneous model of which logging is essential, but to a much more concrete specification of what the system must log, and how important it is: which tests fail when the logging is removed. Our assumption is that in the long run, critical software systems require tests that fail when any important functionality is removed; the novel aspect is that we also believe it would be highly beneficial to have test suites that, at a high level, mark *which* functionality each test checks.

Imagine that the test suite for the system in Figure 2.1 has 250 very thorough tests. Of these tests, only 3 check the behavior of the logging of transmission beginning and ending. An additional 5 tests check that when a transmission fails, that error is properly logged. Furthermore, suppose the 3 tests checking begin/end log messages are labeled `logtransmission` and the 5 tests checking for failure messages are labeled `logfailtransmission`. Finally, 50 tests are labeled `critical`: if these tests fail the system cannot be said to provide its most basic, essential functionality. Other tests may have other labels (and some tests will have more than one label). The most basic version of automatic reduction adaptation proceeds as follows: the system selects some set (to start with, of size 1) of labels that does not include `critical`. Tests marked with these labels are allowed to fail – the labels specify sacrificability of specifications. We use an algorithm based on methods for minimizing test cases and detecting weaknesses in test suites to *automatically*

generate a version of the system that 1) passes all tests that are not marked with the selected sacrificed labels and 2) is otherwise as small as possible, within the limits of the algorithm we use to find code to remove (a locally minimum program). In this case, if the label selected is `logtransmission`, the reduced version of the system will remove both `info` logging calls (and the entire `else` block of the `if` statement) but no other code. If the label is `logfailtransmission` the code removed will be the `error` logging call. Finally, if both labels are sacrificable, all logging calls will be removed and the creation of the `log` object will also be deleted. Running the still-passing tests, a resource monitor can observe that use of storage was reduced on average (not all tests will have transmissions, but if some do the change is visible), and add these removals to a database of adaptations to mitigate storage resource issues. Moreover, in addition to storage usage, the adaptation removing all logging is also applicable in the event that the system migrates to a platform with a new, incompatible API for logging (so the old system will not even build, or will crash when it attempts to access logging).

As discussed below, there are many possible variations on this basic idea, including on-the-fly generation of variants during system operation rather than pre-computing reduced variants to apply; however, the basic concept remains: determine a set of tests representing specification aspects that can be sacrificed in pursuit of resource savings, and compute a version of the system that is reduced with respect to the constraint that it still passes all other tests. Figure 2.2 shows the general concept. Assume that we are willing to sacrifice the

aspects of the specification/functionality represented by test labels A-D[2]. Tests that are not circled have other labels, and must still pass (and tests 1-9, in bold, are critical tests that can never be sacrificed). Our hddRASS tool (available at `https://github.com/amchristi/hddRASS`) takes this sacrificability of specifications and computes a reduced version of the original system that fails most of the tests labeled A-D (one test still passes, because it actually tests only behavior required by other labels), and has less code. The resources used by the removed code are no longer consumed by the system.

2.1.2 Contributions

The contributions of this paper include: 1) a novel way to capture sacrificability of specifications using annotated tests; 2) a reduction tool, hddRASS, that can build an adapted version of a program based on annotated tests; 3) a case study adapting the NetBeans IDE to use significantly less memory simply by labeling 3 tests corresponding to the *undo-redo* functionality and applying hddRASS and 4) an empirical evaluation of reduction achieved using randomly annotated test suites for real-world open source programs.

[2]Note that we say "specification/functionality" for a reason: while the concepts of specifications and functionalities in a software system are not identical, tests tend to conflate specification and functionality, and our approach in many ways does the same, because its concept of either specification or functionality is purely test-based.

2.2 Related work

The field of self adaptive systems has seen renewed interest in recent years. Salehie et al. [70] summarize some recent work. Different engineering approaches to building adaptive software systems are discussed by Kruptizer et al. [47]; fundamental methods include model-based approaches [44] and architecture-based approaches [62]. Cheng et al. [13] suggest *adaptive requirements engineering* to capture uncertainty in an adaptive software system. They envision a new requirement language that captures what a system *might* do instead of what a system *will* do. Our idea of sacrificability of specifications is a limited but simple version of *adaptive requirements*, where the only possible adaptation of requirements is the possibility of removing certain requirements as represented by labeled tests. Delemos et al. [22] categorize self-adaptive systems as self-managing systems which rely on explicit pre-computed adaptations in contrast to self-organizing systems which rely on implicit runtime adaptations. Fredericks et al. [32] suggest choosing only a subset of test cases to execute based on resource constraints for runtime adaptations.

Delta-debugging [81] is an algorithm for reducing the size of failing test cases or test inputs. *Hierarchical delta debugging* [56] was proposed to efficiently reduce test inputs that are hierarchical in nature. *Cause reduction* extends these ideas to a much more general applicability, including our use of reducing programs with respect to tests they pass [38, 37]. The idea that modifications of a program that are both useful and computationally tractable to identify are likely to be deletions-

only was proposed by Qi et al. [66] in their criticism of much work in automatic program repair. Conceptually, this relates to the statement-deletion mutation operator [25], a special instance of deletion mutation operators [24] that achieves a good balance between the number of mutants generated and the subtlety of faults produced. Our hddRASS algorithm is a kind of combination of the ideas behind HDD and statement-deletion, with heuristic optimization to the case where the program is nearly minimal already, and dependencies tend to flow forward in the source code.

2.3 Core concepts

Building resource adaptive systems [42] requires the availability of adaptations corresponding to different resource availability profiles. The adaptations required may either be pre-computed by developers or may be computed at runtime, based on sacrificability of specifications. We use test annotations to capture the relaxation of a specification, to drive the production of a smaller program that still satisfies part of a system's specification. Because it has less code, the adaptation can be expected to use fewer resources, especially as resources are often associated with functionalities of a system.

2.3.1 Test Annotations

We propose test annotation as one of the ways to capture adaptive requirements [13]. The idea is that a formal specification of different aspects of a specification, and how those aspects relate, is difficult to produce, and seldom available for existing systems. However, most systems, especially critical systems, have test suites, and the tests in such suites are often possible to group by *what they test*. In a complete instantiation of the approach proposed by this paper, annotations would likely be multi-dimensional, specifying priority of tests, aspects of behavior covered by tests (as in the example above), and the resources used by the tests (which relate to the resources used by different functionalities of the system). However, the technique works so long as the labeling of tests allows us to select some tests that are not required to pass. In this paper, we assume a simple one-dimensional labeling scheme, mostly based on priority. Tests labeled 0 are critical tests – if these fail, the system is not useful. Higher label values indicate a higher degree of sacrificability of the specification(s) embodied in the labeled test. We primarily assume labeling is applied to tests, but it could also be applied to system invariants that apply across tests, or to individual assertions or checks inside a single test. Such a system clearly has a finer granularity, but also imposes more burden on a user. Labeling some sets of tests as 1) related and 2) potentially sacrificable in exchange for resource adaptation seems to be a relatively light burden for a user. In the long run, we would like to automatically produce approximate annotations, perhaps based on recently proposed schemes for naming tests [21].

2.3.2 (1-)Minimal Program

Zeller and Hildebrandt. [81] define various notions of minimality for test-case reduction. The most important such notion in delta-debugging is the idea of a *1-minimal* test (for us, program). In normal delta-debugging, a test is 1-minimal when no single component of the test can be removed without the test passing (recall that delta-debugging's goal is to produce small failing tests from large failing tests). A program is 1-minimal if no *single* candidate element of a program can be removed without the program failing some required test. That is, given a program P and a test suite $T' \subset T$ (where T is the full test suite for the program), P is 1-minimal iff $\forall t \in T'.pass(t, P)$ and $\neg\exists P'$ such that $\forall t \in T'.pass(t, P')$ and $P \Rightarrow P'$, where $\Rightarrow$ represents a single step of reduction (in our case, removing certain parts of the syntax tree for P). 1-minimality is obviously a local minimality; there may exist some smaller subset of P that passes T', but would require applying multiple removals at once to produce a consistent program. Searching for such a program is prohibitively expensive, in that it would essentially require enumerating all valid subsets of a program.

Is a 1-minimal program just a program that contains only the code covered by the tests in T'? No. While this may be true for some extremely thorough test suites, it is not only possible but quite common for a test to execute code that it does not actually check for correctness. Schuler and Zeller propose a notion of *checked coverage* [71], which only includes code that is not only executed, but that has data flow to values checked by some assertion in a test. Our notion is similar

in that it goes beyond mere coverage, but even stronger, in that code covered and "checked" may be removed from a 1-minimal program. Consider the following Python program:

Listing 2.1: Program fragment

```
1 def toDiceThrow(val):
2     val = val % 6
3     return val
4
5 def test_toDiceThrow():
6     assert(toDiceThrow(1) == 1)
```

Here, line 2 is considered checked (since the value computed flows to the assertion at line 6), but it can be removed without the test failing, and so would not appear in a 1-minimal program for this test.

2.4 Computing (1-)Minimal Programs

The heart of our approach to resource adaptation is the task of converting a program to a version that is 1-minimal with respect to a subset of its test suite. We use a custom tool, called hddRASS (hierarchical delta debugging + Resource Adaptive Software Systems) for this purpose. From a high-level perspective, our tool is very similar to other HDD and delta-debugging tools. All such algorithms can be described abstractly by a very simple loop. Ignoring the details of the strategy

for constructing candidate test cases, reducing a test case t_b is accomplished by iterating the following two steps until the termination criteria is satisfied:

1. Construct the next candidate reduction of t_b, denoted by t_c (where $|t_c| < |t_b|$ because $t_c \subsetneq t_b$). Terminate if no t_c remain (t_b is 1-minimal).

2. Execute t_c by calling $pred(t_c)$. If *pred* returns `True` then t_c is a reduction of t_b. Set $t_b = t_c$.

The purpose of this loop is to reduce a test case (or program) until it has as few components as possible, while still satisfying some property. We adapt this by reducing a program (or class, or other program element) P_b rather than a test, and by using "passes a set of tests" as our *pred*.

From a high level point of view, this change is all that is required to use delta-debugging/HDD for resource adaptation. However, our purposes are quite different, which motivates certain modifications that are intended to improve performance and effectiveness.

2.4.1 Most Programs are Nearly 1-Minimal: Inverted HDD

Delta-debugging and HDD are aimed at minimizing failing tests. Frequently, a failing test can be reduced in size by one or two orders of magnitude. Most of the test is not relevant to the failure. Unsurprisingly, this is not the case in our context. It would be a very unusual software system in which the vast majority of the code base consisted of optional and sacrificable functionality, and we expect

most adaptations to remove only a small part of the code base. Traditionally HDD begins by attempting to remove large portions of the syntax tree by working from coarsest to finest granularity, applying standard delta-debugging with each component size selected. Obviously, unless the set of tests chosen has relaxed a great deal of the specification, most classes, methods, and other high level parts of a program cannot be removed. We therefore invert the traditional order of HDD and begin by attempting to remove the furthest leaf nodes from the root first, then progressively attempt to remove larger and larger sub-trees rooted at nodes closer to the top of the tree.

Algorithm 1 Hierarchical Delta Debugging

```
1: procedure HDDRASS(inputTree)
2:     level ← HEIGHT(inputTree)
3:     nodes ← TAGNODES(inputTree, level)
4:     while nodes ≠ ∅ do
5:         minConfig ← DDMIN(nodes, inputTree)
6:         PRUNE(inputTree, level, minConfig)
7:         level = level − 1
8:         nodes ← TAGNODES(inputTree, level)
9:     end while
10: end procedure
```

Here DDMIN is the standard delta-debugging algorithm [81], as also used as a subroutine in HDD [56]. We modify DDMIN in one additional way: we order attempts to reduce at a given level by reverse order of program elements. For example, if there are two statements, s_1 and s_2, at the same level of the syntax tree of the program, we try to remove s_2 first, since it is possible that s_2 depends on s_1, but once s_2 is removed, s_1 can be removed. For instance, consider a method

that first opens a file, then writes to it four times, then closes it. Our approach will first remove the close, then the writes, then the open. While the order does not matter in all cases, removing the open last is necessary. Combined with moving upwards from deeper nodes (e.g., removing a use of a variable nested in an `if` before its declaration in an enclosing context), this limits failed attempts to remove code, which are costly when checking the predicate requires running the entire test suite. Since DDMIN starts over after every removal, it is useful to try likely-successful reduction attempts first.

2.4.2 Statement Deletion as Fundamental Operation

Finally, the above algorithm is still not what hddRASS does in practice. The gains in removing code are essentially all obtained by removing statements, because statements (including declarations) are the program elements that consume resources. Therefore, hddRASS considers the syntax tree of a Java program to have leaf nodes that are statements. It does not attempt to remove anything smaller than an entire statement, nor does it attempt to remove classes and methods. If all calls to a method are removed, or all code in the method is removed, the effect on resources desired is already obtained, without the problem of some modifications making the program fail to compile, without a useful effect on resource usage. At heart, our approach can be thought of as combining inverted HDD with the statement deletion mutation operator [25]. Moreover, focusing on statement deletion as our smallest granularity of change means that to reject an adaptation as invalid,

tests that have not been removed only need detect a fairly coarse change to a program, not a subtle modification such as a function parameter change or different logical operator.

Note that in theory, Algorithm 1 does not guarantee 1-minimality with respect to statements in the program, and certainly does not guarantee 1-tree-minimality [56]. This can be (for 1-minimality with respect to statement components) easily fixed by applying a final pass over all statements when the procedure terminates, calling the procedure again if any nodes are removed. In practice, this expensive final step does not seem to actually improve results on real Java programs to which we applied our tool, so we omit it and assume the minimized program is either 1-minimal or very close to 1-minimal. In fact, Misherghi and Su noted in their original paper that 1-minimality per se is not the primary goal, in any case. Their original HDD, unlike standard delta-debugging, does not guarantee 1-minimality, but in practice produced much better reductions in size than standard delta-debugging.

