Carnegie Mellon University
School of Computer Science
Master of Software Engineering

Application of Eclat

Automatic test input generation tool for Java

April 24, 2006

Analysis of Software Artifacts

Final Project Report

Team Serendipity

Majid AlMeshari
Hyunwoo Kim
Lucia de Lascurain
Steven Lawrance
Ricardo Vazquez

Table of Contents

Table of Contents 2
Introduction 3
  Goals 3
  MSE Studio Project 3
  Outline 3
Description of Eclat 4
  Motivation 4
  How does Eclat work? 4
    1. Derive a program's operational model 4
    2. Generating test cases 5
    3. Classifying and reducing test cases 5
    4. The output 5
Description of Our Experiment 6
  Our Approach 6
Analysis of Experimental Data 8
  Experiment 1: Code review 8
  Experiment 2: Test suite that covered some of the errors 8
  Experiment 3: Test suite that covered all of the errors 10
Lessons Learned 12
  Benefits 12
  Drawbacks 13
Quality attributes of the tool 14
  Usability 14
  Documentation 14
  Performance 14
Conclusions 15
Appendix 16
References 16

Introduction

Team Serendipity evaluated Eclat – an automatic test case generation tool for Java programs – to see if it can help us save time when creating test cases that can prove if a targeted component has been implemented properly or not. In this report, we discuss the strengths and weaknesses of this tool in the context of how it adds value to our MSE studio project.

Goals

If we decide to use this tool in our project, we will achieve the following goals:

- Apply some knowledge from one of the core courses directly into our studio project, which is one of the main MSE goals.
- Include Eclat in our testing process. Thus, the testing can include both human approaches, such as reviews, and automatic approaches. This will enable us to compare the effectiveness of the two by comparing the number of defects that each approach catches.
- Save some of the team's time spent on the entire testing process. This will enable us to focus on other parts of the project such as enhancing the architecture or improving the readability of the code.

MSE Studio Project

The Configuration Assistant project will help security designers and security component installers at the Bosch Corporation save time through a consistent, computerized process. Using rule-based heuristics and historical data, the project will determine the optimal placements of security components on an input floor plan, shortening their existing process from five weeks to several minutes. Presently, security designers are not always aware of the cost and legal implications of every security component placement. The project will provide this information to the security designers along with coverage volumes and security levels.

Outline

First, this paper briefly outlines this tool. The second part describes the experiments conducted. The third part discusses the lessons learned. The fourth part suggests future improvements. Lastly, it ends with conclusions.

Since this project is about evaluating the tool, we will leave out any tool installation or usage information.

Description of Eclat

Motivation

Writing exhaustive test cases is time consuming. It requires a deep understanding of the targeted software components. This understanding involves knowing the operational models of the components being tested such as global and method-specific invariants. This knowledge typically is not documented thoroughly or, in many cases, at all. The result is that test engineers write test cases that are not likely to reveal all the bugs in a given system. Furthermore, these test cases can overlap and thus increase the time needed for writing and running test cases with little added value.

Eclat tries to help testers by doing the following:

- Deriving the operational model of a program using a proof technique through Daikon, which is an invariant detector.
- Generating test cases by taking the targeted component, which is always a Java source file, as well as a correct execution of this component, which is always a set of test cases
- Classifying and reducing the set of test cases to only represent cases that are likely to reveal a bug in the model rather than cases that represent legal or illegal uses of the methods

After performing the above, test engineers have a set of test cases that they can refine. Tests with false positives can be removed, and tests with no false positives can remain as-is.

How does Eclat work?

We will discuss how Eclat carries out each of the steps mentioned above.

1. Derive a program's operational model

Eclat uses Daikon, which is an implementation of dynamic detection of likely invariants. Invariants can be program-global, meaning that they are true at all program points, or local, meaning that they are true at certain points of the program such as before the beginning of a method or upon finishing the method's execution. Depending on the likelihood that the invariant is globally true throughout the program, Eclat gives each invariant a number called the confidence measure. This measure is used to classify bugs in the system, so bugs that violate invariants with a higher confidence measure are classified as severe. The result of all this process is called the "approximate oracle."

