Resource Adaptation via Test-Based Software Minimization

Arpit Christi
School of EECS
Oregon State University
Corvallis, Oregon, USA
christia@oregonstate.edu

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

Rahul Gopinath
School of EECS
Oregon State University
Corvallis, Oregon, USA
gopinatr@oregonstate.edu

***Abstract*—Building software systems that adapt to changing resource environments is challenging: developers cannot anticipate all future situations that a software system may face, and even if they could, the effort required to handle such situations would often be too onerous for practical purposes. We propose a novel approach to allow a system to generate resource usage adaptations: use delta-debugging to generate versions of software systems that are reduced in size because they no longer have to satisfy all tests in the software's test suite. Many such variations will, while retaining core system functionality, use fewer resources. We describe a tool for computing such adaptations, based on our notion that labeled subsets of a test suite can be used to conveniently describe possible relaxations of system specifications. Using the NetBeans IDE, we demonstrate that even without additional infrastructure or heuristics, our approach is capable of quickly and cleanly removing a program's undo functionality, significantly reducing its memory use, with no more effort than simply labeling three test cases as undo-related.**

I. 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 consuming and error prone. One way to tackle these problems is to let the software evolve itself in response to change [1]. Adaptations may manifest as *restricted functionality*, *altered functionality*, or *enhanced functionality* [2].

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 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 [3] 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) [4] modified the algorithm to produce better results for tree-structured test inputs, especially, e.g., programs input to compilers. *Cause reduction* [5], [6] 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 [5]; 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 proposed in a research roadmaps for software engineering for self-adaptive systems by Cheng et al. [7]. 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.

[1]Often abbreviated as *RASS*.

```
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");
```

Fig. 1: A fragment of a Java program with potentially over-aggressive logging.

A. A Simple Example: Removing Logging

Consider the code in Figure 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, 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 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

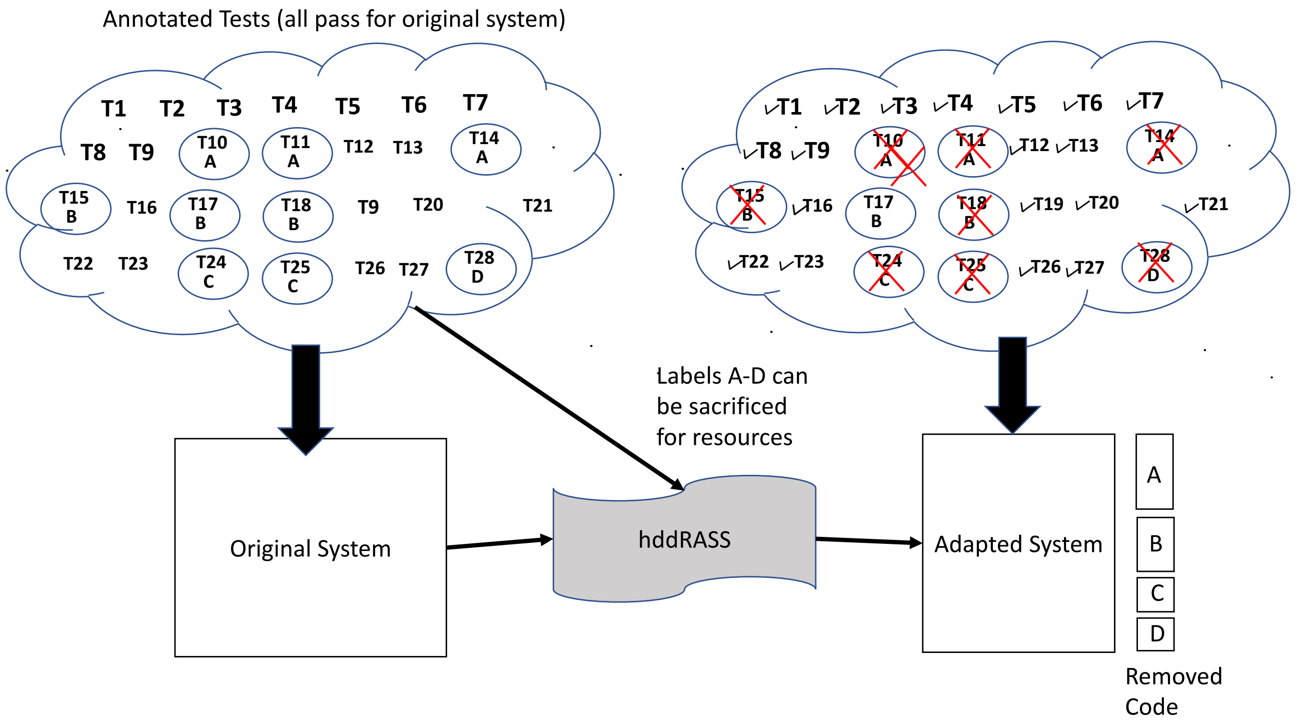
Annotated Tests (all pass for original system)
T1 T2 T3 T4 T5 T6 T7
T8 T9 T10 A T11 A T12 T13 T14 A
T15 B T16 T17 B T18 B T9 T20 T21
T22 T23 T24 C T25 C T26 T27 T28 D
Labels A-D can be sacrificed for resources
Original System
hddRASS
Adapted System
A
B
C
D
Removed Code

Fig. 2: The test-based approach to resource adaptation by program minimization

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 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.

