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 [**?**]. 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 [**?**].

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 mostly 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 among the developer that whenever system adapts and reduction is the chosen adaptations, it may not be able to observe all the invariants all the time. Sacrificability of specifications can be captured using a test suite by extending 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 extended test suite.

This paper proposes a methodology to automatically create a reduced variant(s) of the program based on sacrificability of specification captured by extended test suite. Test suite can be extended by simply labeling/annotating the test cases or using some complex mechanism. Developers can manually annotate test cases or assertions one by one or can employ some automatic mechanism to achieve these annotations. It achieves accurate program reduction by using hddRASS, a new tool that we are proposing, that combines HDD with statement deletion mutation, to automatically create this reduced variant of the program. The basic adaptation workflow is as follows. (1) Developer annotate the test suite of the program. (2) Program is deployed with its annotated test suite and newly proposed tool, hddRASS . (3) When resource adaptation is triggered, using annotated test suite and current program, hddRASS finds out minimal working version of the program.

This paper proposes and implements a tool, hddRASS that takes as its input a program and an extended test suite

and via reduction find out the minimal program such that all tests at certain level in extended test suite will continue to pass. The methodology and tool proposed as part of our work can be considered a step in the right direction in ensuring system reliability of mission critical system that works in an environment where resource availability is unpredictable.

Our contribution is as follows:

1) We proposes a workflow that employs extended test suite, a newly developed reduction tool and existing program to automatically build adaptive program.
2) We propose test case labeling/annotations as a way to capture sacrificability of specifications.
3) We implement a stand alone reduction tool hddRASS to build adapted version of the program using extended test suite. [1].
4) We demonstrate that hddRASS is highly applicable, accurate and works out of the box for most of the java program setups
5) Empirical evaluation of reductions achieved using extended test suite using real world large open source programs.
6) Based of the empirical evaluation of reduction, we propose a simple and cheap heuristic to improve efficiency of automatic reduction.

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. [**?**]. They also summarize some work done in practice by some of the organizations. Kruptizer et. al. [**?**] 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 [**?**]. Tactical Situational Awareness (TSA) system that we worked with falls somewhere between Model based approach [**?**] and Architecture based approach [**?**] 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 [**?**]. Hierarchical Delta Debugging [**?**] 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 earlier using HDD, the reduction criterion were simpler like keeping the crash. Many HDD implementations have been proposed previously, including a very recent one priceley [**?**], [**?**], [**?**]. Early HDD algorithm paper has reducer for C program [**?**]. CReduce was proposed to reduce C Programs generated to find compiler bugs in C program [**?**]. 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. Most of the previous program reducers find minimal configuration using AST (like we do), most of them does not rely on statement deletion as their primary reduction step [**?**], [**?**]

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 [**?**], [**?**], [**?**]. 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 [**?**], 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.

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 ̈put 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, hddRASS is a complete standalone tool that requires no pre-processing if Java/Android program is used with Junit [**?**] test suite. Also, as we will discuss in tool section, our tool is easily extendable to other test framework in the ecosystem.

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

III. Background and Motivation

We are working with a group of developers trying to build real world resource adaptive software system, the system that is being built is known as Tactical Situational Awareness System also known as SA - Situational Awareness. To demonstrate the design and implementation of resource adaptive software system, we will use location provider of SA as an example, that sends location to people or object carrying the device to some server. Location provider in case of SA 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 SA 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 SA and location provider is implemented. Devices that SA is deployed on can have very varying hardware and depending upon the the environment system is forced 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 if 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 [**?**]. 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 environment, giving system a better chance to survive, making system more reliable.

As per objective and system evaluation of BRASS, adaptation for resources by a system can manifest itself in various ways [**?**], 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.

As part of the BRASS effort to build industry strength resource adaptive software system that can last 100 years [**?**], 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? The methodology, test suite extension and hddRASS discussed in further sections are motivated with the intentions to (1) help developers to quickly and automatically build reduced components that can fit into resource adaptive software when reduction is the chosen adaptation. (2) If runtime adaptations are triggered and already built component is not found, system can build a new reduced component at run time.