Note that Daikon might fail to discover all invariants and that might result in inadequate generation of test cases. In this case, the test engineer needs to check that and ensure that all invariants have been discovered.

2. Generating test cases

Eclat takes the program, the operational model discovered in the previous step, and a set of use cases that represent a correct use of the program. After that, Eclat uses the operational model to instrument the program with Java Modeling Language toolset. The result is code whose invariants can be checked during execution. This instrumentation is transparent such that violations of invariants don't alter the behavior of the program.

Eclat also generates new test cases by constructing random sequences of method calls whose arguments are chosen from a pool of objects.

3. Classifying and reducing test cases

By running the tests generated in the previous step and capturing all the violations, guided by the approximate oracle, Eclat recognizes the patterns of the tests. The tests that violate a specific invariant are grouped together and only one test case is chosen to represent the whole group. This makes the test process efficient by omitting the overlapping cases.

4. The output

Eclat can produce and output tests, XML, or JUnit test suites. It contains a list of inputs along with the invariants they violate and a brief explanation of the reason each input is considered fault-revealing.

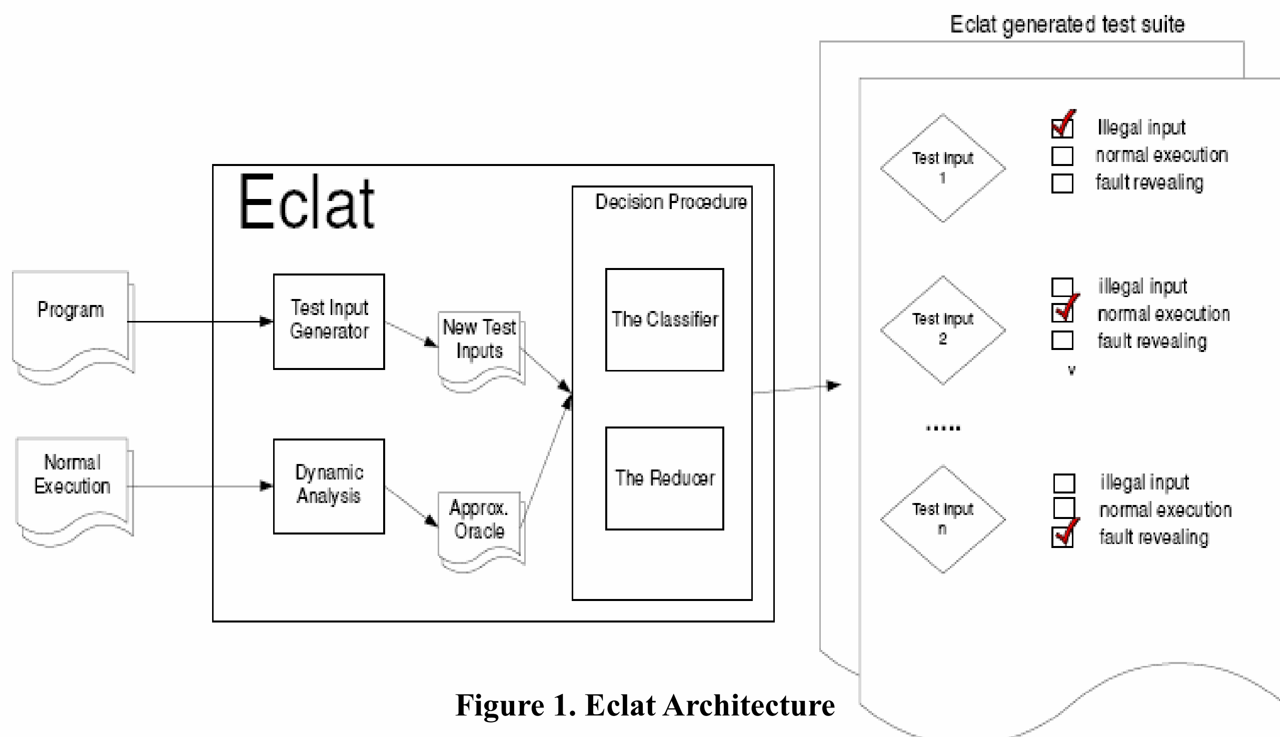

Figure 1. Eclat Architecture

Description of Our Experiment

Since we haven't started the implementation phase of our project yet, we tested this tool against one of the toys (small prototypes) that we have implemented so far: voxel-based volume calculation. This calculation will be used to determine the optimal placements of security components such as motion sensors and smoke detectors. These components have a coverage that our system will try to optimize. Because this toy represents some behavioral aspects of the system we are trying to implement, running Eclat with the voxel-based toy will give us some impression about its performance against the final system.

