Carnegie Mellon University
School of Computer Science
Master of Software Engineering

Evaluation of Eclat
Automatic Test Generation Tool

May 5, 2005

Analysis of
Software Artifacts

Team members:

Min Chen
Bharat Gorantla
Okeno Palmer

Table of Contents

1 Introduction

This paper documents the evaluation of the Eclat tool for automatic test generation by the Sapphire team. Eclat has been in existence since February 2005 and this evaluation was done in May 2005. The rest of this document details the steps and decisions taken during the evaluation of Eclat.

1.1 Purpose

The purpose of this evaluation is to fulfil the requirements of the *Analysis of Software Artifacts* course. The tool evaluation requires a presentation and a final write-up of findings from the evaluation. Another purpose of this evaluation is to give students practical hands-on experience in the use of tools to analyze software artifacts so that students can determine the value that use of these tools brings to the software engineering practice.

1.2 Expectation

This evaluation is expected to result in students understanding how to use the Eclat tool and the situations in which the tool will be useful. It is also expected that students will apply the theory learned in the course to see how closely a practical implementation of the theory applies to software engineering in practice. For this evaluation, Sapphire seeks to do the following:

1. Compare the number of defects found in a peer review versus the number of defects found by test cases generated using Eclat.
2. Determine the types of procedures for which Eclat generates test cases.
3. Determine the number of procedures in a class for which Eclat generates test cases.
4. Determine the number of defects found in an application by Eclat test cases.

1.3 Approach

The approach taken for this evaluation was for students to download the tools being evaluated, install the tool and perform experiments in the use of the tool with existing or new software artifacts.

The software artifact used in this evaluation is source code from the Sapphire team's studio project. The studio project uses source code written in C# but the Eclat tool only analyses java source code. To allow for the use of the Eclat tool, a subset of the C# code of the studio project was converted to Java so that we could evaluate how effective Eclat is in generating automatic unit tests for the studio project.

2 What is Eclat?

Exhaustive software testing is infeasible and expensive. It is desirable to test with a small set of test cases that will reveal as many errors as possible. It is even better if the tests can be generated automatically. Having this in mind, Carlos Pacheco and Michael D. Ernst from the Massachusetts Institute of Technology (MIT) created the Eclat tool to help test engineers select, from a large set of randomly generated test inputs, a small subset likely to reveal faults in the software under test. The development started in 2004, but it was first released on Feb 15, 2005.

Eclat requires Java 1.5 and uses Daikon for invariants detection. Figure 1 shows the architecture of Eclat. It has three tree main components: 1) Input Generator, 2) Classifier, and 3) Reducer.

