Minicopier application:
Case Study

Samer Yousef Khamaiseh
Department of Computer Science
Boise State University
Boise, ID 83725, USA
samerkhamaiseh@u.boisestate.edu

Laxmi Amulya Gundala
Department of Computer Science
Boise State University
Boise, ID 83725, USA
laxmiamulyagundala@u.boisestate.edu

Haritha Akella
Department of Computer Science
Boise State University
Boise, ID 83725, USA
harithaakella@u.boisestate.edu

Abstract- The purpose of this project is to understand and implement various testing techniques to get best productivity from the application-minicopier. Different components of this paper deals with explaining and implementing different approaches for testing a software system using various tools available within the scope. Each technique has limitations over its techniques explained in the later sections through comparison with other techniques.

INTRODUCTION

MiniCopier : Adrian Courreges has developed MiniCopier, an alternative to the traditionally available copy manager provided by OS. It is a graphical user interface application with all the operations done by "drag-and-drops" on the icons. Some of its features are:

1. It lets to create a queue of transfers, so only a file is copied at a time.
2. User can manage the queue by changing the position of any pending transfer.
3. It also handles pause and resume operations.
4. It handles to limit the speed of copy.
5. User can skip current transfer to proceed to next one.
6. User can add new transfers to the queue while the copy is already being processed.
7. If a transfer fails, the bytes already copied are not erased, so MiniCopier can resume the transfer later.
8. User can set the default behavior if target file already exists.
9. User can choose another name for target if file already exists.
10. User can follow or ignore symbolic links(Unix Systems only).

Requirements:

Unix system with: GNOME/KDE/XFCE or Windows or Mac OS X. Java virtual machine 1.5 (or more recent)

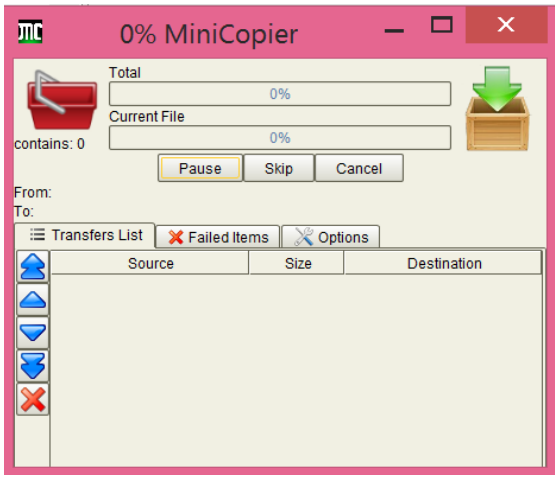
Use:

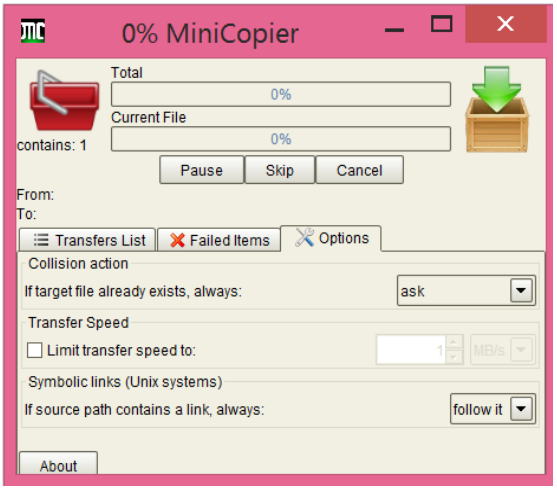
Drag and drop files or folder you wish to copy on the basket icon. Once you're done with filling the basket, drag and drop a destination folder to the box icon. All the content of the basket will be copied to that directory.

License:

MiniCopier is free software released under the GNU General Public License (GPL). The software is also distributed with the PgsLookAndFeel library, (Apache Software License, see https://pgslookandfeel.dev.java.net).

