FixHub

And The Quest For Program Repair

Zachary Ferretti, Chengzhe Li, Zhixiang Huo, Owen Vnek[a], Connor Wong

[a] *University Of Massachusetts Amherst, Amherst MA, 01002*

Keyword: Automated-Program-Repair, Computer-Science, Replication, SimFix

Abstract: Throughout this paper we aim to answer the question of whether or not SimFix (in it's current state) could feasibly be applied on an individual scale code base.

1. Problem

Throughout this Project, we set out to determine whether or not the SimFix Algorithm[1] would be a feasible tool for a lone or small-scale development team to implement. Specifically we intended to evaulate whether a team of five senior collegiate students would be able to set-up a code base which could have bugs fixed automatically using SimFix. The first step in our process will be to reproduce the results that the SimFix team originally reported they achieved.

We will measure this by checking if we are able to attain the 34 correctly patched bugs and the 22 incorrectly patched (false-positive) bugs Jiang reported. Since SimFix has in the past reported incorrectly, Zach developed a script to automate the process of checking each patch that was generated by SimFix. This script is successful in automatically testing developed SimFix patches, but fails when there are more than two patched files.

We will call this paper a success if it is able to answer this question, whether the results are positive or negative. This allows us a good deal of independence and lends itself to true curiosity, as we are not looking for one particular answer, just to see what is out there.

2. Design

During the first portion of our project we spent some time attempting to replicate the results which were obtained from the SimFix team in their original paper.

We spent a lot of time on this step of our project. At first we attempted to use our teams local MacOS machines to host the SimFix project. We configured everything correctly, specifically downgrading Java from 1.8 to 1.7, ensuring other dependencies were present, etc. We then manually set up Defects4J [2] and SimFix. After doing all of these steps we were successful in executing SimFix and Defects4j.

However while they did *run* they did not succeed. That is to say they did not get the same results as were listed in the original paper. Eventually after spending a lot of time trying to figure out why this process was failing, we decided it would be a better investment of our time to attempt working in an Ubuntu environment (the only requirement we did not originally comply with).

First we tried working on the Universities remote connection computers (EdLab) as these run the correct version of Ubuntu (16.04) and have met both projects (Defects4J + SimFix) requirements. These computers were host to shockingly slow performance speed. After a day and half when we finally got back results from 10 tests, again SimFix had failed to repair anything.

Finally we decided to give it one more shot by attempting to run SimFix using local Virtual Machines with Oracle Virtual-Box. At last we achieved success and began getting the same results that the SimFix team had listed. In fact, we achieved their results when we used two distinct methods.

(a) Zach worked on testing the manual set up for SimFix. He was able to replicate the results by going through the installation and setup procedure documented on the SimFix GitHub.

(b) Chengzhe simultaneously was able to replicate the results using the replication package set up by the original SimFix team.

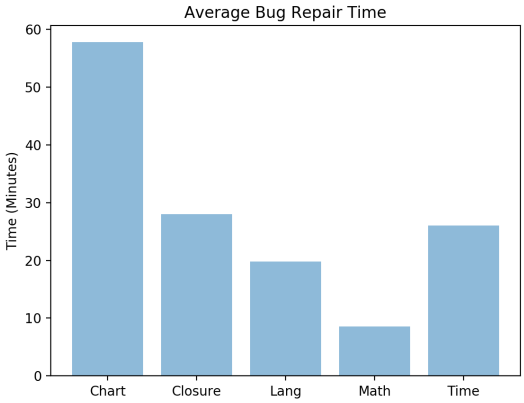
3. Replication

3.1. Evaluation

When we began doing our replication study, we first looked at the 34 *successful* bugs that SimFix was able to repair. We used a shell script to automatically run each of the different tests that SimFix performed on. We decided that it was best to first focus on replicating these 34 bugs. Replicating this was prioritized, as we found replicating their success was harder than their failure. In case you would like to replicate these results yourself it is pivotal that the operating system you use is Ubuntu 16.04. The 34 bugs that we were then able to replicate patch generation for were:

- `Chart` - 1, 3, 7, 20
- `Closure` - 14, 57, 62, 63, 73, 115
- `Lang` - 16, 27, 33, 39, 41, 43, 50, 58, 60
- `Math` - 5, 33, 35, 41, 50, 53, 57, 59, 63, 70, 71, 75, 79, 98
- `Time` - 7

3.2. Results



The shocking thing about these results is just how varied the execution time was for different testing suites. For example if we remove the `Chart` suite from consideration our average test time is only `16:47` (16 minutes, 47 seconds). However if we include the `Chart` suite this average test time goes up to `28:02` (28 minutes, 2 seconds).