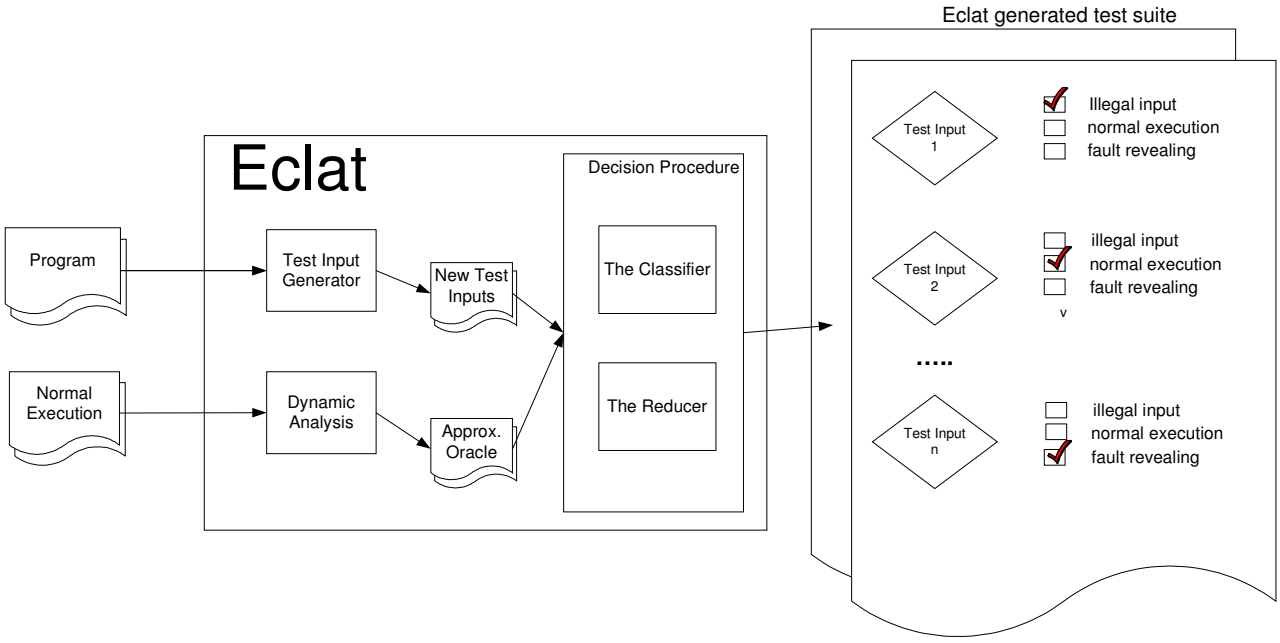

Figure 1: Eclat Architecture

The way that Eclat works is that it receives two inputs files. One input file contains the program that we want to test, and the other is a correct test suite for the program we want to test.

Eclat generates possible program inputs randomly and produces a new set of test inputs. It also uses the Daikon dynamic invariant detector to derive an *operational model*. This operational model approximates the correct behavior of the program based on the test suite. An operational model consists of a set of properties that hold at the entry and exit point of a program's components (e.g., on public method entry and exit), and it is also regarded as *oracle*. These properties have an associated confidence measure which is the likelihood a property is

universally true in all program executions. Properties with higher confidence measure are more likely to be universally true.

The classifier uses the new set of test inputs, the program under test, the operational model, and a confidence threshold. The classifier runs the program on the new set of test inputs and collects the (possibly empty) set of violated model properties. Violations of properties with confidence measure above the threshold are considered severe, and violations of properties with confidence below or at threshold are considered mild.

These inputs are classified based on their violation pattern: the set of violated properties.

1. Illegal. One or more severe entry violations occur, and one or more severe exit violations occur as well.
2. Normal operation. No severe exit violations occur.
3. Fault-revealing. No severe entry violations occur, but one or more severe exit violations occur.

Figure 2 shows how to classify the input based on the set of violated properties.

Severe entry violations?	Severe exit violations?	Classification
yes	yes	***illegal***
—	no	***normal operation***
no	yes	***fault-revealing***

Figure 2: Classification of violated properties

A violation pattern not only determines the classification of an input. It also induces a partition on all inputs, with two inputs belonging to the same partition if they violate the same properties.

The process is repeated in rounds in order to obtain the smallest set of test cases that will reveal as many fault-revealing errors as possible.

After the last round, the reducer discards the illegal and normal operation input and selects one fault-revealing input from each partition.

Eclat generates the test cases based on this subset of inputs.

3 Using Eclat

This section describes how to acquire the Eclat tool, install it, and perform a sample execution of the tool to generate new inputs for a test engineer to use.

3.1 Installation

Eclat is a Java tool. It requires the use of the Java Development Kit version 1.5.0 (J2SDK 1.5.0). To acquire the Eclat tool, visit the website http://pag.csail.mit.edu/eclat/ and follow the instructions in the *Download and Install* section to download the required files. To test that the Eclat installation is successful, run the following commands and verify that output similar to the screenshots is displayed after running the respective commands:

1. java daikon.Daikon

```
C:\>java daikon.Daikon
Daikon error: no files supplied on command line.
Daikon version 4.1.0, released April 1, 2005; http://pag.csail.mit.edu/daikon.
Daikon invariant detector, copyright 1998-2005
Uses the Java port of GNU getopt, copyright (C) 1998 Aaron M. Renn
Usage:
    java daikon.Daikon [flags...] files...
  Each file is a declaration file or a data trace file; the file type
  is determined by the file name (containing ".decls" or ".dtrace").
  For a list of flags, see the Daikon manual, which appears in the
  Daikon distribution and also at http://pag.csail.mit.edu/daikon/.
```

2. java Eclat.textui.Main help

```
C:\>java eclat.textui.Main help
Eclat
======

Eclat is a tool that creates test inputs for Java classes. The inputs
that Eclat creates are deemed potentially fault-revealing and
deserving closer inspection.

Eclat supports various commands, documented below. For more high-level
documentation (and a tutorial on Eclat) please
refer to the manual available from Eclat's webpage:

http://pag.csail.mit.edu/eclat

Type "help" followed by a command name to see documentation.

Commands:

generate-inputs     Generates test inputs for classes that differ from
                    the given execution.
bottom-up           The input generation algorithm behind Eclat (a
                    submodule of the "generate-inputs" command).
help                Displays a help message.
```

Caveat:
Setting the correct value for the CLASSPATH environment variable is very important. The value of this variable is dependent on the operating system that you use. At all times, be sure that your CLASSPATH includes the path to (1) the current directory, (2) path to *daikon.jar*, and (3) path to *Eclat.jar*. An example is shown below. We assume that the Eclat files are installed in the directory C:\eclat on a Micorsoft Windows operating system:

set CLASSPATH=%CLASSPATH%;c:\eclat\daikon.jar;c:\eclat\eclat.jar;.\

* Note there are no spaces between the CLASSPATH entries!

