hddRASS: Hierarchical Delta Debugging for resource adaptive software system

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

Alex Groce
School of EECS
Oregon State University
Corvallis, Oregon, USA
agroce@gmail.com

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

***Abstract*—Hello world.**

I. Introduction

Modern day software systems are very complex consisting of various resources like libraries, models, operating systems, databases, memory systems, processors drivers, browsers, services, data structures etc. While developing such complex software systems, certain implicit or explicit assumptions are made for the resources that either software is going to use or the software is going to be operated under. For example, a mobile application using location assumes that some sort of location provider services is available via network or GPS or through some other mechanism. Advances in underlying technology, sometimes continuous or sometimes drastic forces software developers to adapt/evolve their software to these underlying changes. Users of such systems have to go through updates in software or sometimes buying new technology to continue to use certain software. A change in a library used by the software sometimes forces developers to refactor or rewrite part of their software which can lead to a multiple cycles of development and testing before the application can be fully adapted to the library changes, a costly, time consuming and error prone process.

Having a mechanism to design and implement resource adaptive real world software system that can adapt to changing environment or changing resources can solve the problem. A lot of work has been lately devoted to building self adaptive software system.s. In their survey of engineering approaches for self adaptive software systems Crupitzer et al discussed different approaches to build self adaptive software systems [1]. In order to design and implement resource adaptive software system for mission critical industry strength software system, DARPA launched Building Resource Adaptive Software System project, BRASS, that employs a new clean slate approach that aims to capture the relationship between computations and the resources they use and provide transformations that enable applications to adapt to resource changes without the need for extensive programmer involvement [2].

As per vision of BRASS, When applications adapt to resources that adaptations are manifested as (1) Restricted functionality (2) Altered functionality and (3) Enhanced functionality. We observed and worked with a team of developers trying to build resource adaptive software system and realized that developers design and implement applications react to these adaptations by reduction or replacements mostly. In reduction, the system adapt to depleted or changing resource need by reducing the application functionality, by not respecting less significant or low priority invariants while still observing significant and high priority invariants. Basically, the application continue to provide some(mostly important) functionality while avoiding exercising some features. In replacement, application adapt similarly but by changing some functionality by equivalent functionality such that resource under stress is less utilized.

Resource adaptive software systems (RASS) are designed and implemented normally for mission critical software systems and normally accompanied by a good test suite that captures the specifications of the system well and does a thorough job in verifying these specifications. There is a normal consensus that whenever system adapts, it may not be able to observe all the invariants all the time, particularly when reduction is the chosen adaptation. Sacrificability of specifications can be captured using a test suite by altering the test suite at test case level, at an invariant level or using some other complex combinations for reduction based adaptations. It can be as simple not exercising all the test case that exercise a single feature or turning off some asserts that verify state change of function call. Now if we have a systematic program reducer, we can reduce the program, component, class or method automatically to adapt to resource need based on the new test suite.

This paper proposes and implements a tool, hddRASS that takes as its input a program and a test suite and via reduction find out the minimal program such that all tests in the test suite will continue to pass. Previously reduction tools have been proposed but they are mostly proposed to keep the failure or bug isolation than to produce a useful version of the program, they do not necessarily fit into or optimized for the proposed adaptation work flow. Delta-Debugging(DD) or Heirarchical Delta Debugging(HDD) are also well known reduction algorithms [3], [4]. Tools like chipperJ [5] and picireny [6] were previous implementations of HDD that apart from reducing test input, can reduce java programs also. We will discuss advantages of our tool against these tools in related work section, most important distinction being its ability to optimally fit into adaptation workflow. The hddRASS, as a stand alone tool,can reduce any java program given a JUnit test suite. The tool is also flexible such that by implementing an interface it can be extended to accommodate any arbitrary test suite. We have successfully used it to reduce real world android application and multiple open source java projects. Though tool was originally designed to build reduced component to aid building resource adaptive software system, it is a general purpose reduction tool that can reduce java programs using test suite.

