Analyzing The Software Bug Lifecycle

Jaco Geldenhuys and Willem Visser

Computer Science Division, Department of Mathematical Sciences

Stellenbosch University, Stellenbosch 7601

Tel: +27 21 8084232, Fax: +27 21 8829865

email: {jaco, wvisser}@cs.sun.ac.za

Abstract—We describe the Impendulo tool suite and how we use it to analyze how programmers develop code and more importantly how they introduce and correct errors. In addition we also evaluate state-of-the-art static analysis tools to see whether they can detect the errors the programmers introduce. Our results indicate that these tools find almost no errors. This clearly shows that we need to develop new static analysis tools if we want to improve the software development process. We also illustrate how the results of our programmer observations can be used to determine programmer performance in an objective fashion.

Index Terms—Experimentation, Measurement, Verification

I. INTRODUCTION

It is well known that software errors are costly [9] and the holy grail for the software developer community is how to avoid errors going out into the field and thus hurting the bottom-line when they surface at an untimely moment. However, there is maybe an even more fundamental set of questions that has gone unanswered, namely, what are these errors, where do they come from, why do they go undetected? Our ultimate goal is to answer these questions. The main artifact that we want to produce is a tool that can detect these errors as they are introduced, maybe even stop you from introducing them in the first place. However, the first step is to find out what the errors are and how they get introduced. For this purpose we built the Impendulo tool suite.

The Impendulo framework is built for observation. The primary data we observe is the behaviour of developers as they work on a piece of code. An important aspect of this capability is its granularity: snapshots are recorded as the code is being written. A snapshot is taken every time a programmer saves or compiles his/her work, which happens surprisingly often. Based on this, we can observe fine-grained behaviours that are not captured in the finished product. The snapshots show the thought processes of developers as they make progress, make errors, discover errors, and correct them.

This data is — we believe — a goldmine of information, and can be explored in many different ways. One can focus on the habits of individual developers or identify trends that apply to developers as a group. But there are also other ways to use the data. In our work, we have focused on a second set of observations: the snapshots represent many variations of the evolution of a single piece of code. In our experiments (which we described in more detail in Section IV.B), we provide our developers with a suite of unit tests. These serve as a testing oracle, which means that we can know exactly where in a piece of code the errors are. This combination of oracles and program variations can be used to evaluate the efficacy of program analysis tools.

It allows us to identify tools that find none/some/many/all errors, the rate at which tools report false positives, the kind of errors that particular tools are able to find, and at which stage of development the tools are most effective. In effect, our primary data serve as both a benchmark for program analysis tools and as a guide to the development of future tools that can help developers to become more productive. For this, the tools need to be able to identify true errors conservatively (so that time is not wasted in chasing down false negatives), and notify developers in a non-intrusive manner.

In the rest of this paper we describe the world of program analysis for error detection, the Impendulo framework, our empirical results and lastly some observations on possible other uses for Impendulo and future work.

II. PROGRAM ANALYSIS TOOLS

One way to measure success is to look at commercial impact, and by this measure few techniques from the world of formal methods have been as successful as the use of static analysis for detecting runtime errors in source code. Companies like Coverity [1], Klocwork [3] and PolySpace (now part of MathWorks [5]), to name but a few, have made millions of rands/dollars/euros from selling tools based on the approach of finding errors in code by using a variety of static analysis techniques. However, what these companies

very quickly realized is that although in academia every potential error is considered important, in industry too many potential errors can be a problem. Especially if it turns out that the tool produces too many false positives (where the tool mistakenly thinks an error is possible due to an imprecise analysis) and if it is an embarrassingly simple false positive it will be a deal-breaker. An issue that compounds the problem (of too many errors) is *when* these tools are applied, namely, long after the code is written and even tested. There is a commercial reason for this from the tool vendor's perspective: to demonstrate their ability to find errors, they need to analyse code that is guaranteed to contain many errors, and the more code you analyse, the more likely you are to find errors. Therefore, the issue is now that you are analysing large code bases that contain many possible errors, so the tools narrow their focus to only report errors that are almost without a doubt a real error. The interested reader is referred to [7] for a remarkably frank discussion of these and other issues from the developers of the Coverity tool.