3.3. Conclusion

These results make us wary of the practicality of this software (in it's current state) on smaller test suites, because if you are working with a small application which compiles and can be tested manually or executed in only a few seconds, then dedicating 15 minutes or more to just solving one test seems to be a bad use of the developers time.

I think what these results indicate more is that the feasibility of this software greatly depends on what type of program you are developing. For example `Chart` most likely indicates the development process is occurring with some form of a graphics based program. Whereas `Math` may be a more straightforward code-base (for `Math` the average execution time was `8:57`) which could help to explain why it took less time to evaluate `Math`.

SimFix also has a fairly large requirement for the code base you develop in order for it to run. The code base needs to follow the Defects4J framework otherwise it will not generate patches. This is a fairly large handicap for a solo developer to consider if they are just writing a small script. It is not helpful either that the implementation process for the Defects4J framework remains largely undocumented. In many situations especially on the lower level it would be more practical to try and debug the problems on your own as opposed to using a tool to hopefully try and do it for you.

4. Extension

4.1. Evaluation

When we began looking to evaluate our extended work we just set out with a simple goal. To see if us as senior level developers would be able to implement a very bare bones code base which could be automatically repaired using the SimFix tool as it was evaluated on the paper.

4.2. Results

The results we want to explore here are not so much hard statistical evaluations, but experience-based findings. As a team we spent several days researching and spending time attempting to understand the Defects4J framework. We explored different avenues of implementation while attempting to get our simple code-base implemented in a Java development environment.

The biggest take-away we had from this was that the lack of documentation throughout both projects (Defects4J + SimFix) posed inescapable obstacles. I understand that when doing research your focus is on getting your own results, and clearly both teams were able to. But we found it incredibly challenging to overcome this obstacle during our time working on the research extension. In the SimFix documentation it notes that in order to use their software tool we had to follow the Defects4J framework, yet SimFix's team go into no detail on how to do this.

The logical next step was to examine the Defects4J's GitHub page and try and determine through this means how to implement their framework. They have a folder named `framework` yet go into no detail on how to actually implement this framework. There is one file named `template` but this did not prove useful and still left us searching for answers.

4.2.1. Result Discrepancies

Since we were running into issues as we have outlined above with doing a direct follow-through of what we had set out to do earlier in this report, we decided to extend in different ways. We developed a `bash script` designed to automatically test the patches that SimFix had created. This proved to be really insightful as it allowed us to determine which patches that SimFix had generated were really *true* patches and which were false-positives. There were several false-positives listed on the SimFix GitHub page which was useful as it allowed us to match up and re-produce their results more accurately.

The surprising thing was that there were many patches which we tested and got good results on (i.e. the patches *did* fix all the tests they were crashing) yet were still marked as incorrect if you look here: https://github.com/xgdsmileboy/SimFix/tree/master/final/result-All . Specifically, They list the following patches as false positives:

- *Chart*: `5, 7, 12, 13, 14, 18, 22, 24`
- *Closure*: `80, 106`
- *Lang*: `35, 44, 45, 50, 63`
- *Math*: `1, 6, 8, 9, 20, 28, 40, 46, 62, 67, 72, 73, 81, 85, 88, 95, 98`
- *Time*: `1, 6, 9, 24`

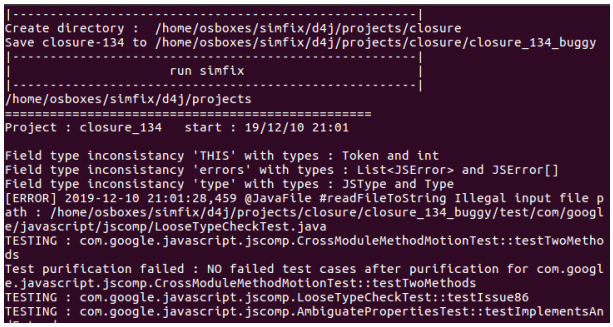
We have a suspicion that the reason's we were not getting these results as being false positives was because perhaps when testing these files for Correctness their team went into more detail and ran the entire test suite over again with the new patch, whereas we only use what was in each bug's test files. This realization was in part fueled by some of the feedback that we received during our final presentation. Yuriy asked us whether or not we had "tested patches on the same tests that they were generated from".

This prompted us to reconsider the original paper, to see just how SimFix checked their patches. As it turns out the way SimFix's team checked the tests was to compare the patches that SimFix generated to the patches Defects4J supplied and check if the results were the same in application. This is different from what we did, however we still think that what we did is a valuable metric.

4.2.2. Testing New Bugs

We also decided near the end of our time with the project to attempt running SimFix on the new bugs in the otherwise previously tested datasets. In order to do this it was required to generate a new `all_tests` and `all_failed_tests` file (these are generated when you run the commands `defects4j compile; defects4j test;`).