2.5 Experimental Evaluation

2.5.1 NetBeans Case Study

We use the NetBeans IDE [4] (a popular IDE among Java developers) as a subject for our proof-of-concept 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,

Table 2.1: Average increase in memory use (mean M^+) for NetBeans IDE versions

	10 minute run mean M^+ (MB)		
Run	Original	Adapted	% Reduction
`Run 1`	34.34	15.75	54.13
`Run 2`	24.46	17.11	30.04
`Run 3`	24.9	19.18	22.97
`Run 4`	34.45	16.73	51.43
`Run 5`	30.87	19.30	37.47
Mean	29.80	17.61	39.21
	5 minute run mean M^+ (MB)		
Run	Original	Adapted	% Reduction
`Run 1`	22.34	13.39	40.06
`Run 2`	17.56	14.66	16.51
`Run 3`	24.22	19.82	18.16
`Run 4`	19.61	17.27	11.93
`Run 5`	23.13	14.10	39.04
Mean	21.37	15.85	25.14

and performance tests for modules. The IDE uses significant system resources including memory and CPU time. For this study, we focus on reducing the memory requirements.

Undo and *redo* are commonly used features of the NetBeans IDE (and most editors). In order for these to work correctly, any editing step within the IDE has to be saved by the IDE. However, saving every step is costly, and NetBeans IDE limits the undo-redo buffer to just 1000 steps[3].

[3]Bug 50411 [5] of NetBeans IDE bugzilla *(increase undo stack size)*, discusses user requests to increase the undo-redo buffer size. The 3rd comment mentions that NetBeans IDE used to have a much larger limit, but that resulted in unacceptable memory consumption and thus the limit was lowered.

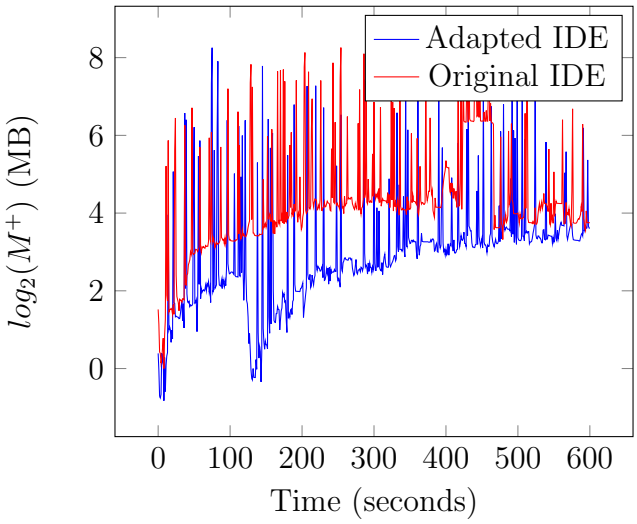

Figure 2.3: Run 1 - 10 min

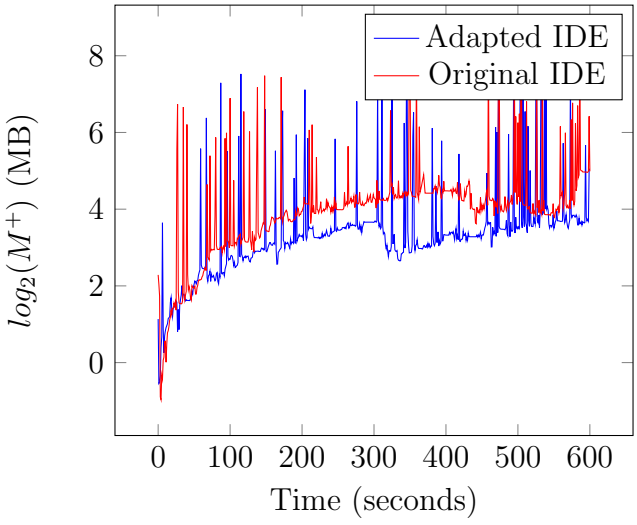

Figure 2.4: Run 2 - 10 min

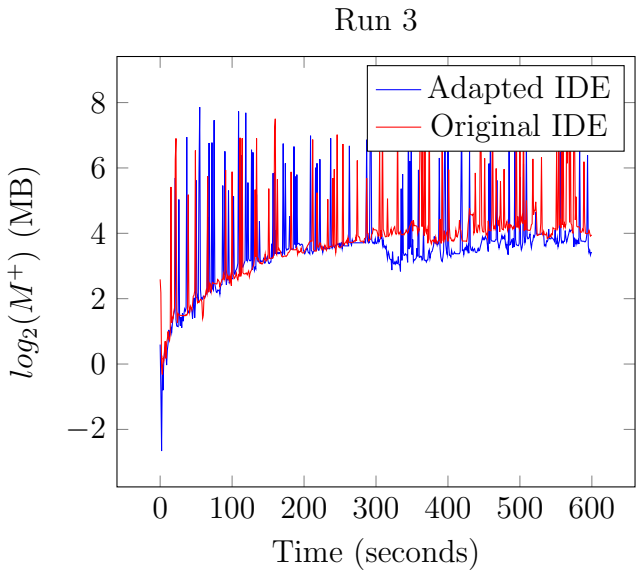

Figure 2.5: Run 3 - 10 min

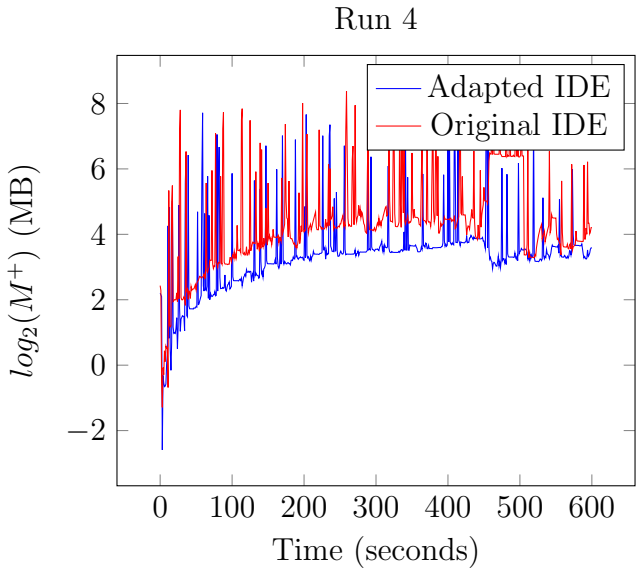

Figure 2.6: Run 4 - 10 min

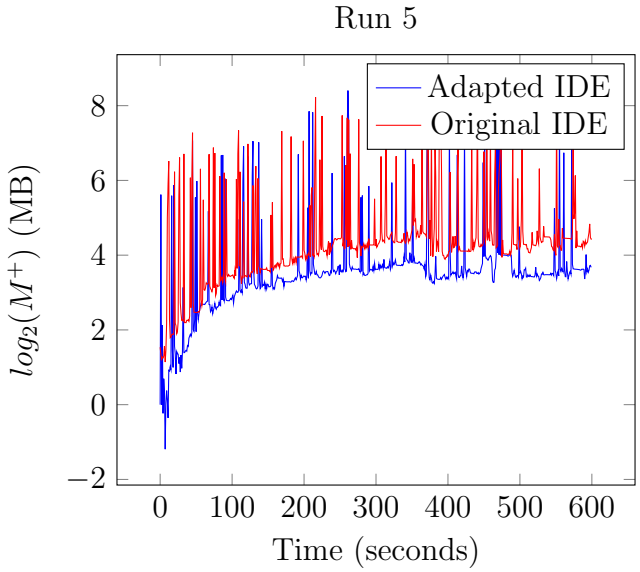

Figure 2.7: Run 5 - 10 min

The *UndoManager* implements the undo/redo functionality and is a part of the *openide.awt* module. This module consists of 11,284 lines of code, with 146 test cases. After studying the module and corresponding tests, we chose three tests and annotated these tests with the label 1 and all others with 0, the designator for essential tests that must pass. We used hddRASS to generate a reduced version of the NetBeans IDE based on the suite without these tests, which took about 4 hours. The tool removed 130 statements, none of which were more than 5 levels above a leaf node. In order to evaluate the effectiveness of resource adaptation via hddRASS, we subjected the IDE to a large number of complex edits.

2.5.1.1 Experimental setup

We ran both the original and the resource-adapted IDE versions for the same sets of edits, holding the runtime and remainder of the environment constant. Each experiment was performed in a VirtualBox-hosted Ubuntu instance with 4GB RAM. To ensure consistent load during the experiment, we ran only the IDE and the data collection tools required in each VM. We used the Linux Desktop Project (ldtp) tool [3] to send the exact same edits with the same delays between edits to both IDE versions. We used a random (but shared between IDEs) mix of small (10%), medium (20%), and large (60%) edits, combined with undo (5%) and redo (5%) keystrokes.

We ran 5 sets of 5 minute and 10 minute runs on both IDE versions (each time using a different seed and thus different shared edits). We measured the memory consumed by the IDE Java objects using the Jmap tool [2]. We called Jmap once per second, collecting 300 data points for the 5 minute runs and 600 data points for the 10 minute runs.

2.5.1.2 Results

We measured average increase in memory utilization (henceforth referred to as mean M^+) over time. The mean M^+ is computed by ploting the data points of memory usage collected during runs and averaging the area under each curve. Table 2.1 shows mean M^+ for each run plus mean values across all runs. Figures 2.3-2.7 show the complete results for the 10 minute runs. The differences in memory

consumption across data points within runs are all highly statistically significant, by Mann-Whitney U test ($p < 1.0 \times 10^{-14}$ in the *least* significant case). This shows two things 1) labeling a small number of tests is a potentially simple and effective way to identify sacrificable functionality and 2) adaptation based on this type of specification of flexibility can produce real improvements in resource utilization. Of course, the reduced memory requirements come with the price of losing undo and redo functionality, but in cases where the choice is between operation with reduced functionality and not operating at all, adaptation is necessary.

2.5.2 Reduction of Java Programs with Randomly Labeled Tests

To further check the robustness of the hddRASS tool and investigate how much reduction can be obtained by removing tests from a suite, we chose 7 Java projects. From each project, we chose 3 classes at random. These classes had an average of 398 LOC. The details are provided in Table 2.2.

For our experiment, we chose to examine only tests that had direct coverage of the class under "adaptation" to avoid extraneous computation. This gave us 30 tests per class on average (*maximum* 58, *mean* 7).

Next, we annotated each test case with a label of 0, 1, or 2, randomly, with probabilities of 80%, 10%, and 10%, respectively. The probabilities reflect what we expect to be the typical case for adaptation: versions of a software system removing more than 20% of all tests (and thus specifications) are not, in most cases, likely to be very useful. Most useful adaptation is probably obtained by

Table 2.2: Subject Class Information: Stmt column counts Java statements inside class methods as defined by java.parser.ast.Statement. Meth 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	Class	LOC	Meth	Tests	Stmt
CruiseControl	AntBuilder	499	33	22	143
CruiseControl	Schedule	383	30	18	127
CruiseControl	Project	685	70	35	291
Ant	Available	289	21	28	133
Ant	Copy	679	48	24	179
Ant	FixCRLF	385	17	34	23
Validator	UrlValidator	218	11	21	82
Validator	RegexValidator	93	4	7	40
Validator	DomainValdiator	1302	15	20	74
Jexl3	Engine	296	30	38	103
Jexl3	JexlArithmetic	781	54	35	289
Jexl3	JexlEvalContext	102	17	37	29
Cli	Option	404	48	9	85
Cli	GnuParser	64	1	58	23
Cli	PosixParser	141	6	58	37
Jena	OntTools	289	29	4	65
Jena	LocationMapper	292	21	10	138
Jena	OntlClassImpl	464	60	27	133
Text	ExtendedMessageFormat	301	17	14	137
Text	LevenshteinDetailedDistance	220	6	12	142
Text	AlphabetConverter	277	13	10	82

removing at most 10-20% of the system specification. We first reduced each class by removing tests labeled 2 from the suite, then further sacrificed the specifications represented by tests labeled 1.

We repeated the process for each of the 21 classes giving us in total 420 "adapted" classes, 210 for label 2 and 210 for labels 1 and 2. We collected the following information from each run: (1) the *reduction size*, – the number of statements removed – and (2) the *reduction height* – the highest distance above a leaf node that was removed. The *reduction size* measures the quantity of changes while the *reduction height* measures the complexity of changes (if it is one, only leaf statements were removed, no `if`s or more complex structures were removed). Note that height here is measured from the bottom, whereas in the algorithm we use level in the opposite sense, where the leaf nodes have higher values, as they are deeper in the tree.

2.5.2.1 Results

Table 3.3 shows that relaxing the specification of systems by removing a small number of randomly selected tests, where the number of tests removed is similar to the number found to be associated with a major feature of the NetBeans IDE, does produce reduction in the code base. The amount of reduction scales with the number of tests removed from the system, and on average was interestingly similar to the portion of tests removed. We suspect these reductions are somewhat less than might be seen in real systems with correctly labeled tests, since there is little

Table 2.3: Reduction size. Tests removed and reduction size are averaged across 10 runs. %Reduction is measured against total statements in the class as defined by Table 2.2

.

	Label 2			Label 1		
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
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
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
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
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	0	0	0	0	0	0
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
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
MEAN		12.53	11.75		16.69	14.74
MEDIAN		7.18	8.75		10.5	12.18

cohesion between the tests selected. It is likely that more than one test forces a program element to exist, in many cases, and only reduction based on removing all tests of that functionality (which should be labeled the same) will allow any aspect to be removed.

