Bug Hunting 101: A Case Study in Squishing the Bugs Before They Grow and Eat Your Robot
Jeremy Rand
Team SNARC (Sooners / Norman Advanced Robotics Coalition)
jeremy.rand@ou.edu

Bug Hunting 101: A Case Study in Squishing the Bugs Before They Grow and Eat Your Robot

1 Introduction

As a veteran Botball programmer and hacker, I'm sometimes asked how I managed to code something extremely complex or reliable in a short amount of time. Many Botball programmers will tell you that approximately 10% of the programmer's time is spent writing new code; the rest is spent debugging problems in existing code. Unfortunately, debugging code can be so time-intensive that many teams either run out of time, or simply abandon their bug-hunting efforts and hope that the occasional bug they encountered won't show up at the tournament. In many situations, this will not be the case.

However, there are some good tips to make fixing bugs much easier and more efficient. In Bug Hunting 101, I will present some general strategies for finding and fixing bugs in your code. As an illustrative example, I will describe a major bug in the CBC firmware, which has caused at least 2 Botball teams to lose a match in the past 2 years. I will discuss how I identified the bug after less than 2 hours of testing (the night before our regional), and how I fixed the bug so that it didn't affect us on tournament day.

Most of the strategies in this paper are well-known to expert programmers, so crazy-awesome programmers and hackers probably won't gain much from this paper — this paper is aimed more at the beginning to intermediate programmers who know how to write the code to accomplish what they want, but may not know what to do when the code fails.

All code revisions referenced in the paper are included in a .zip file in the GCER Proceedings. The paper is guaranteed not to make sense unless you follow along with the code revisions, so please do follow along.

2 Case Study: Norman Advanced 2011 Pom Sorter

As our example for this paper, we will be looking at Norman Advanced Robotics' pom sorting robot from the 2011 Oklahoma regional. This robot is of moderate complexity in terms of programming. Imagine the following:

It's about 10:30PM, the night before the 2011 Oklahoma regional tournament. You're the lead programmer for Norman Advanced's flagship robot. Your team is up late in the Norman High School physics room, and things aren't going well. You just finished the code for sorting poms about an hour ago. Things seem to work great in a test program that simply sorts poms without doing anything else. But when you run the entire program, the CBC crashes randomly. Clearly this would be a very bad thing if it happened at the tournament. You have 30 minutes before your team is required to leave Norman High (Mr. Askey's policy is that the team has to leave at 11:00PM — no exceptions). The stress is on.

Stop. Think. What do you do?

3 Try to Reproduce the Bug

When you have a bug, the first step is to find a way to determine whether you've fixed it. In other words, you need detailed steps which are guaranteed (or at least have a high probability) to reproduce the bug. This is called reproducibility. Reproducibility also has the benefit of telling you roughly where the bug in your code is, since the code executing while the bug occurs is a good place to start looking.

You run the program a few times with the light start enabled. It doesn't crash. You're not 100% sure why it worked these times. Then you start running the program again. The robot is ready to go when you turn on the starting light, but the documentation people need to talk to you for a few minutes. Frustrated, you leave the robot unattended while answering the documenters' questions. You come back to the robot to find it crashed.

Current time: 10:36PM.

Stop. Think. What happened?

There was a correlation between leaving the robot unattended and it crashing. You're sure that no one messed with the robot (you were only on the other side of the room, and your teammates aren't jerks; they want you to figure it out too). So what do you do? Reboot the CBC, get to the exact same point, and leave it alone for about 3 minutes. Sure enough, the CBC suddenly becomes unresponsive.

You've reproduced the bug.

Current time: 10:40PM.

4 Look for Really Obvious Mistakes

Your first thought is, what's happening in my code there? You look at your code (Revision 0).

The problem has occurred while waiting for the starting light. This means that any code which ran before or during this point could be the culprit. Do you see anything obviously wrong with your code? Did you make a really obvious typo in your code? Accidentally call the wrong function? You decide that nothing jumps out at you. You decide to look at it more analytically.

But first, stop. Think. What should you do before modifying any of your code?

5 Backup Your Code in Every Step

When bug-hunting, you're going to be modifying your code to see what happens. Before each modification, you should make a separate backup. If you know how to use Git [1] or SVN [2] (I recommend Git, since it doesn't require a network connection), use it to make a separate commit before each change. This way, you're guaranteed to have a way to undo changes. Using tags [3] and branches [4] is also a good idea.

Luckily, you already have your code in Git, so you quickly make a branch for your bug-hunting activities. This only takes a few seconds.

Current time: still 10:40PM.

