Automatic Bug Fixing

Presented at Microprocessor and SoC Test and Verification (MTV 2015), Dec 2015, Austin Texas

Daniel Hansson
Verifyter AB
Lund, Sweden
daniel.hansson@verifyter.com

***Abstract*— Several EDA tools automate the debug process[1,2] or part of the debug process[3,4] . The result is less manual work and bugs are fixed faster[5]. However, the actual process of fixing the bugs and committing the fixes to the revision control system is still a manual process. In this paper we explore how to automate that last step: automate bug fixing.**

First we discuss how the automatic bug fix flow should work. We implemented the automatic bug fixing mechanism into our existing automatic debug tool[1] and ran an internal trial. Then we list the various issues that we learned from this experience and how to avoid them.

Our conclusion is that automatic bug fixing, i.e. the art of automatically modifying the code in order to make a failing test pass, is very useful, but it is done best locally, i.e. the fix should not be committed. Instead a bug report should be issued to the engineers who made the bad commits and let them take action. Automatically committing the identified fix is very simple (unlike the analysis that leads to the fix), but it this leads to a list of issues such as human-tool race conditions, fault oscillation and removal of partial implementations.

Keywords—bug fixing; automatic debug; regression testing

I. INTRODUCTION

A. *Unrolling Entire Commit vs Fixing Root Cause*

There are two ways of fixing a bug (see Figure 1) which has been checked in to the revision control system. Fixing in this context means removing the bug from the revision control system in one way or the other.

The most straight-forward way is to fix the actual root cause. This is hard to automate as it requires a lot of intelligence, both about what the code is currently doing as well as trying to understand what the intention was.

The other way to fix a bug is to unroll the entire commit. In this case the whole commit that contains a bug is removed from the revision control system. The idea is that the developer who committed this bug should work on the code locally, find a fix for the issue and then commit the code again. During this time nobody else should be exposed to the bug in order to avoid being slowed down. This way of fixing the bug, by simply removing it and all associated code from the revision control system, is easier to automate.

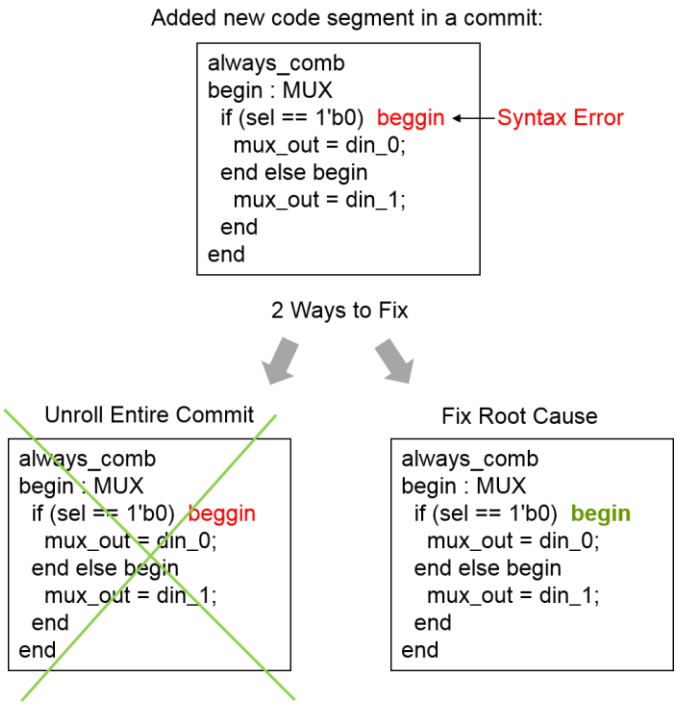

Figure 1. Two ways of fixing a bug: Unroll Entire Commit or Fix Root Cause. It is easier to unroll the entire commit than fixing the root cause, but in both cases the problem is fixed in the revision control system, i.e. the code compiles again. In the case of unroll, the code is removed from the repository and a bug report is sent back to the committer, who will have to fix the root cause and commit the code segment again. Unrolling is a way of strictly policing the quality of the revision control system and force engineers to fix issues locally and only check in good code.

