Impendulo: Debugging the Programmer

Willem Visser
Computer Science Division
Department of Mathematical Sciences
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa
visserw@sun.ac.za

Jaco Geldenhuys
Computer Science Division
Department of Mathematical Sciences
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa
jaco@cs.sun.ac.za

ABSTRACT
We describe the Impendulo tool for fine-grained analyses of programmer behavior. The initial design goal was to create a system to answer the following simple question: “What kind of mistakes do programmers make and how often do they make these mistakes?” However it quickly became apparent that the tool can be used to also analyze other fundamental software engineering questions, such as, how good are static analysis tools at finding real errors?, what is the fault finding capability of automated test generation tools?, what is the influence of a bad specification?, etc. We briefly describe the tool and some of the insights gained from using it.

1. INTRODUCTION
One of the fastest growing research areas in software engineering is that of data mining software repositories [3] to try and learn almost anything: from how many null pointer dereferences occur to what bug characteristcs make it more likely to be fixed in a future release. We wanted to do something similar, but with the more modest goal of finding out where we should put our energy when it comes to building tools that will detect errors that actually makes a difference. However, it was obvious that mining repositories has two immediate drawbacks: (1) we don’t know which bugs existed and (2) we don’t know which bugs were fixed. The reason for this is that bug tracking tools don’t contain precise information and the time (and code changed) granularity of commits vary dramatically and thus don’t accurately reflect all the changes. The simple way to avoid these two problems are to collect our own data in a setting where we know what a “correct” program is and where we force commits at every save. The tool we describe here works on Java code and is an Eclipse plugin that saves a snapshot of the user’s work every time they save (including a timestamp) and in addition we provide a large test suite which can be run on each snapshot to determine whether the code contains errors. Running the test suite gives immediate feedback to the developer, but we also use it to determine afterwards which errors occurred, how long they persisted before being fixed, which errors tools could detect, etc.

In a nutshell we are creating our own repositories to mine, where we have near perfect information. Of course this setting also has its own issues, namely, the programs and the programmers we are studying may not be representative and to a lesser extent our test suites might not be as good as we think. To mitigate the first concern we are building the framework to be open for anyone to use and thus we believe we will quickly leverage enough information to overcome any specific biased sample. Currently we are using programming exercises akin to Google Code Jam exercises that we hand to developers while we monitor their progress. These kinds of examples are easy to specify precisely and to generate vast amounts of test data for.

Once we have the repository information it can be analyzed. Our initial goal was just to try and classify the types of errors encountered and to see how long they persisted before being fixed. The tool supports a graphical view for looking at snapshots ordered in time with text highlighting and alignment to show differences. In addition one can also observe the errors if any) that occurred when compiling and running the snapshot against the test suite. We use a simple classification based on whether an error is exceptional behavioral (i.e. uncaught exception) or simply one of the tests failed (in our case a JUnit assertion was violated). We would have liked to conclude from this whether errors are generic mistakes or whether they are fundamental (requirements misunderstanding or design errors). Unfortunately, some fundamental errors in understanding can exhibit themselves as exceptions being thrown. Once we have enough data we hope to be able to answer how accurate this classification scheme is in practice. Note that this is an important question, since the whole field of static analysis for runtime error detection is built on the premise that there are large numbers of generic errors and the field of requirements (or specification) based testing on there being lots of functional errors. Our tool gives a way of answering this question and thus allows a way to allocate testing/analysis resources.

Probably the most important aspect of our framework is that we provide an oracle (test suite) that is considered perfect, i.e. if a program passes the oracle it is considered correct. Maybe more importantly, if it fails to pass the oracle, the errors are “real”. Hence, by design, there are no false positives when an error occurs when run against the oracle. This characteristic allows us to evaluate a number of interesting questions in software engineering, most notably, since
2. INITIAL EXPERIMENTS

We collected data from two trial runs of our tool. The first trial consisted of one Google Code Jam 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. One of these (“watersheds”) was another Code Jam example, “kselect” was our own invention and “triangle” was found on the web (and is not the classic triangle problem from the testing literature).

An overview of the data collected is shown in Table 1; S shows the number of students, and P is 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. Note for example how the more mature students (“welcome”) saved more often than the third year students. We don’t know why, but we would like to believe they work in a more test driven style since they have had more exposure to software engineering principles.

As stated earlier we wanted to see how well static analysis tools predicted the errors that actually occurred. We first tried FindBugs [6, 1] and although many people consider this tool to be the best available for Java, it found essentially no errors in any snapshot. Note, more than 80% of the errors in the code were of the kind FindBugs can detect (namely, ArrayIndexOutOfBoundsExceptions and NullPointerExceptions). We were quite surprised therefore at this result. Next we tried a tool we believed is more prone to giving false positives, namely Lint4J [4], but this tool produced absolutely zero warnings. Lastly we tried TPGEN [8] that is a research tool we believed is more prone to giving false positives, but in their case it is more metrics such as coverage, number of errors, etc. and not actual code.

3. RELATED WORK

The related work to ours is of course rather too plentiful to enumerate here, so we will give selected highlights we believe is most closely related. Firstly, the Mining Software Repositories (MSR) Conference [3] has been running for more than 5 years and contains a wealth of information on related work. Our work differs from most of this in the sense that we are creating our own repositories, rather than mining open source or projects from large companies (such as Microsoft). Probably the most closely related project to ours is Marmoset [7], which also consists of an Eclipse plugin to record snapshots, but they use it in a teaching context to determine how to help students become better programmers. In theory our work can be added to Marmoset. Lastly the HackyStat project [2] is similar to ours in that it collects data from real projects via development environment plugins, but in their case it is more metrics such as coverage, number of errors, etc. and not actual code.