IV. Motivation

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. Methodology

This section discusses the methodology to automatically build reduced components for resource adaptive software using extended test suite and hddRASSÀs we discussed previously either adaptations are pre computed by developers or it can trigger at run time if a pre computed adaptations are not available. Developers based on the context knowledge of system and its environment, chooses pre computed adaptations or run time adaptations. Providing multiple reduced components for each of the system component can make system bloat in its size and may create performance issues while system is still functioning in normal environment in some cases. Run time adaptations will pause system until adaptations are completed and then swap it with original component. Pre computed adaptations are useful if adaptations are needed frequently and providing pre computed adaptations does not create serious performance issue. Run time adaptations are useful if adaptations are needed rarely and can be triggered quickly or pre computed adaptations are not available.

A. Labeling test cases

Extending test cases is nothing but to prioritize test cases as per its importance and embed this information as part of test case itself. For Junit test cases, this is achieved by providing annotation for each of the test case, we call this labeling of test cases. Test cases are labeled from 0 to n, test cases with label 0 being the most important test cases. If any of these tests fail that means system is not usable. Test cases with label n are the least important and the specifications observed by these test cases can be sacrificed without affecting system seriously. Test cases with label closer to 0 are more important then test cases with label closer to n. This process divides test cases in n different level. Test case labeling can be manual or automatic. In manual labeling, developer will go through the tedious task of going through each test case pertaining to a component and label them. Work on automatic labeling is under progress hence for rest of the paper, we assume manual labeling. Also, labeling cane be applied at coarser or finer levels giving us tighter or loose reductions. For Junit test cases, labeling can be applied at test class level or at each assertion level in turn making life of developer easy or difficult.

B. Workflow for pre computed adaptations

1) Developer extend the test suite by labeling each test cases or each assertions.
2) For each of the level, starting from level n we do the following.
3) If current level is x, remove all the test cases whose label is k, k x.
4) Provide remaining test suite and program or component as input to hddRASSàbcd
5) The tool outputs minimal program/component for level x.

C. Workflow for run time adaptations

1) Developer extend the test suite by labeling each test cases or each assertions.
2) Run time adaptations are triggered and we assume reduction is chosen adaptation.
3) Start with level n and do the following
4) remove all tests labeled as k where k is greater then n.
5) Provide remaining test suite and program as input ot hddRASS
6) hddRASS outputs minimal program. check it against resource contraint.
7) If resource constraint is not violatedm keep the reduced program.
8) If it is violated, continue by reducing by one level.
9) If level 0 is reached, stop and output that system cannot be used under given resource constraint.

D. Choosing only subset of tests

For object oriented components, components are reduced by reducing all the classes and classes are reduced by reducing all the methods. When a class is being reduced, as we will discuss in Tool section, after each unit reduction step, system needs to be compiled and tests need to be executed. Running all the tests of a component when we just reduce a single class can be prohibitively time consuming as this step has to be repeated large number of times. To speed up the process, we only choose a subset of tests from all the tests such that only most relevant tests for the class will be selected. For instance, for Apache commons validator project, changing specification of URLValidator does not affect behavior of CreditCardValidator or CalendarValidator and hence all or some tests testing CreditCardValidator or CalendarValidator need not to run when reducing URLValidator. We find similar situation while working with TSA system also. To accommodate the situation and to allow quick convergence, hddRASS allows its user to specify set of tests out of whole test suite at test class level. hddRASS will only run those tests during the reduction process, saving significant amount of time and resources. Selecting tests is not possible when run time adaptations are triggered on a deployed system.In their cloud refactoring work, kwon and Tilervich proposed a way to choose *corresponding methods/classes* by a process that they call profile based recommendations that rely on static analysis and dynamic traces. Based on that, they define two terms CC - Class connection and CR - Class Relation and provide recommendations based on that. Value of CC and CR can be from 1 to 0, with 1 means most tight connection or strong relation and value 0 means no connection or no relation. We also do static and dynamic analysis of our program and tests and recommend tests for a class based on Kwon's method but employing much simpler formula. For simplicity, we select a test case only if CC and CR both are 1. Allowing CC and CR thresholds to be less then 1 will