Figures below: MiniCopier Application





Version: v0.5

Testing Techniques:

White-Box Testing- It is a method of software testing the internal structures of an application as opposed to its functionalities. The tester should be able to understand the code to design test cases. Some of the testing tools used for current project are:

1. OpenJML

2. AspectJ

3. JavaMOP

Black-Box Testing- It is a method of software testing the functionalities of an application without the knowledge of its internal code/functional implementation. It is a higher-level testing where the tester is only aware of the system specifications. Some of the testing tools used for current project are:

1. Randoop

2. TSL

3. CASA

It is crucial also to test the design of the application through code coverage as it determines the efficiency of the testing approaches. EclEmma is used as a coverage testing tool.

The later part of this paper deals with each of the testing approaches and implementations used towards MiniCopier.

I.TEST METHOD BASED ON FORMAL SPECIFICATION

Specification of a program's properties in a language defined by a mathematical logic is Formal specification. Software requirements and software design could be well understood with the help of formal specifications. Formal system specification and a formal programming language definition can be used to check if a program conforms to its specifications. Mathematical methods are used to study and analyze formal specifications. Formal specifications help testers to identify appropriate test cases for a component. Activities such as Validation, Verification, Testing Consistency- and Completeness-Check are the main properties of formal specifications.

DISCUSSING OPENJML:

OpenJML is a tool for verification (checking consistency of code and specifications) of Java programs. It is an implementation of the Java Modeling Language (JML) and is built using the OpenJDK compiler. JML is a behavioral interface specification language for Java. It was designed primarily to support design-by-contract (DBC) Invented by Gary T. Leavens in the 90s. Design By Contract (DbC) is a software correctness methodology. It uses preconditions and postconditions to document (or programmatically assert) the change in state caused by a piece of a program. OpenJML can be used both as a command-line tool and through a GUI built on Eclipse. To describe our experience working with the tool, OpenJDK and Eclipse, there were many advantages in specification-based software verification.

This paper demonstrates writing specifications for properties using OpenJML. OpenJML was chosen because of its advantages. It Checks JML structure by parsing and type checks it. It has Eclipse plug-in and jars for command-line use. Its RAC capabilities which allows to check for property violations during program execution are commendable. There are 21 properties that have been defined for MiniCopier software which reflect the desirable behavior of the program. 16 of these are single properties and rest 5 are sequential properties. One such property from MiniCopier software is ‘Creating file to be transferred’. This property creates the file that has to be transferred and is from a class called FileToTransfer.java. JML was used to define invariants, pre and post conditions of the method FileToTransfer as shown in Figure1.

Pre-conditions: FileToTransfer initially does not create file in the source and destination file/folder.

Post-conditions: FileToTransfer should create files to source and destinations and should have references to them.

```
/*@
       @ requires _path2source !=null;
       @ requires _path2destinationfolder !=null;
       @ ensures this.sourcePath !=null;
       @ ensures this.name !=null;
       @ ensures this.destinationFilePath !=null;
       @ ensures this.destinationFilePath !=null;
       @
       @*/


public FileToTransfer(String _path2source,
                String _path2destinationfolder) {

            //Using the argument Strings directly work,
but the
            //display is not fine for Windows, so we
create Files
            //and get the path (OS-dependent).

            //this.sourcePath = _path2source;

            File sourceFile = new File(_path2source);

            this.sourcePath = sourceFile.getPath();

            this.name = sourceFile.getName();
            this.size = sourceFile.length();

            //this.destinationPath =
_path2destinationfolder;


            File destFolder = new
File(_path2destinationfolder);
            this.destinationPath = destFolder.getPath();

            this.destinationFilePath =
this.destinationPath
            + File.separator + this.name;


      }
```

Figure: OpenJML for FileToTransfer method

Key terms in JML implementation include:

1. Comments: JML specifications are written inside Java comments immediately before class members, e.g., fields, methods. (Field invariants can be written after fields also).
   - Newline comments of the form: //@
   - Block comments: /*@...@*/
   - Multi-line block comments: /*@... @... @*/
2. Invariant: Invariant is a property that should always hold at observable points, e.g., entrance/exit of a method, end of a constructor. Invariant can be an instance (default) or static.
   - Static invariant constraints class static fields.
   - Instance invariant constraints class non-static fields. Only instance invariants have been considered in the example provided.
3. Method Specifications: Lightweight specifications have only been considered. Method (or constructor) clauses include

- Pre-conditions: requires clause, It specifies method preconditions.
- Post-conditions: ensures clause, It specifies method postconditions, as mentioned in the example in Figure 1.
- Frame conditions: assignable clause which declares what locations the methods can modify

II. ASPECT-ORIENTED PROGRAMMING (AOP):

Aspect-oriented programming (AOP) has been proposed as a technique for improving separation of concerns in software.

