Automatic Debug Down to the Line of Code

Daniel Hansson, Patrik Granath
Verifyter AB
Ideon Science Park
Lund, Sweden

***Abstract*— Automating the debug of regression test failures can be done using either the continuous integration methodology [1] or an automatic debug tool for regression tests [2]. In both cases the goal is to identify the bad commit. However, for large commits it would be even better if we could identify the exact bad lines of code within the commit.**

There is a methodology called delta debugging [3] that may be used to address this problem, but it has issues. In this paper we show how to overcome these challenges.

Keywords—regression testing; automatic debug; delta debugging; continuous integration

I. Introduction

A. Debugging Regression Failures

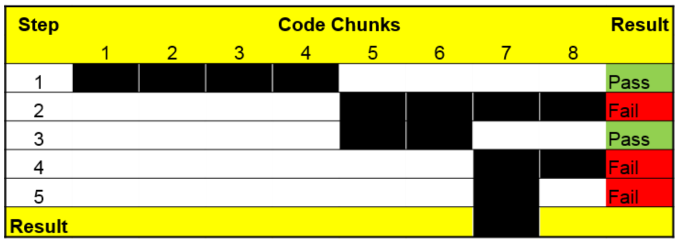
During product development of ASIC's or software, new bugs are continuously introduced by mistake causing the quality of the product to deteriorate. These bugs are called regression bugs. Regression bugs are captured by running regular test runs, so called regression tests, which typically are RTL simulations that are run once or several times per day.

Automatically debugging the regression failures can be done with the continuous integration methodology [1]. This popular approach to regression testing consists of testing every commit to the revision control system with a short non-random test suite. The idea is that if this short test suite fails then we know which commit caused the problem, and the committer can be automatically notified about his or her mistake. For larger test suites and for random tests it is possible to use tools such as PinDown[2]. In both cases the goal is to identify the bad commit.

For larger commits it would be useful to know the exact lines that caused a test to fail, but as commit sizes vary, it would also be interesting to know how many of the commits that would benefit from line granularity.

B. Delta Debugging

Delta Debugging [3] is a methodology where different combinations of code chunks are tested together in order to narrow down a problem. The advantage is that it can find the exact line of code, but only under certain conditions.



Step	Code Chunks								Result
	1	2	3	4	5	6	7	8	
1	⬛	⬛	⬛	⬛					Pass
2					⬛	⬛	⬛	⬛	Fail
3					⬛	⬛			Pass
4							⬛	⬛	Fail
5							⬛		Fail
Result							⬛		

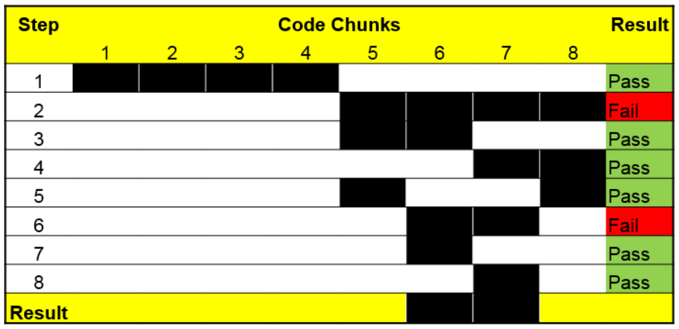
Figure 1. Delta Debugging Example with 8 code chunks. Chunk no 7 contains the bug.

Figure 1 contains an example of delta debugging. The starting point is two versions of the code, a version that passes the tests and a newer version which fails the same tests due to a regression bug having been introduced. Comparing the two versions of code we see that there are 8 chunks of code that are different in this example. The purpose of delta debugging is to identify which chunk or chunks that contain the regression bug. This is done by splitting the chunks in sub-groups and the re-run the failing tests on each sub-group in order to find the minimum number of chunks which still reproduce the failures.

In step 1 the chunks 1-4 are tested but they fail to reproduce the failure as the test is passing. In step 2 the failures are reproduced on chunks 5-8, which means the algorithm has now successfully narrowed down the bug to one of these 4 chunks. When such a milestone is reached the algorithm splits the remaining 4 chunks into 2 sub-groups and continue the testing. In step 3 chunks 5-6 fail to reproduce the failures but chunks 7-8 successfully reproduces the failure. Splitting up chunks 7 and 8 the algorithm achieves a result in step 5 where it manages to narrow down the problem to chunk no 7.