So, we know there is a commercial incentive to run these tools late in the development cycle, but it is also well-known that the later one finds an error the more costly it is to fix. The question is, can we apply these static analysis tools earlier, preferably as the code is being written? The focus of the work described here is an attempt to answer this question. We created a framework in which we can record code snapshots during actual development, plug in different static analyses, and see which one of these reported errors actually occurs during the execution of the code. In order to know which errors can occur, we provide developers with a very precise specification of a problem and a sufficient (to the best of our knowledge) set of test cases; if all tests pass we consider the program "correct", and if not, the reported error(s) are recorded and matched to the output of the static analyses. Currently our framework is based on the analysis of Java programs and uses an Eclipse plug-in that records a snapshot whenever the user saves their work.

The current system supports three static analysis tools for Java: FindBugs [2, 8], Lint4J [4] and Tpgen [11]. FindBugs is a very popular tool for detecting errors statically and is used extensively at Google [6]. It essentially looks for patterns that indicate errors and uses some control and data dependencies, but focuses on intraprocedural analysis (within one method). Lint4J also performs intraprocedural analysis, but looks almost exclusively at errors that occur on one line of code. Tpgen is based on interprocedural symbolic execution and is thus path-sensitive; it also generates test cases for possible errors and only reports those that can be found when executing the code. Our aim was to initially select tools that cover a range of static analysis approaches and thus we have a tool that mostly reports local, or state-based errors (Lint4J), one that is more behavioural or path-based (FindBugs), and one that is fully path-sensitive, even across method calls (Tpgen).

The experiments conducted for this work is based on student programming exercises. To our surprise the static analysis tools found hardly any of the errors reported when the code is run. In the following we describe our framework for conducting the experiments and elaborate on the results.

III. FRAMEWORK

The framework for our experiments consists of two parts: *Intlola*, a data-gathering IDE plug-in, and *Impendulo*, a data analysis and visualisation tool [12]. As they are used during two distinct phases, the components are entirely independent of each other.

A. Gathering the Data

We have implemented a plug-in for the Eclipse IDE platform that operates in the background. It is configured with a location for data storage that can be switched on and off for individual projects, but requires no further interaction. As the user develops their program, the plug-in takes a complete snapshot of the project whenever code is saved, either explicitly by the user or implicitly by the IDE. This happens, for instance, when the user compiles or executes the code. The program code is recorded along with a small amount of meta-data. Collecting the data is fast and therefore unobtrusive, and requires a reasonably small amount of space. The data-gathering plug-in is not tied inexorably to the Eclipse platform, and, in fact, we plan to develop similar plug-ins for other environments. Once the experiment is completed, the stored snapshots are collected in an archive and passed to the next phase of the experiment.

B. Analysing the Data

While individually small, the collected snapshots for a group of participants represent a large amount of raw data, and need to be organized and managed effectively. We have implemented a visualization tool to help with this. The tool has three components:

1. a central data manager that keeps track of different projects, different participants for each project, different snapshots for each participant, and the different files in each snapshot;
2. a modular architecture for managing different forms of analyses; and
3. a modular architecture for visualizing the output of the different analyses and the overall view on a participant or project.

The data manager and visualization modules are, if not trivial, at least straightforward, and the biggest challenge is integrating diverse forms of analyses into a uniform design that makes it easy to add new techniques to the tool. Some basic tools such as the compiler and unit tests are standard fixtures, but to be able to add different forms of analyses, we settled on the following plug-and-play style:

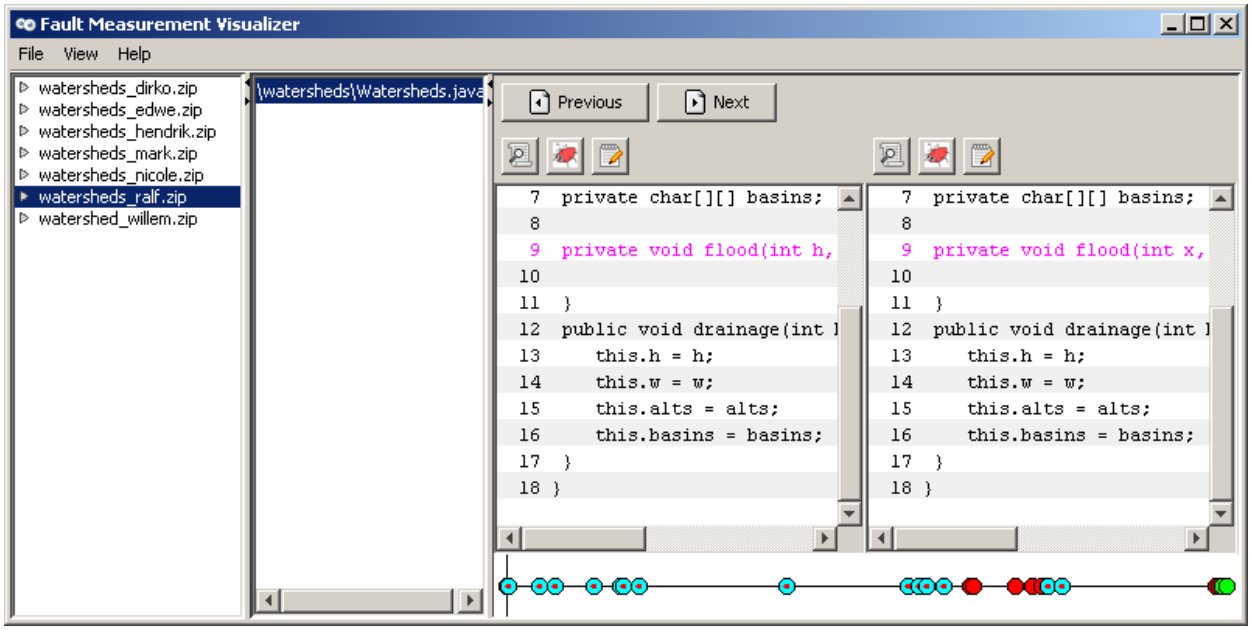

Figure 1 Sample screenshot of the tool

- When the user requests an analysis of a participant's work, the relevant snapshots are unpacked one by one. For each snapshot, a copy of the participant's work is reconstructed.
- The analysis may involve one or more tools, depending on how the system is configured. The location of the unpacked snapshot is passed to each tool in turn.
- The tool performs the analysis and captures and interprets the output. Mechanisms for persistent storage of results are provided, so that the same snapshot need not be analysed more than once by the same tool.
- Tools are expected to produce both verbose and summary output.

As a simple example, the compiler is implemented as such a tool. Given a snapshot location, it invokes the Java compiler and records the compiler's output for the user to inspect. It also parses the output to collect information about the number and nature of the errors.

A screenshot of the tool is shown in Figure 1. This particular visualization shows the difference between two snapshots of the same file. A navigable timeline of all of the snapshots appears at the bottom of the screen.

C. Visualization

The visualization tools can be quite useful. For example, the graph in Figure 2 shows the progress of a particular student at a glance. In this context, each snapshot must pass three stages. First it is compiled, then it is subjected to a small set of tests ("Easy"), and finally to a large, complete set of tests ("All"). The smaller set is convenient for programmers to find initial bugs. For each of the two sets, a snapshot can either produce failures (terminate abnormally), or errors (terminate normally but produce the wrong answer).

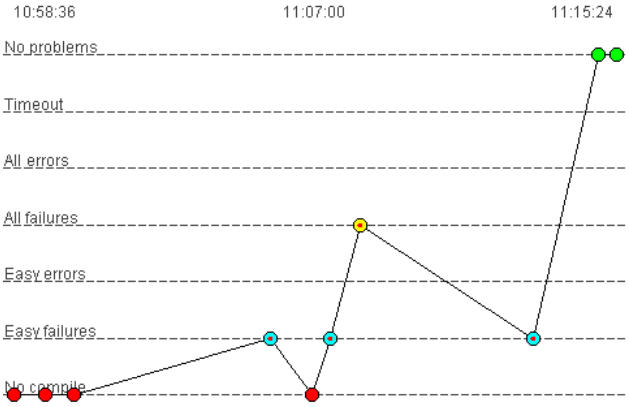

Figure 2 Visualization of a participant's progress

Each dot in the figure represents a snapshot: its horizontal position denotes the time it was taken, and its vertical position is its status. Those snapshots on the first (lowest) line do not compile at all, those on the next line compile but produce "Easy" failures, those on next line produce "Easy" errors, and so on. Those on the top line pass all the tests. If a snapshot appears at a certain level, this means that it has successfully passed all of the levels below it.