DISCUSSING ASPECTJ

Aspect-oriented extension to Java is AspectJ. In developing AspectJ, the aspects for any program are the cross-cutting conerns. Concerns determine the abstracted functionalities of a class or method in the program. The key terms in AspectJ model include:

1. Join Points: well-defined points in the execution of the program constitute Join Point;

2. Pointcuts: Collections of join points make a pointcut;

3. Advice: Special method-like constructs are advices and can be attached to pointcuts;

4. Aspects: A class like feature depicting the concerns along with advices and pointcuts.

This paper presents an overview of AspectJ, including a examples on how AspectJ can be used by implementing AspectJ on the single properties. These examples show that using AspectJ we can code, in clear form, crosscutting concerns that would otherwise lead to tangled code. AspectJ for MiniCopier was implemented using pointcuts and the advices that were used were- before, after and around. Major discussions were involved while implementing pointcut, to choose call() at call site of the fully defined method or execution() inside the method body initial. Initialization of any object that is started with the defined constructor happens using initialization() method. Aspects were written to monitor the 16 properties. One such example can be found in Figure 2. AspectJ monitor for FileToTransfer method was generated.

Commonly used advices are before() advice which runs upon reaching join point. after() returning – runs after a normal return from the method. after() throwing – runs after the call to the method throws an exception. after() – runs regardless of how the method was exited. around() – runs instead of the advised method, if the method has to be called then proceed() has to be used. before() and after() have been implemented in the example provided.

An advice declaration identifies arguments and connects those arguments to three primitives of a pointcut:

1. This: this is the currently executing object

2. Target: target is the target object on which is method defined in the pointcut is called on , ex: ftt is the target object

3. Args: args are the arguments of the advised method, ex: _path2Source, _path2destinationfolder are the arguments.

AspectJ encapsulates several concerns into a single “class” called an aspect, located in their own file is an advantage with aspectj.

```
pointcut fileToTransfer(FileToTransfer ftt, String _path2Source,
                String _path2destinationfolder):
initialization(FileToTransfer.new(String,String))
                && target(ftt)
                && args(_path2Source, _path2destinationfolder);
      before(FileToTransfer ftt, String _path2Source,
                String _path2destinationfolder): fileToTransfer(ftt,
_path2Source, _path2destinationfolder)
                {
                if(_path2Source == null)
                {
                      System.err.println("path to source cannot be null");
                }
                if(_path2destinationfolder == null)
                {
                      System.err.println("path to destination cannot be null");
                }
                }
      after(FileToTransfer ftt, String _path2Source,
                String _path2destinationfolder): fileToTransfer(ftt,
_path2Source, _path2destinationfolder)
                {
           if(ftt.sourcePath == null)
           {
                System.err.println("source path cannot be null");
           }
           if(ftt.destinationPath == null)
           {
                System.err.println("destination path cannot be null");
           }
           if(ftt.size == 0)
           {
                System.err.println("size cannot be 0");
           }
           if(ftt.name == null)
           {
                System.err.println("name cannot be null");
           }
                }
}
```

Figure: Generated AspectJ monitor for FileToTransfer method

III. MONITORING-ORIENTED PROGRAMMING (MOP):

A software development and analysis technique in which monitoring plays a vital role is Monitoring-oriented programming (MOP). Users of MOP have a freedom to add domain-specific requirements specification formalisms into their framework. This is done by means of logic plug-ins. Those plug-ins generally comprise of monitor synthesis algorithms for properties expressed in form of formulae. The properties are specified with declarations stating how and where to automatically integrate monitor into the software system.