The challenge with unrolling the entire commit is to only remove the bug and its associated code, which includes all code that was changed in the same commit, but it can also include code updated in other commits which are too tightly linked to be able to be handled separately. These commits are not normally the last commits made to the revision control system, other unrelated commits have been added. Consequently the bad commits cannot simply be unrolled in a simple way. There may be hundreds of commits, most of them

good, and the challenge is to select the bad commits made at different times and remove them, without affecting all the good commits. Essentially this will create a new version of the code that has never existed before but where the test or compilation now passes.

Unrolling only works for issues that have been introduced recently, so called *regression bugs*. Fixing the Root Cause on the other hand is always a way to solve a bug.

B. *Regression Test Setup*

There are two ways to prevent that regression bugs are entered in to the database under revision control: first, continued integration where a short test suite acts as a gate keeper. A small test suite of directed tests are run and if all tests pass then the commit is allowed into the main repository database. The test suite has to be short in order that you can run it on every commit. This methodology does not support constrained random testing because a random test failure may be due to a new scenario being tested and not due to a recently introduced regression bug.

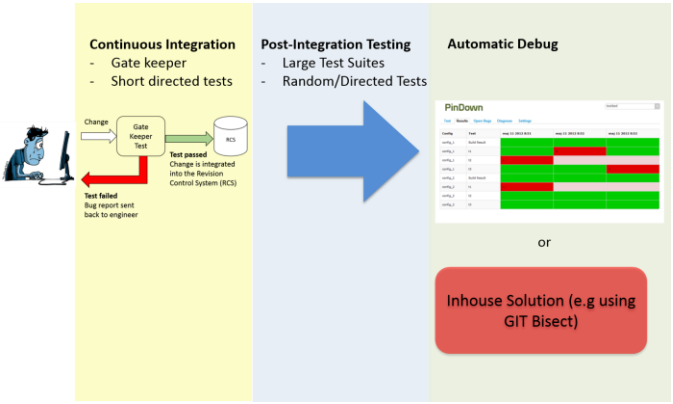

Figure 2. Regression Test Setups

For larger test suites and for constrained random tests you must use a different methodology. This is where you can use automatic debug tools[1,2] or more commonly: debug the failures manually.

C. *Automatc Debug Tools*

Today's automatic debug tools[1,2] points out where the bug is but leaves it to the human engineer to decide what to do. The engineer has to validate that the proposed fix actually solves the problem and makes the test pass again. Before committing the engineer should make sure that he or she has rebased to the latest version of the code so that the fix is still feasible to do. The engineer must also make sure that the failure is still open, as it may have been fixed by someone else recently.

The automated debug process for PinDown is described in Fig 3. The test failures are grouped and the fastest test in each group is re-run on different versions of the product and testbench. The goal is to fix the bug locally by backing out the one or many bad commits in order for the failing test to pass again. Please note that this is different from a roll-back to the last known good state. There may have been hundreds of commits since the last good state but typically only one or two commits need to be backed out again in order for the test to pass again.

Once the test pass again, locally, a validated bug report can be sent out to the engineer(s) that have made the bad commits. The next step after that is to commit the change to the revision control system. This is an optional step which PinDown can automatically do, but as we will see in this paper, that step has some side-effects.