4. CONCLUSION

Our long-term vision is a programming assistant that monitors the activity of the programmer and intervenes when it detects possible mistakes. Such a tool would be aware of the idioms programmers use, and how they deviate from these norms. It would employ model checking and similar techniques to analyze and detect potential bugs on-the-fly; issue discrete warnings and suggest corrections, and, perhaps most importantly, generate test cases for later use. It would also learn the programmer’s behavior and adjust itself accordingly. In the short term our goal is to allow other users to replicate our experiments in their own setting, since the threats to validity produced by using a system like this can only be properly mitigated by vast amounts of data from all over the world.

Besides the future work already mentioned (classifying errors and evaluating test generation techniques) we also hope to conduct experiments where we provide users with inadequate or ambiguous specifications to determine what errors that may lead to. Other options are to look at the influence of programming environments (our framework can easily be ported to other IDEs), programming languages and even programmer background.

5. REFERENCES


---

Table 1: Overview of collected data

<table>
<thead>
<tr>
<th>Problem</th>
<th>S</th>
<th>P</th>
<th>P/S</th>
<th>Average interval</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>welcome</td>
<td>3</td>
<td>256</td>
<td>85.3</td>
<td>36.5</td>
<td>5h12m</td>
</tr>
<tr>
<td>kselect</td>
<td>19</td>
<td>835</td>
<td>44.0</td>
<td>110.7</td>
<td>19h46m</td>
</tr>
<tr>
<td>triangle</td>
<td>8</td>
<td>199</td>
<td>24.9</td>
<td>193.7</td>
<td>8h22m</td>
</tr>
<tr>
<td>watersheds</td>
<td>7</td>
<td>495</td>
<td>70.7</td>
<td>100.7</td>
<td>8h18m</td>
</tr>
<tr>
<td>Total</td>
<td>37</td>
<td>1529</td>
<td>41.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

we know which errors can occur, we can tell how good bug finding tools really are. Often times these tools are evaluated on small contrived examples, on the one hand, or on large code bases where we don’t know which errors are being missed, on the other hand. In section 2 we give some results of applying static analysis tools in our setting.
APPENDIX

A. DEMONSTRATION

The demonstration will mainly focus on the analysis phase of the tool, since the data capture phase is rather trivial and just involves a user saving via the Eclipse IDE. We will however show what a specification looks like, see for example section C as well as an example of a test suite.

We will show all the features of the tool, focussing on navigating between the snapshots from a user and also looking at the output results (when the code is run) and the analysis results (when the tools are run on the code). The tool provides the capability for the user to also add annotations to each snapshot to maybe record what kind of errors occurred etc. See figures 1 and 2 for example screen shots from the tool.

In the last phase of the demonstration we will show how to add a new tool to the framework, to allow others to plug in their favorite analysis tool they might like to compare to the existing tools in the framework. Our main goal with the demonstration is for experts from the field to suggest ways we can mine the data. Clearly collecting the data is easy, but analyzing the vast amounts of the raw data to extract useful information is difficult. Plugging in new tools is one option to make it useful to the community and another is to add to each snapshot.

The current status of the system is that it is not yet open source, but we are handing it out to anyone that wants to try it. We plan to opensource it before the conference. If all goes well we will have our first outside users (from the University of Nebraska at Lincoln; Sebastian Elbaum) run an experiment in capturing data in August, in time to be demonstrated at the conference.

B. SCREENSHOTS

Figure 1 shows a timeline of the development activity of one user on a specific example. During the demonstration we hope to show a variety of these, to illustrate how one can see patterns in this type of graphical view that can be correlated to correct and incorrect programs. The vertical axis indicate what happened at each save point and the horizontal is the timeline. One can see saves that lead to compilation errors, cases in which a simple test suite found exceptions (“Easy errors”), “Easy failures” refer to the case where the simple test suite computed the wrong results (but no exceptions were thrown), and then the same two options for the complete test suite (referred to as “All” here) and finally correct programs on the top line. Even in this simple example one can see the user fails back every now and then and then back eventually getting it right. In some other examples it is clear a user is making random changes with very frequent saves but each time exceptions are thrown; which clearly extremely bad programming practice that a tool can easily pick up and report on.

Figure 2 is the main view of the Impendulo tool and shows how one can see exactly what changed between different versions. The timeline at the bottom right is color coded to show which snapshots had compilation errors, static warnings, execution errors and lastly worked correctly. One can navigate left and right on the timeline and then the code changes to show the diff which is color coded in magenta to make it more obvious. Above each code fragment there is also a button to show the output of that run, the static errors detected and annotations that the user of the tool can add to each snapshot.

C. KSELECT

We give an example of the specification we give to the programmers. Specifically we show here the one homegrown example we created for our first set of experiments.

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,

\[
(a, b) < (c, d) \quad \text{if and only if} \quad a < b \quad \text{or} \quad a = b \land c < d.
\]

The \(k\)-selection problem is to find the \(k\)-th smallest pair in a list of pairs. When \(k \leq 0\), 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. For example, given the list

\[
\begin{array}{cccccc}
(3, 1) & (4, 1) & (5, 9) & (2, 6) & (5, 3) & (5, 8) \\
1 & 2 & 3 & 4 & 5 & 6
\end{array}
\]

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.

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.

```java
public int kselect(int k, int values[])
```

For example, if \(A\) is an array containing the values

Figure 2: Visualization of a participant’s progress
\{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8\},

then

\[
\text{kselect}( 1, A) \quad \text{should return} \quad 4 \\
\text{kselect}(-3, A) \quad \text{should return} \quad 5 \\
\text{kselect}( 7, A) \quad \text{should return} \quad 0
\]

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 \( 0 \leq x \leq 10^6 \).