DISCUSSING JAVAMOP:

MOP is an environment for developing robust Java oriented application programs. In MOP, monitoring is the fundamental principle supported. Monitors are integrated at appropriate places in the program, as per the tester requirements. It was a great experience working with JavaMOP to monitor sequencing properties. AspectJ were successfully derived from the JavaMOP files. The generated monitor could be compiled using any AspectJ compiler. Monitors are automatically synthesized from formal specifications.

To begin working with JAVAMOP, properties that are to be monitored should be specified. The specifications must be stored in files with .mop extension. Figure3 is an example of JavaMOP file called add2basketBeforetreatQueue.mop. Sequential property - add2basketBeforetreatQueue has been specified in this file. Extended regular expressions have to be written as shown below:

ERE: for events {”Basket : add2Basket ” , ”Copier : treatQueue ”}
~[”treatQueue”]∗(~[”treatQueue”,”add2Basket”]∗; ”add2Basket”; .∗)

```
package test;

import java.io.*;

import java.util.*;

import minicopier.*;

add2basketBeforetreatQueue(Basket i){

    event add2Basket before(Basket i): call (* add2Basket(..)) &&
target(i){}

    event treatQueue before(Basket i): call (* treatQueue()) &&
target(i){}

    ere: ~treatQueue* |~(treatQueue | add2Basket)* add2Basket *

    @match{
        System.out.println("pattern matched");
}
    @fail {
        System.err.println("! add2Basket() not called before
treatQueue()");
        __RESET;
    }
}
```

Figure. MOP specification for add2basketBeforetreatQueue

Extended regular expressions extend regular expressions with complement (negation).They specify properties non-elementarily more compactly and are more complicated to monitor.

After writing monitors using JavaMOP for the sequencing properties as mentioned, AspectJ was generated. Figure 4 refers to sample AspectJ code generated from JavaMOP.

```
263 »   // Trees for References¶
264 »   static javamoprt.map.MOPRefMap add2basketBeforetreatQueue_Copier_RefMap = add2b
265 ¶
266 »   pointcut MOP_CommonPointCut() : !within(javamoprt.MOPObject+) && !adviceexecuti
267 »   pointcut add2basketBeforetreatQueue_add2Basket(Copier i) : (call(* add2basket(.
268 »   before (Copier i) : add2basketBeforetreatQueue_add2Basket(i) {¶
269 »   »   add2basketBeforetreatQueue_activated = true;¶
270 »   »   synchronized(add2basketBeforetreatQueue_MOPLock) {¶
271 »   »   »   add2basketBeforetreatQueueMonitor mainMonitor = null;¶
272 »   »   »   javamoprt.map.MOPMap mainMap = null;¶
273 »   »   »   javamoprt.ref.MOPWeakReference TempRef_i;¶
274 ¶
275 »   »   »   // Cache Retrieval¶
276 »   »   »   if (i == add2basketBeforetreatQueue_i_Map_cachekey_0.get()) {¶
277 »   »   »   »   TempRef_i = add2basketBeforetreatQueue_i_Map_cachekey_0;¶
278 ¶
279 »   »   »   »   mainMonitor = add2basketBeforetreatQueue_i_Map_cachenode;¶
280 »   »   »   } else {¶
281 »   »   »   »   TempRef_i = add2basketBeforetreatQueue_i_Map.getRef(i);¶
282 »   »   »   }¶
283 ¶
284 »   »   »   if (mainMonitor == null) {¶
285 »   »   »   »   mainMap = add2basketBeforetreatQueue_i_Map;¶
286 »   »   »   »   mainMonitor = (add2basketBeforetreatQueueMonitor)mainMap.getNode(Te
```