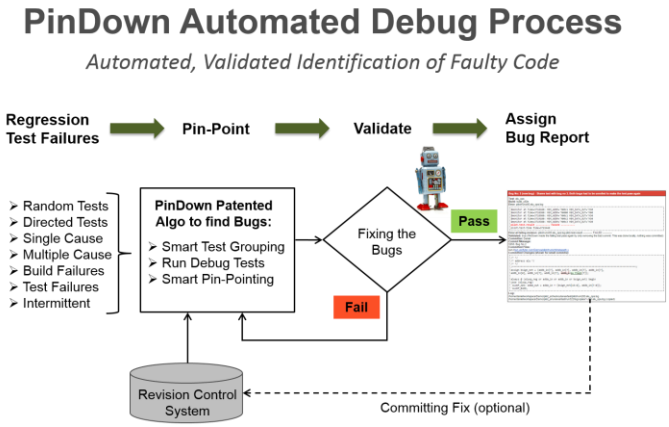

Figure 3. PinDown Automated Debug Process

In this paper we will not analyze what PinDown does in detail in order to debug each failure. However, it is the identification of the fix which is the most important part of the debug process. However, this process has already been described in an earlier paper[7].

II. IMPLEMENTATION

A. *Automatic Bug Fix Timeline*

First, let's look at the timeline for the automatic bug fix process (see Figure 4). It all starts with checking out a revision from the revision control system. A test suite is run which results in a report containing test results, both passing and failing tests. The next step is to debug the failures, check that the proposed fix is good and then commit the fix. This all takes some time, which is one of the challenges. In this paper we call the latest revision *head*, which is different from the tested revision (if anyone has made a commit after the testing started). When a fix has been identified it is important to check that the fix is still valid at head and that the failure is still open and has not yet been fixed, before committing the fix.

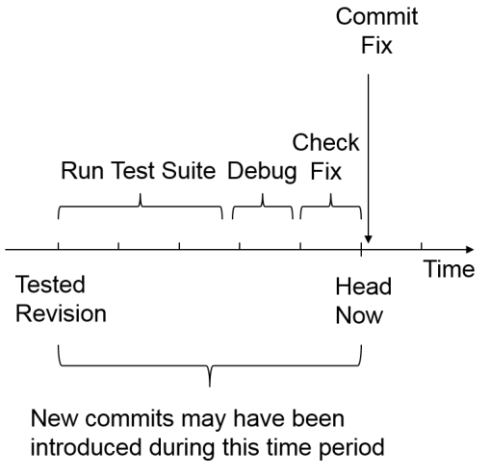

Figure 4. Automatic Bug Fix Timeline – Additional commits may be performed by engineers between the tested revision and the committed fix, which may affect the fix. The latest revision that exist right now is called *head*. It is necessary to check if the bug detected in the tested revision still exists at head and whether the fix is still valid before committing the fix. This is what human engineers should also do before committing a fix.

B. Automatic Bug Fix Flow

The flow for the automatic bug fix process is described in Figure 5. The flow starts with running a test suite and collecting the results. Subsequently the failures are analyzed by the automatic debug tool. If the tool cannot identify a cause it will send out an open bug report. If it can identify the cause it will propose a fix.

The next step is to Validate Fix, which is the most important step. In this step it will check whether the failing test will pass if the proposed fix is applied. If it does then the fix has been validated as a good fix. The automatic debug tool may propose many fixes, good or bad, some more speculative than others, but this step will filter out any bad proposed fix, which is why this step is crucial. There are many reasons why a test can fail, but to actually make a failing test pass again is a very strong indicator that the fix was good.

