Towards Checking the Usefulness of Verification Tools

Willem Visser and Jaco Geldenhuys

Computer Science Division
Department of Mathematical Sciences
University of Stellenbosch, South Africa
{wvisser,jaco}@cs.sun.ac.za

Abstract. Building verification tools is what researchers in the field are good at, but evaluating if they are actually useful and useable is often a much harder problem. We propose a very simple idea to address this problem, namely, a framework in which we capture what a programmer does during a programming exercise and mechanisms to then allow us to incorporate verification (including testing) tools to analyze the resulting code snapshots. The ultimate idea is to build up a very diverse set of code benchmarks on which verification tools can be evaluated. However unlike current benchmarks that are somewhat cherry-picked for showing off certain tools' strengths, this one will be based on actual code being captured during programming.

1 Introduction

This work stems from our desire to capture the errors that programmers make during the actual code development; more precisely we want to know *exactly* when an error is introduced and when is removed. In addition we want to capture the *type* of the errors. This requires a way to closely observe programmers as they develop code. For example, looking at revision repositories is not at a fine enough level of granularity: by the time code is committed many errors may already have been removed and we don't really know how much time was wasted on finding and repairing the code. We therefore take a code snapshot whenever a developer saves a file.

Once we have the snapshots we can analyze each snapshot to see which errors occurs. Of course one needs to know what the code is supposed to do. In this first iteration of our approach we fix the domain to observe students doing well-specified programming puzzles, akin to Google Code Jam or the ACM programming competitions. We can thus construct what we refer to as the *perfect* oracle, which concretely consists of a large test suite that the code must pass to be considered correct.

The code snapshots plus the test suite makes it possible to classify each snapshot in terms of which snapshots are correct and which ones have errors. A natural extension of this approach is to not just run the code to determine which errors are present, but to also run tools on the code to try and find the errors

automatically. Our current framework only supports one type of tool, namely static analysis tools for runtime error detection. However the results obtained already seem to be in stark contrast to common beliefs about the usefulness of these kinds of tools: we found that even well-respected tools like FindBugs [2, 8, 6] didn't find a single one of the errors that actually occurred (we also tried Lint4J [4] and Tpgen [10]).

In the following we describe our current framework in more detail and also discuss the evaluation of the static analysis tools. Note that although our current system only works within the Eclipse IDE, it can trivially be extended to other languages and IDEs. We currently focus on bug finding tools, but nothing prevents us from also adding verification tools to analyze the snapshots to check which ones are easy to prove correct or invalid, the level of annotations required, the need for human intervention, etc.

2 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 [11]. As they are used during two distinct phases, the components are entirely independent of each other.

2.1 Gathering the Data

We have implemented a plug-in for the Eclipse IDE platform that operates in the background. All that is needed is for the user to switch on recording for a particular project. 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.

2.2 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:

- 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.

2.3 Visualization

The visualization tools can be quite useful. For example, the graph in Figure 1 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). Note that the large set also checks the scalability of the code hence we additionally record if tests timeout.

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.

2.4 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.

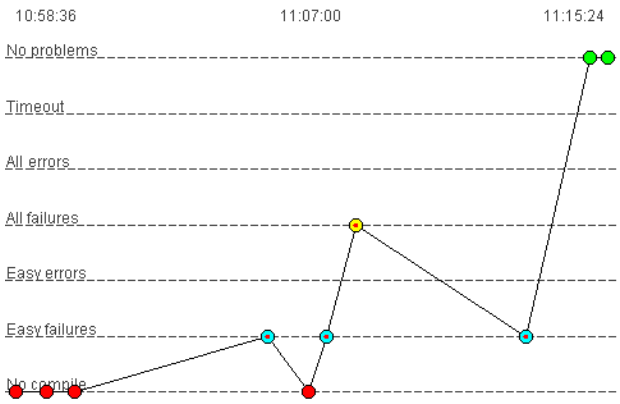

Fig. 1. Visualization of a participant's progress

3 Experimental Results

We have conducted several trials runs of the system. 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 total time spent on the problem by all of the students combined.

Table 1. Overview of collected data

Problem	S	P	P/S	Average 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
`watersheds`	7	495	70.7	100.7	8h33m
`triangle2`	13	452	34.8	162.5	9h53m
`LRS`	42	2136	50.9	93.2	37h41m
`welcome2`	4	677	169.3	151.3	8h55m
Total	**96**	**5050**	**52.6**		

3.1 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 belongs 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. As an example of what a problem specification looks like we added the `kselect` specification to the Appendix.
- 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). `triangle2` refers to the same problem given to a different set of students.
- `LRS` is a problem in which the programmers needed to find the longest recurring substring. This problem was given to a class of first year students and contains the most snapshots of all.

– 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. `watersheds2` refers to the same problem given to a different set of students.

The `welcome` and `watersheds` problems were taken from the annual Google Code Jam programming competition, 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 the students to make different kinds of errors both across the problems, and within a single problem, and we hoped that the analysis tools would be able to identify some of these errors.

3.2 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 is shown in Figures 2 and 3; 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 index-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 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 are an important avenue for future work in applying static tools early in the development cycle.

```
 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 }
```

Fig. 2. The source code corresponding to Figure 3

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
```

Fig. 3. Typical output produced by FindBugs and Tpgen, and error output produced by the East tests.

4 Related Work

There is a vast amount of related research to ours, but the closest is the Marmoset system [9]. 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.

5 Conclusions

The framework we have shown here and the initial experiments with static analysis tools indicate that the snapshots will be a fruitful source of benchmarks to evaluate testing and verification tools. The current focus has been on testing tools, and given that these tools can be evaluated on fault-detection capability, evaluating these tools are straightforward. Verification tools on the other hand will be much harder to evaluate. One nice feature though is that the programs are relatively small, and thus should not pose any scalability issues for the verification tools.

References

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. 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.
10. 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.
11. W. Visser and J. Geldenhuys. Impendulo: Debugging the Programmer. To appear in *Proc. Intl. Conf. Automated Software Engineering*, 2010, tool paper.

6 Appendix

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,

$$(a, b) < (c, d) \quad \text{if and only if} \quad a < b \quad \text{or} \quad a = b \wedge c < d.$$

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

$$\underset{1}{(3,1)} \quad \underset{2}{(4,1)} \quad \underset{3}{(5,9)} \quad \underset{4}{(2,6)} \quad \underset{5}{(5,3)} \quad \underset{6}{(5,8)}$$

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

$$\begin{aligned} \text{1-th smallest pair } (2,6) &\text{ is in position } 4 \\ \text{4-th smallest pair } (5,3) &\text{ is in position } 5 \\ -\text{1-th smallest pair } (5,9) &\text{ is in position } 3 \text{ and} \\ -\text{4-th smallest pair } (4,1) &\text{ is in position } 2. \end{aligned}$$

Your 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`

You may assume that

- the list will contain n pairs, where $1 \leq n \leq 10000$,
- none of the pairs are equal, and
- each integer x in the list will satisfy $0 \leq x \leq 10^6$.