Figure: AspectJ file generated from MOP specification

VI. TEST METHOD BASED ON FEEDBACK DIRECT RANDOM TEST GENERATION

Random testing is a type of black box testing technique where, test cases are generated randomly based on independent inputs. This approach has been used where the testing time is short and the complexity of application preventing the testers to test every combination. According to many studies [1, 2, 3] random testing is not powerful and efficient as systematic testing due to different reasons such as test cases duplication, illegal test cases and missing test cases. Feedback-Direct Random Test Generation is a technique represented to improve and enhance the random test cases generation. The new test inputs are generated by Feedback-Direct Random Test Generation guided and extended by the previous inputs execution. Feedback-Direct Random Test Generation start from an empty set of sequences of inputs and build the inputs incrementally by using a random selection of method call and finds arguments through previously constructed inputs. Once the sequence of inputs is built, it is executed and check against a set of filters and constraints to determine the situation of the input such as redundant, illegal, contract violating or can be used to generate more inputs. Such techniques can help towards finding new and legal sequences of object states. The advantage of this technique is eliminating the opportunity to create illegal and redundant test cases as random test generation do. In terms of error detection and coverage, many studies show that Feedback-Direct Random Test Generation is more efficient than systematic and randomized test generation [4]. Feedback-Direct Random Test Generation has been implemented with Randoop tool and can be used without any need for initial inputs from the user.

DISCUSSING RANDOOP:

Randoop is an automatic test case generator, it uses Feedback-Direct Random Test Generation to generate test cases. Randoop executes the sequences that are created based on Feedback-Direct Random Test Generation technique .It will create assertions that capture and cover the behavior of the program. Randoop can be used to create two types of Unit Tests, the first one is contract-violating tests that covers scenarios where the code of the program will violate the contract, and the second one is Regression test that helps to find out inconsistencies among different versions of the programs. Randoop has been used to generate test cases for Minicoper application. Source code [excluding the GUI package] with all classes in the packages are used as targets for testing. For each iteration, different seed number was specified and a different number of test cases are generated.

Randoop has many benefits such as,

1. easy to use.
2. no prior knowledge required to test the program.
3. Automatically generates test cases for the given packages/ classes.
4. does not require additional inputs from the users.
5. The ability to ensure no change in program behavior.

Randoop also has weak points such as

1. The difficulty to understand and debug the test cases generated.
2. A Large number of test cases to get good coverage for the targeted source code.
3. No support of the category-partition method where, the tester cannot select specific arguments for specific test cases.

V. THE CATEGORY-PARTITION METHOD FOR SPECIFYING AND GENERATING FUNCTIONAL TESTS

For any application, the tester uses the given system specifications and develops test suites. However, it isn't always possible to allow the tester to change the test specifications. One of the methods to make this approach feasible is the category-partition method. Through this approach, the tester can analyze the system specifications, understand to write test specifications. Then, the tester can use an automatic generator tool to get test descriptions that can be used to write test scripts. Using this approach, test

specification can be easily modified depending on necessity and complexity. And can limit the number of test cases that are sufficient to test all specifications.

Functional testing a software system should not show any dissimilarities between the implemented functions and desired system behavior. To ensure this, tests have to be executed for all the functions and find maximum errors as they are required for maximum productive testing. Also, eats should be focused on the most vulnerable parts of the system implementation. However, writing excellent test cases focused only on some parts of the software or only on certain characteristics is not enough.

The main characteristics of category-partition method according to authors are

1. The test specification is a uniform document of test function implementation.
2. The test specification can be easily modified based on requirement changes and necessity.
3. The test specification is logically feasible to limit the number of test cases.
4. The generator tool produces tests for each function avoiding undesirable or impossible combinations on specifications/ parameters.
5. The method focuses on maximum coverage over a specification and aspects prone to errors.

This approach was developed by THOMAS J. OSTRAND and MARC J. BALCER.

DICUSSING TSL TOOL

TSL interprets the implementation of category partition method by Oswald and Balcer.

For TSL, there are certain terminology.

1. Comments: '#' denotes a common until a new line.
2. Categories: The one with ':' ending is called a category. However 'Parameters:' and 'Environments:' category divisions are ignored by TSL.
3. Choice: Those with '.'(period) character ending are called choices.
4. Constraints: Anything between '[' and ']' are the constraints marked on choices. They are not applicable to categories and are tae sensitive.
5. Property Lists: It is a type of constraint with 'property' keyword to identify it. The keyword is stated only once for every constraint. They are case sensitive and can have 1-10 properties in a single constraint.
6. Error Marking: It is a type of constraint with 'error' keyword. It gives only one single test frame. It denotes for error test cases.(optional)
7. Single Marking: It is a type of constraint with 'single' keyword. It gives only one single test frame. Different from Error marking. (optional).
8. Selector Expressions: It is a type of constraint with 'if' or 'else' keyword followed by a expression. '[else]' is optional. Expressions can be consisting of logical operators '&&','||', '!' with precedence order as not>and>or.