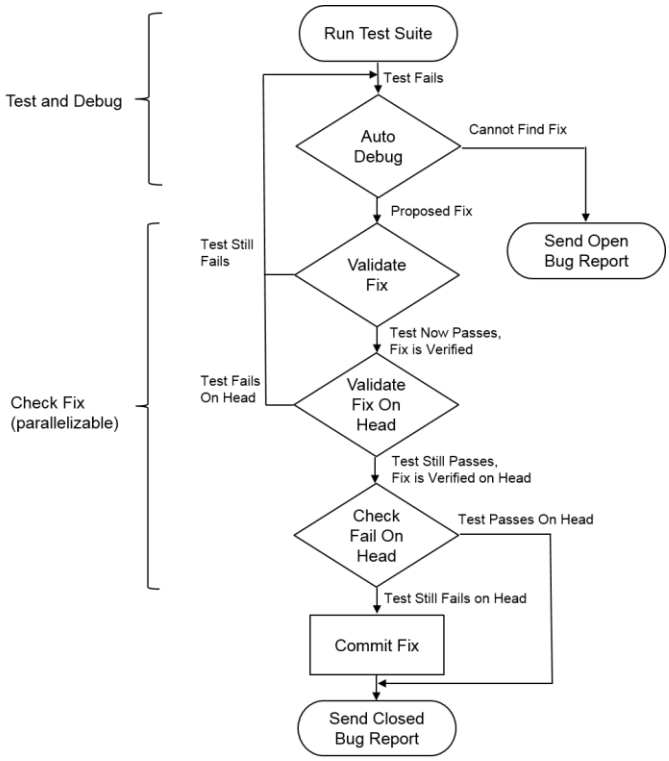

Figure 5. Automatic Bug Fix Flow. There are two outcomes – 1) send an open bug report or 2) commit a fix and send a closed bug report. The section "Test and Debug" is the same as in an automatic debug flow. The unique section for Automatic Bug Fixing is the "Check Fix" segment where we make sure that we have a good fix before it is allowed to be committed. All checks can be done in parallel,

After the fix has been validated we have done the most important part of the automatic bug fix flow. However, before we can commit the fix we must make sure that nobody has fixed the issue already or made the issue even harder to solve due to some additional closely related commits. This is done in the steps called "Validated Fix On Head" and "Check Fail On Head" which can be run in parallel to "Validate Fix" to save time.

First let's explain the concepts a bit further. "Validate Fix On Head" means running the test using the fix and the new commits. If the test pass then we know the fix is still good even after new commits was introduced. "Check Fail On Head" means running the test using the new commits, but without the fix. If the test still fails then this means that no one has fixed the issue manually. If "Validate Fix On Head" produces a passing test and "Check Fail on Head" produces a failing test then fix is still good even when taking the new commits into account.

If the fix was validated on the tested revision, but does not work on head then further commits have occurred which complicates the scenario. In this case we need to go back to the automatic debug step and further analyzing the scenario, using this new information. However, if the fix also works at head

and the test still fails, i.e. nobody has fixed the bug since the tested revision, then we commit the fix and send out a bug report where we mark the bug as closed, and describe what the tool did to close the issue. We also send this closed bug report if the issue was solved by a human engineer, before the tool committed the fix.

III. Methodology

A. *Implemented Automatic Bug Fix into our existing Automatic Debug Tool*

We implemented the automatic bug fix flow described in Figure 5 into our existing automatic debug tool called PinDown.

PinDown already performs the "Validate Fix" step in order to ensure that the outcome of the automatic debug analysis was correct. If not it will continue analyzing until it can make the test pass, which may require one or many commits to be unrolled. The validation of the fix allows PinDown to solve more difficult cases when many commits are involved. It also makes the users able to trust the results more as the tool made the test pass before issuing the bug report.

B. *Tested Internally*

We ran the extended version of PinDown internally in our development project to see how well automatic bug fixing worked and made notes on the lessons that could be learned.

IV. Results

A. *Human-Tool Race Condition*

One consequence of letting the automatic debug tool committing the fix is the race condition that may occur between the tool itself and a human that in parallel fixes the same issue (see Fig 6).

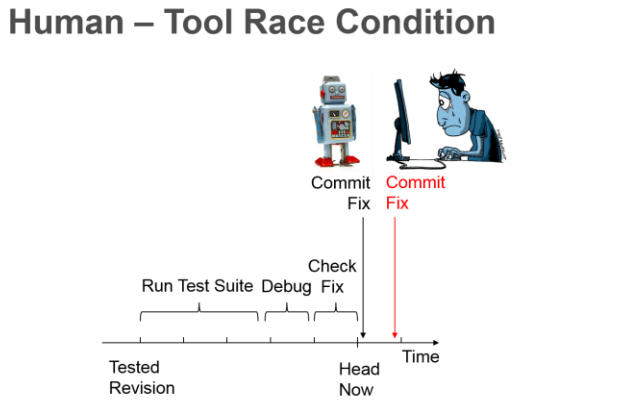

Figure 6. Human – Tool Race Condition

This issue can be reduced by the steps described in Fig 5 but it cannot be completely eliminated. The human engineers must also look at what the automatic debug tool is currently working on before committing a fix in order to avoid committing the fix to the revision control system at the same time as the tool, leading to two fixes being committed on top of each other.