B. 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.

II. Related work

The field of self adaptive systems has seen renewed interest in recent years. Salehie et al. [8] summarize some recent work. Different engineering approaches to building adaptive software systems are discussed by Kruptizer et al. [1]; fundamental methods include model-based approaches [9] and architecture-based approaches [10]. Cheng et al. [7] 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. [11] 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. [12] suggest choosing only a subset of test cases to execute based on resource constraints for runtime adaptations.

Delta-debugging [3] is an algorithm for reducing the size of failing test cases or test inputs. *Hierarchical delta debugging* [4] 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 [6], [5]. 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. [13] in their criticism of much work in automatic program repair. Conceptually, this relates to the statement-deletion mutation operator [14], a special instance of deletion mutation operators [15] 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.

III. Core concepts

Building resource adaptive systems [2] 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.

A. Test Annotations

We propose test annotation as one of the ways to capture adaptive requirements [7]. 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 [16].

B. (1-)Minimal Program

Zeller and Hildebrandt. [3] 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* [17], 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 1: Program fragment

```python
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.

IV. 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.

A. 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:      $level \leftarrow$ HEIGHT($inputTree$)
3:      $nodes \leftarrow$ TAGNODES($inputTree, level$)
4:      while $nodes \neq \emptyset$ do
5:          $minConfig \leftarrow$ DDMIN($nodes, inputTree$)
6:          PRUNE($inputTree, level, minConfig$)
7:          $level = level - 1$
8:          $nodes \leftarrow$ TAGNODES($inputTree, level$)
9:      end while
10: end procedure
```

Here DDMIN is the standard delta-debugging algorithm [3], as also used as a subroutine in HDD [4]. 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.

B. 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 [14]. Moreover, focusing on statement deletion as our smallest granularity of change means that to reject an adaptation as invalid, tests that

TABLE I: 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

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 [4]. 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.

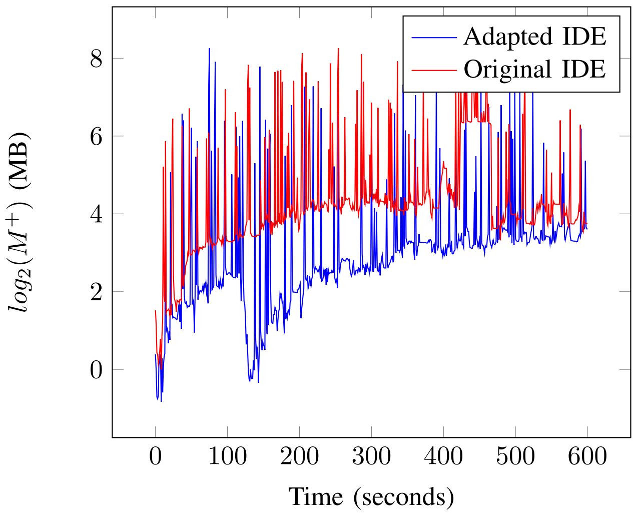

Fig. 3: Run 1 - 10 min

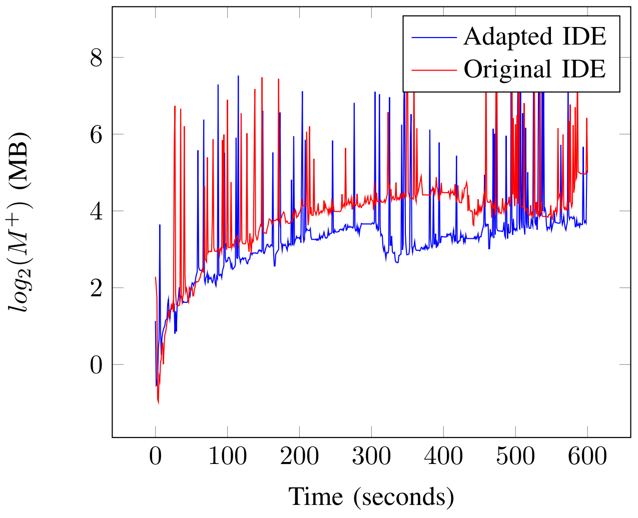

Fig. 4: Run 2 - 10 min

V. Experimental Evaluation

A. NetBeans Case Study

We use the NetBeans IDE [18] (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, 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 [19] 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.

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.