Delta debugging can also manage the situation when a bug is introduced in two chunks. Figure 2 shows such an example where a regression bug only causes tests to fail if both chunk 6 and 7 are present at the same time. The delta debug algorithm starts in the same way as in Figure 1, but after step 4 the two examples deviate.

In Figure 2 the algorithm does not succeed in narrowing down the problem further in step 3 and 4. Consequently it groups the remaining chunks differently and tries again in step 5 and 6. The latter combination manages to reproduce the failure as it contains both the chunks 6 and 7. The algorithm continues in step 7 and 8 and tries to narrow down the issue further, but fails, and the final result is that the bug is contained within chunks 6 and 7.

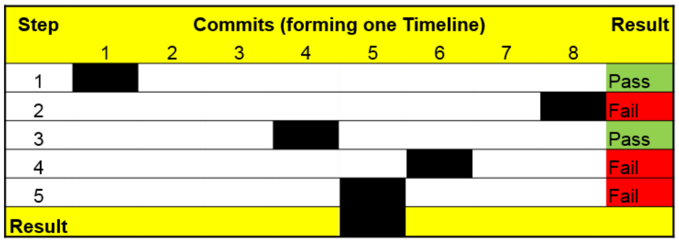


Step	Code Chunks								Result
	1	2	3	4	5	6	7	8	
1	■	■	■	■					Pass
2					■	■	■	■	Fail
3					■	■			Pass
4							■	■	Pass
5					■			■	Pass
6						■	■		Fail
7						■			Pass
8							■		Pass
Result						■	■		

Figure 2. Delta Debugging Example with 8 code chunks. Chunk No 6 and 7 are faulty.

C. *Binary Search*

Another approach to narrow down problems is to use binary search along the timeline of the commits to the revision control system [4].



Step	Commits (forming one Timeline)								Result
	1	2	3	4	5	6	7	8	
1	■								Pass
2								■	Fail
3				■					Pass
4						■			Fail
5					■				Fail
Result					■				

Figure 3. Binary Search Example with 8 commits to the revision control system, forming one timeline. Commit no 5 is faulty.

Figure 3 provides an example of binary search across the timeline that the 8 commits to the revision control system is forming. A test passes on the older commit no 1 (step 1) and fails on the newer commit no 8 (step 2). The binary search algorithm tests commit no 4 in step 3 as it is roughly in the middle and finds that it is passing. In step 4 commit no 6 is tested as it is in the middle of commit no 4 (where the test passed) and commit no 8 (where the test failed). As it fails the algorithm tests commit no 5 in the last step and finds that it fails, which is the end result.

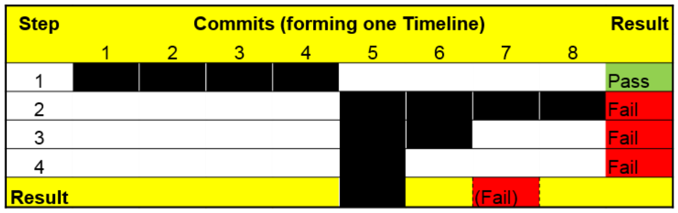
II. COMPARING DELTA DEBUGGING WITH BINARY SEARCH

A. Is Delta Debugging better than Binary Search?

In the paper about Delta Debugging [3] the algorithm is presented as a better algorithm for automatic debugging than binary search. Is that true? Before answering that question, let us first look at some limitations of the two different algorithms.

B. Limitations Of Delta Debugging

Delta debugging has some limitations. The delta debug algorithm is not aware of any timeline as it only compares two different versions of the code and this may lead it astray.



Step	Commits (forming one Timeline)								Result
	1	2	3	4	5	6	7	8	
1	■	■	■	■					Pass
2					■	■	■	■	Fail
3					■	■			Fail
4					■				Fail
Result					■		(Fail)		

Figure 4. Delta Debugging Example with 8 commits to the revision control system, forming one timeline, where commits no 1-4 must be present in order for chunks 5-8 to be applied. Commit no 7 is the only faulty chunk, but the algorithm thinks it is commit no 5, because everything fails if commits no 1-4 are not present.

In Figure 4 the algorithm will come to the incorrect conclusion (commit no 5) because it is not aware of the timeline. In this example commits no 1-4 are required to be committed first in order for later commits (no 5-8) to be applied, otherwise the tests fail. The delta debug algorithm will find failures whenever it tests commits no 5-8, but it cannot distinguish real failures from just the limitation that is not aware of the timeline and is thus excluding commits no 1-4.