B. *Fault Oscillation*

The automatic debug tool must never unroll a commit which itself created earlier when unrolling another commit. Otherwise the tool will go into fault oscillation where it will unroll one of the commits every second time for eternity.

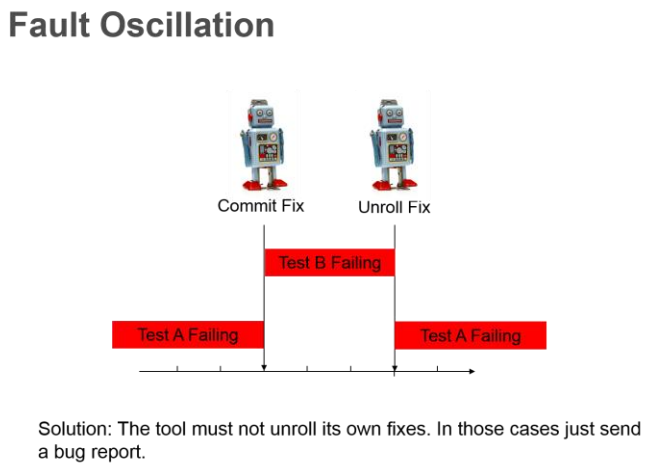

Figure 7. Fault Oscillation

C. *Partial Introduction of a new Feature.*

If the engineers are allowed to commit a partial implementation of a new feature which causes some tests to fail then there will be a conflict with the tool that will immediately remove it. This may cause irritation because the engineer probably intended to commit the remainder of the implementation at some time later. See Fig 8.

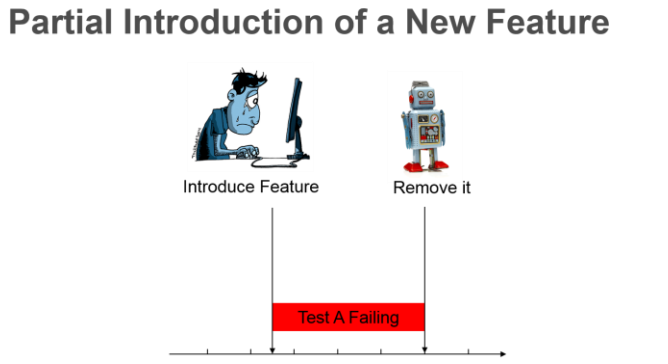

Figure 8. Partial Introduction of a New Feature

The solution to this problem is to disallow commits of partial implementations. However, this may be a challenge when using constrained random testing. Initially when the partial implementation has been committed there may be no failing tests, but a week later a random seed may trigger a scenario

that reveals a failure in the partial implementation. The tool will then back out this change, one week later (!) which may be annoying for the engineers.

In a continuous integration setup with a gate keeper test you cannot allow a commit of a partial implementation that causes any of the gate keeper tests to fail. However if the gate keeper tests all pass then there will be no surprise removal of the implementation at a later time. Having a commit being declined to commit a partial implementation because the gate keeper test is failing is logical and well accepted. This does not cause the same issues and surprise as a removal of the commit at a later time.

D. Communication.

We learned several lessons from this project. First, robustness and accuracy are both very important. The human developers must fully trust the tool in order avoid spending time analyzing whether the tool took the correct action. This is true for a pure automatic debug tool as well, but it is even more important for an automatic bug fix tool as it becomes a contributor to the revision control system without any human involvement. This is different from an automatic debug tool which automatically generates a bug report showing where the problem is, but lets a human actually take the action to fix it.