Our contribution is as follows:

1) We demonstrate that a test suite and reduction tool can be used to automate reduction based adaptation workflow.
2) We implement a stand alone reduction tool , to reduce programs using test suite [1], modified HDD, provide reasons for modifications.
3) We provide some initial evaluation of the tool.

Remaining paper is organized as follows. In section 2 we provide we describe some of the related work and how our tool differs from existing tool. In section 3 we provide background of component based approach of building adaptive software system in practice and need to such a tool. In section 4 we motivate need of such a tool by providing scenarios in practice and motivating examples. In section 5 we provide description of the tool, the theoretical aspects, practical considerations, architecture, algorithm, implementation, usage and challenges. In section 6 we evaluate the tool by applying it on industry application and open source projects. Section 7 contains future work and conclusion.

II. Related work

The current status of work and challenges for self adaptive systems are well summarized in the work of Salehie et. al. [7]. They also summarize some work done in practice by some of the organizations. Kruptizer et. al. [1] discusses different engineering approaches used in building adaptive software systems. The approaches they discuss include but not limited to model-based, architecture-based, reflective, programming paradigms, control theory, service oriented, agent based, formal modeling and verification based, machine learning based and others [1]. Tactical Situational Awareness (TSA) system that we worked with falls somewhere between Model based approach [8] and Architecture based approach [9] to build self adaptive software systems.

Alex or Arpit - if space permits, add something mentioning slicing.

Delta-debugging (DD/dd for short) is an algorithm for reducing the size of failing test cases or test inputs [3]. Hierarchical Delta Debugging [4] or HDD/hdd was proposed to efficiently reduce test inputs that are hierarchical in nature, for example, html inputs, xml inputs and programs. When programs were reduced before using HDD, the reduction criterion were simpler like keeping the crash. Many HDD implementations have been proposed previously, including a very recent one priceley [4], [5], [6]. Early HDD algorithm paper has reducer for C program [4]. CReduce was proposed to reduce C Programs generated to find compiler bugs in C program [10]. First of all our tool differs from those tools in its intended usage. Almost all other tools were designed to reduce test inputs to find the smallest failing test input. Our tool try to reduce program with intention to find a reduced but useful version of the program that does not respect certain specifications as specified in test suite.

Also, all other tools, because they do not need to respect any property that does not contain failure, normally produce outputs that are significantly smaller then original output. Most of the papers boast their reduction factors and compare with other approaches using reduction factors [4], [5], [6]. In order to quickly generate those reduced input, they tend to have bigger chops earlier when they reduce. In our case, a program and its reduced version will have few dissimilarity as the reduced program is expected to be a useful artifact itself, our algorithm (1) starts with the smallest unit possible (sentence, in our case) and (2) Start with leaf nodes and build upwards.

Authors of the most recent tool picinery [6], lamented the fact that no newer HDD tools came out for long years and older tools use very outdated technology. Authors also acknowledge the fact that HDD tools are not used in practice. They proposed a new tool, picinery, where they use homogeneous technology component (every underlying component of their tool uses python) and used modern libraries like ECFG (Extended CFG) instead of CFG(Context Free Grammer). Authors definitely modernize HDD, but did not mention how their tool will encourage widespread industry usage.

chipperJ reduces java programs for automated bug isolation and was successfully applied to reduce complex java programs for bug isolation [5]. chipperJ rely on very naive symptom like null pointer exception or looking for particular string in the output and cannot be used for general purpose standalone tool with modern day software without significant changes.

Most HDD (and DD) implementation assumes ability to provide test results into HDD implementation in the format that the tool wants. That means a practitioner will have to make sure that they write some sort of test script and plug in the results in the format the tool expects. picinery has `test p̈ut your test script here¨` option in command and chipperJ architecture figure has a test script block supplying results to chipperJ. Before any HDD reductions will start, a pre processing step is expected and mostly performed when HDD tools are run for research purpose.