Table 3.4 show mean and maximum reduction heights over the Java programs. The reduction is non-trivial (it is above height 1 in most cases, on average, and in only one case was the maximum reduction a leaf node), but is also very seldom higher than a few nodes above the bottom of the syntax tree. This suggests that our inversion of HDD is a good heuristic. In fact, it suggests that the computational effort expended checking the tree far above the leaf nodes is likely to be wasted. However, we do suspect that more coherently labeled tests would produce higher

Table 2.4: Reduction height

	Label 2		Label 1	
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
Available	0.8	2	2.6	3
Copy	2.5	8	3.6	8
FixCRLF	1.6	2	2	3
UrlValidator	1.6	3	2.2	3
RegexValidator	1.6	3	1.9	3
DomainValidator	1.3	3	1.85	3
Engine	3	3	3	3
JexlArithmetic	0.4	2	0.6	2
JexlEvalContext	1	1	1	1
Option	2.3	3	2.6	3
GnuParser	1.4	6	1.8	6
PosixParser	0	0	0	0
OntTools	2	2	2	2
LocationMapper	1.8	2	2	4
OntlClassImpl	2	3	2.5	3
ExtendedMessageFormat	2	2	2.3	3
LevenshteinDetailedDistance	2	2	2	2
AlphabetConverter	2	2	2	2
MEAN	1.55	1.85	2.76	3.14
MEDIAN	1.6	2	2	3

reductions, and the effort spent on nodes far up the tree is usually relatively small, since there are many fewer program elements to use as candidates for reduction at those levels.

2.6 Threats to validity

The key threat to validity is that our results concern one actual use of our approach on a single larger application, and a number of reductions (without any *meaningful* adaptation with respect to valid labels) for a small set of additional Java programs. Moreover, due to the computational resources required, we opted to reduce with respect to only a small subset of classes from the programs we selected. In short, our results should be interpreted as a proof-of-concept for a proposed method, not as complete experimental evaluation of our approach to adaptation.

2.7 Discussion

There are numerous engineering decisions to be made about how to use the basic idea proposed in this paper, and many open related research questions. Resource adaptation via test-based software minimization, where removing tests (or possibly invariants) from a test suite is used as a simple way to capture sacrificability of specifications, can be incorporated into many adaptation workflows. Adaptations and resource gains can be pre-computed during offline efforts before deployment, with the adaptations simply enabled, rather than discovered, in the field. A good

method for predicting the resource impact of each adaptation is important: in some cases, dynamic analysis of test behavior can give good hints, but in many cases tests do not resemble actual use in terms of resource profiles. Is it possible to safely compose adaptations? That is, if we compute an adaptation based on removing one feature (represented by one set of tests) and then compute another, based on a different set of tests, will the system produced by applying both of these tend to 1) work correctly, except for the sacrificed aspects of the specification and 2) obtain resource gains similar to the sum of those obtained by each separate adaptation. If composition works, then the effort to produce useful adaptations may be much less than might be expected, based on the number of test labels.

However, in a sense all of these questions are ignoring what would appear to be the elephant in the room: our method assumes that software systems have highly effective, comprehensive, and easy-to-label test suites. There are three responses to this objection. First, for many critical systems, we suggest that if there does not exist such a test suite, one should be created as soon as possible. For instance, in the Tactical Situational Awareness System effort that brought our attention to the problem of self-adaptive software, it would be negligent to not produce an effective test suite for all system functionalities and important specifications. Second, we believe that even if test suites are good but imperfect, we can distinguish the signal of reduction based on removing some tests from the noise of general test suite inadequacy. In particular, we propose to establish a baseline for each software system based on reducing it *without removing any tests at all*. For very good test suites, this should not produce any reduction at all, or reveal opportunities to

simplify or optimize the original system. For systems with useful but incomplete test suites, the baseline can be locked in place: when reducing by removing tests, any parts of the system that are removed, even if those tests are present, is *not* removed. It is not related to sacrificing the specification represented by the set of tests, but an artifact of the inadequacy of the test suite in general.

Finally, for pre-computed adaptations it should be possible to run a *shadow* version of a system in field use, and compare behavior with the real system over the same inputs. If the adaptation diverges in a way deemed unacceptable, the adaptation can be removed from the database of adaptations, and the behavioral difference stored as a basis for a future test to capture the missing specification. Such *shadow* execution of adapted versions of a system in conjunction with a standard version can also serve to capture detailed and accurate resource profiles for adaptations in actual use.

In the long run, we believe that producing complex, highly adaptive, reliable software systems requires producing extremely good tests. Moreover, recent work on automated test generation and advanced static and dynamic analysis brings us closer to this goal. We therefore also propose that this paper presents a first step towards thinking about what benefits, besides improved system reliability, can be obtained in a world where important software systems do, for the most part, have extremely good test suites.

2.7.1 Heuristics for Faster Minimization

HDD and DDMIN are potentially expensive algorithms. In practice, if reduction is mostly applied at the class level, our current implementation may be fast enough for offline pre-computed adaptations, especially if such adaptations compose. However, it is far too slow for practical online use, and not convenient for developers wanting to experiment with the method. Moreover, for systems with very slow tests (such as our own TSA), reduction is costly. A few heuristics to improve the process are low-hanging fruit. First, if a statement is not covered by a test suite, it is obvious that it can be removed. Second, we can use checked coverage [71] to further remove statements: if a statement's computation does not flow to any assertion in a test it is likely going to be removed (one exception is that checked coverage cannot detect statements that cause a crash rather than assertion violation). Our tool also lets developers, or perhaps automated methods [32] select and prioritize tests to use in reduction. One obvious approach is to prioritize tests that tend to reject reductions.

2.8 Conclusions and Future Work

Building robust resource-adaptive systems is critical if we are to produce 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 a grave need for methods for allowing software to adapt without human inter-

vention. In this paper, we show that by removing labeled tests from a software system's test suite, we can represent the sacrificability of specifications in a simple and relatively low-burden way. This representation also allows us to automatically construct adapted versions of a system that, on the one hand, sacrifice some functionality, but trade this for advantages in resource usage. We demonstrate our approach by marking three tests of the NetBeans IDE as sacrificable, and obtaining a version of the IDE that lacks undo/redo functionality but also uses significantly less memory during operation. Examination of a set of Java classes shows that program size reduction obtained over randomly labeled tests seems to usually be proportional to the fraction of tests removed, and most reduction is performed near the bottom of the syntax tree.

As future work, we plan to extend and improve hddRASS, especially in terms of improving its efficiency, and to integrate our approach into a realistic, complex, self-adaptive system such as the Tactical Situational Awareness System (TSA), in the context of a complete framework for resource adaptation. A key element will be effective prediction of resource profiles, perhaps through shadow execution.

Acknowledgements: this work was partly funded by the DARPA BRASS [42] program, and the authors would like to thank our collaborators at OSU and Raytheon/BBN.

Manuscript 2

Target Selection for Test-Based Resource Adaptation

Arpit Christi, Alex Groce

2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), Lisbon, Portugal.

pp. 458-469

Chapter 3: Target Selection for Test-Based Resource Adaptation

3.1 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 han-

dle 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 [42]. Different engineering approaches to build adaptive software systems are summarized by Kruptizer et al. [47]. 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 [18] 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 adap-

[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.

tation. 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 unnecessary 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.

3.2 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 [13]. Adaptations manifest in 3 different ways: (1) restricted functionality, (2) altered functionality, or (3) enhanced functionality [42]. 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 [53, 29] 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 2.2 [18].

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 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 [42], 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) [56], optimized for the RASS workflow [18]. 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 [18]. Like HDD, hddRASS is still a greedy search-based algorithm, with similar worst-case time

complexity $O(n^3)$ [18][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.

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

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

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.

3.2.1 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.

3.3 Experiments

3.3.1 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 [35, 66]; 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 [80, 54], 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 3.1. 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

Table 3.1: 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 3.2: 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

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 3.1). For 32 out of 40 subjects, statement coverage of the tests is more than 80%, and it is more than 70% for 37 subjects, as shown in Table 3.2. 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%).

3.3.2 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.

3.3.3 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 [1] 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.

3.4 Empirical Results

3.4.1 Reduction Size

Table 3.3 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.

Table 3.3: 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 3.1

.

	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

Table 3.4: 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

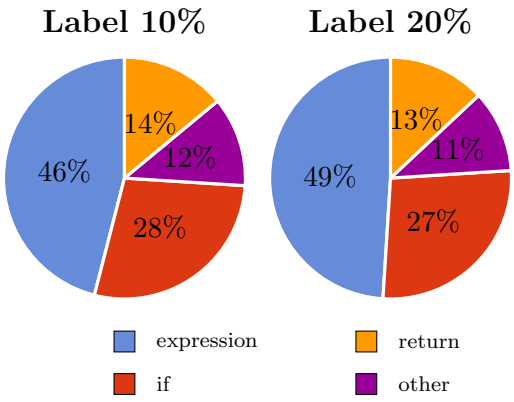

Figure 3.1: Reduction type: what kind of statements are removed?

3.4.2 Reduction Height

Table 3.4 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.

3.4.3 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 categorized removals into the four categories defined above (`If`, `Return`, `Expression`, and

`Other`). Figure 3.1 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 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 3.2).

3.4.4 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

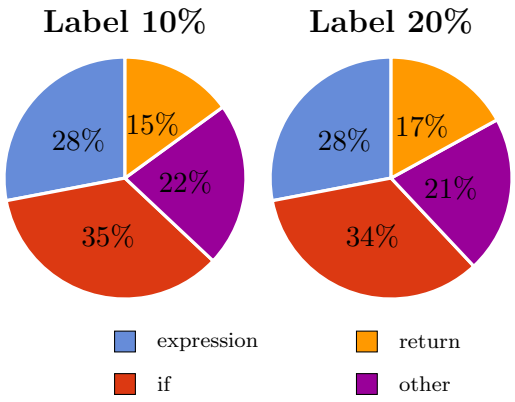

Figure 3.2: Reduction probability: what is the breakdown of removal types, adjusting for statement type distribution differences?

statements) should be covered by some test. In reality, of course, a program may have a poor, highly inadequate test 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

Table 3.5: 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

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 3.5 and Table 3.6 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 3.6: 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

3.5 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.

3.5.1 Heuristic H1: Only Attempt to Remove Simple Statements

The results in Sections 3.4.1 and 3.4.2 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).

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

Section 3.4.3 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.

3.5.3 Heuristic H3: Only Attempt to Remove Statements Contained in *CBLS*

Section 3.4.4 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.

3.5.4 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.

3.5.4.1 Accuracy

Accuracy measures the similarity of heuristic-based results to baseline results. We measure accuracy as follows: $BR = \{\, x \mid x \textit{ is a statement removed by the original approach}\}$ $HR = \{\, x \mid x \textit{ is a statement removed by the heuristic-based approach}\}$. As the heuristics all involve sub-setting 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%.

Table 3.7: 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

3.5.4.2 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: $BS = \{ x \mid x \textit{ is a statement the baseline approach attempts to remove}\}$ $HS = \{ x \mid x \textit{ is a statement the heuristic-based approach attempts to remove}\}$. 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].

[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 3.8: 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

Table 3.7 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 3.8), 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 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.

3.6 Case Study

We use the NetBeans IDE [4] (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 [18].

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!

In order to confirm that our heuristics are accurate and provide efficiency gains

Table 3.9: 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

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 3.9 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 3.10 (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.*

Table 3.10: 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

3.7 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* [56], 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.

3.8 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 [18], 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. [70] summarize much of the earlier recent work in the field. Different engineering approaches to building adaptive software systems are categorized by Kruptizer et al. [47]. Cheng et al. [13] 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. [75] 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. [22] 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. [32] 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 [81] is an algorithm for reducing the size of failing test cases or test inputs. *Hierarchical delta debugging* (HDD) [56] 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 [38, 37].

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. [66] 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 [25], 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 [36].

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 [50]; 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 [25], a special instance of deletion mutation operators [24] 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 [18] 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 [40] to see if they promise further improvements in the runtime of hddRASS.

3.9 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 [18] 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 [71]). 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 [42] program, and the authors would like to thank our collaborators at Oregon State University and Raytheon/BBN.

Manuscript 3

Evaluating Fault Localization for Resource Adaptation via Test-based Software Modification

Arpit Christi, Alex Groce, Rahul Gopinath

2019 IEEE International Conference on Software Quality, Reliability and Security (QRS), Sofia, Bulgaria.

pp. 26-33

Chapter 4: Evaluating Fault Localization for Resource Adaptation via Test-based Software Modification

4.1 Introduction

Modern day software systems have varying, complex resource needs that change frequently based on environmental variations, technology, and integrating system components. To ensure survivability in changing resource contexts, these systems must self-adapt based on changing resource needs and availability [41]. The inability of software systems to handle varying resource needs can lead to inferior and potentially vulnerable software. A *self-adaptive software* (SAS) changes its behavior in response to changes in operating conditions. A *resource-adaptive software* (RAS) is a SAS where the reason for adaptation is unavailability or variability of one or more resources. Researchers are devoting significant efforts to devise methods to effectively construct resource-adaptive software systems, and DARPA has launched BRASS (Building Resource Adaptive Software Systems), a major initiative devoted to the problem [42].