6 Start Removing Apparently Irrelevant Code

As you remember, the test program which just sorted poms didn't crash. You've used `mrp`, `mav`, `bmd`, servo commands, and `msleep` hundreds of times throughout the season with no issues either; you suspect that the crash is more exotic. And this code all executed after the crash occurred, meaning it's probably not relevant (there are rare exceptions to this rule). So you remove all of the code in `main()` after line 330, which was where the crash occurred. Your result is Revision 1.

You test Revision 1 to see if the removed code was relevant. It still crashes in the same place.

Current time: 10:44PM.

7 Use Binary Elimination, Not Linear Elimination

You're short on time. You have 16 minutes before you're going to get kicked out, and you still have 57 lines of code in main() which could be related to the crash.

Stop. Think. What should you do?

If you individually remove each piece of suspect code, you'll have to test n times, where n is the number of pieces of suspect code. But what if you remove 50% of the suspect code each time? You'll cut the search area in half every time you test, meaning that you'll only have to test $log_2(n)$ times. This will drastically reduce the number of times you have to test, so that you can get the maximum effect from your 16 minutes.

You start looking for suspects in Revision 1.

8 Hacks are Suspect

Of the remaining code in `main()`, there are 4 significant sections. Your team's resident hacker (Jeremy) made a modification to the firmware which adds the ability to switch between cameras; lines 316-324 utilize this feature. Using multiple cameras slowed down the CBC, so your team's resident hacker also gave you a hack to prioritize your code above everything else; lines 263-265 utilize this. There is also a simple option selection menu, a `set_servo_position`, a `start_process` for a `black_button()` emergency stop, as well as the `wait_for_light` code itself.

Stop. Think. What should be most suspect?

If you've seen my talks on hacking, you remember the videos of the exploding microwaves, as well as the warnings that "We do these experiments so you don't have to. Do not try this at home." [5] If not, just remember for now that hacking tends to cause unexpected problems at unexpected times, and should not be taken lightly. Given these dire warnings, it seems logical that the 2 most suspect of the 6 code sections are the hacks.

Stop. Think. Which 50% of the code sections do you remove?

You may be tempted to remove the two hacks, and one other piece of code. This is not as good an idea as it sounds. If we remove the two hacks, and that fixes the problem, that doesn't tell us much — we already knew those two hacks were probably the issue. However, if we only remove one hack, and remove one of the less suspect sections, then we have a good idea of which hack was the problem, since the less suspect sections are most likely irrelevant. So, you remove the camera hack, the `set_servo_position`, and the options menu.

As a side note, in many cases you would want to replace the options menu with setting default options. In this case, most of these options are only used by code we already deleted, so we can save time by simply removing them outright. The only exception is `light_start`, so we replace that variable later in the code with a 1 (since we were testing earlier with the light start enabled).

Your result is Revision 2.

You test Revision 2. It still crashes. You need to remove more code.

Current time: 10:48PM.

Stop. Think. What do you remove now?

There is 1 hack remaining, and 2 other sections of code. Do you remove one of them too? No. The hack is far more likely to be the culprit, since it messes with the CBC's Linux internals (as most hacks do), and removing the hack by itself would confirm this. So, you remove the priority hack (lines 263-264). Your result is Revision 3.

You test Revision 3. It still crashes.

Current time: 10:52PM.

Stop. Think. What do you remove now?

9 Get a Reduced Test Case

Of the 2 remaining sections, the start_process is more suspect, simply because it's your code and not KIPR's code (KIPR's code is typically well-tested). You also figure that the `if(1)` can't be relevant, and nor can the other functions and `#defines`, so you remove the `start_process`, the `if(1)`, and everything outside `main`. You get a compile error, because you deleted `light_port`'s definition. You quickly hardcode the light port, and get Revision 4 (excerpted below).

```
int main()
{
    wait_for_light(1);
}
```

You test. It still crashes. You now have what is called a reduced test case (the simplest possible code which still triggers the bug).

Current time: 10:57PM.

You conclude that wait_for_light is indeed crashing the CBC at some point.

Stop. Think. What is probably happening?

10 When in Doubt, Reload Firmware

`wait_for_light`, like the rest of the firmware functions, are on an internal flash drive on the CBC. If the filesystem is corrupted (either by hacking, or by switching off the CBC at the wrong time, or by loading a bad firmware, or by some other weird event), `wait_for_light` could have gotten corrupted. It's too late to reload a firmware and test on the game board. You ask your mentor if you can bring the robot home. He says yes (he trusts you), but he warns you not to stay up late.

You drive home with the robot.

Current time: 11:15PM.

You reload the firmware that your team's resident hacker gave you. It installs.

Current time: 11:21PM.