Our Approach

The approach that we followed was:

- We took one class and developed a set of use cases that we thought were comprehensive. We wanted to see if Eclat would spot any defects that this suite failed to spot.
- We injected five defects into this class:
    1. The first defect we inserted assigned an incorrect value to a variable that contained the size of an array in our class. The variable was then used to check for array bounds errors. With an incorrect initial value, the mapping between this variable and the actual size of the array was different in most cases.
    2. The second defect we inserted consisted of changing a check for an incorrect variable in an if-statement that also controlled the correct access to array sizes. This check was in the accessCheck() method of our class.
    3. The third defect removed a check for the bounds of the arrays completely by removing the call to the accessCheck() method in one of the array uses.
    4. The fourth defect did not add any array bounds errors, but instead assigned an incorrect value to the contents of the arrays within the color() method. The system would lose this value and not be able to count the volumes later.
    5. The last defect was an off-by-one problem in a while.
- We performed the following three experiments:
    1. Gave this class to one team member to review to see how effective code reviews are compared to unit testing in general and find which errors are easy to find by a review but not necessarily so in tests.

2. Created a test suite that captures some of these errors but not all of them. The goal of this task was to check if Eclat can capture all errors injected or if it would be affected by the input test suite.
3. Created a test suite that captures all these errors. The goal of this task was to check if Eclat would capture all of these errors as well as any other errors such as the errors that we might have injected indirectly by injecting the original defects.

The results of this experiment as well as our analysis are discussed in the following section.

Analysis of Experimental Data

Experiment 1: Code review

In this experiment, one team member, unfamiliar with the code, was asked to review it to find any defects. No information of where the defects we injected were nor how many they were was given to him. The results are displayed in the following table.

Injected errors found	3
Which injected errors were found	1, 2, and 4
False negatives	2
False positives	0
Unexpected errors found	0
Time spent in review	0.25 hours

Table 1. Results of the first experiment

The errors found by the reviewer were all errors where, by the names of the variables, it was clear that the code did not follow the intention of the user. For example, in defect 1, a variable called *x* is assigned to *y_size* instead of *x_size*. These errors would be hard to find through some analysis techniques, but could be detected by using invariants and with some example of the original intent, like Eclat does with the sample test cases.

Experiment 2: Test suite that covered some of the errors

For this task, two of the five defects were checked in the test cases. The first unit test created was for the accessCheck() method in our second defect. The assertion added to this method was to check if the method assigns the x, y, and z elements correctly in the voxel space.

The second unit test created was for the color() method in our fourth defect. The assertion added to this method was to check that the colors were stored correctly in the voxel and if the assignment of the voxel's colors corresponds to those specified in the input parameters of the method.

Instructions created to compile the classes and to generate the test suit using Eclat:

```
java Eclat.textui.Main generate-inputs --create-regression-suite
--test cmu/voxeltoy/VoxelSpace.java src.cmu.voxeltoy.VoxelSpaceTest
```

VoxelSpace.java is the class that we are analyzing, and VoxelSpaceTest the JUnit test class used by Eclat to generate the set of tests for Voxelspace.java.

Eclat generated a unit test class that contained a total of 29 test cases for the VoxelSpace class. Of these, four were reported as errors while running the JUnit framework. Reviewing the code, the problems that Eclat found were true. The main cause of the problems in the code was initialization of the voxelSpace to null and the values of the matrix dimensions to negative numbers or zeros.

Eclat created test cases for methods that weren't specified in the input test class, but it didn't find the some of the errors in the code. For example, it didn't find the third or fifth defects that we had injected.

A summary of the results of this experiment can be found in the following table.