Researchers have proposed numerous tools, techniques, and approaches to build SAS that rely on modeling techniques, architectural specifications, domain-specific languages, and formal methods [70, 47]. While numerous tools exist, these tools are mostly domain or application specific. Further, they are seldom reusable nor

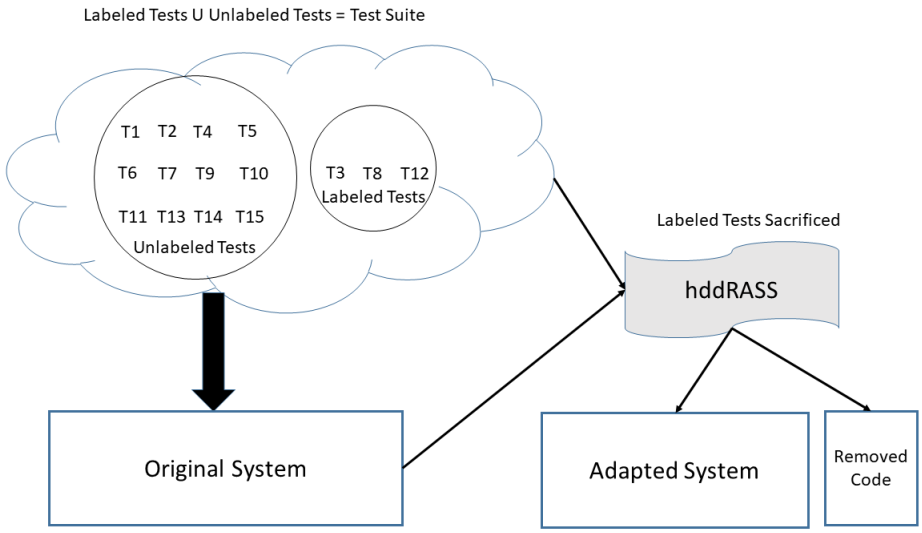

Figure 4.1: TBSM approach to build adaptation for single adaptation objective scenario.

sufficiently applicable to different systems [15, 7]. These tools usually require developers to learn modeling techniques, formal specification methods, domain-specific languages, etc., and map their applications accordingly [52, 26, 33], which presents a high barrier to entry.

While working with software developers building real-world RAS for a mission-critical software system, the Tactical Situational Awareness System (TSAS), Christi et al. proposed *a conceptually simple but highly applicable* technique called *Test-Based Software Modification* (TBSM) [18], inspired by delta-debugging variations [37, 38]. TBSM is similar to Automatic Program Repair (APR) for patch generation and relies on existing test infrastructure, combined with automatic program modifications, to build adaptations [35]. The technique relies on developers' under-

standing of tests and how tests relate to features and resource usage. Developers encode this information by merely *labeling the tests* related to functionalities. As developers do not need to learn any modeling or architectural technique, any specification languages, formal methods etc. *the entry barrier is low to developers.*

To simplify presentation and analysis, we assume in the remainder of this paper that a single resource change requires adaptation. Figure 4.1 is a simplified version of a figure in the original TBSM paper [18], explaining the basic idea of the approach. Tests in the labeled set mark tests that encode the adaptation objective: they test a functionality that is to be removed. The unlabeled tests define functionality the adapted system must retain. The reduction tool hddRASS takes as its input the original program and labeled and unlabeled test suites, and produces a "minimized" program that passes all the unlabeled tests, but does not have to pass the labeled tests. Ideally, this is a program with the targeted functionality removed. The operation of hddRASS is conceptually simple: it repeatedly removes parts of the code, and checks if the modified program still passes all unlabeled tests. If so, this becomes the new baseline program, and hddRASS attempts to minimize it, until no changes produce a program that still passes all required tests. If the labeled tests define a functionality that consumes resources, the modified program will consume fewer resources.

The major drawback of TBSM is efficiency — it is a slow technique. Since building resource adaptations with TBSM is similar to patch generation with APR, it has similar reasons for inefficiency — (1) an extremely large search space, (2) potentially long running time for test suites, and (3) inefficiency due to the un-

derlying algorithm, a modified form of Hierarchical Delta Debugging (HDD) [56], which has worst-case complexity of $O(n^3)$. Of these problems, the sheer size of the search space is perhaps the worst (and drives HDD complexity): the search space in TBSM is the set of possible modifications to a whole program, and since it is a generate-and-validate technique, in principle TBSM may have to run the entire unlabeled test suite for every step of the search. We proposed three heuristics [17] to reduce the search space, of which one, called H3 or CBLS (Coverage-Based Likely Statements) relies on statement coverage information of labeled and unlabeled tests. This heuristic performed best in terms of both accuracy and reduction in search space size, often by a large margin. We therefore use CBLS as the starting point for our exploration of improving the efficiency of TBSM. Even with CBLS, computing an adaptation still required over an hour in a realistic case study.

While proposing Fault Localization (FL) as an early step of APR to tackle the search space problem, Le Goues et al. noted it as a significant repurposing of FL — the original purpose mainly being automated debugging [50]. The presence of the fault and the availability of passing and failing tests makes FL a natural choice for APR. Unfortunately, with TBSM, neither is present: TBSM aims to generate a minimal program to fit a test suite, not cause failing tests in a suite to pass. In this paper, we argue that the presence of labeled tests can be used to overcome this limitation. We propose changes in the definition of Fault Localization, primarily in the usage of passing and failing tests, to repurpose it for TBSM. We call this tweaked FL Adaptation FL, or AdFL. We demonstrate that AdFL can successfully predict which statements will be modified (removed) during adaptation more

effectively than CBLS. Moreover, we show how to modify the TBSM process to incorporate the information provided by AdFL to produce accurate adaptations much more quickly. The contributions of this paper are: (1) We repurpose fault localization to tackle the search space problem for resource adaptations via Test-based Software Modification. This is accomplished by modifying the definition of FL. (2) We evaluate five different FL techniques repurposed for AdFL. (3) We conduct an empirical analysis to study the usefulness of AdFL on 800 data points across 40 subjects of 10 open source Java projects, using two test labeling schemes. (4) Using two real-world adaptation scenarios, we demonstrate that AdFL can predict modifications for real systems. (5) We consider the *stopping rule* problem for TBSM, and propose some heuristics, as well as an incremental version of TBSM.

4.2 Related Work

4.2.1 Self Adaptive Software Systems

Self-adaptive systems are well described in two roadmaps (Cheng et al. [13] and Delemos et al. [22]). Salehie and Tahvildari provide a summary of much early work in SAS [70], and describe some of the notable techniques proposed to build SAS. Krupitzer et al. summarize different engineering approaches to build SAS [47]. The approaches presented include model-based approaches, architectural-based approaches, control theory-based approaches, and learning-based approaches. The above-mentioned techniques, tools, and approaches target specific scenarios or ap-

plications with limited applicability and little reusability. This was noted by Garlan et al. while developing the RAINBOW framework [33], an early attempt to solve the reusability issue, and relies on architectural modeling combined with control and utility theory.

Elkhodary et al. argue that using feature or functionality as the core building block in SAS construction can alleviate some of the key challenges by abstracting underlying complexities [28], and develop a feature-oriented SAS framework [27]. Fredericks et al. suggest elevating tests to the status of first class citizens in SAS specification and verification. They propose MAPE-T, a test aware feedback loop where T stands for tests [32]. Self-adaptive software requirement specification is an open research question. Formal specifications, modeling languages, and domain-specific languages have been proposed [53, 29, 75]. Whittle et al. developed a DSL called RELAX that provides expressions to capture uncertainty in requirements [75]. Our work, in contrast, focuses on improving the performance of a recently proposed technique, TBSM, that avoids the need for additional specification, modeling, or architectural information.

Casanova et al. discuss the issues and limitations of utilizing FL to diagnose unobserved components in self-adaptive systems where the information collected for diagnostics may be insufficient or incomplete [12].

4.2.2 Fault Localization and Program Repair

Fault Localization ranks statements by the likelihood of the statement being faulty. Jones et al. used *spectra* of passing and failing tests to define the Tarantula technique [43]. Following that seminal work, researchers have proposed many (Spectrum-Based) FL techniques. Wong et al. summarize in detail recent advances in FL: according to Wong et al., Tarantula, Ochiai, Barinel, Op2, and DStar are the most studied FL techniques [76].

The recent surge of interest in Automated Program Repair (APR) largely began with GenProg, which modifies the Abstract Syntax Tree (AST) of a program until all tests in a suite pass and the fault is (presumably) fixed [35]. Le Goues et al. subsequently identified multiple major issues with APR, with a key issue being the search space problem [50], and argued that it is sufficient to modify only *likely faulty statements*. They mentioned FL as an imperfect, but best-available technique for the purpose. Further research in APR frequently uses FL as the first step of APR in order to tackle the search space problem [49, 73, 77, 60]. While evaluating different size search spaces, Long and Rinard noted that a small fault space might result in missed faulty statements while a larger fault space results in overfitting, making selecting the ideal fault space a difficult problem [51]. In their evaluation of FL techniques using APR, Qi et al. raised concerns about the usefulness of FL as an early step of APR [65]. Rather than using FL to improve APR, we modify FL and re-purpose it to the needs of TBSM.

4.3 Adaptation FL

Fault Localization (FL) is a natural fit for Automatic Program Repair (APR) because both methods assume (1) there is a fault, identified by at least one failing test and (2) there are both passing and failing tests to focus attention on the faulty aspects of a system. Both are missing when building adaptations using TBSM: the original program is assumed to pass all labeled and unlabeled tests, and a proposed adaptation step may not pass any tests. In this section, we present core concepts behind TBSM, the process of building an adaptation, and the use of labeled and unlabeled tests in the process. In doing so, we will construct a mapping between the core components of FL (*fault*, *passing tests*, *failing tests*) and core components of TBSM (*adaptation*, *labeled tests*, and *unlabeled tests*).

We define *minimization* as the process of applying hddRASS/TBSM to reduce a program's size. The modified program produced by applying TBSM is called the *adaptation*. The set of program statements modified (usually removed) by applying TBSM is called the *modification*; the modification is the diff between the original program and the adaptation. Tests that are marked as pertaining to a feature to be removed from the program are called *labeled tests*. Labeled tests are also known as removed tests, as the TBSM process involves removing them from the test suite so the program is no longer required to pass those tests. Tests that are not labeled tests are called *unlabeled* or *retained* tests. Adaptation via TBSM guarantees that the adaptation passes all unlabeled tests.

At a high level of abstraction both APR and TBSM modify a program until a

certain set of tests passes. APR starts with a program that does not pass all tests, and aims to modify it so that it passes all tests; TBSM starts with a program that does pass all tests, and aims to modify it until it can no longer be modified and still pass all unlabeled tests. There are other differences — e.g. in APR we expect repairs to be small, involving changes to only a few lines of code, while in TBSM changes may be very large, depending on the functionality in question — but this is the essential difference. Our problem, then, is to map FL for APR to the different setting of TBSM.

4.3.1 Mapping FL to TBSM

4.3.1.1 Faults and Modifications

The underlying idea of spectrum-based FL is to identify code that is likely to be faulty; however, another way to think about this idea is that FL aims to identify code that is *part of the fix to a fault* — almost by definition, faulty code is code that needs to be changed.

The "faulty code" in TBSM is code that needs to be modified or removed for an adaptation: the difference between the original program and the adaptation. Such code is even, in fact, "faulty" when seen from a larger perspective. The purpose of resource adaptations is to avoid faults (either in correctness or performance) that occur in low-resource settings. We may not have any tests that represent such a scenario, but nonetheless we can easily conceive that code that should

be modified in TBSM is the code that is involved in some resource-based fault scenario. E.g., if a program crashes because it allocates too much memory in a low-memory environment, we can say that the code that is 1) not essential to the functioning of the system and 2) allocates memory is *all* faulty. TBSM (and other resource adaptation) can be seen as APR for such faults. However, there remains a very significant difference between the techniques: in APR there is at least one test case that fails due to the fault being repaired. TBSM instead begins with a functionality to remove, since adaptation is often a pre-emptive effort rather than a response to a particular concrete failure. APR does not address purely "hypothetical" bugs, obviously, while minimization, in contrast, operates without access to a concrete failure scenario.

4.3.1.2 Failing Tests and Labeled Tests

FL for APR relies on the existence of at least one failing test, and identifies code more associated with failing than passing tests. In TBSM, we do not have failing tests. However, we do have labeled tests, and these can serve the same purpose. Recall that our failures, while unavailable, are due to resource uses of the functionality to be removed. This means that code more associated with the removed functionality is the code of potential interest for our repair. Not all code in labeled tests is resource-using, or related to removed functionality, but this is exactly the situation in FL in general: most code executed by failing tests is not faulty; the problem is to identify code that is somehow (causally, statistically, etc.) more

related to failing than passing tests, and thus highly suspicious. Given this understanding, it is clear that *the "failing tests" in FL for TBSM are the labeled tests*, the tests that will be *allowed* to fail in the adaptation. Note that we are *using tests that are not executed at all in the minimization process* to approximate the *goal of that process*.

4.3.1.3 Passing Tests and Unlabeled Tests

Given the above mappings, it should be clear that *unlabeled tests, the tests the adaptation must still pass, correspond to the passing tests in FL for APR.* These tests serve as the "background" for the "figure" of the labeled tests.

4.3.1.4 Putting it all Together: FL for TBSM

To perform FL for TBSM, Adaptation FL (AdFL), then, we apply any fault localization algorithm, but replace failing tests with the labeled tests and passing tests with unlabeled tests.

To make the mapping here concrete, consider one of our case studies in 4.4.2, adapting the NetBeans IDE by removing undo/redo functionality. The labeled tests for the functionality do not fail, but they all (1) test an undo and/or redo action and (2) allocate memory for an undo buffer. The unlabeled tests never test undo/redo actions, but do allocate memory for the undo buffer in many cases. The strong association of the "fault" (undoing/redoing) with the labeled tests should

still make effective FL algorithms label code that implements undo and redo as suspicious.