D. Extensibility

The architecture allows users to develop and add their own visualizations, analysis tools and tool chains — tools that invoke a sequence of other tools. Also, it remains flexible enough to accommodate almost any kind of analysis.

IV. Experimental Results

We have conducted two trial runs of the system. The first

Table 1 Overview of collected data

Problem	*P*	*S*	*P/S*	Aver. interval	Total time
`welcome`	3	256	85.3	36.5	5h12m
`kselect`	19	835	44.0	110.7	22h47m
`triangle`	8	199	24.9	193.7	8h22m
`watershed`	7	495	70.7	100.7	8h33m
Total	**37**	**1529**	**41.3**		

trial consisted of one problem completed by three fourth-year Computer Science students, while the second trial consisted of three problems completed by 21 third-year Computer Science students. An overview of the data collected is shown in Table 1. The *S* column shows the number of students, and the *P* shows the number of snapshots. This is followed by the average number of snapshots per student, the average number of seconds between snapshots, and the combined time spent on the problem by all of the students.

A. Problem Selection

The choice of problem is a critical aspect of the experiments. The problems were chosen to be challenging but doable within one hour each. We have tried to avoid problems that require deep mathematical insight and concentrate on more traditional computer science problems.

- In the `welcome` problem the programmer is given an input string and asked to count how many times a second, fixed string appears as a non-contiguous substring of the first.
- `kselect` presents the programmer with an array of integers that belong together in pairs (i.e., the first two numbers form the first pair, the next two numbers form the second pair, and so on); the task is to sort the pairs lexicographically.
- The `triangle` problem asks the programmer to find the longest path from a specified vertex in a triangle-shaped acyclic directed graph (with a potentially exponential number of paths).
- Lastly, in the `watersheds` problem the programmer is given a two-dimensional integer array of function values; the task is to identify the local minima and the "closest" minimum for each array element according to a fixed definition of distance.

The `welcome` and `watersheds` problems were taken from the annual Google Code Jam programming, the `triangle` from a puzzle website, and the `kselect` problem is original. (It appears in the appendix.)

Generally speaking, we tried to make the students think in a programmatic way, and to make them exercise their technical programming skills. For example, because the numbers in the `kselect` problem are paired, the students are not able to use a standard library sorting routine. We expected them to make different kinds of errors both across the problems, and within a single problem, and hoped that the analysis tools would be able to identify some of these errors.

B. Results

To our surprise the results from the tools were quite dismal. Lint4J reported basically no errors on any of the examples, FindBugs reported some possible errors but not the ones that occurred when running the code, and Tpgen reported a few errors that actually occurred but also missed many. Note that the code fragments were relatively small and all the tools ran within seconds; we also tried to fine-tune each tool to make it perform to the best of its ability.

One of the primitive visualizations provided by our tool allows the user to view the raw output of the analysis tools. A typical sample of code and the corresponding output are shown in Figures 3 and 4; both trimmed for the sake of brevity. FindBugs complains about a futile assignment in line 32, while Tpgen reports a null pointer exception on line 13 (a false positive) and an array-out-of-bounds error on line 46. In fact, most of the Easy tests produced errors because of an array-out-of-bounds error on line 33.

Lint4J can be somewhat exonerated because the errors that did occur often, namely null-pointer dereferences and array index out of bounds accesses, are not errors that it looks for. Still it seems that even these novice programmers didn't make the kind of silly mistakes Lint4J catches. FindBugs on the other hand should have found these errors but didn't. We believe it is due to FindBugs being tuned for large code bases and thus it doesn't report errors unless it is quite certain it will be triggered during execution. Tpgen, which is the tool that most closely follows real executions caught the most real bugs, but it did suffer from scalability issues since some of the errors occur at path depths beyond which it can scale to. However it is just a research prototype and we believe that it shows that path sensitive analyses is an important avenue for future work in applying static tools early in the development cycle.