With modern day software system, there is an ecosystem of programming and testing technology. Most of the open source and even industry Java projects have their tests written in JUnit. Same is true with other programming languages like C#, javascript etc. HDD tools fail to leverage this fact leaving some of the work as pre-processing for developers. As we will discuss later, our tool is a complete standalone tool that requires no pre-processing if Java/Android program is used with Junit [11] test suite. Also, as we will discuss in tool section, our tool is easily extendable to other test framework in the ecosystem.

III. Background

To demonstrate the design and implementation of resource adaptive software system, we will use location provider of Tactical Situation Awareness (TSA) as an example, that sends location to people or object carrying the device to some server.

[1]our tools is available at https://github.com/amchristi/hddRASS

Location provider in case of TSA consists of location, image and sometimes streaming video of the location, more complex then normal location providers where location itself is enough. Also location provider of TSA can be used in extreme locations like in remote battlefield, forests or under sea where resource availability is drastically different then the environment in which the TSA and location provider is implemented. Devices that TSA is deployed can have very varying hardware and the environment forces system to use network or GPS for location. The quantity and quality of resources are not known in advance and vary drastically.

Specification of such systems are well documented and verified sometimes using a good test suite. Though, to adapt to extreme resource need some of these specifications are relaxed or compromised at run time in practice to let system function in deteriorated way. For instance, if system sends latitude, longitude, altitude and image as part of location provider, in case of low network bandwidth, system choose not to send the image, at least you know where the person or vehicle carrying the system is. Another example of adaptation can be turning off logging functionality of disk space is really low, helpful in case your disk is inside a satellite where immediate maintenance is not possible.

Component based approach to adaptive software development assumes a software system to be built of loosely connected components that are replaceable, sometimes even at run time [9]. So, when resources are sparse or unavailable making a particular software component unusable in its current state, system tries to swap it with a replaceable component in such a way that component and hence the system will continue to run in the changing and most often deteriorated resource availability giving system a better chance to survive.

As per objective and system evaluation of BRASS, adaptation for resources by a system can manifest itself in various ways [2], the most basic being

- Restricted functionality - Resource based adaptations will not observe certain system specifications, making system limited in its functionality. Just like low network bandwidth will trigger adaptations that will not send images.
- Altered functionality - Resource based adaptations will observer some specifications in altered way then originally intended. In previous example, images will be send periodically while location will be sent at normal rate.
- enhanced functionality - Resource based adaptations will meet all the specifications and may now meet some new specifications that the system was not originally implemented for. These kind of adaptations are not going to part of the discussion of this paper.

IV. Motivation

In this section we motivate the need of a tool that combined with test suite can automatically produce reduced version of the components used as building block of adaptive software system. We also argue that though the tool is designed to aid building/mutating components of resource adaptive software systems, it is general tool that can be used to reduce any java program with respect to a test suite.

As part of the BRASS effort to build industry strength resource adaptive software system that can last 100 years [12], we closely work with a team of developers attempting to do so. We observed how they try build the components with possible adaptations in mind. Though authors acknowledge the lack of empirical analysis partly because unavailability of subjects as rarely systems are implemented as resource adaptive software systems, we observe that developers tend to employ two different natural strategies.

1) Reduction - Developers build reduced version of original components that does not observe certain less critical specifications.
2) Replacement - Developers build components that are altered from original components, new components use different libraries, data structures, hardware, resources etc.

Reduction makes most sense when it is possible to shield the component and in turn system by relaxing some of its specifications either by turning off some features or by not executing certain part of the code such that resource under stress will be less utilized or not utilized at all. Replacement makes most sense when it is possible to come up with completely different components using completely different libraries, hardware, data structures etc such that it will alter the resource usage also making application weather the resource degradation. Reduction associate with Restricted functionality and Replacement with altered functionality objectives mentioned by BRASS objectives as mentioned in background section. For remaining of the discussion we are only going to consider reduction strategy.

Currently both the strategies are executed manually. A developer has to carefully observe the component, observe its resource need, consider the specifications, wrap her head around component interactions in order to come up with correct reductions. Also the reduced components have to be identified, associated with and verified against its resource consumption. They have to be verified against the *new or reduced set of specifications* they are going to observe. All these steps are manual, error prone and tedious. It also triggers few cycles of development and testing.