Input test cases	2
Generated test cases	29
Injected errors found	3
Which injected errors were found	1, 2, and 4
False negatives	2
False positives	0
Unexpected errors found	4
Time spent creating input test cases	0.75 hours
Time spent running Eclat and interpreting results	1.5 hours

Table 2. Results of the second experiment

As we had mentioned earlier, the same errors that the reviewer found were detected by Eclat when we gave appropriate test scenarios. It would seem from the results of this experiment that Eclat efficiently takes a normal use of the system and generates more test cases that use different inputs with the same use. It does not, however, find errors in parts where the test cases are not provided.

Experiment 3: Test suite that covered all of the errors

In this experiment, a test suite was built to thoroughly test the VoxelSpace class in addition to specifically finding the injected defects. At the end, Eclat was run to see if any more could be found with an established test suite base. The goal of this experiment was to determine how much time it takes for a software engineer to find all injected faults in the system and to see if Eclat can find anything that we missed even after explicitly finding all injected defects in our test suite. After creating a comprehensive test suite, we ran Eclat and found one extra real defect.

This experiment was built in the following phases:

1. **Build our manual test cases**: A test suite was manually written to specifically test each method's postconditions. After running that and fixing an unrelated preexisting bug, we wrote two extra tests to explicitly catch two injected defects that our manually-written test suite did not have. We ran this test suite against the original, pre-defect-injection version of VoxelSpace and, in the process, found and corrected a defect in the test suite. Once all these changes and fixes were in place in the test suite, it was able to catch all five injected defects correctly.
2. **Run Eclat against our test suite to see if any more defects can be found:** Because our test suite could find all the injected defects, we ran Eclat to see if it could find any additional defects. Eclat actually did find two additional defects, though one was a false positive based on a false invariant that it generated. It correctly noticed that if getVolume() is called before any calls to color() take place, then a NoSuchElementException gets thrown. This is directly related to the injected defect 5, though it is a defect that our manual test suite did not catch. Eclat's other identified defect assumed that the z_size object instance field must always be greater than or equal to 1, which is the size of the z-axis in the voxel space. It's theoretically possible to have a zero z_size, though that would basically mean that no voxels exist. Because it's theoretically possible to have a zero z_size, Eclat incorrectly derived an invariant, though it was a good try.

A summary of the results of this experiment can be found in the following table.

Test cases: Our manually-created tests	10
Test cases: Eclat-created tests	85
Injected errors found	5
Which injected errors were found: Manual tests	1, 2, 3, 4, and 5

Which injected errors were found: Eclat, over and beyond our manual tests	5
False negatives	0
False positives	1
Unexpected errors found	4
Time spent creating our manual test cases	2.25 hours
Time spent running Eclat	0.25 hours
Time spent interpreting the results of our manual tests	0.75 hours
Time spent analyzing Eclat's output	0.25 hours

Table 3. Results of the third experiment

This experiment took about three and a half hours to perform from start to finish. While our manually-created test suite could find all defects, partly due to knowledge of which defects were injected, Eclat was able to find one additional real defect related to injected defect 5 and one false positive. As a result, even with a comprehensive test suite, Eclat was still helpful to us as it was able to correctly find one more real defect.

Lessons Learned

Benefits

- **Saving Tester's Time**

Eclat takes a normal operation test case and generates all possible input combinations from it. This would be extremely useful in our tests, because these combinations can be many and time-consuming to write. Eclat even optimizes the combinations for those that affect different invariants, making our testing more efficient.

- **Finding Abnormal Errors**

Eclat can find complex errors that go beyond just wrong inputs. For example, in our experiment, Eclat found a bug related to Null pointers. These kinds of errors are sometimes neglected by developers.

- **Exploring extensive set of inputs**

One big benefit of Eclat is the creation of the set of input information to test methods in a class. The variety of information goes from negatives numbers, zero, and nulls. This set of information is created automatically by Eclat, if we wanted to create this inputs manually, it could take long time to come up with a set of information similar to the Eclat's.