After a successful installation, Eclat is now ready to be used to automatically generate test inputs for a test engineer.

3.2 Operational options

Eclat can be operated in two modes. These modes are *generate-inputs* or *bottom-up*. The modes are enabled by specifying which one Eclat is to run in at its entry point. The entry point for Eclat is in the form of the command line command:

java Eclat.textui.Main <*command*> <*command-options*> <*command-arguments*>

The <*command*> option specifies Eclat's mode of operation. In *generate-inputs* mode, Eclat generates test inputs, for examined classes, that deviate from the behavior observed when running the program that specifies a correct sample execution of the class being examined. In *bottom-up* mode, Eclat generates random new input for the observed class by executing several rounds of invocating public methods and constructors of the observed class. In each round of execution, inputs generated from the previous rounds are used to invoke the public methods and constructors in the current round. This produces an increasing number of inputs per round which are eventually classified by Eclat as *normal, fault-revealing,* or *illegal.* A subset of these inputs is used to create JUnit test classes that serve as the output of Eclat.

Each command has specific command options. Details of these command options can be found in the documentation of the Eclat tool. This information is located at http://pag.csail.mit.edu/eclat/manual/index.php#Usage

3.3 A sample execution

During our evaluation of Eclat we used Java code that represents a Java implementation of a C# project that Sapphire had been working on. This code was the implementation of a structure that represents the concept of a *Preference* in a system. A preference is defined as an object that contains a key and possibly one or more values associated with that key. A JUnit test for this *Preference* object was also created and used during the evaluation of Eclat. The java code for these classes can be found in Appendix A's code samples.

To start, we specified to Eclat, the class to be examined and the unit test that represents a correct execution of this class. The operational mode used for executing Eclat is *generate-inputs*. This can be done as follows:

java Eclat.textui.Main generate-inputs --test Preference.java PrefereceUnitTest

It is important to note here that the files Preference.java and PreferenceUnitTest.java must have already been compiled before running Eclat. This is necessary because Eclat will attempt to locate the class files for *Preference* and *PreferenceUnitTest*, and use those class files in its dynamic analysis of the *Prefernce* class to determine invariants for the class' usage as it observes how the class is used in the correct execution of the class as determined by the supplied unit test, *PreferenceUnitTest*.

The output from Eclat details information about the inputs Eclat creates as it performs each round of generating inputs for the class. Eclat begins executing the correct normal execution of

the code supplied to it so that it can observe the values used in the normal execution of the class. It then performs dynamic analysis of the input by using Daikon to determine invariants over the observations it has done. These invariants state what conditions are true in the correct execution supplied to Eclat. Figure 3 below shows Eclat's output that is associated with this part of its processing on the *Preference* class.

```
Executable command: PreferenceUnitTest
Parsing file Preference.java
Compiling test sources.
Observing PreferenceUnitTest's values as it executes.
Entering Daikon to detect invariants over observations.
Daikon version 4.1.0, released April 1, 2005; http://pag.csail.mit.edu/daikon.
  (read 0 decls files)
Processing trace data; reading 1 dtrace file:
[6:49:46 PM]: Finished reading PreferenceUnitTest.dtrace.gz
Creating implications
Exiting Daikon.
```

Figure 3: Eclat observing invariants of the correct normal execution of the Preference class

After acquiring these invariants, Eclat instruments the source code provided to allow it to check new inputs it creates against the invariants that were discovered during dynamic analysis. In this step Eclat attempts to generate an approximate oracle that it can use to create new inputs and determine what category these inputs fall into. These inputs are categorized as illegal, fault-revealing, or normal execution. Input generation is done over several rounds of invocation of the public methods and constructors of the class being analyzed. The number of rounds that Ecalt performs is a configurable option, but the default value is four. Figure 4 below shows Eclat's output that is associated with this part of its processing on the *Preference* class.

```
Instrumenting sources for runtime invariant checking.
Compiling instrumented sources.
Generating new inputs.
Preference

Generating new inputs (round 1)...Generated 1 new inputs.
Evaluating inputs:
.

Generating new inputs (round 2)...Generated 29 new inputs.
Evaluating inputs:
.F...F...FFF....FF.F...FFF..i

Generating new inputs (round 3)...Generated 264 new inputs.
Evaluating inputs:
i.......i.....i...F.i.....FF.F.............F....F.FFF.F.........i.FF.F.....F...
.F...........i..F.........FF.............F......FFF.FF.F......iF.FFFF.F.F.FF.F
F..FFFF.....F.FF.FF.Fi..iF.FFFiFF.F.....i...FF.F.................F......F....F
....F.....i............

Generating new inputs (round 4)...Generated 626 new inputs.
Evaluating inputs:
.F.FF...F.....F......F.F.....................F...F.....................F......F.
.FF.......FFF.F.......F.........................F....F..........F...F......F...
........F.....................F..F.....F..F.....................FF..Fi.........
.........F.................FF....i.........F......FFF.....F.......F...F...F....iF
..........F.........F.F..........F.........FFF...........FF.......F.F.F..FF.....
........F..F......FF...F.F.....F..FFFF.F..........F..F............F............
....Fi.....................F...FiF..i.............F...F.....................FF...
iF.F.F...F.....F.....................................FFFFF.......
Creating JUnit class.
```