select more tests as *corresponding tests*. This will increase the time required to converge the reduced class and decreases the amount of code reduced as the class is tested against more specifications. CC and CR value being 1, if simply put in words means, test case directly calls a method of the class in code and that method was indeed executed during the execution of the tests. CC and CR values should be left as configurable parameters that can be adjusted based on the context.

VI. Tool

In this section, we discuss intuition behind combining HDD with statement deletion mutation to develop hddRASS such that it fits into adaptation workflow, modified HDD algorithm, architecture, implementation and usage of the tool in detail.

A. coverage and dead code removal

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 hddRASS 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.

Listing 1: URLValidator reduced code

```
if (!isValidAuthority(authority)) {
  return false;
}
```

The code was reduced when we relaxed specifications by randomly turning off 2 tests. The code was reduced though it was *covered 42018 times during execution*.

B. 1-minimal program

Original delta debugging work defines different notions of minimality for reduction in detail [**?**]. 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 [**?**], [**?**]. 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 hddRASS is to output the 1-minimal program for a given program given a test suite such that all tests continue to pass.

C. Statement Deletion mutation - SDL

If resource adaptation is triggered because of resource constraint violation and if a reduced program/component is generated, as the reduced program is still a useful artifact that will continue to function as it is mostly, we hypothesize that small local changes are applied to the program. The newly created program is a perturbation of the original program and mutation captures perturbations well. In traditional mutation testing, purpose of mutant is to act as an artificial bug while in our case mutant is a useful perturbed version of original program. Statement Deletion mutation operator was proposed by xyz and during empirical analysis conducted by abc it was found to be very effective and it subsumed other mutation operator significantly. When HDD is traditionally applied on programs, it relies on tree structure of the program like AST, CFG or ECFG etc. For example, picireny relied on ECFG of programs. Its reduction run even deeper then program statements, it even reduces expressions. Our approach does not go beyond SDL because (1) SDL operator was found to be subsuming other mutation operator that are used in expressions (2) As reduced program still needs to be working as mostly correct program, we expect simple removals. (3) Minimally reducing expressions, conditions etc further may be very time consuming and hence run time adaptations may not converge quickly.

D. Intuitive modification to HDD for RASS

Now we describe modifications that we propose to original algorithm and intuition behind the proposals. Later during evaluation we find some evidence for the intuition. We need to put on our *Resource Adaptive Software System developer* hat on to understand this modifications. As the reduced adaptive program is a useful artifact that will be used in place of the original software, we hypothesize that (a) it may not be drastically different then the original program. (b) Any reduction that does not compile is useless.

(1) 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 are going to be local and smaller, we propose modification to original algorithm. Our algorithm always start with leaf nodes at highest depth level 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 some posibilities of 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) Consider any two statements s1 and s2 existing at different level in AST such that s1 is at higher level. As our algorithm starts from leaf node and moves upward, s2 will be deleted before s1 ensuring that if def and use are at different levels, use will be deleted before def, again reducing possibility of non compilable code.