4.3.1.5 The Need for a Stopping Rule

CBLS has one obvious advantage over FL. CBLS provides a set of statements, not a ranking. In CBLS it is clear how the search space is reduced: only statements in the CBLS set are considered for modification. AdFL provides a *ranking* of statements. Having a ranking is clearly useful in TBSM, providing an ordering attempting to modify/remove statements, and making the program smaller much more quickly, but it does not, on its own, reduce the search space . To reduce the search space, we also need a *stopping criteria*. Constructing such a criteria, however, is not a simple matter, and previous work in FL for APR does not, in a sense, *need* such a criteria: when the modified program passes all tests, APR can stop. The only use for a stopping criteria is to abandon the search and consider the program unrepairable. In TBSM, however, we do not have such a convenient way to detect when we are done.

As it turns out, one (weak) stopping criteria requires no empirical data to justify. Statements not in CBLS are, by definition, never covered by any labeled test. This has two consequences. First, for any widely used SBFL approach we are aware of, such statements will have a suspicousness score of 0.0, and thus rank at the bottom of any ranking of statements in AdFL. Second, intuitively, it seems absurd to consider a statement not covered by any test "defining the functionality

to be sacrificed" as a candidate for modification. Such statements are clearly, if our tests are any good, not part of the functionality we are adapting. Therefore, as a simple, default, stopping criteria for using AdFL in TBSM, we ignore all statements not in CBLS, as these statements will always have a suspicousness of 0.0. In fact, when statements are both covered by unlabeled tests (so presumably part of non-sacrificed functionality) and not covered by labeled tests, it seems *dangerous* to remove them. We therefore call AdFL using CBLS to limit the space *Guarded* AdFL (G-AdFL) because it guards against this possibility. This stopping criteria also points out an interesting corner case. CBLS includes "dead code" statements not executed by any labeled or unlabeled test. For many FL formulas (e.g., Tarantula) this results in an undefined suspiciousness. We assign such code a suspciousness of $\top$, since we know hddRASS without heuristic guidance will always remove such statements.

4.4 Experiments

Having defined a procedure for Adaptation FL, we need to determine if the proposed equivalences are useful for building adaptations using TBSM. We demonstrate the utility of AdFL using two real-world single adaptation objective scenarios, building class-level adaptations in a controlled setting. We also show that AdFL performs well for a large set of synthetic adaptation problems using open source subjects used in previous evaluations of TBSM improvements [17].

In what follows, the *baseline* is the result of the application of TBSM in its

original form without any heuristic or AdFL guidance. *CBLS* refers to application of TBSM where the search space is first reduced using the H3/CBLS heuristic [17], and *AdFL* to application of TBSM where the search space is first reduced using our new AdFL technique. We use the five most commonly studied SBFL techniques as our FL algorithms for AdFL [76]. Hence, for each AdFL, we produce five separate results.

4.4.1 Case Study: TSAS Elevation-API

For TSAS, we present a scenario that the development team calls the *Elevation-API*. Here, the *variable resource* is one of the libraries and the *variation* is the availability of a newer library. Via test labeling, the developer indicated sacrificablefunctionality. With the availability of a newer library, certain features implemented in TSAS are implemented by the library, making the TSAS implementation code redundant. The test labeling here was in terms of features only, not resources. The sandboxed TSAS server component that we used consists of 70 Java files and is 5571 LOC in size. The developer labeled 5 tests, and the remaining tests were considered unlabeled. We applied the *baseline*, *CBLS*, and *AdFL* techniques to produce adaptations. Developers confirmed that all four necessary modifications were performed correctly by all the techniques, and that the adapted TSAS versions all worked correctly.

4.4.2 Case Study: NetBeans IDE undo-redo

The original work on TBSM discussed a NetBeans IDE undo-redo adaptation scenario in detail where memory consuming functionality was correctly modified to preserve memory [18]. The module under consideration, the `openide.awt` module, consists of 69 Java files with 11,284 LOC and 146 tests. We continue to use the three labeled tests that were labeled as *undo-redo feature related* in the original work. We applied the baseline, CBLS, and AdFL TBSM techniques to build a memory-adaptive NetBeans IDE. We confirmed, again, that all the techniques correctly removed all 19 resource consuming statements (as identified in previous research [17]). By building the adapted NetBeans IDEs and using them for a while, we were also able to confirm the normal operation of NetBeans IDE with undo-redo ability removed.

4.4.3 Synthetic Adaptation Analysis

We also compared AdFL to other methods using synthetic scenarios used in previous TBSM research. This involves 800 adaptations of 40 subjects from 10 open source Java projects, using two random labeling schemes. Our intent with the synthetic problems is not to produce practically useful adaptations, but to produce accurate adaptations for a given test suite and labeling. We control all variables such as test suite, labeling scheme, and class subjects. The only variation allowed is the choice of search space selection technique.

4.4.3.1 Subjects and Tests

Previous work on TBSM noted that at the class level, adaptations are meaningful and useful [18]. Hence, we focused our study on class-level adaptations. Subject details are available in our previous paper on TBSM heuristics [17]. Subjects have an average of 387 LOC (for the whole class) and 132 statements in non-constructor methods.

Previous work noted that arbitrary subsets of tests are not interesting (reduction will usually target some actual functionality), and some tests are not relevant to a particular class. Based on this, previous work used direct coverage as a criterion for test selection, and we adopted the same approach. However, in our case, test selection is not relevant as long as tests and labels remain the same across techniques. We have 24 tests per subject class, on average. For 32 out of 40 subjects, statement coverage of the tests is more than 80%, and it is more than 70% for 37 subjects. We can, therefore, say that we generally have subjects with good coverage. This is an important experimental criteria, since TBSM would only be applied to systems with good test suites.

4.4.3.2 Procedure

For each class, we first randomly labeled 10% of the tests and computed the adaptation. By keeping the test suite and labeling same but using the search space defined by the CBLS and AdFL techniques we repeated the process (details in Section 4.4.3.3). We repeated this procedure 10 times for each class, randomly

labeling tests each time, yielding 10 results per class, generating 400 results for 40 subjects. During each run, if the labeled tests overlapped with the unlabeled tests, or concerned only minor functionality, modifications are minimal, but if a very important test is labeled the modifications may 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 in our open source projects' 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 labeled tests representing a feature. Hence, we repeated the same procedure, but with 20% of the tests labeled, to produce another 400 adaptations. For each of the 800 data points, we have 3 results: baseline, CBLS, and AdFL. Each AdFL result can be broken down into 5 separate results for the different SBFL techniques.

4.4.3.3 Measurement

TBSM performs adaptations by modifying (in the current implementation of hddRASS, removing) program statements. The set of program statements that are considered by TBSM is the search space. For AdFL without a stopping criteria, the real search space is the same as for baseline, except with the statements prioritized so that more likely-removed statements are considered first. However, we can use the ranking of the worst-ranked modified statement to see *what stopping criteria could have been used* while preserving a "perfect" adaptation. It is impor-

tant to note in what follows that we are comparing real search spaces for baseline and CBLS with a theoretically ideal search space for an AdFL with a perfect cut-off/stopping rule. This ideal search space will inform our effort to devise concrete methods for using AdFL to improve the efficiency of TBSM.

For *baseline*, the search space is the whole program. For *CBLS*, the search space is statements in the CBLS set. For AdFL's theoretical ideal search space, we use the *EXAM SCORE*, the most common measure used to compare FL techniques [63]. We apply AdFL to all the program statements and sort them by ranking. Then we look for the modified statement with the worst ranking. If the modified statement with the worst ranking is tied with other statements, we resolve the tie randomly. For AdFL, the "search space" is then all the statements up to the last statement that was modified. We have five distinct AdFL results based on the SBFL techniques: Tarantula, Ochiai, Barinel, Op2, and Dstar; the most widely studied SBFL formula [76]. The formula can be found in Pearson et al. [63]. For the remainder of the paper, % improvement of technique B over technique A is measured as $\%\text{improvement} = ((|B| - |A|)/|A|) * 100$. With a fixed cutoff rule (CBLS or a percent of highest ranked statements for AdFL) we measure % accuracy as % of statements modified out of statements modified by the baseline technique that searches the entire space of modifications.

Table 4.1: MEAN/MEDIAN are for EXAM SCORE. FLT=Fault Localization Technique. #Worse presents the number of techniques worse by tournament ranking. FLT Rank is mean across 800 data points.

FLT	MEAN	MEDIAN	#Worse	FLT Rank
Tarantula	0.490	0.521	0	1.966
Ochiai	0.511	0.540	0	2.095
op2	0.509	0.534	0	2.033
Barinel	0.505	0.541	0	2.053
DStar	0.513	0.537	0	2.13

4.5 Evaluation

We focused on three key research questions: **RQ1:** Is there a best formula for AdFL, among the five SBFL techniques evaluated? **RQ2:** How does AdFL compare to the baseline and CBLS for synthetic adaptations? **RQ3:** Most importantly, is AdFL an effective technique for real-world adaptations?

4.5.1 RQ1: Comparison of SBFL techniques for AdFL

To compare each of the five techniques for AdFL, we used the same comparison methodology used by Pearson et al. for evaluating and improving fault localization techniques (FLTs) [63], with three evaluation metrics: (1) mean EXAM SCORE, (2) tournament ranking: pairwise comparison to determine if one FLT is statistically superior to another, and (3) mean FLT rank: using each data point to rank SBFL techniques from 1 to 5 and then averaging the rank across 800 data points

For tournament ranking, we used a *paired t-test*. Table 4.1 shows results. The

Table 4.2: MEAN/MEDIAN represents mean/median % improvement while choosing AdFL vs. baseline. AvgAdFL and WorstAdFL have 100% accuracy. G-AvgAdFL and G-WorstAdFL have mean accuracy of 79% (identical to CBLS).

Method	MEAN	MEDIAN	p-value	std dev	MIN	MAX
AvgAdFL	47.06	43.71	<0.005	29.31	0.34	97.01
WorstAdFL	41.40	37.70	<0.005	31.81	0.22	96.91
AvgG-AdFL	60.88	65.32	<0.005	23.85	4.93	97.28
WorstG-AdFL	42.71	41.30	<0.005	29.28	0.00	97.14

#Worse columns shows that, for all 10 pairings, no technique was statistically significantly better than the other. MEAN, MEDIAN and FLT Rank values are also very close for all techniques. Hence, we can say that no clear winner was found among the five SBFL techniques. Therefore, in the following results, we report both average FLT result (AvgAdFL) and worst FLT result (WorstAdFL).

4.5.2 RQ2: AdFL vs. Baseline and CBLS

Table 4.2 compares AdFL, with and without the guard (pruning statements by CBLS) to baseline. If we used an ideal cutoff (stopping the search after the last modified statement), how many percent fewer statements would we have to examine if we used AdFL (or G-AdFL) instead of the full program? The results show that, with an ideal cutoff, (G-)AdFL performs considerably better than baseline, often removing nearly half the statements proposed from consideration. Guarded AdFL improves on baseline by a larger margin, at the cost of some accuracy compared to non-guarded AdFL. Guarded AdFL is, of course, exactly as accurate as CBLS, since CBLS provides the cutoff for AdFL.We also conducted paired t-tests,

Table 4.3: Percent improvements over baseline. AdFL vs. CBLS shows the % improvement of average AdFL over CBLS.

Application	CBLS	Tarantula	Ochiai	Op2	Barinel	DStar	AvgAdFL	WorstAdFL	AdFL vs. CBLS
Elevation-API	55.17	91.72	93.79	90.34	91.72	92.41	92	90.34	66.75
NetBeans IDE undo-redo	90.71	92.28	93.42	96	93.85	94.42	94	92.28	3.62

comparing AvgAdFL with CBLS and WorstAdFL with CBLS. For AvgAdFL vs. CBLS, the mean difference is 15.2 ($p < 0.005$) with 95% confidence interval (12.68, 17.77). Also, when comparing AvgAdFL with CBLS, the effect size is large (Cohen's $d > 0.5$). For WorstAdFL vs. CBLS, the mean difference is 9.48 ($p < 0.005$) with 95% confidence interval (6.87,12.26). From all the results, we can say that AdFL performs better than CBLS in search space reduction. Given that AdFL is better than CBLS, we also want to know the average improvement by choosing AdFL over CBLS. For that, we used AvgAdFL and compared it directly with CBLS. We define the % improvement $((|AvgAdFL| - |CBLS|)/|CBLS|) * 100$. where $|CBLS|$ is the size of the CBLS set and $|AvgAdFL|$ is the mean EXAM SCORE across all 5 techniques. A one-sample t-test on % improvement shows mean improvement of 23.20% with $p < 0.005$ and 95% confidence interval of (19.71, 26.69).