Figure 4: Eclat generating new input based on Daikon invariants and an approximate oracle

When Ecalt is finished generating new inputs, it produces a new unit test that contains a subset of the new inputs it created. This subset is intended to be the smallest set of inputs that Eclat determines will be useful to test engineer. The subset of inputs will include inputs that lead to fault-revealing, illegal, or new normal execution scenarios. The inputs that lead to fault-revealing and illegal scenarios are used by Eclat to create unit tests that demonstrate these scenarios. Inputs that lead to new normal execution scenarios are also used to create unit test cases that demonstrate use of the new inputs. Figure 5 below shows Eclat's output that is associated with this part of its processing on the *Preference* class.

```
JUnit test suite has been created: PreferenceEclatTest.java
This file contains the subset of inputs most likely
to reveal faults or new behavior.
```

Figure 5: Eclat specifying the name of the unit test file it created with new unit test cases

Along with creating these new unit tests, Eclat also makes a log of all the new inputs it generated and the results of executing those inputs against the class being examined. This is done by Eclat to allow test engineers to see what exactly Eclat used during its operation and can give a test engineer an idea of the kind of coverage Ecalt gives for variation of inputs used that could cause fault-revealing, illegal, or new normal execution of the class they supplied to Eclat. Figure 6 below shows Eclat's output that is associated with this part of its processing on the *Preference* class.

```
Creating inputs text file.
Created file containing inputs in text format: Preference.txt.zip
This file contains ALL the inputs generated,
in case you want to see them.
```

Figure 6: Eclat storing inputs it created to a zip file for future reference by a test engineer

At this point, Eclat has completed its analysis of the class and has created the following files for a test engineer to use:

1. PreferenceEclatTest.java
2. PreferenceUnitTest.dtrace.gz
3. PreferenceUnitTest.inv.gz
4. Preference.txt.zip

The two Gzip (i.e. *.gz) files created contain invariant information and a trace generated by Daikon during Eclat's dynamic analysis of the *Preference* class. The Zip (i.e. *.zip) file contains all the inputs generated by Eclat during its analysis. The Java file contains the unit test generated by Eclat. Samples of the contents of these files can be found in Appendix A's input samples.

At this point, it is now up to the test engineer to examine the unit test generated by Eclat to determine how viable those unit tests are for testing the target class. The test engineer is now

equipped with both new unit tests, knowledge of the hidden invariants of the code, and samples of inputs generated by Eclat that he can use in his determination of the right tests for a class. The new unit tests created by Eclat can be refined by the test engineer and then used to further test that refinement with another run of Eclat.

4 Evaluation

In this section we evaluate the Eclat tool by stating observations from the use of the tool and by answering several questions that focus on the effectiveness and usefulness of the tool.

4.1 Usability

To use Eclat, a user needs to know how to run commands from the command line of an operating system. Eclat is not have a graphical user interface. Eclat's command line syntax is similar to any average command line oriented application. Users familiar with using unix or windows commands from the unix shell or windows shell respectively will have no difficulty using the tool. Eclat's command options are easily accessible via a help command as well so at any point when a user is unsure of how to use Eclat, they can issue a help command to the tool and it will display its usage instructions.

Eclat displays progress information to the command line in a very readable manner and it is easy to see the points of the analysis as Eclat does its operations. The display of the rounds and number of inputs for each round is also useful and easy to identify and immediately gives the user an idea of how large the input set generated for the observed class is.

Eclat's output becomes unreadable when it is run in debug mode and/or with verbose modes set to true. In these modes, Eclat outputs numerous debug information intended for a more technical audience that is interested in seeing the specifics of Eclat's analysis as it performs its processing. An example of a run for Eclat that generates this type of output is as follows:

java eclat.textui.Main generate-inputs --verbose --debug --test Preference.java PreferenceUnitTest

This ability does not reduce the usability of the tool. It instead makes it more useful as the tool allows both simple users and more technical users to gain the information they need from the same interface, namely the command line. The default behavior for Eclat is to output simple results which are in a very readable form and provide sufficient detail for the progress of the analysis and the location of files generated by the tool.