We attempted running it on new projects from Defects4J (such as Mockito) but this continually failed. We then attempted running it on Closure bug 134. We moved the test files to where they needed to be in the directory `/simfix/runnable/d4j-info`, and this lead to us being able to run SimFix on new bugs. Due to time constraints we were not able to run a lot of bugs or see if SimFix was able to generate patches. You can see the beginning of Closure 134's evaluation below.



4.3. Conclusion

While the results we list above may sound negative, in fact they are nothing but positive. We still were capable of answering our originally posed question. No, as it currently stands SimFix cannot evaluate or repair code-bases used by small scale developers or development teams. Furthermore I think that for their software to ever hope to become truly feasible in a wide-scale market they need to either work to make the framework for Defects4J easier to implement or do away with requiring that framework all together.

5. Replicating Our Results

After spending so much time exploring the process of replicating the results of a paper, we believe it is pivotal to clearly state how you can replicate our results. So throughout this section we plan on giving you detailed instructions about how you can achieve the same results that we did.

1. Begin by first downloading the `Replication Package` for SimFix.
2. Set up your environment by changing directories into the `SimFix` directory that exists after unzipping the replication package download and run the command: `source source_me` .This sets the environment variables and the like all to be what SimFix needs to run.
3. To manually run SimFix and then test a patch:
   (a) Run the command `./run.sh Chart 1` to run SimFix on a specified Project and Bug (here `Chart 1`).
   (b) After this, switch into the bug directory and you should be able to run `defects4j compile; defects4j test` and receive some amount of output documenting the success or failure of your tests.
   (c) Then if you go into the folders `‘(simfixRoot)/runnable/patch‘` and move these patches to the corresponding folders for the Defects4J bug, once you run `defects4j compile; defects4j test` again you ought to see that all the tests pass.
4. The simpler method is to automate this process, which can be accomplished by downloading our GitLab repository and making use of our `‘automateReplication.sh‘` and `‘patch.sh‘` scripts.
   (a) Move the `‘automateReplication.sh‘` into your `/(simfixRoot)` and then execute it to automatically run SimFix on the 34 tests we patch.
   (b) Then if you go into the folder `‘(simfixRoot)/runnable/patch‘` and bring our `patch.sh` file into here, by running it it will automatically validate all of the patches. *Note: This script does not work well with patches containing multiple files, but is successful on a large enough percentage of them to be useful regardless, as many of them are single file patches.*

6. Overall Conclusions + Takeaways

Overall we have concluded that in it's current form SimFix is not a feasible tool to be used by solo-developers or even small teams. In fact we honestly feel that it would not currently even be feasible to help a large team. Not because of the accuracy (as we admittedly were not able to test this on non-defects4J code-bases) but because of the difficulty we had setting it up, and the large requirement of implementing the Defects4J framework to gain program repair functionality.

If you are a large-corporation with a relatively large code base then completely re-working it in order to fit this framework would be incredibly time-expensive. More importantly the results that you *could* achieve when using SimFix frankly still leave a lot to be desired. It takes a long time to figure out the patches and in many scenarios we believe the bugs would most likely be able to be fixed in the same amount of time or less with conventional methods. The benefit of SimFix is that you could run it offline while the developers are asleep so even when no one is working, work is still being done.

We also learned alot about the research and software-engineering process. Specifically following the paper *exactly* would have left us with more time to focus on our extension. It is also useful to compare the paper's results to our own for each section. We also got familiarized with just how difficult it would be to extend a study. Replicating it was hard enough, but then attempting to extend it also required a deep understanding of many inter-connected parts that when looking strictly at literature or GitHub repositories may not be accessible.

The research being done by the SimFix team *is* ground-breaking. It leaves our team excited for the future of automated program repair, and many of the issues described above are purely (we believe) growing pains. Whenever new software is being developed it makes sense for the technology to have issues and need to grow more before it will be widely applicable. As we can report, SimFix is not there yet, but in the future very well could be a standard tool in the developers tool belt. SimFix has built upon and stands on the shoulders of the other automated program repair programs that have come before it. By combining the two useful states to avoid the low patch accuracy of other APR repositories is a brilliant step in the right direction. That is the point of research, to take steps in the right direction allowing your successors to innovate even faster.

7. Useful Links

- Our GitLab: https://gitlab.com/smalllicheng/fixhub - *Contains the scripts we wrote for this project, as well as the patches we found when testing.*
- SimFix GitHub: https://github.com/xgdsmileboy/SimFix - *Where you can download the Replication Package to get started replicating our results!*
- Original SimFix Paper: https://xgdsmileboy.github.io/files/paper/simfix-issta18.pdf
- Defects4J GitHub: https://github.com/rjust/defects4j- *The repository containing the Defects4J library for more information.*

References

[1] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. 2018.

[2] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. 2014.