You reload the program, and run it. It still crashes.

Current time: 11:24PM.

Finally, in desperation, you reload an official firmware. You know it won't work with your custom program, but you figure it can narrow things down so you can contact the hacker and get

him to check his code before the tournament.

Current time: 11:30PM.

You reload the program, and run it. It still crashes.

Current time: 11:33PM.

11 Switch CBC's.

You conclude that there must be something wrong with the CBC hardware. Luckily, you have a spare CBC at home. You load the program onto that CBC. It still crashes.

Current time: 11:36PM.

12 Copy Code from the CBC Firmware

At this point, you realize that there must be a problem with the `wait_for_light` function in KIPR's code. KIPR rarely makes mistakes, but all the signs point to it. But you don't have the ability to rebuild the CBC firmware — you're not a hacker, and besides, reloading firmware takes time, which you don't have.

Luckily, you know where KIPR's GitHub is, and you quickly find the `wait_for_light` function from KIPR [6]. You rename it `wait_for_light2`, and copy it into your own code. You get a few duplicate declaration errors, so you also rename the helper function and its global variable. Your result is Revision 5.

You test it. It still crashes.

Current time: 11:39PM.

13 Fast Forward a Bit...

Following the same logic described previously, you start removing pieces of the code, until you have a reduced test case which still crashes.

This reduced test case is Revision 6 (exerpted below).

```
int main()
{
    while(1)
       {beep();}
}
```

Current time: 12:05AM.

14 Creating a Fix

Calling the `beep()` function repeatedly seems to cause the CBC to crash.

Stop. Think. Should you try to fix the beep() function?

No. `beep()` may be defective, but it doesn't matter why or how it crashes, because `beep()` is completely unnecessary for your purposes. You're calibrating a light; beeping is irrelevant. So what is the solution? Simply comment out all beeping from Revision 5 (the full `wait_for_light` function), and test.

You've hit the jackpot! It doesn't crash! All you have to do now is copy the modified wait_for_light function from Revision 5 into Revision 0 (Revision 7), and test it. Obviously you won't be able to run on a game board, but through binary elimination we already determined that that wasn't the problem. It works.

Current time: 12:10AM.

Now reload the hacked firmware which your full program requires, and test the program again. Again, it works.

Current time: 12:19AM.

15 Mission Accomplished!

You're done. It's only 12:19AM; less than 2 hours after the bug was discovered (including the drive home). Get some sleep, go to the tournament in the morning, and watch as the robot's light start works perfectly every time!

16 Conclusion

We've followed how to hunt bugs, using a real-life example. Effective bug-hunting skills are vital in both Botball and the real world. In Botball, this exact bug is believed to have cost at least 2 teams a tournament round. And in the real world, if a bug is threatening to make a NASA rover crash-land, they had better get it fixed before anything bad happens. And if a bug is threatening the security of a bank or a nuclear facility, not fixing it quickly could result in even worse consequences.

As a side note, I never did figure out how the `beep()` bug was happening. I can't prove whether the problem is in KIPR's code, Qt's code, or Chumby's code, but based on incomplete research and experience, I suspect the issue is not KIPR's fault. In any event, it's not something I would expect KIPR to notice in testing; I can't picture KIPR wasting development time putting a `beep()` function in a loop just to see what happens.

You can find me on the Botball Community [7] if you have any questions.

Also, regardless of whose code contains the bug, I strongly recommend that KIPR remove the `beep()` function from their `botball.c` libraries until the problem with `beep()` is corrected.

Happy Hunting!

18 References

[1] Wikipedia contributors. Git (software). https://en.wikipedia.org/wiki/Git_%28software%29 , June 2012.

[2] Wikipedia contributors. Apache Subversion. https://en.wikipedia.org/wiki/Apache_Subversion , June 2012.

[3] Wikipedia contributors. Revision tag. https://en.wikipedia.org/wiki/Revision_tag , April 2010.

[4] Wikipedia contributors. Branching (software). https://en.wikipedia.org/wiki/Branching_%28software%29 , March 2012.

[5] Jeremy Rand, Matt Thompson, Braden McDorman. Hacking the CBC Botball Controller: Because It Wouldn't Be a Botball Controller if It Couldn't Be Hacked (GCER 2009 video). https://www.youtube.com/watch?v=zYdm-Y064MM , July 2009.

[6] KISS Institute for Practical Robotics. botball.c . GitHub, https://github.com/kipr/cbc/blob/8b27eead824752069e75b087fec53ae5c1774cde/userlib/libcbc/src/botball.c , January 2011.

[7] Botball Youth Advisory Council. Botball Community. http://community.botball.org , June 2012.