- **Ensuring program invariants**

Another, may be indirect, benefit of Eclat's is that it can help the software engineer ensure that the invariants he/she intended to be in the program are captured correctly by the tool. Eclat tries to discover the invariants and assign weights to them. If the developer finds that an invariant that he wanted to be in the program was not discovered or assigned a lower weight, then he/she can go back and change his program accordingly to stress this invariant.

With this Eclat can be introduced to software engineers as a unit test generator, but also introduce them to other analysis techniques if they have never been exposed to them before.

- **Shortening test process**

Eclat makes the testing process shorter and more efficient. This is addressed by reducing the test suite to test cases that are only likely to reveal bugs in the system, and also by omitting test cases that have the same input category and reveal the same bug

Drawbacks

- **Omitting illegal inputs**

Eclat is claimed to omit illegal test inputs. However, we couldn't prove that since we had some illegal inputs to some methods as part of the final test suite that Eclat generated. This thing might hinder the developer and make him spend more time isolating these unreasonable cases.

- **Total dependency on correct inputs**

If the errors are not covered by the input set of test cases, then Eclat doesn't find them. We need to guarantee that we cover the correct path of all of our code to ensure that Eclat's findings are more complete.

- **False positives**

Testers need to check Eclat's outputs for any false positives. With anyone with less experience, false positives can waste someone's time trying to figure out where the source of this bug is. Eclat seems to be suitable with users who are familiar with the concepts of invariants and false positives, so they would understand Eclat's outputs.

Quality attributes of the tool

Usability

The current version of the tool is hard to use in command line. There is an Eclipse plug-in version, but we were unable to use it with version 3.1.2 of Eclipse. There are some quirks in the command line interface. For example, it only accepts paths ending with a file separator / when specifying the classpath. Another example is that you need to specify every option of the program in the command. This makes it hard to use because it generates confusion within all the directories it generates and setting the classpath.

Documentation

The documentation is very basic. It does not cover any problem solving or troubleshooting sections. The tutorial gives you only basic tips and mixes UNIX and Windows instructions. This can be due to the tool being a research tool. However, with such impressive functionality, we think that the tool developers would benefit from having better documentation by increasing their user base.

Performance

Eclat took between 5 and 10 seconds to generate 124 test cases for our system. It generated all the preconditions, postconditions and invariants of the class.

Conclusions

Eclat provides a pragmatic solution to the problem of producing extensive test suites. This solution focuses on the program's properties (i.e. invariants) as well as the efficiency of the resulting suite.

By doing this, Eclat proves the applicability of some analysis techniques, such as model checking, in the software industry. It demonstrates a perfect combination between analysis techniques and traditional ones such as unit testing. However, this might limit its usage to experienced users who have background in model checking. Users with less experience might get stuck in the trap of false positives resulting mainly from wrong invariant discovery.

If Eclat could overcome this problem, it would open the way to novice users with no experience of analysis techniques such as formal model checking to use it thoroughly.

Appendix

The results of our experiments are stored in the submitted zip file's Experiment2 and Experiment3 folders. Significant files are noted in the list below:

- "Serendipity – Analysis Final Project Report.doc" and "Serendipity – Analysis Final Project Report.pdf": This file
- Experiment2
  - output.txt: Contains the command-line outputs of running the Eclat tool
  - src/: The source code being checked
    - VoxelSpace.java: The class being checked
    - VoxelSpaceTest.java: The manually-created test suite
  - eclat-*/: The Eclat-generated output files and tests
- Experiment3
  - output.txt: Contains the command-line outputs of running the Eclat tool
  - src/: The source code being checked
    - cmu/voxeltoy/VoxelSpace.java: The class being checked
    - cmu/voxeltoy/VoxelSpaceTestAll.java: Our manually-created test suite
  - eclat-*/: The Eclat-generated output files and tests

References

[Daikon] Daikon page http://pag.csail.mit.edu/daikon/

[Eclat] Eclat page http://pag.csail.mit.edu/eclat/

[Ernst 2005a] Ernst, M., Pacheco, C., Eclat: *Automatic Generation and Classification of Test Inputs*. MIT, 2005. Available at http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-968.pdf

[Chen 2005] Chen et al, *Evaluation of Eclat Automatic Test Generation Tool.* CMU, May 2005