Is it possible to automate this process?

We hypothesize that most of the mission critical system and industry strength software, specifications are well documented and verified using test suite. So, for rest of the paper we assume availability of a *good test suite*. Specifications that need to be relaxed to allow resource adaptations can be done at test suite. To accommodate certain adaptations, developers will alter the test suite by simply removing some test cases that executes some feature or not executing some invariants of test cases. Some other complex combinations can also be created but for the discussion of our tool, we simply say that a modified test suite will capture the reduced specifications for adaptations.

We propose that if test suite can be modified by developer to reflect relaxed specifications, using the test suite and original program, we can automatically produce reduced component using a reduction tool like HDD. We implemented one such tool and did some preliminary evaluation for the tool.

V. Tool

In this section, we discuss intuitive modification to original HDD algorithm to make it optimally fit adaptation workflow, modified algorithm, architecture, implementation and usage of the tool in detail.

A. coverage and mutation

It is important to note that if the program parts that can be reduced only contains non covered entities, just computing coverage and removing non covered entities will do the job. But even if entities are covered, but does not contribute to the correctness of program as defined by test suite, they qualify as reduction candidates. Though can be optimized by allowing non covered entities to be reduced first, something we are planning to do in future. To provide an example, consider following code that was removed as part of our reduction from URLValidator class.

```
// Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
    public void paintComponent(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }
}
```

B. 1-minimal program

Original delta debugging work defines different notions of minimality for reduction in detail [3]. For our further discussion, we define 1-minimal program or 1-minimal mutant as follows. It is important to note that this is local minimal as DD or HDD are greedy and our algorithm is also greedy [3], [4]. Given a program P and a test suite T, we define 1-minimal (local) program P' of P is the reduced version of P such that all tests continue to pass and the program cannot be reduced further. For component level adaptations, the idea of 1-minimality is applied at component level. The intention of is to output the 1-minimal program for a given program given a test suite such that all tests continue to pass.

C. Intuitive modification to HDD for RASS

Now we describe modifications that we propose to original algorithm and intuition behind the proposals. We need to put on our *Resource Adaptive Software System developer* hat on to understand this modifications.

(1) As the reduced adaptive program is a useful artifact that will be used in place of the original software, we hypothesize that it may not be drastically different then the original program. Reduction may incur many small local changes to software based on the relaxed specifications. As original HDD setups were designed to quickly converge to reduced input that is significantly *smaller* then original input, they tend to make bigger chops earlier in the process. As changes to RASS

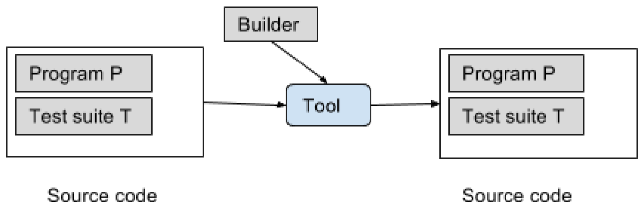

Fig. 1: Overview of the tool

are going to be local and smaller, we propose modification to original algorithm. Our algorithm always start with leaf nodes and it chops smallest possible unit at the beginning (statements for our tool).

(2) Consider any two statement s1 and s2 existing at the same level in AST in the order s1,s2. Our algorithm always delete s2 before s1. Our deletions are always from right to left at same level in AST. (a) This prevents non compilable code and hence costly reverts, use will always be deleted before def. (b) Later read/write will be deleted before earlier read/write. Will this have better chance to keep program correct, for Alex to answer? Reverts will be needed only if any test in test suite failed.

(3) Intuitively, changes for adaptations are going to be local somewhere at deeper depth level in program, we hypothesize that an early exit criterion is possible. We hypothesize that algorithm may not need to reach all the way to root node, It can stop much earlier. Accommodating the early exit possibility provides one more reason to start at leaf nodes first. Early exit criterion can be arbitrarily specified like 1/3rd of the depth of the tree or static or dynamic analysis or some feedback loop can be employed to define more accurate early exist criterion, if such criterion exists. our experiments validates such possibility.