4.2 Documentation

The documentation for Eclat as of May 2005 is sufficient for specifying how to install and use the tool for getting output. However, this documentation can be improved. While using the tool, we encountered installation issues with setting the java CLASSPATH variable and some nuisances involved with getting this to work on the Microsoft Windows operating system. Though this may seem outside the scope of the Eclat tool, this type of documentation should be a part of Eclat because the tool relies on the correct configuration of the Java environment for it to run. Having this documentation can save users time during setup.

Documentation for use of the tool and the options it supports is very good. The tool creators do a good job of specifying what each command to Eclat does along with the purpose of each

command and the options it supports. There a few typos in the documentation of the syntax of the commands on the website but these typos do not exist in the tool's usage documentation accessed by issuing the help command to the tool.

4.3 Performance

Eclat's performance is determined by the operational commands and the options specified for those commands that affect the way Eclat analyzes the input files to the tool. Eclat's analysis involves observing the behavior of the provided correct execution of a class and the determination of invariants over that observation. Eclat performs numerous rounds of input generation which can be configured from the command line. In each round of input generation that Eclat performs, the user can specify the number of invocations per round. To specify these parameters to Eclat a user can use one or more of the following command line options to the *generate-inputs* command:

1. *--num-rounds N*
    a. Do N rounds of generation (default=4)
2. *--invoc-per-round N*
    a. On each round, try to create N new invocations of each method. (default = 100)
3. *--nesting-depth*
    a. Depth to which to examine structure components (default=2)

These command options directly affect the runtime performance of the tool in terms of how long it will take t produce new inputs and the coverage that those inputs will provide. A sample run of Eclat was done with the command below:

java eclat.textui.Main generate-inputs --nesting-depth=100 --num-rounds=10 --invoc-per-round=1000 --test Preference.java PreferenceUnitTest

The execution of this command took 2 hours 11 minutes seconds consuming 483MB of RAM on a laptop PC with 1GB of RAM and a 3GHz Multiprocessor; created 6825 new inputs and 4 test cases in the generated unit test. These measures indicate that doing more detailed analysis of a simple class does not necessarily provide better results. We still had 4 test cases that identified the same results as when we ran Eclat with the default values for the options. In this latter case, the run took 2 hours whereas in the default configuration case, the run took under 5 seconds.

The performance for Eclat is good with its default values for these command options with execution completing in fewer than 5 seconds with the Preference class found in Appendix A. It is necessary to note though that depending on the level of analysis that a test engineer requires, the tool can consume significant computer resources and take a long time to complete its analysis.

4.4 Validity of results

During the evaluation of the tool, several questions were asked that we sought answers to. The questions are:

1. Does Eclat find defects that were not captured during Sapphire's peer review of the Preference code?
2. What types of procedures does Eclat generate test cases for?
3. How many procedures in a class does Eclat generate test cases for?

During our evaluation Eclat found several defects that were not captured during the team's peer review session. Eclat was able to generate 4 test cases that identified two fault revealing cases and 2 new behavior cases. None of these were identified in our peer review. On revising these results, the team realized that the results were valid and the defects did exist in the code base. The identification of these defects also leads to the further investigation of the rest of the code base for these types of defects. This is a very good result of using Eclat and shows immediate value for the tool. The types of defects it found were out of bounds exceptions that could occur while accessing a collection. These types of defects were not identified in the peer review because the peer review did not inspect the detailed technical aspects of methods calls or invocations. This is something that Eclat does well by observing behavior and exercising inputs that are able to get at these types of defects better than a peer review would do.

Questions two and three can be answered together. Ecalt makes invocation of all public methods and constructors of a class during its analysis. Therefore, it will analyze all methods depending on their level of access. Our observations show that Eclat generate test cases for all public methods that it finds a fault-revealing, illegal, or new behavior for. This is helpful for a test engineer because it identifies methods that can have defects and also allows the test engineer to see what other ways his methods can be further tested.

One of the difficult things when doing testing is determining the granularity of testing and the trying to gain the right coverage. Creating input for tests is a challenge and Eclat offers a solution to overcoming that challenge. Our evaluation show that Eclat's results are often valid and its use of Daikon and the invariants it produce offer a test engineer insight into the hidden invariants that often occur in software development. Having these invariants show up in code is a good way of documenting the design of modules. The fact that Eclat makes these explicit in unit tests also allows a test engineer to now use his unit tests to verify use cases that may exist for which his unit test is being created. This also offers the added benefit of traceability of requirements to code.