4.5.2.1 Using Real Cutoffs

Results up to this point rely on using an EXAM SCORE provided ideal cutoff that is, in practice, not possible for developers. However, the results also show that the ranking position of the last removed statement is such that we could,

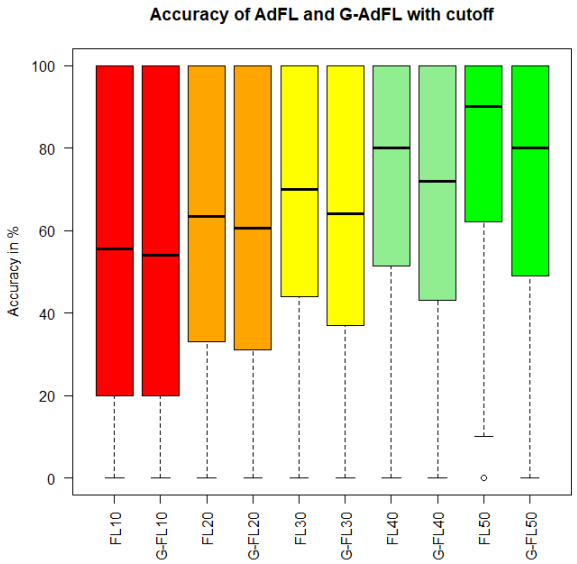

Figure 4.2: % Accuracy of AvgAdFL and AvgG-AdFL with different cutoff criteria. FL10 and G-FL10 show 10% cutoff.

almost always, prune much more of the search space than just using CBLS. In practice, developers using an AdFL technique to prune the search space must pick some cutoff, and ignore statements below that suspicousness. Given a cutoff, note that the effectiveness of search space pruning is fixed — if a developer decides to only consider the top 10% of AdFL-ranked statements for removal, the reduction compared to baseline is obviously 90%. What can vary is the *accuracy* of the produced adaptation. Real cutoffs are highly context-specific, trading more time required for adaptation off for less likelyhood of not removing some statements. We therefore used 10%, 20%, 30%, 40% and 50% cutoff values, with 10% cutoff meaning that TBSM only considers the 10% top ranked statements by AdFL. Figure 4.2 shows how accuracy varies with these cutoffs for our 800 synthetic

adaptations. In non-guarded AdFL, we directly apply the cutoff; in G-AdFl we may end up with a smaller search space than the cutoff, if CBLS itself prunes the space more than the cutoff would. Even with synthetic test suites with non-meaningful labels (the equivalent of a fault with no reliable location), for non-guarded AdFL 45% of data points have 100% accuracy at a 50% cutoff.

4.5.3 RQ3: AdFL in real-world adaptation scenarios

How does AdFL perform in the real world? Table 4.3 shows EXAM SCORE derived ideal cutoffs for two actual adaptation scenarios with meaningful test labelings. For the industrial scenario from the TSAS system, CBLS is only able to prune 55% of the program statements, but using AdFL and a cutoff of 10% would produce a 100% accurate result, even using the very worst FLT. For the NetBeans IDE, CBLS performs quite well, but even the worst FLT still improves on it by a few percent. For the elevation-API adaptation, baseline TBSM without heuristic guidance requires 460 minutes. Using CBLS, this can be reduced to 118 minutes, and using AdFL with Op2 (the worst performing FLT) and a 10% cutoff only requires 49 minutes. For the NetBeans IDE adaptation, baseline TBSM without heuristic guidance requires 175 minutes. Using CBLS, this can be reduced to 61 minutes, and using AdFL with Op2 and a 10% cutoff only requires 57 minutes, slight improvement. Developers do not know in advance how well CBLS will perform, but in both scenarios here, they can safely use AdFL, which has no additional computation cost, and a 10% cutoff, without loss of accuracy.

Table 4.4: Different TBSM strategies with corresponding time (in minutes) needed to build correct adaptations.

Application	baseline	CBLS	AdFL-10% cutoff	AdFL-inc
Elevation-API	460	118	49	35
NetBeans IDE undo-redo	175	61	57	20

4.6 Threats to validity

We used open source Java projects and the NetBeans IDE in order to compare with previous work on TBSM. The only proprietary program used in our analysis is TSAS. For TSAS, tests were labeled by developers, again avoiding any bias on our part. We used the standard EXAM SCORE measure to study the effectiveness of SPFL techniques.

All the projects used in the synthetic study as well as case study scenarios are Java projects. In order to verify generality, we need an hddRASS-like tool that can modify/reduce programs written in other programming languages. The primary threat to Java generalization is that while we suspect our synthetic results are typical of similar artificial adaptation problems, the most meaningful data is our two realistic case studies. Our non-proprietary data and results are available online at `https://github.com/amchristi/AdFL`.

4.7 Discussion: Best-Effort Incremental TBSM

The primary question raised by our experiments is how AdFLizer decides which statements are "unlikely" targets for modification: does the developer provide a

cutoff? Our synthetic results suggest that developers *could* provide a 50% cutoff, and likely see no significant loss in accuracy. This would be useful, and is better than CBLS on synthetic adaptation problems, but is far from ideal. Our results on real-world adaptations suggest that a 10% cutoff might be safe for actual adaptation based on meaningful test labels, even if the developer picks a "bad" SBFL technique. However, we only have two real-world scenarios, so we hesitate to claim that 10% is really safe for most adaptations. What cutoff should we pick if only 20 minutes are available to achieve the resource adaptations? There is a way to sidestep the entire issue: provide AdFLizer directly with a time budget, rather than a fixed percent cutoff.

The problem of determining a stopping criteria for TBSM is more acute than in APR: APR can stop whenever it has a version of a program that passes all tests. However, this distinction can be considered in a different light, to the advantage of TBSM. Namely, APR is useless until it produces a program that passes all tests, while TBSM is useful so long as it has removed some resource-using statements, and at least partly disabled problematic functionality, while preserving a useable system. If we assume that passing all unlabeled tests is usually an indicator a system is useable, even if not optimally adapted, then we can see that TBSM could be used as an incremental algorithm.

In order to demonstrate the effectiveness of best-effort incremental TBSM, we used AdFL-driven TBSM to produce resource adaptations for both of the case study scenarios. We started with a computation budget of 5 minutes and continued to increment it by 5 minutes until all resource consuming statements were

correctly modified. Table 4.4 represents the time budget required to build a correctly adapted version for different TBSM strategies: baseline, CBLS, AdFL with low cutoff, and AdFL with fixed computation budget, approximating incremental TBSM. The time budgets required to compute correct adaptations for the Elevation API scenario and NetBeans IDE are 35 minutes and 20 minutes respectively, saving almost 30% and over 60% of the cost for even a low cutoff. Because incremental results always pass all retained tests, the approach is both safe and effective, even when the time budget available is very limited.

4.8 Conclusions and Future Work

Resource-adaptation is crucial for the survivability of modern, mission-critical software systems. TBSM is a technique for building resource adaptations that relies on *test labeling* and *program modification*, and has been applied by the developers of TSAS in real world adaptation problems. This paper presents a novel application of Fault Localization (FL) techniques to improve the efficiency of TBSM, inspired by the application of FL to Automated Program Repair (APR) and Spectrum-based Feature Comprehension. We show that using FL in real world scenarios would allow up to 90% of the search space to be pruned in TBSM. Furthermore, using FL in TBSM makes it possible to reconsider TBSM as an incremental best-effort adaptation method.

Manuscript 4

Building Resource Adaptation via Test-Based Software Minimization: Application, Challenges, and Opportunities

Arpit Christi, Alex Groce, Austin Wellman

2019 The International Symposium on Software Reliability Engineering (ISSRE — Industry Track), Berlin, Germany.

Accepted for publication.

Chapter 5: Building Resource Adaptation via Test-Based Software Minimization: Application, Challenges, and Opportunities

5.1 Introduction

Ultra Large-Scale Systems, Internet of Things applications, and Cyber-Physical Systems face increased resource variability compared to programs of the past, with real-world physical consequences. Resources may be explicit: e.g., CPU, memory, or storage; or they may be implicit, e.g., libraries and protocols. The inability of software to handle resource changes effectively and automatically can produce inferior or potentially vulnerable systems [42].

Self-Adaptive Software Systems (SAS) modify their own behavior in response to changes in operating conditions. Resource Adaptive Software Systems (RASS) are a subset of SAS where the trigger for adaptation is variability or unavailability of resources. For longevity and survivability of modern systems, self-adaptation for resource changes is essential hence the problem of self-adaptation is well studied [41]. To this end, DARPA started BRASS (Building Resource Adaptive Software Systems) program, a major initiative to bring researchers and practitioners together to solve the problem of building resource adaptive software.

Researchers have proposed many techniques, approaches, and tools to build SAS automatically [42, 70, 47]. We previously proposed a solution called Test-

Based Software Minimization to build SAS to overcome some of the limitations of previous approaches based on our work with Raytheon developers as part of DARPA BRASS program [18]. In this work, we applied TBSM to produce adaptations of a mission-critical military system and the popular NetBeans `Java` IDE. Based on our observations:

1. We demonstrate that unlike most other previous approaches, resource adaptation via TBSM is a conceptually simple, usable, and applicable technique.

2. We present challenges we faced while attempting to produce accurate resource adaptations automatically.

3. We present the solutions we employed to solve some of those challenges.

4. We explore a synergy between some of the solutions to the problems of TBSM and the problems of reliable software development.

In the process, we identify opportunities to further aid developers in effectively employing test-based minimization to achieve resource adaptations.

While working with developers attempting to build SAS for a real-world, mission-critical system, we observed that (1) most developers speak the language of tests fluently, unlike that of formal methods or architectural descriptions, (2) tests are thus, for even most complex projects in the real world, the *only* semi-formal specification available. To exploit both the widespread availability of tests for mission-critical systems and developers' familiarity with tests, we proposed capturing resource adaptation specifications using tests. TBSM relies on developers' understanding of tests, and how tests relate to program features and resource usage.

Developers encode this information by simply labeling the tests. Test labels can be a multi-dimensional space in feature, resource, and priority. To demonstrate the concept, we assume a single adaptation objective: a single resource faces variability or unavailibility. The concept is demonstrated in Fig. 4.1 (a simplified version of Fig 2. in the original paper proposing TBSM [18]). Labeled tests define functionality that can be sacrificed to reduce resource usage. Unlabeled tests define the functionality that is to be retained and not modified. A tool called hddRASS takes as its input the test suite with labeled and unlabeled tests and the original program, and produces a minimized program such that the functionality marked by labeled tests is removed. hddRASS achieves this by temporarily removing the labeled tests from the suite and using the Hierarchical Delta Debugging (HDD) algorithm to find a minimal program, such that retained tests continue to pass [56]. The basic idea is that if the labeled tests define a functionality that uses resources, removing that functionality will ensure resource adaptation. TBSM builds adaptations, in a sense, in the same way that Automatic Program Repair fixes faults. In APR, the goal is to modify the program to pass a set of previously failing tests (without failing previously passing tests) [57]. In TBSM, the goal is to modify the program while removing as much code as possible, while still passing a set of unlabeled tests.

5.2 Evaluation

We evaluate resource adaptation via TBSM/hddRASS along several dimensions: effectiveness, usability, applicability, and scalability.

5.2.1 Effectiveness

For the *Elevation API* scenario, all 4 necessary modifications were correctly performed automatically by TBSM. The development team was able to confirm the adaptations for library change by (1) examining the code for necessary modifications (2) running the tests and (3) using the application. As TSAS is a proprietary system, we have to rely on conformation from the development team. All the necessary modifications to adapt the *NetBeans IDE* were also performed correctly. We observe that all 19 resource consuming statements (as identified in previous work [17]), the statements that fill the undo-redo buffers, were correctly modified by the technique. We have shown previously that the adapted IDE 1) cannot perform undo-redo operations and 2) uses far less memory, in a controlled experiment setting allowing only edits to a single text file [18]. We also confirm otherwise normal operation of the IDE.

5.2.2 Usability

Most adaptation approaches require developers to learn a new specification language or add complex annotations to code. To use such techniques, develop-

ers typically need to use modeling approaches, architectural specifications, formal methods, etc., to map their application to a view that the adaptation technique supports; this is obviously a time consuming and error-prone task [70, 47]. TBSM only requires developers to look at their tests and make a "good guess" about the feature(s)/resource(s) relevant to each test. Developers are familiar with tests and can likely perform this task without additional training. For the *NetBeans IDE* case, it took us only a few hours to determine 3 tests to label out of 146 tests, despite the fact that we are not developers, or familiar with the NetBeans code in any way before we examined it. For the *Elevation API*, actual developers were almost instantly able to determine the tests pertaining to certain features.

5.2.3 Applicability

While developing the RAINBOW framework, Garlan et al. noted that most previous approaches for SASS were scenario-based or application-specific, and not very reusable [33]. To evaluate applicability, we measured the effectiveness of hddRASS as a tool. We believe hddRASS to be complete for `Java` 7: it applies to any programs written in `Java` 7. To confirm this, we checked for thrown exceptions or unprocessed statements when minimizing real-world systems. While applying hddRASS to TSAS, we processed 70 `Java` files with 338 non-constructor methods. For the *NetBeans IDE*, we processed 2 `Java` files with 38 non-constructor methods. We also applied hddRASS to 40 randomly selected classes with a total of 1,207 methods across 10 open source projects, using a random labeling scheme [17]. We

observed neither exceptions nor unprocessed statements across these experiments. Based on this, we can say that hddRASS as a tool, and TBSM as a technique, is generally applicable to `Java` applications. To extend it to other programming language, one needs to build hddRASS like tool for that programming language.

5.2.4 Scalability

For the *Elevation API* the application of hddRASS, using no heuristics, on all 70 Javafiles took 7 hours and 50 minutes. For *NetBeans IDE*, adapting two files in `openide.awt` using no heuristics took 2 hours and 35 minutes. As the program size increases or test suite runtime increases, scalability becomes the limiting factor in TBSM. For offline adaptations, scalability may not be a limiting factor. But for live systems deployed in the wild, the system needs to be halted until adaptations are performed and a long offline period is not normally acceptable. The speed of the most thorough version of TBSM without heuristics to guide adaptation is likely acceptable for use before deployment, e.g., to prepare a specialized version of a system for a resource-limited platform, but unsuitable for field use. However, as we discuss below, even without additional developer effort, use of heuristics to compute approximate or incremental best-effort adaptations can mitigate this problem substantially.

5.3 Challenges

We describe the major challenges that we faced while applying TBSM to build real-world RASS, and how we attempted to solve these challenges for the case study scenarios. In the process, we identify major research challenges in TBSM. We note the similarity between the challenges in TBSM and the challenges observed by researchers in APR [50].