D. Algorithm

The basic algorithm is described below. It still follows all the standard steps of HDD, though it is modified to fit into adaptation workflow better by taking into consideration modification mentioned in previous section.

Algorithm 1 Hierarchical Delta Debugging

```
 1: procedure HDD($inputTree$)
 2:     $level \leftarrow$ HIGESTDEPTHLEVEL($inputTree$)
 3:     $nodes \leftarrow$ TAGNODES($inputTree, level$)
 4:     while nodes are not empty do
 5:         $minConfig \leftarrow$ DDMIN($inputTree$)
 6:         PRUNE($inputTree, level, minConfig$)
 7:         $level = level - 1$
 8:         $nodes \leftarrow$ TAGNODES($inputTree, level$)
 9:     end while
10:     return $level$
11: end procedure
```

E. Architecture and implementation

The basic architecture of the tool from a user's perspective is shown in figure 1. For Java-JUnit combo, user just needs to provide the source code that consists of program and tests for java projects.The only extra information needed is build command - ant,gradle or maven command for modern day java projects. No separate mechanism is needed to feed DD with pass-fail result by setting up a tester, running it and collecting the results.

Figure 2 describes the detailed architecture of the tool. Components in oval gray are input/output, original program P, Test suite T, and builder constitute input and Program P' is output. Components in white rectangle constitute part of the tool. Architecture follows standard HDD/DD procedure of reduction while taking into account the changes proposed in intuitive modification subsection. Program is passed to AST Parser [13], we used JavaPaser for this. We implemented our own HDD to reflect changes mentioned in intuitive modification subsection. Similarly, we have our own implementation of DD for the reasons mentioned in intuitive modification subsection. Apart from implementation/algorithmic changes that we proposed, the major architectural change consists of how test results are being fed to dd. Tools like picireny and chipperJ require some sort of test script and pre-processing for developers, while testing is built in as part of our tool.

F. Usage

Tool takes as its input, a program (or a component if only component needs to be reduced), a test suite or a tester, a build system and produces a *minimally reduced program* such that all test cases continue to pass. Though right now it works as a stand alone tool with JUnit tests only, the tool is very extendable and flexible to accommodate any other testing technologies. Just like JUnitTester in the 2, providing an extra implementation, NUnitTester, will make the tool capable of working with any java projects consisting of NUnit test suite out of the box. Tool can be easily extended to work with other arbitrary tester by implementing a simple interface that consists of only one method. We were able to extend it for TSA system under consideration that has a customized tester written in python by implementing the interface using less then 70 lines of java code. Tool is made available as command line tool with many flexible options giving developers very fine control over how it can be used. Tool can be used at method level, class level, program level or component level giving its user finer control over reduction.

Because the tool is capable of method level reduction, it can be made capable of test case reduction in place of program reduction, as JUnit tests are also methods. When running the tool in test case reduction mode, provide the test case as method. The tool can switch from *reduce program until all tests pass* to *reduce test case until test case continue to fail* mode automatically, making it a test case reducer also like normal hdd.

G. Challenges

The challenges that we faced are very similar to other reduction tools.

1) Non complilable code - Most of other tools discussed in this paper, just revert the code if code is not compilable at any step of reduction. This issue is not present in our implementation by design, we attempt to produce code that compiles only. Though we still compile at the end of each reduction step (just like other tools) in order to make sure that our reductions are at any point of time are compilable indeed. Because of our proactive approach, we are able to avoid reverting, one of the most time consuming step, reverting mostly require writing back java files.
2) Infinite loops - This problem is just not avoidable as reduction can delete loop control statements. We employ very crude strategy of providing some threshold and giving up if the threshold is hit and returning Fail as result of tests.
3) Local minimum - As mentioned earlier, nature of ddmin or hdd just produces local minimum and as our tool is built on hdd, our tool also produces local minimum program.

H. Envisioned usage of tool in RASS work

We envision threefold usage of with respect to resource adaptive software system work.