E. 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 ← HIGESTDEPTHLEVEL(inputTree)
3:     nodes ← TAGNODES(inputTree, level)
4:     while nodes are not empty do
5:         minConfig ← DDMIN(inputTree)
6:         PRUNE(inputTree, level, minConfig)
7:         level = level − 1
8:         nodes ← TAGNODES(inputTree, level)
9:     end while
10:    return level
11: end procedure
```

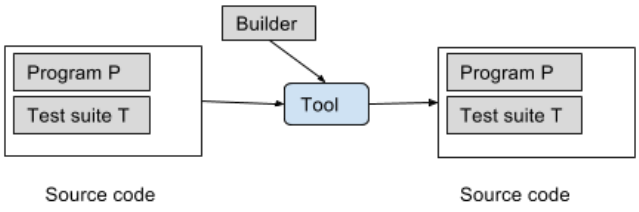

Fig. 1: Overview of the tool

F. Architecture and implementation

The basic architecture of the tool from a user's perspective is shown in figure **??**. For Java-JUnit combo, user just needs to provide the source code that consists of program and labeled 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 **??** 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 [**?**], 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.

G. 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 **??**, 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.

H. 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.

VII. Evaluation methodology

acbd

In order to measure the applicability and accuracy of hddRASS as well as evaluate the reductions produced by our methodology, we chose 21 java classes from 7 projects of varying size, purpose and complexity. From each of 7 open source java project, we chose 3 classes randomly. These classes have average LOC (non empty, non comment) 398 lines, with maximum being 1302 and minimum being 64. These classes contains on average 24 methods, with max and min being 64 and 1. Information related to all 21 classes are shown in Table. For experiment purpose, we choose class as a component.

In real world scenario, developers will be well aware of their system and tests and will label the tests accurately based on their importance. They will choose the tests pertaining to certain functionality if they want to reduce component associated with that functionality. For experimental purpose, to demonstrate use of our tool, we first selected test cases based on its CC and CR values as described in section subsection, CC and CR values should be 1. It prevents all tests from running while a class is being reduced and converges faster.

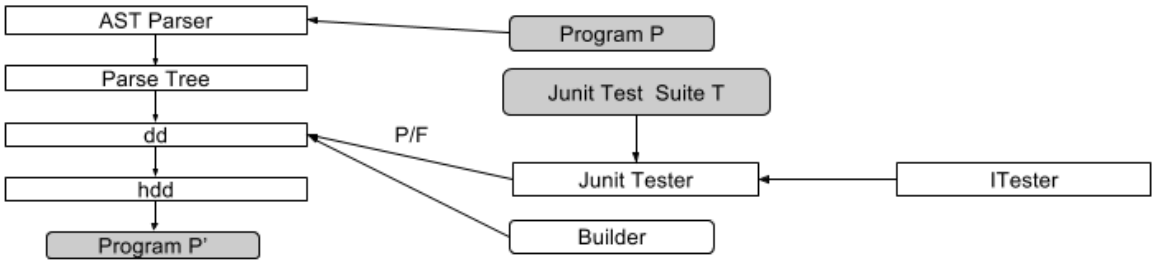

Fig. 2: Architecture of the tool

Project	Class	LOC	Methods	Tests	Statements
CruiseControl	AntBuilder	499	33	22	xx
CruiseControl	Schedule	383	30	18	xx
CruiseControl	Project	xx	xx	xx	xx
Ant	Available	6.45	25	4.54	23
Ant	Copy	6.45	25	4.54	23
Ant	FixCRLF	6.45	25	4.54	23
Validator	UrlValidator	6.45	25	4.54	23
Validator	RegexValidator	6.45	25	4.54	23
Validator	DomainValdiator	6.45	25	4.54	23
Jexl3	Engine	6.45	25	4.54	23
Jexl3	JexlArithmetic	6.45	25	4.54	23
Jexl3	JexlEvalContext	6.45	25	4.54	23
Cli	DafaultParser	6.45	25	4.54	23
Cli	GnuParser	6.45	25	4.54	23
Cli	PosixParser	6.45	25	4.54	23
Jena	LocationMapper	6.45	25	4.54	23
Jena	OntlClassImpl	6.45	25	4.54	23
Jena	OntlClassImpl	6.45	25	4.54	23
Text	ExtendedMessageFormat	6.45	25	4.54	23
Text	ExtendedMessageFormat	6.45	25	4.54	23
Text	ExtendedMessageFormat	6.45	25	4.54	23

TABLE I: Reduction size for subject classes

The process gave us certain number of tests to consider for each of the class chosen from reduction. The process gave us close to 30 tests per class on average, while maximum tests selected was 58 and minimum was 7. for further discussion, we call these tests *selected tests*.

We then did test case labeling using the following process. We allowed 3 labels 0,1 and 2, 0 being most important test cases that determines minimum allowable system. System generated by label 1 and 2 are the reduced systems generated based on adaptation, System generated by removing tests with label 2 is the system closest to original system. It is important to note that when we remove tests at label x, we also remove any tests at label x+1, as label x test cases are more important then label x test cases as per our design. We iterate through the *selected test cases* and label each test case with 80% probability