V. PROGRAMMER BEHAVIOR

The visualizations from Figures 1 and 2 give one an interesting insight into how a developer goes about coding a solution to a problem. Note for example that according to this output, the developer made some syntax errors to begin with then got a running program, but it failed the easy tests, a few minutes later it passed the easy tests but then failed the complete test suite. However when fixing the reason for failing the complete test suite, an error was introduced that made it fail the easy suite, fixing this eventually led to a working program. A more interesting case was where a student introduced an array indexing error, but instead of just thinking about the error and fixing it, the student started changing array bounds "randomly", which we could observe

FindBugs:
```
Dead store to $L6 in kselection.KSelection.kselect(int, int[])
At KSelection.java:[line 32]
```

Tpgen:
```
Running 5 test(s)...
1) Solution (0)
   REAL? Caught expected exception: java.lang.NullPointerException
   Occurred at kselection.KSelection.kselect:13
...
3) Solution (2)
   REAL? Caught expected exception: java.lang.ArrayIndexOutOfBoundsException: 0
   Occurred at kselection.KSelection.kselect:46
```

Easy tests:
```
Time: 0.063
There were 13 failures:
1)  java.lang.ArrayIndexOutOfBoundsException: 12
    at kselection.KSelection.kselect(KSelection.java:33)
2)  java.lang.ArrayIndexOutOfBoundsException: 12
    at kselection.KSelection.kselect(KSelection.java:33)
...
FAILURES!!!
Tests run: 15,  Failures: 13
```

Figure 3 Typical output produced by FindBugs and Tpgen, and error output produced by the Easy tests.

by many frequent saves but the error was still there; after a while it obviously occurred to the student that it might be worth thinking about the problem which one can observe by a few minutes gap before the next save, after which the error went away.

At Stellenbosch we have empirically observed that students without a proper programming background struggle with our Honours course (first postgraduate year after the 3 year B.Sc. Computer Science degree). We now plan to use the Impendulo framework to determine which students might require extra programming classes before starting Honours. Since these are small numbers of students we can do the classification by hand, but in the long run we hope to automate the approach. In addition we also have a project to evaluate undergraduate students programming proficiency throughout their studies to measure progress.

One would of course also be able to use the framework to judge candidate's programming skills during job applications in an objective way. For example they can develop the code outside of the stressful interviewing environment (at home for example) and we just capture and analyze the snapshots.

VI. Related Work

There is a vast amount of related research to ours, but the most closely related is that of the Marmoset system [10]. As with our framework it can record programming iterations within Eclipse, but their goal is not to analyse the types of errors found, but rather to analyse students' programming behavior and to provide a more stimulating learning environment for novice programmers. One should be able to extend Marmoset to do the same analysis we perform.

VII. Conclusions

One can justifiably argue that the poor results obtained so far on the suitability of static analysis tools early in the development life-cycle suffers from many threats to validity, most notably the kind of programs the students wrote and the small sample of tools. However, we believe the main contribution of our work is not the results we obtained so far, but rather the framework we have created. We intend to open this framework to others to plug in new analyses and to run their own experiments. The ultimate goal is to establish a repository of program snapshots that can form a benchmark for tool developers.

Note that we have focused here on using static analysis tools in our framework and tried to determine how well they predict real errors, but one can also analyse a myriad of other aspects in this controlled environment. Perhaps one of the most interesting would be to analyse the impact of poor specifications on the type of errors people make. This can be done by giving a control group a proper specification and another group a less than adequate one and then to see which errors are made and analysing the difference. Other possibilities include evaluating different test generation techniques, say, based on obtaining high code coverage, for their capability to find real errors.