5.3.1 Search Space

TBSM indiscriminately processes all the statements of a program, so the search space is accordingly vast. This can be significantly reduced by selecting only statements likely to actually be modified. We empirically evaluated a large number of adaptation related modification for 800 synthetic adaptations to derive heuristics to guide target selection. We derived statistics-based heuristics (H1, H2) and dynamic-analysis-based heuristics (CBLS, AdFL) based on our evaluation [17, 16]. The CBLS heuristic relies on coverage information for the labeled and unlabeled test suite. We repurpose Spectrum-Based Fault Localization (SBFL) to derive the more general AdFL heuristic by establishing equivalence between core components of SBFL and core components of TBSM. We found that dynamic-analysis-based heuristics perform better in empirical analysis. For the *Elevation API* scenario, CBLS reduces the search space by 55% and AdFL heuristics reduces search space by 92% bringing the wall clock time to build adaptations down from 460 minutes to 118 minutes for CBLS and 49 minutes for AdFL. For the *NetBeans IDE* scenario,

CBLS heuristics reduces search space by 90% and AdFL heuristics reduces search space by 94% bringing the wall clock time down from 175 minutes to 61 minutes and 57 minutes for CBLS and AdFL respectively.

The CBLS heuristics provides search space reduction while AdFL prioritizes the search space based on likeliness of modification. We demonstrated that the likely targets of modifications would appear earlier in the sorted order if we use AdFL [16]. We used this fact to circumvent the search space issue all together. We proposed best-effort incremental TBSM as a technique were developers provide a time limit to finish the adaptations [16]. The best-effort incremental TBSM will always produce a useable system in the given amount of time, with resource adaptation achieved entirely or partially. For *Elevation API* and *NetBeans IDE* scenarios, time limits of 35 minutes and 20 minutes, respectively, were sufficient to perform complete resource adaptation using best-effort incremental TBSM. Best-effort incremental TBSM performs better than AdFL heuristics with 90% search space reduction for both scenarios.

5.3.2 Test Suite Runtime

Like APR, TBSM is a generate-and-validate technique that must execute a potentially large test suite to evaluate each modification. TBSM benefits from a large test suite, but running all tests is often prohibitively slow. The version of *NetBeans IDE* we used has a test suite that requires 7+ hours to run for all 1193 modules. TSAS also exhibits unacceptably slow complete test suite runtime. For

both case studies, we observe that running only modified-module-related tests is sufficient, as modules are self-contained and have good individual test suites. We observe that even such a simple ad-hoc test selection reduces test runtime to 34 seconds and 18 seconds for the *NetBeans IDE* and *Elevation API*, respectively. For most resource adaptation scenarios, such an ad-hoc technique is not feasible and scalable. We need test selection and prioritization (since early failures also limit runtime) tailored to TBSM.

5.3.3 Inefficient Algorithm

Our hddRASS reducer uses a modified HDD* algorithm with worst-case running time of $O(n^3)$, where n in our case is the number of statement nodes of the abstract syntax tree of the program [56]. We employ HDD* as the underlying driver for TBSM because it guarantees convergence and minimality. By minimality, we mean that any output program produced by TBSM is minimal and cannot be modified any further. One solution we employ is to forgo the minimality guarantee of HDD* and use a pure greedy search, trading accuracy for efficiency. With a greedy search strategy, we can build resource adaptations 1.72 times faster while retaining 94% accuracy for the *Elevation API* and 1.90 times faster while retaining 100% accuracy for *NetBeans IDE*. We need further evaluation of different search strategies and heuristics-based modifications to HDD*.

5.3.4 Overfitting

The issue of overfitting is well studied in APR, and we face the same problem[72]. TBSM produces test-adequate adaptations, adaptations that are correct with respect to a test suite and test labeling scheme. For our case studies, Type 1 errors are rare, and even related statements are usually removed; e.g., if a single resource consuming statement is the body of a for loop, the loop is removed as well. We consider directly or indirectly resource-adaptation-related modifications as correct modifications. We do observe a large number of false modifications, modifications that are not directly or indirectly related to resource adaptations: these are the more common Type 2 errors. Indeed, the false modifications outnumber true ones in our scenarios. A total of 49 and 121 modifications are performed by TBSM for the *Elevation API* and *NetBeans IDE* scenarios, respectively. We observe 91% and 51% false modification rates for the *Elevation API* and *NetBeans IDE* adaptations, respectively. As TBSM generated test-adequate adaptations overfit a given test suite and labeling scheme, TBSM is vulnerable to both (1) inadequacy of the test suite and (2) labeling errors. Developers reported instances where TBSM-generated modifications removed untested but desired functionality.

5.4 Test Adequacy and Adaptation Quality

Developers using TBSM expressed more concern over overfitting than any other problem. In this section we note the origins of this problem, and propose that solving it is synergistic with addressing a long-standing problem in software devel-

opment.

5.4.1 Test Inadequacy

TBMS only knows what the tests tell it. If tests are grossly inadequate, it is bound to produce sub-standard adaptations; in particular, TBSM is aggressive, and if a test does not force the inclusion of code in the adapted program, TBSM is likely to remove that functionality. There are two basic sources of inadequacy. One is true inadequacy, where the feature that TBSM "accidentaly" removes is simply not tested at all. The solution to this problem is to test all features of a system that are of any actual importance, a worthy goal in any case. The second problem is that there may be some tests for a feature, but they are all mixed-in with tests for a feature that is to be removed. That is, the tests for a system are not well decomposed, and some functionality is only tested in conjunction with a feature that a developer wants to "adapt away." Fortunately, these problems can be addressed by taking a better approach to tests in general.

5.4.2 Code Coverage

TBSM will remove any code that is not covered by any tests, and not required for compilation. The easiest way to avoid this problem is to start with a high-coverage, high-quality test suite. Developers can target poorly covered code with manual tests or use targeted test generation tools to cover such code. Improving code

coverage using random testing reduced overfitting by 23% and 8% for *Elevation API* and *NetBeans IDE* respectively. Naïve test generation will not necessarily improve matters, however. Developers have to label any generated tests, and if tests mix multiple features, there still may often be cases where a feature not to be removed is never tested in isolation. We propose using delta-debugging, in particular *cause reduction* to help isolate features in generated test cases [37]: if a generated test covers two features, perhaps one can be removed by reducing the test with respect to targeted code for one feature only. In fact, in principle, we can automatically take each individual line of code, and produce a tests whose only purpose is to cover that line of code, with any other behavior required to retain that coverage. If developers can then label files or functions in a program by functionality, the targeted code's location can automatically provide a label.

5.4.3 Better Tests

Based on our study (with the developers) of overfitting, we identified a few categories covering most of the incorrect removals. For the *Elevation API*, there were no tests for logging, and so all `LOG` statements were removed. This problem is likely very common, but can be easily remedied by explicitly testing system logging as a feature. Testing the logging output for performed actions in the tests for the functionality is a seldom-performed, but likely beneficial practice. Similarly, exception-handling code was often inadequately tested.

We worked with the developers to produce a new test covering each major

overfitting category identified. We produced just three and four (relatively small) tests, respectively, for the *Elevation API* and *NetBeans IDE* scenarios, resulting in 20% and 66% reduction in overfitting. Identifying overfitting categories can be a highly context-specific task, but we suspect new automated test generation methods targeting specific under-tested categories such as logging and exception handling may be useful here.

5.4.4 Motivating Better Tests

Of course, the reason tests are inadequate in the first place is that developers did not produce extremely high-quality, fine-grained tests. Developers often fail to test important functionality, perhaps on the grounds that it is "only logging" (despite the fact that logging is, in the long run, the only visibility and debugging aid for many systems). More seriously, even developers of high-quality systems may not test certain exceptional behavior paths. Targeting often-overlooked functionality is a good future goal for the automated testing community; perhaps logging code is not tested because developers expect to change logging output frequently in reponse to debugging needs, and don't want to change the tests. Automated methods could generate (and re-generate) properly labeled tests just to test the logging output of existing unit or system tests, without making this a significant burden on developers. Aggressive random testing and fuzzing is likely to expose untested exceptional scenarios. Interestingly, TBSM can also be seen as a way to identify untested code: just run TBSM on a code base with *no* labeled tests.

Anything removed is a good candidate for additional testing!

But there is a larger picture here. We suspect that developers do not put as much effort as TBSM would ideally expect into testing because the payoff of testing is not always clear. The relationship between test quality, even measured by mutation testing rather than coarse-grained coverage only, and detection and anticipation of important defects worth fixing, is not always extremely strong [6]. However, there is clearly some relationship between test quality and eventual system reliability, and one way to see TBSM (and future techniques of the same kind) is as a new way to get better tests. Because most code does not have a critical bug, effort to write tests is currently seen as a burden, a "cost-center," not a "revenue-center" for developers, to adapt a business analogy. It may have to be done, but testing is purely to ward off disaster. However, TBSM offers a different way to look at tests: high quality tests allow developers to be more productive, in that they enabled the *automation of certain kinds of changes to the software.* Writing high-quality tests may become a much more pleasant part of software development if the payoff is not having to write as much complex code to, e.g., adapt a system to a more limited platform, or handle changes to system libraries, or produce a more secure version with a smaller attack surface due to removing insecure functionalities. If tests can produce "productivity revenue" rather than simply allowing the detection of faults in previous productivity, they may be more valued, and receive more attention. The upshot will be more reliable systems that are also easier to adapt to resource-limited situations.

5.5 Tool and Data Availability

The hddRASS tool is available at `https://github.com/amchristi/hddRASS`. It supports the baseline TBSM technique as well as heuristics guidance (H1, H2, CBLS, or AdFL). It also provides support to execute TBSM in greedy mode. As TSAS is a proprietary system, we cannot make its source code, test suite, or tests labels available. The NetBeans IDE (and hence `openide.awt` module) source code is available online. We also provide the exact `openide.awt` module source code, test suite and test labels that we used `https://github.com/amchristi/AdFL`. The same link contains 800 synthetic adaptation scenarios with source code, tests, and test labels for empirical evaluation and comparison.

5.6 Ongoing Work and Future Directions

As part of DARPA BRASS project, we work with multiple software development teams and other collaborators. Our communication with these stakeholders as well as the feedback that we received from the development teams using our work are driving our ongoing efforts.

5.6.1 Ongoing Work

5.6.1.1 Improving Applicability

The underlying tool hddRASS that drives TBSM only supports Java. Raytheon development team uses TBSM for TSAS, an application written in Java. We also developed, and plan to release in the near future, a C++ version of hddRASS. We developed it specifically for use by ROS (Robotics Operating System) application developers to adapt against ROS version changes and other ROS package changes, but it can be applied to any C++ code base.

5.6.1.2 Minimization vs. Modification

One core current limitation of TBSM is that it only offers minimization (reduction), rather than code *modification*. Adaptation, however, can generally include (1) reduction, (2) replacement, and (3) enhancement [42]. The published TBSM version only supports reduction. While developing the C++ version of hddRASS, we incorporated many modification operators from APR used to fix faults, making some replacement capabilities available [31, 9, 23]. We plan to continue to improve hddRASS to provide more modification capabilities.

5.6.2 Future Directions

We are currently investigating multiple other ways to speed up TBSM, including using static and dynamic analysis to precompute the effects of program statements on test oracles and using test case selection and prioritization to reduce the running time of tests in TBSMs generate-and-validate loop.

The original work in TBSM emphasized the need for a "good" test suite. Previous heuristics suggested coverage as a way to define goodness of a test suite; the CBLS heuristic depends on coverage information. Because of the success of AdFL heuristics in isolating and prioritizing modification targets, we plan to consider test suite diagnosability metrics as a more refined way to define the goodness of a test suite for TBSM, using approaches proposed by Baudry et al. and Perez et al. [10, 64]

For our work, we mostly used existing test suites provided by the developers. As discussed above, sometimes a test that a developer labeled as pertaining to onefeature may contain code that exercises other features. Similarly, an unlabeled test, apart from testing functionality that needs to be retained, may exercise sacrificial features. Presence of such tests makes it harder for TBSM to differentiate between an adaptation-related modification and an accidental modification, resulting in underfitting or overfitting. To mitigate the situation, we plan to extend work on test-case purification [78] and test-case decomposition [37, 20] to *automatically produce more fine-grained tests*, and ideally, automatically label them.

5.7 Conclusions

Test-based software minimization offers a conceptually simple, widely applicable, easily understood approach to generating resource adaptations. While TBSM faces significant scalability challenges, and cannot be applied where tests are impossible to label, or to resources not associated with removable features, it also benefits from a powerful synergy: namely, the best way to improve TBSM is to improve software test suites. Making test suites faster to execute, improving their granularity so that tests cover distinct features, increasing code coverage (and other test quality measures), and improving test prioritization techniques all lead not only to more efficient and effective TBSM, but to better fault detection and debugging.

Acknowledgments: this work was partly funded by the DARPA BRASS [42] program, and the authors would like to thank our collaborators at Oregon State University and Raytheon/BBN.

Chapter 6: Conclusion

The central goal of this dissertation is to come up with a technique to build self-adaptive software that is highly applicable, usable, and efficient.

In the first part of the dissertation, we proposed a way to capture adaptive software requirements via test labeling. Based on the labeled test suite that captures sacrificial requirements, we introduced a new technique, TBSM, to build self adaptive software. We implemented TBSM using a new tool called hddRASS. Using 2 case study scenarios, we demonstrated that TBSM is applicable, usable, effective, but inefficient [19].

In the second part of the dissertation, we explored multiple ways to improve the efficiency of TBSM. To do so, we considered the most critical issue affecting efficiency; the search space problem. We proposed four different ways to improve efficiency and found out that AdFL is the most efficient technique.