of being label as 0, 10% probability of being label as 1 and 10% probability of being label as 2. Default label of test is 0, so if a lable is not present, it is considered 0. The process keeps 80% system as minimum allowable system while still giving us 2 reduced versions. If during the labeling process, if all the 3 labels don't appear, we throw that data point out and re generate the data. In real world, developer may choose not to label any test case with lable 1 or lable 2 test cases. For our experiment purpose, we want to have all the labels and 2 extra reduced versions to be generated for each class and hence the bias.

Consider a component C with test suite T. Initially no tests with any test labels are removed so we call this test suite T_∞ and corresponding component C_∞. We call a test suite T_n if it is generated by removing all the tests whose labels are n and above. We call reduced version generated by test suite T_n as component C_n. The way labeling and corresponding test removal is defined, it is important to note that $T_\infty \supseteq$ .... $T_{n+1} \supseteq T_n \supseteq T_{n-1}$ ..... $\supseteq T_1$ and C_∞ ..... $\supseteq C_{n+1} \supseteq C_n \supseteq C_{n-1}$ ..... $\supseteq C_1$. T_0 and P_0 are not possible because they violate minimal allowable functionality as defined by label 0.

In order to prevent bias in test case labeling (an important test labeled as 2, may remove too much functionality), we repeat the process 10 times, allowing random labeling each time. For each of such labeling, we reduce the class, giving us 10 versions for label 1 and 10 versions for label 2. We repeat the process for each of the 21 classes giving us in total 420

reduced versions, 210 for label 1 and 210 for label 2.

We also want to evaluate and compare reduced versions of the class. In order to evaluate reductions, we measure (1) number of statements removed (2) highest level from leaf level the statement is removed in AST of class. If reduction is higher, large number of statements are removed, and reduced version differ significantly from its original version. If average highest level from leaf level is small, mostly simple statements are removed while highest level from leaf level is big, blocks are removed.

In order to measure accuracy of the tool, we randomly chose 20 reductions out of 420 reductions generated. For each of such reduction; using test case labeling and *selected test cases* we manually generated 1-minimal class. As we are not following any systematic algorithm like hdd to do it automatically, we produced global minimal class. We measure accuracy in terms of statement reduction. For each reduction produced by hddRASS and hand, we define following terms.

- *TruePositive* : a statement is true positive if it is removed in both hand reduction and reduction by hddRASS
- *FalsePositive* : a statement is reduced by hddRASS while it is not reduced in hand reduction, hddRASS incorrectly removed a statement.
- *FalseNegative* : a statement is reduced in hand reduction but it is not reduced by hddRASS hddRASS missed the reduction.

VIII. Results

In this section we want to (1) Demonstrate applicability and accuracy of our tool. (2) Study reduced version generated by our methodology. We ask following Research Questions.

- **RQ1:** How applicable is hddRASS ?
- **RQ2:** How accurate is reductions produced by hddRASS ?
- **RQ3:** What are the size of the reductions produced by our methodology?
- **RQ4:** Does reductions occur in small blocks or bigger blocks?

A. RQ1 Applicability