Depending on the software, its system functionalities are grouped into 'categories' with set of 'choices' with 'constraints' on them.

Below is the example (part of) of a TSL formal test specification file for unzip application:

```
#start of file.
Parameters:
    Function:
        compress.                       [property Comp]
        decompress.                     [property Dcom]
    Info:
        help.                           [single]
        checking.                       [single]
    Suffix:
        myGZ.                           [property myGZ]
    Outputs:
        regular.                    [if !myGZ || !Dcom]
    Input file name:
        good file name.
        no such file.                   [error]
    File state:
        file compr.                     [if Decomp]
        file uncompr.                   [if Comp]
        incorrect format.               [error]
#end of file.
```

using the TSL tool, test frames are generated. One of the test frames is:

```
Test Case 6                 (Key = 2.0.1.0.1.1.)
  Function       : decompress
  Info           : <n/a>
  Suffix         : myGZ
  Outputs        : <n/a>
  Input file name : good file name
  File state     : file compr
```

'Test Case 6' denotes the test frame with frame number. 'Key' gives the combination used for current frame. Next are the chosen choices on categories.

MiniCopier with TSL

For MiniCopier, TSL was used to identify various categories and set of available choices. Initially few complexities were observed while testing using TSL. MiniCopier is a GUI based application and most of categories and choices were dynamic and targeting the internal threads of the application. Also, few button click events were individual functions. All such functions are categorized to Ramblings and the choices are manually tested.

```
#TSL file
Parameters
        Input basket:
                file.                   [property file]
                folder.                 [property folder]
        file size:
                <1024.                  [if file]
```

>1024. [if file]

output basket:

folder.

collision action:

ask. [property ask]

cancel copy.

overwrite.

overwrite if older.

resume.

Transfer speed:

limit.

do not limit.

symbolic links:

follow it.

ignore it.

ask:

overwrite. [if ask]

rename. [if ask]

resume. [if ask]

cancel. [if ask]

Ramblings:

checkEmpty. [single]

pause. [single]

skip. [single]

cancel. [single]

isBusy?. [single]

unPause. [single]

clearBasket. [single]

Buttons. [single]

TSL generated 104 test frames with various combinations on the available set of choices. However, almost all of the MiniCopier functionalities are based on user choices and hence, there was little scope for errors in the application.

Ex: one of the test frames:

Test Case 9 (Key = 1.1.1.1.1.1.1.0.)

Input basket : file

file size : <1024

output basket : folder

collision action : ask

Transfer speed : limit

symbolic links : follow it

ask : overwrite

Ramblings : <n/a>

A JUnit test drive is created to test those 104 test frames. Almost all the test cases are automated and around 15 test cases are manually tested. A new method is added to copier to ease the usage of various choices on collision action.

VI. THE COMBINATORIAL DESIGN APPROACH FOR AUTOMATIC TEST GENERATION

For a software system with thousands of lines of code (or sometimes even more), there might be innumerable number of test cases. Testing each one of them takes lots of time and effort. Instead, tester has to choose among them that are sufficient to get maximum productive testing. Also, according to ISO standards, every individual system requirement must be tested. However, there is no guarantee that these individual requirements will give a desired functionality when combined. The tester needs a methodology to select few test cases among the astronomical number of test cases. The combinatorial design method gives a scope to identify test cases that can give maximum productive testing in short time. This approach substantially reduces test costs with good code coverage and fault detection ability.

This approach was designed by DAVID M. COHEN, SIDDHARTHA R. DALAL, JESSE PARELIUS, AND GARDNER C. PATTON.

DISCUSSING CASA TOOL

CASA is one such tool that implements combinatorial design approach with constrained interaction testing. (CIT-Combinatorial Interaction Testing).