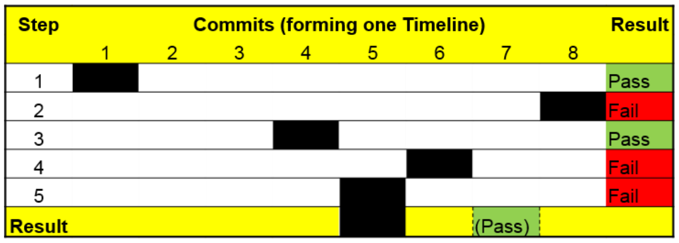
A second limitation of the delta debug algorithm is that it assumes that there are no complicated dependencies with commits or chunks far apart. Once it has narrowed down the problems to the commits no 5-8 the problem must be in some of the commits within range. However, in reality as we see in Figure 4, there can be dependencies to commits outside this range that complicates the picture. In step 2 in Figure 4 the failure has been narrowed down to commits no 5-8, but when the algorithm tries to further narrow down the problem it runs into a problem as it misses the dependencies to commits no 1-4.

To sum it up, delta debugging is good at narrowing down failures to a sub-set as long as there is no timeline and no complicated dependencies with chunks far apart.

C. Limitation Of Binary Search

Binary search performs better than delta debugging when there is a timeline because it respects the timeline. It does not try to test combinations of commits that have never existed together. Each commit was made by an engineer in a way that made sense at the time (even though it may contain bugs). This is different from delta debugging which tests any combination of code without caring whether the resulting code base makes sense. For example, in Figure 4 the delta debug algorithm tested commits no 5-8 without commits no 1-4, which produced a version of the code that had never existed. This is because commits no 5 and onwards built upon the changes made in commits no 1-4, but the delta debug algorithm ignores this.

However, binary search does not work when there is more than one faulty commit. In Figure 5 there is a faulty commit (no 5) which has already been fixed in commit no 7. But as the algorithm never tests commit no 7 it fails to recognize that the test is passing on this commit. It consequently zooms in on the already fixed commit no 5 instead of zooming in on the only existing open commit no 8.



Step	Commits (forming one Timeline)								Result
	1	2	3	4	5	6	7	8	
1	■								Pass
2								■	Fail
3				■					Pass
4						■			Fail
5					■				Fail
Result					■		(Pass)		

Figure 5. Binary Search Example with 8 commits to the revision control system, forming one timeline. Commit no 5 and 8 are both faulty, but commit no 5 have been fixed in commit no 7. However as the binary search algorithm fails to see the test pass when running on commit no 7 the algorithm fails to zoom in and find the faulty algorithm in commit no 8, which is the only open issue.

Binary search only works when there is only one faulty commit affecting the failing test. It does not go completely lost in the example in Figure 5, commit no 5 is a real bug but it has already been fixed. For multiple bugs binary search may miss some bugs, but the faulty commits it does find are real but may not still be open.

D. Best Way to Combine Delta Debugging and Binary Search

Let's summarize the strengths and weaknesses of the two algorithms:

No of Bugs	**Timeline Exists**	**No Timeline**
Single Bug	Binary Search	Both
Multiple Bugs	None	Delta Debug

Table 1. Debug Scenarios supported by the Delta Debug and Binary Search algorithms

In ASIC projects there is always some sort of a timeline with commits being made to a revision control system. The timeline may be complex, e.g. a project is spread out over different repositories, each with a slightly different timeline, but even in this case it is important to respect the overall timeline and not create combinations of code that have never existed in reality.

In some projects there is a small set of directed tests run on each commit to quickly detect simple mistakes, e.g. compilation mistakes, which can be covered by these short tests. This setup is called continuous integration. If a test fails, then there is no timeline between the test that fails and the previous revision where the test passes, because the small test is run on each commit.

However for large test suites and for constrained random tests the previous commit has not been tested, either because the test suite is too large to be able to run it on every commit or because it has not been run with the same seed as in the last run. Most tests fall into this category in ASIC projects.

Inside a single commit, there is no timeline. All lines were updated at the same time.