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

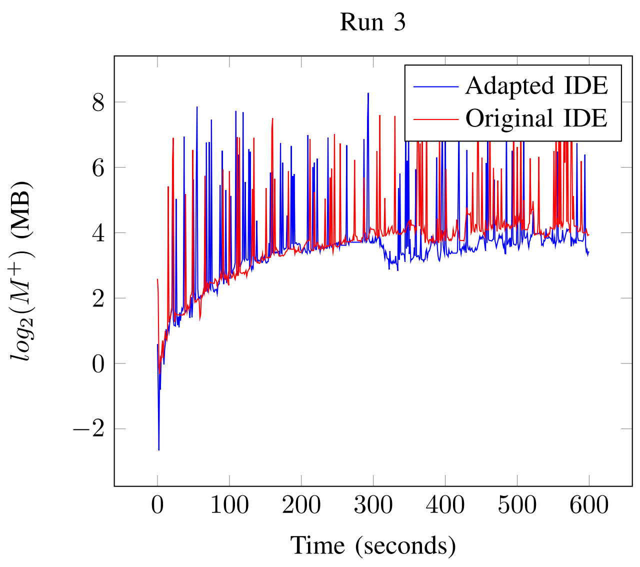

Fig. 5: Run 3 - 10 min

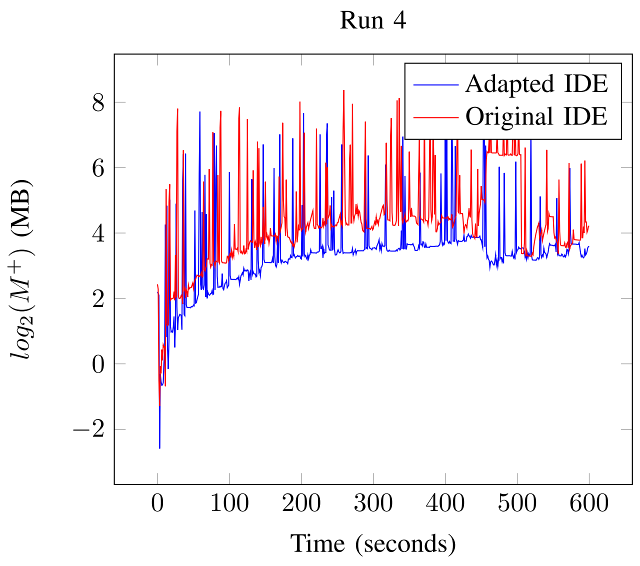

Fig. 6: Run 4 - 10 min

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 [20] 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 [21]. We called Jmap once per second, collecting 300 data points for the 5 minute runs and 600 data points for the 10 minute runs.

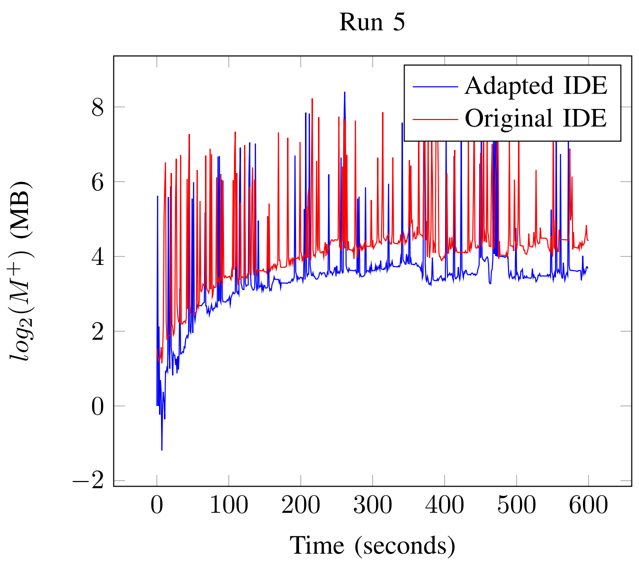

Fig. 7: Run 5 - 10 min

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 I shows mean M^+ for each run plus mean values across all runs. Figures 3-7 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.

B. 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 II.

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

TABLE II: 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

adaptation is probably obtained by 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.

1) Results: Table III 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 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 IV 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 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.

VI. 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.

VII. 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

TABLE III: 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 II

.

	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

TABLE IV: 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

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.

A. 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 [17] 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 [12] select and prioritize tests to use in reduction. One obvious approach is to prioritize tests that tend to reject reductions.

VIII. 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 intervention. 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 [2] program, and the authors would like to thank our collaborators at OSU and Raytheon/BBN.

References

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

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

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

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

[5] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: Delta-debugging, even without bugs," *Journal of Software Testing, Verification, and Reliability*, accepted for publication.

[6] ——, "Cause reduction for quick testing," in *IEEE International Conference on Software Testing, Verification and Validation*, 2014, pp. 243–252.

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

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

[9] G. Karsai and J. Sztipanovits, "A model-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 46–53, May 1999.

[10] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May 1999.

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

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

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

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

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

[16] E. Daka, J. M. Rojas, and G. Fraser, "Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?" in *International Symposium on Software Testing and Analysis*, 2017, to appear.

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

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

[19] "Netbeans IDE bug 45011." [Online]. Available: https://netbeans.org/bugzilla/show_bug.cgi?id=50411

[20] "Linux desktop project." [Online]. Available: https://ldtp.freedesktop.org/wiki/

[21] "Jmap." [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html