4.5 Issues

There were two main issues that arose during the evaluation of Eclat. We found that Eclat does not operate well with:

1. Interactive programs
2. Top-level Java exceptions thrown within an analyzed class

When we attempted to have Eclat generate inputs for a class that was used in an interactive program – i.e. a program that requires input from a user – Eclat got stuck at the program point where input was required from the user during the execution of the program. Eclat provided no

way for this input to be done explicitly and simply halted at that program point. Although Eclat supports the provision of program arguments to a class during its execution via the command given to Eclat, it does not allow input to a program while analysis is being done on that program.

Another issue we encountered during evaluation was that Eclat would not do analysis on a class that had methods that declared that the method throws a java.lang.Exception. When this situation occurred, Eclat would throw an exception stating that a java.lang.RuntimeException was already being caught. When this type of situation occurs, Eclat produces the source code the instrumented version of the class that is uses for dynamic analysis. The exception thrown by Eclat mentions the line in that source file where the exception occurs. This allows a test engineer or user to further investigate the error. This is exactly what we did.

On examining the source file generated, we observed that the instrumented file that is generated by Eclat does exception handling by first catching a java.lang.Exception and then more specific types of exceptions. An extract of the faulting code is below:

```
1      try
2       {
3          retval_instrument = internal$getValue();
4          daikon.tools.runtimechecker.Runtime.numNormalPptExits++;
5       }
6       catch (Exception t_instrument)
7       {
8          daikon.tools.runtimechecker.Runtime.numExceptionalPptExits++;
9          methodThrewSomething_instrument = true;
10          throw t_instrument;
11       }
12       catch (java.lang.RuntimeException t_instrument)
13       {
14          methodThrewSomething_instrument = true;
15          daikon.tools.runtimechecker.Runtime.numExceptionalPptExits++;
16          throw t_instrument;
17       }
18       catch (java.lang.Error t_instrument)
19       {
20          daikon.tools.runtimechecker.Runtime.numExceptionalPptExits++;
21          methodThrewSomething_instrument = true;
22          throw t_instrument;
23       }
```

The exception thrown by Eclat occurs on line 12 of the above code. The message given by the thrown exception is “exception java.lang.RuntimeException has already been caught”. This indicates that this exception was already caught by some handler previously in the code. The team suspects that the issue may lie with the ordering of the exception catches from lines 6 – 18 of the code extract. The catching of a java.lang.Exception before the other more specific types of exceptions may be the culprit of this error.

However, we cannot draw a conclusion on this as we are not familiar with the inner workings of the Daikon analysis tool. The exception message thrown by Eclat does however indicate that our suggestion may be the likely cause of this error.

5 Future Improvements

From the previous sections, we have tried to demonstrate the various ways the Eclat tool can be used and how it can be used to help a developer to do his/her unit testing more efficiently. By applying Eclat to a small-scale application, we noted some things that we liked about Eclat as well as what we would have liked to have seen. In this section, we will air out a number of things we think that could greatly improve Eclat's usability and functionality.

The following describes some of the improvements that we identified.

5.1 GUI based application

The tool at the moment is a command-driven application. There are a number of commands and parameters that are deemed useful in the generation of Eclat test cases. And as described in the installation as well as in the execution of the tool, a developer would need to refer constantly for the commands and parameters. By creating a GUI-based application, this takes some pressure of the developer and makes his/her life much simpler.

5.2 Handling interactive applications

The tool at the moment does not have the ability to handle applications that deal with user-interactions. The reason for this is that code that involves user input, somehow and for some reason causes the tool to stall and stay idle. It informs the developer that a user input is needed but the tool doesn't have the ability to take in any input and hence just stalls. This is an important aspect for real-time and embedded applications.

5.3 Incorrect handling of base exceptions

In the current version of Eclat, there seems to be some inconsistency in the way the tool analysis the application when the base exceptions are used. Exceptions, such as java.lang.Exception which is the base exception, cause the tool to terminate with some errors. By inspecting the invariant files generated by the tool, it informs the developer that it needs a specific type of exception and not a base exception. So if a method is supposed to throw a NullPointerException, the method should throw this specific exception instead of the base exception. This is a concern to the developer as they might not know what kind of exceptions a method is throwing but would like to catch all kinds of exceptions but the tool doesn't promote it.

5.4 No descriptive information on failures

Any time the tool crashes or fails in generating a test file, the developer is left at figuring out what exactly made the tool to behave in that manner. It is unclear by looking at the tool's log messages on exactly what has happened. This could be improved greatly as it will save time on the developer's end in preventing the developer to spend hours and hours on the reasons on why the application code or the parameters that were inputted failed.

5.5 Inadequate documentation

The current documentation that exists for the Eclat tool is in a form as a development manual. This manual consists of a brief installation guide, followed by a simple example on how to run the tool. The improvement that could be made here is to extend the documentation to include: Information on providing users with some commonly asked questions from other developers or testers.