So what is the best way to achieve automatic debug down to a single line (or a couple of lines) in an ASIC project? The answer is: you need to use both delta debugging and binary search as one algorithm is not better than the other one in all respects. Start with binary search in order to identify the faulty commit, while respecting the timeline, and then use delta debugging to further narrow down the problem within a commit.

III. PERFORMANCE

A. Commit Sizes in Real ASIC projects

For small commits of a couple of lines we don't need to debug further in order to know which lines of code that caused a test to fail. However for larger commits it would be useful to know which code lines that triggered the failure. What proportion of the commits would benefit from further fine grained debugging beyond just the commit?

We looked at the commit sizes and bug frequency in two large ASIC projects. We investigated the following commits:

Project	No of Commits Between Bad Commits	Days Between Bad Commits	Time Period	Total Commits
1	40.5	2.8	Jan 1– Aug 11 2014	4690
2	45.0	1.3	Aug 1 2013 – Mar 1 2014	8585

Table 2. Median Commits and Days between Bad Commits

We plotted the distribution of commit sizes, separating bad commits that triggered test failures from all commits.

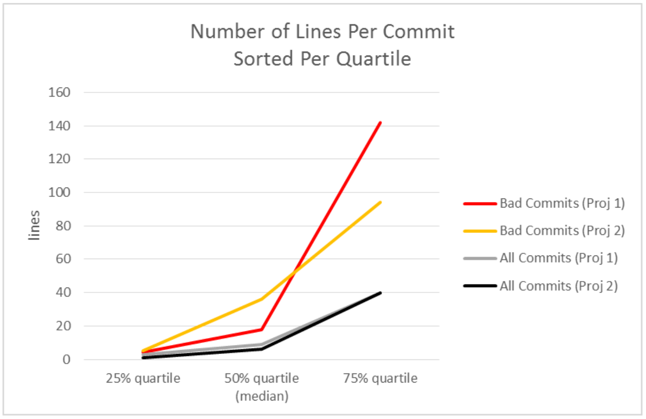


Figure 6. Number of lines per commit for bad commits compared to all commits, sorted per quartile of commits. Median number of lines in a commit is 7.5 lines across both projects, compared to a median 27 lines for bad

The median commit (see Figure 6) is just 7.5 lines across both ASIC projects, which is fairly small and there would be little benefit to further debug these commits. However the median *bad* commit was 27 lines, probably reflecting the fact that errors are more likely to be introduced the more lines that are changed. For the 75% quartile the commit sizes grow up to 140 lines and the rest of the commits are larger than that. The conclusion is that debug with line granularity would be beneficial for roughly half the bad commits, which are 27 lines or larger. In the example that follows in the next section we use the median commit sizes (7.5 lines for any commit, 27 lines for a bad commit).

B. *A Scenario Based on Median Commit Sizes From Real ASIC projects*

First, let us define a scenario (Figure 7) based on the median commit sizes (taken from Figure 6) measured in real ASIC projects.

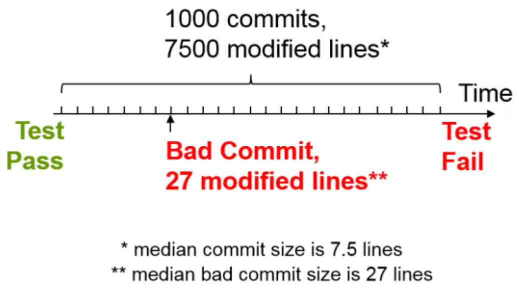

1000 commits,
7500 modified lines*
Time
Test Pass
Bad Commit, 27 modified lines**
Test Fail
* median commit size is 7.5 lines
** median bad commit size is 27 lines

Figure 7. A scenario with 1000 commits between two runs, where the commit sizes are set to the median values, i.e. 7.5 lines for any commit and 27 lines when looking at only the bad commits

We have already concluded that the best way to debug is to first use binary search in order to find the bad commit and then use delta debugging to find the exact line of code.

We will use binary search with one performance improvement: instead of just testing one revision in the middle of the timeline we will test 10 evenly distributed revisions. In the example in Table 3, we start with 1000 commits and test 100 commits in parallel in the first iteration, then zoom in where a transition is detected from pass to fail and then test every 10 commits in parallel. In the third iteration we use the binary search algorithm to test every single commit in the tranche of 10 commits where we have detected a transition from pass to fail in iteration 2.