We are exploring multiple ways to improve efficiency further that includes looking into (1) improving the underlying algorithm by using checked coverage (2) improving test suite running time by considering test selection and prioritization [79].

As the effectiveness of the results produced by TBSM heavily relies on test suite quality and accuracy of test labeling, we plan to (1) evaluate test suite quality criteria for TBSM (2) use test case purification, decomposition, etc. to generate fine-grained tests.

Bibliography

[1] Javaparser. `http://javaparser.org/`.

[2] Jmap. http://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html.

[3] Linux desktop project. https://ldtp.freedesktop.org/wiki/.

[4] NetBeans IDE. https://netbeans.org/.

[5] Netbeans IDE bug 45011. https://netbeans.org/bugzilla/show_bug.cgi?id=50411.

[6] Iftekhar Ahmed, Rahul Gopinath, Caius Brindescu, Alex Groce, and Carlos Jensen. Can testedness be effectively measured? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 547–558, New York, NY, USA, 2016. ACM.

[7] Youssif Al-Nashif, Aarthi Arun Kumar, Salim Hariri, Yi Luo, Ferenc Szidarovsky, and Guangzhi Qu. Multi-level intrusion detection system (ml-ids). In *Proceedings of the 2008 International Conference on Autonomic Computing*, ICAC '08, pages 131–140, Washington, DC, USA, 2008. IEEE Computer Society.

[8] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano-sla based management of a computing utility. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*, pages 855–868, 2001.

[9] Andrea Arcuri. *Automatic software generation and improvement through search based techniques*. PhD thesis, University of Birmingham, UK, 2009.

[10] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 82–91. ACM, 2006.

[11] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. Moses: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering*, 38(5):1138–1159, 2011.

[12] Paulo Casanova, David Garlan, Bradley R. Schmerl, and Rui Abreu. Diagnosing unobserved components in self-adaptive systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 75–84, 2014.

[13] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[14] Betty H. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 468–483, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] S. W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141, May 2009.

[16] A. Christi, A. Groce, and R. Gopinath. Evaluating fault localization for resource adaptation via test-based software modification. In *2019 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 26–33, July 2019.

[17] Arpit Christi and Alex Groce. Target selection for test-based resource adaptation. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*, pages 458–469, 2018.

[18] Arpit Christi, Alex Groce, and Rahul 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*, pages 61–70, 2017.

[19] Arpit Christi, Alex Groce, and Austin Wellman. Building resource adaptation via test-based software minimization: Application, challenges, and opportunities. In *The 30th International Symposium on Software Reliability Engineering (ISSRE- Industry Track)*, 2019. Accepted for publication.

[20] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Memphis, TN, USA, October 15-18, 2018*, pages 184–191, 2018.

[21] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67. ACM, 2017.

[22] Rogério De Lemos, Holger Giese, Hausi A. Muller, Mary Shaw, Jesper Andersson, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Marin Litoiu, Antonia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezze, Christian Prehofe, Wilhelm Schäfer, Rick Schlichting, Bradley Schmerl, Dennis B. Smith, Joo P. Sousa, Gabriel Tamura, Ladan Tahvildari, Norha M. Villegas, Thomas Vogel, Danny Weyns, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In Rogério De Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems*, volume 7475 of *Dagstuhl Seminar Proceedings*, pages 1–26. Springer, 2013.

[23] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 65–74. IEEE Computer Society, 2010.

[24] Marcio Eduardo Delamaro, Jeff Offutt, and Paul Ammann. Designing deletion mutation operators. In *International Conference on Software Testing, Verification and Validation*, pages 11–20, 2014.

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

[26] Jim Dowling and Vinny Cahill. Self-managed decentralised systems using k-components and collaborative reinforcement learning. In *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, WOSS '04, pages 39–43, New York, NY, USA, 2004. ACM.

[27] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 7–16. ACM, 2010.

[28] Ahmed Elkhodary, Sam Malek, and Naeem Esfahani. On the role of features in analyzing the architecture of self-adaptive software systems. In *4 th Workshop on Models@ run. time at MODELS 09*, 2009.

[29] Franck Fleurey and Arnor 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*, MODELS '09, pages 606–621, Berlin, Heidelberg, 2009. Springer-Verlag.

[30] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, March 2006.

[31] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 947–954. ACM, 2009.

[32] Erik M. Fredericks, Andres J. Ramirez, and Betty 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*, SEAMS '13, pages 169–174, 2013.

[33] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, Oct 2004.

[34] Heather J. Goldsby, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Danny Hughes. Goal-based modeling of dynamically adaptive system requirements. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS '08, pages 36–45, Washington, DC, USA, 2008. IEEE Computer Society.

[35] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan 2012.

[36] 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)*, pages 737–748, Nov 2015.

[37] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability*. accepted for publication.

[38] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 243–252, 2014.

[39] Svein Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George Angelos Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *Journal of Systems and Software*, 85(12):2840–2859, 2012.

[40] Renáta Hodován and Ákos Kiss. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, pages 31–37, New York, NY, USA, 2016. ACM.

[41] Danny Hughes. Seams 2018 keynote speech. `https://conf.researchr.org/track/seams-2018/seams-2018-papers#program`. Accessed: 2018-08-09.

[42] Jeffrey Hughes, Cassandra Sparks, Alley Stoughton, Rinku Parikh, Albert Reuther, and Suresh Jagannathan. Building resource adaptive software systems (BRASS): Objectives and system evaluation. *SIGSOFT Softw. Eng. Notes*, 41(1):1–2, February 2016.

[43] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[44] Gabor Karsai and Janos Sztipanovits. A model-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):46–53, May 1999.

[45] Christian Krupitzer, Felix Maximilian Roth, Christian Becker, Markus Weckesser, Malte Lochau, and Andy Schürr. Fesas ide: An integrated development environment for autonomic computing. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 15–24. IEEE, 2016.

[46] Christian Krupitzer, Felix Maximilian Roth, Sebastian Vansyckel, and Christian Becker. Towards reusability in autonomic computing. In *2015 IEEE International Conference on Autonomic Computing*, pages 115–120. IEEE, 2015.

[47] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive Mob. Comput.*, 17(PB):184–206, February 2015.

[48] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards requirements-driven autonomic systems design. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[49] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[50] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[51] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *International Conference on Software Engineering*, pages 702–713, May 2016.

[52] Joseph P. Loyall, David E. Bakken, Richard E. Schantz, John A. Zinky, David A. Karr, Rodrigo Vanegas, and Kenneth R. Anderson. Qos aspect languages and their runtime integration. In *Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '98, pages 303–318, London, UK, UK, 1998. Springer-Verlag.

[53] Markus Luckey and Gregor 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*, SEAMS '13, pages 143–152, Piscataway, NJ, USA, 2013. IEEE Press.

[54] Matias Martinez and Martin Monperrus. Astor: A program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 441–444, New York, NY, USA, 2016. ACM.

[55] Daniel Menasce, Hassan Gomaa, Joao Sousa, et al. Sassy: A framework for self-architecting service-oriented systems. *IEEE software*, 28(6):78–85, 2011.

[56] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 142–151, 2006.

[57] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, January 2018.

[58] Mirko Morandini, Frédéric Migeon, Marie-Pierre Gleizes, Christine Maurel, Loris Penserini, and Anna Perini. A goal-oriented approach for modelling self-organising mas. In Huib Aldewereld, Virginia Dignum, and Gauthier Picard, editors, *Engineering Societies in the Agents World X*, pages 33–48, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[59] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *International Conference on Architecture of Computing Systems*, pages 124–138. Springer, 2005.

[60] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: program repair via semantic analysis. In *International Conference on Software Engineering*, pages 772–781, 2013.

[61] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[62] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[63] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *International Conference on Software Engineering*, pages 609–620, 2017.

[64] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 654–664, 2017.

[65] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, pages 191–201, 2013.

[66] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[67] Dong Qiu, Bixin Li, Earl T Barr, and Zhendong Su. Understanding the syntactic rule usage in java. *Journal of Systems and Software*, 123:160–172, 2017.

[68] Nauman A. Qureshi, Ivan J. Jureta, and Anna Perini. Towards a requirements modeling language for self-adaptive systems. In *Proceedings of the 18th Inter-*

national Conference on Requirements Engineering: Foundation for Software Quality, REFSQ'12, pages 263–279, Berlin, Heidelberg, 2012. Springer-Verlag.

[69] Paul Robertson and Robert Laddaga. Self-star properties in complex information systems. chapter Model Based Diagnosis and Contexts in Self Adaptive Software, pages 112–127. Springer-Verlag, Berlin, Heidelberg, 2005.

[70] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009.

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

[72] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 532–543, New York, NY, USA, 2015. ACM.

[73] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering*, pages 356–366, 2013.

[74] Jules White, Douglas C Schmidt, and Aniruddha Gokhale. Simplifying autonomic enterprise java bean applications via model-driven development: a case study. In *International Conference on Model Driven Engineering Languages and Systems*, pages 601–615. Springer, 2005.

[75] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE*, RE '09, pages 79–88, Washington, DC, USA, 2009. IEEE Computer Society.

[76] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Softw. Eng.*, 42(8):707–740, August 2016.

[77] Qi Xin, Steven P. Reiss, and Shriram Krishnamurthi. Program repair using code repositories. Technical report, Brown University, 2016.

[78] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63. ACM, 2014.

[79] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[80] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. *CoRR*, abs/1703.00198, 2017.

[81] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.

APPENDICES

Appendix A: Appendix to demonstrate the statement type coverage of selected subjects for Empirical Analysis

We present the distribution of statement types in our 40 subject classes in Fig A.1. The distribution of statement types establishes that we have covered commonly used statement types that exist in the Java program. To understand the syntactic rules in Java, Qiu et al. studied 19 Statement types in Java [67]. We looked at the ten most commonly used java statement types and verify that our randomly selected subjects also cover those statements frequently. These ten most commonly used java statement types cover more than 95% of syntactic statement rules in Java programs. For other statement types, we observed consistent, very low distribution in both their analysis and our analysis. We also observe frequent occurrence of 3 most common java statements that they observed - `ExpressionStmt`,`IfStmt`, and `ReturnStmt`. Consider 2 declaration statements within a method (1) `int a;` (2) `A a` where `A` is a Type. `Javaparser`— treats both the statements as `ExpressionStmt` while Qiu et al. classify them as variable declaration. Also, Qiu et al. did not consider `SwitchEntryStmt` but only `Switch` statement. Due to the use of `JavaParser`, we observe both `SwitchStmt` and `SwitchEntryStmt`. As number of `SwitchEntryStmt` are normally larger than `SwitchStmt`, our results include `SwitchEntryStmt` ahead of `SwitchStmt`.

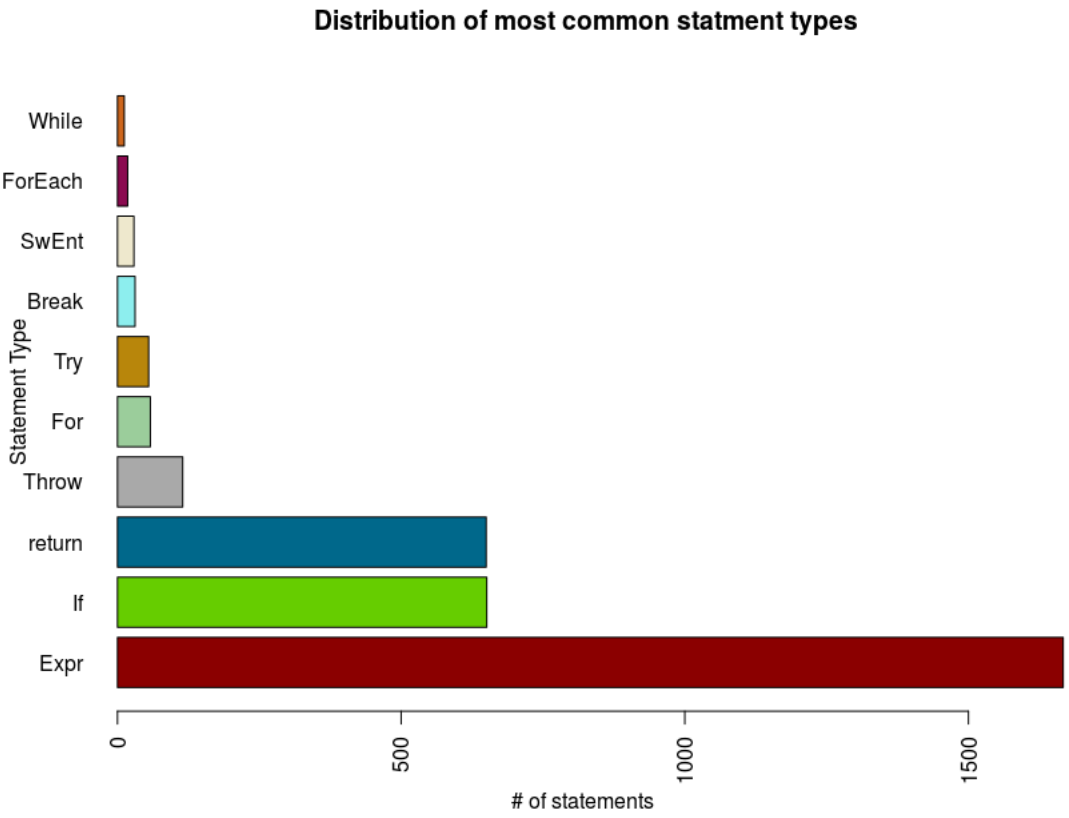

Figure A.1: Distribution of common JavaParser statement types: Expr represents `ExpressinStmt` and SwEnt represents `SwitchEntryStmt`.