- A better comprehensive documentation for each of the bugs that are listed in their bug tracker.
- Information on the various commands presented in a helpful readme.txt or as a help option in the tool.
- Providing more examples on various types of applications. For e.g. applications using threads, command-line applications, exceptions, etc.

6 Conclusion

In the end, we found Eclat to be a useful tool with a lot of potential, although it is not yet in a state that we think will greatly benefit developers and testers in the industry. Eclat has a nice concept that will greatly be appreciated especially in the testing phase of a software project. It will also cut-down on costs and improve the quality of the product that is being tested on. The neat thing about this tool is the ability to work with Daikon in providing the various invariants for generating the inputs when simulating the behavior of the application.

However, on small-scale applications, we think that by using Eclat to generate our test cases as indicated in Section 3.2, Eclat was able to identify a number of different inputs that would have caused the application to fail. We don't think we have enough data to support our claim on whether the tool will help developers/testers on a large scale as we have not done any sort of analysis on these applications.

Appendix A

Eclat Input samples

Input sample 1: Normal input

```
======================================================================
round: 1
INPUT CLASSIFIED AS <normal>
======================================================================

  1. Preference var1 = new Preference();


  ----------------------------------------------------------------------
  Test expression evaluation (line 1).

  EXCEPTIONS: none.

  INVARIANT VIOLATIONS:
  none.

  ----------------------------------------------------------------------
  Explanation:
  No properties were violated and no errors were thrown. This
  indicates normal execution.
```

Input sample 2: Fault-revealing input

```
======================================================================
round: 2
INPUT CLASSIFIED AS <fault>
======================================================================

  1. Preference var1 = new Preference();

  2. int var19954 = var1.getValue(1);


  ----------------------------------------------------------------------
  Prep code evaluation (lines 1 through 1).

  EXCEPTIONS: none.
  INVARIANT VIOLATIONS:
  none.

  ----------------------------------------------------------------------
  Test expression evaluation (line 2).

  EXCEPTIONS:     java.lang.IndexOutOfBoundsException
  INVARIANT VIOLATIONS:
     violated on entry : precondition  : index == size(this.prefValue[])-1
     violated on entry : precondition  : this.prefKey has only one value
     violated on entry : precondition  : size(this.prefValue[]) == 1


  ----------------------------------------------------------------------
```

```
  Explanation:
  During execution of the last method call, a throwable was
  thrown. No important preconditions were violated, so this
  suggests a meaningful input. Since the throwable is
  considered severe, this suggests a bug.
```

Input sample 3: Illegal input

```
========================================================================
round: 2
INPUT CLASSIFIED AS <illegal>
========================================================================

  1. Preference var1 = new Preference();

  2. var1.setListValue((java.util.ArrayList)null);


  ------------------------------------------------------------------------
  Prep code evaluation (lines 1 through 1).

  EXCEPTIONS: none.
  INVARIANT VIOLATIONS:
  none.

  ------------------------------------------------------------------------
  Test expression evaluation (line 2).

  EXCEPTIONS: none.

  INVARIANT VIOLATIONS:
     violated on entry : precondition  : this.prefKey has only one value
     violated on entry : precondition  : value has only one value
     violated on entry : precondition  : size(this.prefValue[]) == 1
     violated on exit  : postcondition : this.prefKey has only one value
     violated on exit  : postcondition : this.prefValue has only one value
H    violated on exit  : obj invariant : this.prefValue != null


  ------------------------------------------------------------------------
  Explanation:
  During execution of the last method call, at least one
  important postcondition was violated. Since one of the
  arguments was null, I will classify this input as illegal.
```

Code samples

Code sample 1: Preference.java

```
import java.util.*;

public class Preference
{
       private String prefKey;
       private ArrayList prefValue;

       public Preference() {
              prefValue = new ArrayList();
       }
```

```
	public String getKey() {
		return prefKey;
	}

	public void setKey(String value) {
		prefKey = value;
	}

	public int getValue(int index) {
		return ((Integer)prefValue.get(index)).intValue();
	}

	public void setValue(int value) {
		prefValue.clear();
		prefValue.add(Integer.valueOf(value));
	}

	public ArrayList getListValue() {
		return prefValue;
	}

	public void setListValue(ArrayList value) {
		prefValue = value;
	}
}
```