In the same way, we are testing up to 10 different combinations of line chunks during delta debugging of the faulty commit. As the median bad commit is 27 lines, we are going to split that group in two halves (13 and 14 lines) in the first iteration and in parallel their sub-groups as well (6 and 7 lines). In total the parallel delta debug algorithm will test groups of 14, 13, 7, 6 and 3 line chunks in the first iteration. In the second iteration groups of 4, 3, 2 and 1 lines will be tested. If we are lucky the algorithm may already complete here if the first single lines that we test in isolation contains the bug. However, in this example we are going for the most pessimistic result. The last single bad line that is tested happens to contain the problem, which happens after 7 iterations.

Iteration	**Find Bad Commit, Then Delta Debugging** (commits)	(line chunks)
1	Every 100	
2	Every 10	
3	Every 1	**Within bad commit:**
4		14, 7, 3 chunks
5		4,3,2,1 line chunks
6		1,2 line chunks
7		1 line chunks

Table 3. How many times do we need to re-run the failing test (iterations) based on the scenario described in Figure 7? The answer is 5-7 times, depending on how lucky we are: if the first single line we test in isolation contains the bug the algorithm completes after 5 iterations and if it is the last single line that we test in isolation then it takes 7 iterations.

Using median commit sizes for the ASIC projects we have measured that we would be able to automatically debug a failing test down to the single bad line of code from a range of 1000 commits in the time it takes to re-run the failing test 5-7 times (including compilation time and the time it takes to check out the code from the revision control system).

It is also worth noting that already after 3 iterations the person who committed the bad commit can be notified that the commit is bad.

C. File Granularity

Up to now we have looked at line granularity and compared the result with commit granularity. In the previous section we saw that in an example with 1000 commits using median commit sizes it took 3 iterations to reach commit granularity and an additional 2-4 iterations, in total 5-7 iterations, to reach line granularity. However, a third approach is possible: file granularity. This means pointing out which files that have been updated within a commit, but not pointing out the exact line within the file. Performance-wise file granularity will end up somewhere in between commit granularity and line granularity. This can be useful for large commits, e.g. merger of branches, with several hundreds of files. In this scenario it can be useful to debug down to the file and that result can be produced with reasonable performance, unlike line granularity which may take too much time.

From an algorithmic point of view file granularity is identical to line granularity but the definition of code chunk changes. Instead of being a minimum number of updated lines or characters, all updates in one file is counted as one chunk.

Let's first look at the median commit sizes in terms of number of files that were updated in each commit. Here we are using the same data set as we did for line granularity.

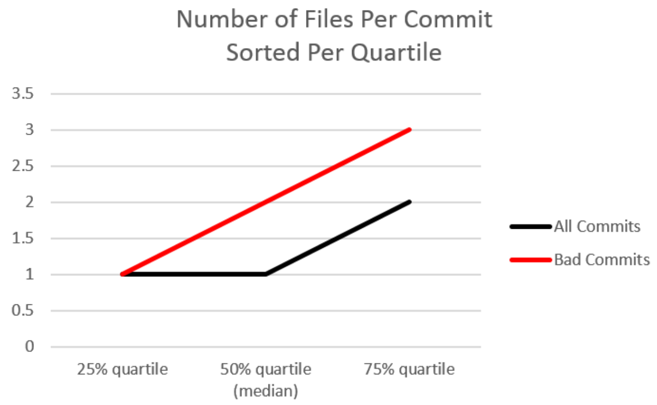


Figure 8. Number of updated files per commit for bad commits compared to all commits, sorted per quartile of commits. Median number of updated files in a commit is 1 file across both projects, compared to a median of 2 files per commit for bad commits

The median commit (see Figure 8) is just 1 file across both ASIC projects, which cannot be furthered debugged with file granularity. However the median *bad* commit was 2 files across both projects, probably reflecting the fact that errors are more likely to be introduced the more code that is changed. For the 75% quartile the commit sizes grow to 3 files for bad commits. The conclusion is that debug with file granularity would be beneficial for roughly half the bad commits, as they have 2 files updated per commit or more. However it is really the 75% quartile that benefits from file granularity as there are 3 or more files changed. This is different from line granularity which was useful in both the 50% and 75% quartile.

Let us look at an example with 1000 commits using the median sizes (1 file for any commit, 2 files bad commits).