Our current framework only captures results from our running of the tool suite. We plan to next release the tools to others to also capture snapshots so that we can obtain much more data to measure tool and developer performance.

```
 5 public int kselect(int k, int[] V) {
 6   int temp1 = 0, temp2  = 0;
10   if (k < 0) { k = k*(-1); }
13   if (k > ((V.length)/2)) { return 0; }
18   for (int i = 0; i < V.length; i += 2){
19     for (int j=i+2; j < V.length; j +=
2)
20       if (V[j] <= V[i]){
21         temp1  = V[i];
22         temp2  = V[i+1];
23         V[i]   = V[j];
24         V[i+1] = V[j+1];
25         V[j]   = temp1;
26         V[j+1] = temp2;
27       }
28   }
31   for (int i = 0; i < V.length; i += 2){
32     int j = i;
33     while (V[i] == V[i+2]) {
34       if (V[i+1] > V[i+3]){
35         temp1  = V[i];
36         temp2  = V[i+1];
37         V[i]   = V[i+2];
38         V[i+1] = V[i+3];
39         V[i+2] = temp1;
40         V[i+3] = temp2;
41       }
42     i = i+2;
43     }
44   }
46   return V[k];
47 }
```

Figure 4 The source code corresponding to Figure 3

Acknowledgements

This work was supported by the Stellenbosch Unit of the Telkom-Nokia-Siemens-Networks Centre of Excellence in ATM and Broadband Networks and their Applications.

[1] Coverity. www.coverity.com
[2] Findbugs. findbugs.sourceforge.net
[3] Klocwork. www.klocwork.com
[4] Lint4j. www.jutils.com
[5] Polyspace. www.mathworks.com/products/polyspace
[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In Proc. 7th Workshop Program Analysis for Software Tools and Engineering, pages 1–8, 2007.
[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *CACM*, 53(2):66–75, 2010.
[8] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
[9] National Institute of Standards and Technlogy (NIST), Department of Commerce. Software errors cost U.S. economy $59.5 billion annually. NIST News Release 2002–10, 2002.
[10] J. Spacco, D. Hovemeyer, and W. Pugh. An eclipse-based course project snapshot and submission system. In *Proc. 3rd Eclipse Technology Exchange Workshop (eTX)*, 2004.
[11] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proc. 2007 Intl. Symp. Software Testing and Analysis*, pages 97–107, 2007.
[12] W. Visser and J. Geldenhuys. Impendulo: Debugging the Programmer. To appear in *Proc. Intl. Conf. Automated Software Engineering*, 2010, tool paper.

APPENDIX: THE K-SELECTION PROBLEM

Given two pairs of integers (a,b) and (c,d) we can compare them by first comparing the first component and then the second. For example,

(a,b) < (c,d) if and only if *a<b* or (*a=b* and *c<d*).

The *k*-selection problem is to find the *k*-th smallest pair in a list of pairs. When *k<0*, the task is to find the *-k*-th largest pair. If *k=0*, or if the absolute value of *k* is greater than the length of the list, we shall say that the answer is zero. Given the list

(3,1)	*(4,1)*	*(5,9)*	*(2,6)*	*(5,3)*	*(5,8)*
1	2	3	4	5	6

Then (2,6) < (3,1) < (4,1) < (5,3) < (5,8) < (5,9) and we know that the

1-th smallest pair (2,6) is in position 4
4-th smallest pair (5,3) is in position 5
-1-th smallest pair (5,9) is in position 3 and
-4-th smallest pair (4,1) is in position 2.

The task is to write a routine in the `KSelection.java` class that accepts two parameters: (1) the value of k and (2) the list of integer pairs, stored in a single array. Your routine must return the position of the k-smallest pair as an integer.

```
public int kselect(int k, int V[])
```

For example, if `A` is an array containing the values `[3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8]`, then

`kselect( 1, A)` should return `4`
`kselect(-3, A)` should return `5`
`kselect( 7, A)` should return `0`

Willem Visser holds a PhD in Computer Science from the University of Manchester (1998) and is currently a Full Professor at Stellenbosch University. He is on the Editorial Board of ACM Transactions on Software Engineering and Methodology and on the steering committee of the Automated Software Engineering conference.

Jaco Geldenhuys holds a DTech in Software Systems from the Tampere University of Technology (2011) and is a Senior Lecturer at Stellenbosch University. His research interests include formal methods, model checking, software engineering, and automata and language theory.