1) Tool produces useful reduced components, these reduced components can be used readily to swap components in case of resource degradation.
2) Different techniques like static analysis, dynamic analysis etc are being used to map resources with computation. Delta of original components and reduce components can be the extra information that will help the process.
3) If reduced components are not readily useful, it will at least serve as a starting point for developers. Relaxing specifications using test suite and generating and observing reduce components will help them to quickly discover, how specification changes map to computation.

We are successfully able to reduce programs for TSA project, though the state of the current program and its test suite is such that we are unable to verify accuracy and usefulness of our reductions at this point of time.

VI. Evaluation

Alex makes an argument why we are unable to run it against BRASS project. In order to evaluate our tool for its final purpose, we need to get a real world system, ask developers or testers to relax some of the low priority specifications and see what kind of components our tool can generate. We want to demonstrate the following about our tool.

1) Tool is a general purpose too, it can be used with any arbitrary java program with JUnit test suite out of box.
2) If specification are relaxed, Tool is indeed able to reduce programs

To achieve this, We apply our tool on apache commons validator project. It takes very long for tool (as this is true for all HDD setups) to reduce full program, in order to evaluate

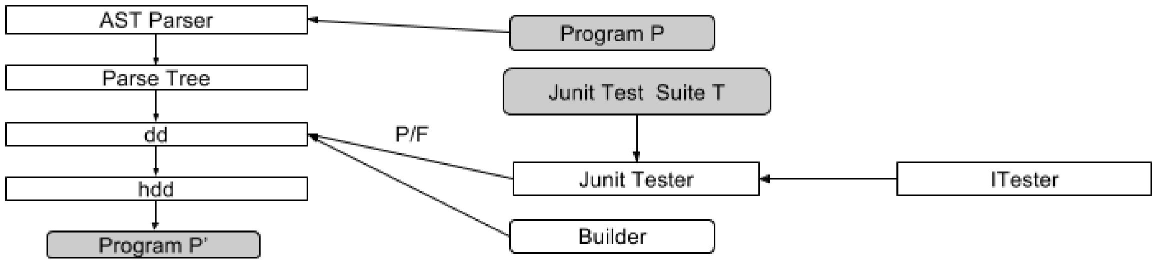

Fig. 2: Architecture of the tool

its usage, we apply it in class level mode, we are interested in reducing one class at a time. Apache command project has one test class associated with each class and we chose to run only tests associated with the corresponding test class. For example, URLValidator class of apache commons validator consists of 21 class that tests specification of URLValidator.

We chose to relax 10% 20% percent and 30% percent specification of of each of the test class by turning off the tests. In order to prevent bias by a particular test method turned off, we repeat the process 3 times each time, each time randomly chosen tests turned off. We measure the reduction in terms of number of nodes deleted for abstract syntax tree. We also measured at what depth level each nodes are. We present our results in the following table.

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. [Online]. Available: http://dx.doi.org/10.1016/j.pmcj.2014.09.009

[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. [Online]. Available: http://doi.acm.org/10.1145/2853073.2853081

[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. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[4] G. Misherghi and Z. Su, "Hdd: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 142–151. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134307

[5] C. D. Sterling and R. A. Olsson, "Automated bug isolation via program chipping," in *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, ser. AADEBUG'05. New York, NY, USA: ACM, 2005, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/1085130.1085134

[6] R. Hodován and A. Kiss, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, ser. A-TEST 2016. New York, NY, USA: ACM, 2016, pp. 31–37. [Online]. Available: http://doi.acm.org/10.1145/2994291.2994296

[7] 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. [Online]. Available: http://doi.acm.org/10.1145/1516533.1516538

[8] 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. [Online]. Available: http://dx.doi.org/10.1109/5254.769884

[9] 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. [Online]. Available: http://dx.doi.org/10.1109/5254.769885

[10] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254104

[11] "Junit home page." [Online]. Available: https://en.wikipedia.org/w/index.php?title=LaTeX&oldid=413720397

[12] "Darpa news article." [Online]. Available: http://www.darpa.mil/news-events/2015-04-08

[13] "Java parser." [Online]. Available: http://javaparser.org/