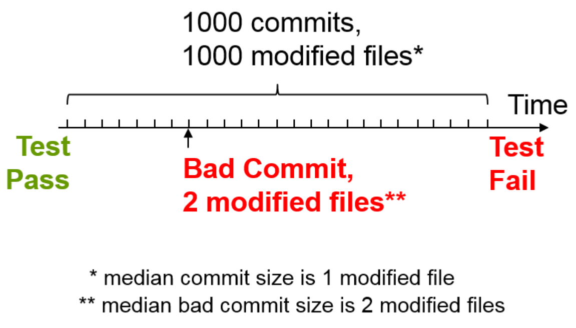

1000 commits,
1000 modified files*
Time
Test Pass
Bad Commit, 2 modified files**
Test Fail
* median commit size is 1 modified file
** median bad commit size is 2 modified files

Figure 9. A scenario with 1000 commits between two runs, where the commit sizes are set to the median values, i.e. 1 updated file for any commit and 2 modified files for bad commits

Debugging the example in Figure 9 with file granularity requires 4 iterations as shown in table 4.

Iteration	Find Bad Commit, Then Delta Debug
1	Every 100 commit
2	Every 10 commit
3	Every single commit
4	Validation of All Combos of Files

Table 4. How many times do we need to re-run the failing test (iterations) based on the scenario described in Figure 9? The answer is 4 times, one extra time to get file granularity for the median bad commit

File Granularity takes only 1 additional iteration in the median case, compared to commit granularity. This can be compared to the extra 2-4 iterations required by line granularity. By changing the definition of a code chunk from a minimum set of line updates to all updates in the entire file we can improve the performance substantially.

D. Trade-off between Debug Granularity and Performance

There is a trade-off between debug granularity and performance as we have seen in this chapter. Choosing the right balance is something that has to be decided on a case per case basis depending on the characteristics of the verification environment and the individual test times. The simplest way would be for the user to decide what debug granularity to use for each project.

Another, more dynamic way, would be for the user to define what debug granularity to use depending on the commit size. As the first step is to find the bad commit, the commit size will be known before starting the file or line granularity debug. It is consequently possible to wait until the size of the bad commit is known before choosing granularity. For example, the user may decide to use line granularity for commits where up to 10 files have been updated, file granularity for larger commits up to commit sizes of 100 files and commit granularity for all commits larger than this limit.

IV. METHODOLOGY

A. How the measurements were done

We first reasoned theoretically on the limitations of the binary search algorithm and the delta debug algorithms in order to combine them in the most optimal way.

Then we measured commit sizes for 13275 commits in two real ASIC projects (see Table 2) and used the median values to calculate how well the combined algorithm performs for a fairly large example of 1000 commits. We did this exercise first for line granularity. Then we repeated the exercise for file granularity and compared the results.

V. RESULTS

The delta debugging algorithm is not better than the binary search algorithm in all aspects, mostly because the delta debugging algorithm doesn’t respect the timeline which is formed by the string of commits to the revision control system that typically are done in an ASIC project. Consequently it is better to start with the binary search algorithm that does respect the timeline, in order to first narrow down the issue to one commit. Only when the problem has been narrowed down to one commit then it is time to let loose the delta debug algorithm in order to find the individual bad line.

Using commit sizes from real ASIC projects we saw that for the median case it will require 5-7 re-runs of the failing test (including compilation and checkout from the revision control system) in order to narrow down the problem to a single line, when analyzing a fairly large range of 1000 commits.

Based on the same data set from real ASIC projects we saw that the median case for file granularity will require 4 re-runs.

The conclusion is that there is a trade-off between debug granularity and performance. Finding the right balance will be different for different projects as it depends on individual test times, project sizes and verification environment in general. We believe that the ability to select debug granularity to line or file may be useful in real ASIC projects.

REFERENCES

[1] Booch, Grady (1991). Object Oriented Design: With Applications. Benjamin Cummings. p. 209. ISBN 9780805300918.
[2] PinDown from Verifyter, http://www.verifyter.com
[3] Zeller, Andreas (1999). Yesterday, my program worked. Today, it does not. Why? (Software Engineering—ESEC/FSE’99 doi:10.1007/3-540-48166-4_16 ed.). Springer.
[4] [NESS, B., AND NGO, V. Regression containment through source code isolation. In Proceedings of the 21st Annual International Computer Software & Applications Conference (COMPSAC ’97) (Washington, D.C., Aug. 1997), IEEE Computer Society Press, pp. 616– 621.