6.1.1 Code sample 2: PreferenceUnitTest.java

```
import junit.framework.Assert;
import junit.framework.TestCase;
import java.util.*;

public class PreferenceUnitTest extends TestCase {

	Preference pref = null;

	public static void main(String[] args) {
	    junit.textui.TestRunner.run(PreferenceUnitTest.class);
	}

	protected void setUp() throws Exception {
		super.setUp();
		pref = new Preference();
	}


	protected void tearDown() throws Exception {
		super.tearDown();
		pref = null;
	}

	public void testPreference()
	{
		pref.setKey("one");
		pref.setValue(1);

		Assert.assertTrue(pref.getValue(0) == 1);

		ArrayList listVals = new ArrayList();
		listVals.add(0, Integer.valueOf(100));
```

```
            listVals.add(1, Integer.valueOf(200));
            listVals.add(2, Integer.valueOf(300));
            listVals.add(3, Integer.valueOf(400));

            pref.setListValue(listVals);

            listVals = pref.getListValue();
            int initVal = 100;
            for(Iterator iter = listVals.iterator(); iter.hasNext(); initVal+=100)
            {
                  int currVal = ((Integer)iter.next()).intValue();
                  Assert.assertEquals(initVal, currVal);
            }
      }
}
```

Code sample 3: PreferenceEclatTest.java

```
// This file of JUnit tests was automatically created by the
// Eclat tool for generating test cases.  See
// http://pag.csail.mit.edu/eclat/ for more details.

// TODO: Add correct package name.
// package your.package.here;

import junit.framework.*;
import junit.textui.*;
import java.util.*;

public class PreferenceEclatTest extends junit.framework.TestCase {

  public PreferenceEclatTest(String name) {
    super(name);
  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }

  public static Test suite() {
    TestSuite suite = new TestSuite();

    // Inputs labeled as fault
    suite.addTest(new PreferenceEclatTest("test_0_getValue"));
    suite.addTest(new PreferenceEclatTest("test_1_getValue"));

    // Inputs labeled as normal, new behavior
    suite.addTest(new PreferenceEclatTest("test_2_getValue"));
    suite.addTest(new PreferenceEclatTest("test_3_setValue"));
    return suite;
  }

  // Eclat labeled this input as potentially fault-revealing.
  public void test_0_getValue() throws Exception {

    Preference var1 = new Preference();
    var1.setValue(3);
    int var30603 = var1.getValue(0);
    var1.setValue(-1);

    // Execution of the following method call violates properties
    // that were true in the execution provided to Eclat:
```

```
    // Execution of the following method throws java.lang.IndexOutOfBoundsException.
    int var41136 = var1.getValue(var30603);

    // TODO: Replace this call by the correct assertion.
    junit.framework.Assert.assertTrue(false);
  }


  // Eclat labeled this input as potentially fault-revealing.
  public void test_1_getValue() throws Exception {

    java.util.ArrayList var0 = new java.util.ArrayList();
    Preference var1 = new Preference();
    var1.setListValue(var0);
    var1.setValue(-2);
    var1.setValue(-1);
    int var30503 = var1.getValue(0);

  // Execution of the following method call violates properties
  // that were true in the execution provided to Eclat:
  // Execution of the following method throws java.lang.ArrayIndexOutOfBoundsException.
    int var41212 = var1.getValue(var30503);

    // TODO: Replace this call by the correct assertion.
    junit.framework.Assert.assertTrue(false);
  }


  // Eclat labeled this input as a normal input that exhibits.
  // behavior different from the execution provided to Eclat.
  public void test_2_getValue() throws Exception {

    Preference var1 = new Preference();
    var1.setValue(3);
    var1.setValue(-5);

    // Execution of the following method call violates properties
    // that were true in the execution provided to Eclat:
    //   precondition  violated on entry: var1.prefKey != null
    //   postcondition violated on exit : var1.prefKey != null
    //   postcondition violated on exit : var41217 == 1
    //   postcondition violated on exit : var41217 == var1.prefValue.size()
    int var41217 = var1.getValue(0);

    // TODO: Replace this call by the correct assertion.
    junit.framework.Assert.assertTrue(false);
  }


  // Eclat labeled this input as a normal input that exhibits.
  // behavior different from the execution provided to Eclat.
  public void test_3_setValue() throws Exception {

    java.util.ArrayList var0 = new java.util.ArrayList();
    var0.add(0, 1);
    Preference var1 = new Preference();
    var1.setListValue(var0);

    // Execution of the following method call violates properties
    // that were true in the execution provided to Eclat:
    //   precondition  violated on entry: var1.prefKey != null
    //   precondition  violated on entry: value == 1
    //   postcondition violated on exit : var1.prefKey != null
```

```
    var1.setValue(-3);

    // TODO: Replace this call by the correct assertion.
    junit.framework.Assert.assertTrue(false);
  }
}
```

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

[Ernst 2005b] Ernst, M., Pacheco, C., Automatic Generation and Classification of Test Inputs. MIT, 2005. Available at http://www.csail.mit.edu/research/abstracts/abstracts04/html/82/82.html