Communication is also very important. The automatic bug fix tool must send out a bug report to everyone in the project in general, but especially to the committer that introduced the error that was fixed. The bug report should be marked as closed and contain information about what the problem was and how the issue was solved. This is important so that nobody else tries to fix the issue which has already been addressed.

It is also important to assign responsibility for the automatic bug fix tool in general to someone and include this person in all bug reports sent out by the tool. If an engineer wants to discuss what the tool did they must know who to talk to. Without the human owner there is nobody to talk to as the complete flow is automated.

V. SUMMARY

There are three regression test setups which all prevent bad commits by either a gate keeper or an automatic bug fix tool that fixes the problem after the fault has been committed, see Figure 10.

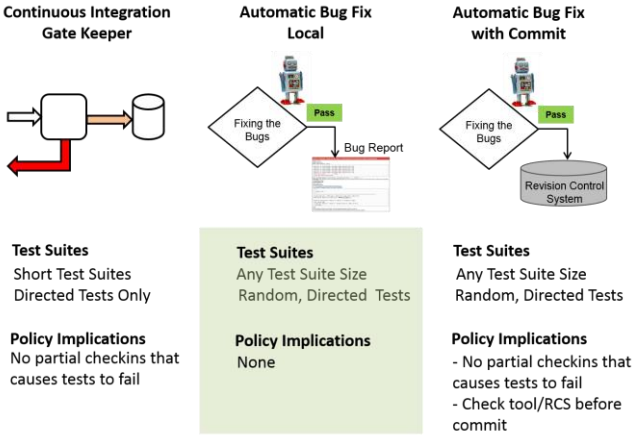

Figure 10. Comparing Automatic Bug Fixing, Local and Committing plus Continuous Integration with a Gate Keeper

It's interesting to note that all issues relating to automatic bug fixing appear because the fix is committed to the revision control system. If the fix is done only locally (i.e. not committed) and reported to the people who made the bad commits then there are no issues. The fact that the tool managed to automatically fix a bug locally means that it can send a validated bug report, i.e. the content of the bug report is not speculation; it contains information about how to make the test pass again.

Please note that this conclusion is valid for all types of bugs, not only for regression bugs, even though this paper has only talked about regression bugs. Automatically fixing any bug locally and reporting it is all good as long as the fix is not committed to the revision control system.

One idea for a way to solve this could be to let the tool automatically produce a validated bug report which allows the user to click on a "Fix" button. The decision is then taken by the human engineer which means there would be no issues with race conditions, fault oscillation or removal of partial implementation. The labor cost for clicking the "Fix" button would be very small. However there is a chance that further commits have been made before the user clicks on the "Fix" button in which case the fix may no longer work. To solve that issue the "Fix"-button could be active for a limited period of time until the involved files had been updated in the revision control system.

Continuous Integration does not have any complications. It is a very simple setup. The only downside with continuous integration is that it can only operate on small test suites and those tests cannot consist of random tests. The reason random tests does not work is that the results need to be directly comparable with results from the previous run in order to determine if a failure was introduced in the last commit, something which requires tests to be directed, not random.

To sum it up, for short directed test suites you can use continuous integration to prevent regression bugs. For larger test suites and random test suites a tool that automatically fix the regression bugs locally (i.e. no commits

to the revision control system) and then issues validated bug reports to the committers is very powerful and the most optimal way of handling regression bugs or any other type of bugs for that matter.

REFERENCES

[1] PinDown from Verifyter, http://www.verifyter.com

[2] Onpoint from Venssa, http://www.venssa.com

[3] Verdi from Synopsys, http://www.synopsys.com

[4] Indago from Cadence, http:/www.cadence.com

[5] "Measuring the gain of automatic debug" (Daniel Hansson, Heli Uronen-Hansson, MTVCon 2013)

[6] http://jenkins-ci.org

[7] "Standard Regression Testing Does Not Work", DVCon 2015, San Jose, Daniel Hansson