hddRASS is complete in Java 7 statements specified in japa.parser.ast.stmt, as long as java parser can parse a program, hddRASS can reduce it. As hddRASS is going to be used my developers in practice (at least by the developers of TSA), we want it to have high applicability and hence we are maintaining it well. Because it is complete in statements, we expect it to be highly applicable. During our experiments, we run into issues where hddRASS threw exception, but we continued to fix it along the way. For the current 21 classes, consisting of total 6781 lines of non comment, non empty code and 408 methods, hddRASS is 100% applicable. Among 22 distinct statement types captured by java parser, 21 of them were encountered at least 1 time during our reduction of 21 classes, again demonstrating applicability of the tool. If program uses java 8 features like functional programming using stream, hddRASS is unable to reduce statements with this feature and currently it ignores them, though it will continue to reduce rest of the program as it is. We are planning to address this in future.

B. RQ2 Accuracy

Two factors contribute to inaccuracy of the reductions. (1) Error in the implementation of the tool, this can be fixed (2) As hdd and dd generates local minimum reductions, reduced 1-minimal program is only local minimal and not global minimal. Rarely any reduction tool measures their inaccuracy against global minimal because local minimum is part of definition of dd. We want to consider local minimum as a contributing factor in inaccuracy. Any missed reductions produces bigger program then the true minimum program. We measure accuracy by choosing 20 random reductions and comparing it against reductions generated by hand. We calculate precision and recall using the standard definitions:

$$precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

We found hddRASS has precision of 100% and recall of 94%. This means, all the reductions applied by hddRASS were correct but hddRASS missed 6% of potential reductions. (Alex, Rahul: This numbers are dummy, will be updated soon.)

C. RQ3 Size Of Reduction

Remaining two subsections, we discussed quality of the reductions. We measure reduction in terms of number of statements deleted (both simple and block) by the method and the tool. We only did some preliminary analysis of reductions. 420 reductions across 21 classes consisting of 210 reductions at label 2 and 210 reductions at label 1 cannot be considered as data coming from a random source and hence cannot be used for thorough empirical analysis

Table demonstrate class,number of selected tests,Average reduction for label 2 and label 1, Average number of tests removed with label 2 and label 1 ,maximum number of reductions for label 1 and label 2, standard deviation in reduction for label 1 and label 2, % reduction for label 1 and label 2. It is important to note that when tests for label 1 are removed, we also remove tests for label 2 and hence $T_2 \supset T_1$ and hence $C_2 \supseteq C_1$. By design of experiment, $T_2 \neq T_1$. Reductions are many to one and hence $C_2 = C_1$ is possible. In order to verify this, we chose 20 reductions out of 420 reductions such that 10 reductions are at label 1 and remaining 10 reductions are *corresponding label 2 reductions*. We have 10 pairs of Label 1 and Lable 2 reductions and corresponding test cases. We found above equalities to hold 100% of the time. Also though we are preemptively removing one test at least every time, it is quite possible that class does not reduce at all. Out of 420 reductions we have observed XX number of reductions that does not remove any code. We also plot all 20 reductions for 8 of the classes with their test removals in graph.

Average tests removed For Label 2, across all reductions runs are YY and for Label 2 is ZZ. Average reduction for label 1 across all 210 instances are 6.31 statements. Out of

	Label 2					Label 1				
Class	Tests Removed	Reduction	Max	% Reduction	% Max	Tests removed	Reduction	Max	% Reduction	% Max
AntBuilder	3	6.45	25	4.54	17.48	4.4	9.53	30	6.73	20.97
UrlValidator	2.8	7.18	14	8.75	17.07	6.1	13.54	25	16.51	30.48
DomainValidator	2.6	9.45	13	12.77	17.56	5.4	14.8	20	20	27.02
RegexValidator	1.4	2.4	6	6	15	2	3.6	9	9	22.5
Engine	5.3	0	0	0	0	7.9	0	0	0	0
JexlArithmetic	4.2	1.2	4	0.41	1.38	7	4.4	18	1.52	6.22
JexlEvalContext	3.9	7.1	8	24.48	27.58	8.4	7.4	11	25.51	37.93
Schedule	2.7	0.8	2	0.62	1.57	3.5	1	4	0.78	3.14
Project	3	11.27	114	3.87	39.17	6.1	43.27	118	14.86	40.54
Available	34.1	26.05	82	24	9	7.14	23.2	41	17.44	13.82
FixCRLF	3.1	10.1	15	16.55	24.59	6.4	12.3	20	20.16	32.78
Copy						0	0	0	0	0
GnuParser	4	2.2	11	9.56	47.82	4	2.2	11	9.56	47.82
DefaultParser										
PosixParser										
LocationMapper	1.6	11.66	14	8.44	10.14	2.83	12.66	28	9.17	20.28
OntlClassImpl										
ExtendedMessageFormat										
AVG		6.31	19.21	7.86			10.82	24	10	

TABLE II: Reduction size for subject classes

21 classes considered, only for 2 classes average reduction is above 10 statements. Percentage reduction for Label 2 across all 210 instances is 7.86%. Out of 21 classes considered, only for 4 classes, percentage reduction is above 10%.

Average tests removed For Label 2, across all reductions runs are YY and for Label 1 is ZZ. Average reduction for label 2 across all 210 instances are 10.82 statements. Out of 21 classes considered, only for 5 classes average reduction is above 10 statements. Percentage reduction for Label 1 across all 210 instances is 10%. Out of 21 classes considered, only for 5 classes, percentage reduction is above 10%.

Should we display maximum numbers and minimum numbers and discuss it?

Out of all 420 instances of reduction, only 62 times more then 10 statements were removed, while only 32 times more then 15 statements are removed.

From above analysis, it is clear that reductions are small, not many statements are reduced if program is reduced after throwing 10% or 20% test cases.

D. RQ4 Max level from leaf node

In case of statement deletion using class, all the simple statements are at leaf level. As you move above the leaf level, blocks statements are encountered. As you move further above, complex blocks statements are encountered, that consists other block statements. We want to know what kind of statements are removed? Table demonstrate class,number of selected tests,highest level from leaf level where statement removal occur for label 2 and label 1, Max for the same for label 2 and label 1 ,% for the same for label 2 and label 1.

On overage for level 2 across all 210 reductions, highest level was 1.07. For level 2, across 21 classes, average highest removal level was 2 only once. *Maximum of highest removal level* was 2.38 across all 210 reductions. For 21 classes, *Maximum of highest removal level* was 3 or above only for 5 classes and it was 4 or above for only 1 class.

On overage for level 1 across all 210 reductions, highest level was 1.62. For level 1, across 21 classes, average highest removal level was 2 only 5 times and above 3 only once.*Maximum of highest removal level* was on average 3 across all 210 reductions. For 21 classes, *Maximum of highest removal level* was 3 or above for 9 classes and it was 4 or above for only 2 class. So for 21 classes into consideration, even the *Maximum of highest removal level* for 16 classes was 3.

As far as removal level are concerned. across all 420 reductions, removal at level 1 happened xx number of times, removal at level 2 happened at yy number of times, removal at level 3 happened at zz number of times and removal at level 4 or above happened only aa number of times.

It is clear from above discussion that simple statements are simple blocks have best chance of being removed.

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

	Label 2		Label 1	
Class	Reduction	Max	Reduction	Max
AntBuilder	0.9	3	0.9	3
UrlValidator	1.6	3	2.2	3
DomainValidator	1.3	3	1.85	3
RegexValidator	1.6	3	1.9	3
Engine	0	0	0	0
JexlArithmetic	0.6	error	2	error
JexlEvalContext	1	1	1	1
Schedule	0.1	2	0.2	4
Project	0.6	6	2.4	6
Available	0.8	2	2.6	3
FixCRLF	1.6	2	2	3
Copy	error	error	error	error
GnuParser	error	error	error	error
DefaultParser	error	error	error	error
PosixParser	error	error	error	error
LocationMapper	1.83	2	2	4
OntlClassImpl	2	2.5	2	3
ExtendedMessageFormat				
AVG	1.07	2.38	1.62	3.0

TABLE III: Reduction size for subject classes

[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/