For CASA, there are 2 types of input files.First if for CIT model and second is for constraints.

The model file has the following format:

[strength of testing (t)] \n

[number of options (k)] \n

[number of values in the first column (v_0)] [more v_is] \n

For example, if we are 4-way testing a system with four binary options and a ternary option, the model file could be written:

4

5

2 2 2 2 3

Here, options(CASA) is same as categories(TSL) and values(CASA) is same as choices(TSL).

From left to right each option is given symbols starting from 0. In the example given, symbols 8–10 are given to the ternary option because the four binary options have taken the symbols 0–7.

Constraints are written as a conjunction of disjunctions over these symbols, in the following format:

[number of disjunctive clauses] \n

[number of terms in this disjunctive clause] \n

[+ for plain, - for negated] [variable] [more plain or negated variables...] \n

[more disjunctive clauses...] \n

explaining with previous example:constraint 1- let us suppose if the ternary option takes its value with lowest symbol then the first binary option cannot its value with lowest symbol.

constraint 2- if the second binary option chooses its second value, then first value of third binary option can be chosen or second value of the fourth binary option can be chosen.

so, the constraints can be encoded as

2

2

- 0 - 8

3

- 3 + 4 + 7

The output for the two CASA files (.citmodel and .constraints) follows the same symbol numbering as done in model file and constraints file. The output file format is:

[number of configurations (rows)] \n

[chosen value for the first option in the first configuration] [more chosen values for the first configuration] \n

[more configurations] \n

hence, the output for above given example is

27

0 2 4 6 10

0 2 4 7 9

0 2 5 6 10

0 2 5 6 9

0 2 5 7 10

0 3 4 6 10

0 3 4 6 9

0 3 4 7 10

0 3 5 7 10

0 3 5 7 9

1 2 4 6 8

…… (27 combinations)

The test cases derived through CASA are part of test frames derived through TSL.

MiniCopier with CASA

CASA is used to give constraints on the available test cases using combinatorial design approach so as to reduce test cost and time.

CIT model:

2

4

5 2 2 5

Here, the options with respect to TSL are collision action, Transfer speed, Symbolic links and ask (5th choice is <n/a>).

Contraints are:

5

5

-0 +9 +10 +11 +12

2

-1 +13

2

-2 +13

2

-3 +13

2

-4 +13

Constraints are based on condition that only when if user selects ‘ask’ from ‘collision action’, 4 of the available choices under ‘ask’ category can be chosen.

CASA generated 16 valid test cases with these combinations:

16

0 5 8 9

2 6 8 13

4 5 7 13

4 6 8 13

2 5 7 13

0 5 7 10

0 6 7 9

3 6 8 13

3 5 7 13

0 6 8 10

1 6 8 13

1 5 7 13

0 6 8 11

0 5 7 11

0 6 8 12

0 5 7 12

All these test cases are part of TSL test frames.

VII. COVERAGE TOOLS

For any software system, with all the test cases available, it is difficult to identify how much of the code is covered by running test cases. It is important to be able to identify the coverage as it will give the analysis on how efficient all the thousands of lines of code written for the application. Also, it is crucial to have test cases that can give maximum code coverage. These factors lead to usage of coverage tools.

DISCUSSING ECLEMMA

Code coverage analysis can be achieved by using EclEmma as an eclipse plug-in. It is the Eclipse integration of JaCoCo, a free code coverage library. EclEmma gives Rich-coverage analysis and is the fastest of the available.

EclEmma provides coverage analysis for

1. Instruction counters
2. Branch counters
3. Line counters
4. Method counters
5. Type counters
6. Complexity

COVERAGE COMPARISON

Different tools showed different percentages on coverages. It was possible to check for black box testing tools. White box testing tools focused on target methods and hence didn't give acceptable results.

Coverage for Randoop: Coverage data is obtained for Branch counters. For each iteration we specified different seed number and different number of test cases generated, and then use ECLEMMA to get coverage for the MiniCopier. Table below shows coverage with respect different seed values.

Seed Number	*Number of Test Cases*	*Branch Coverage*
0	8749	49.9
1	9568	46.1
3	6879	47.4
5	7895	47.2
8	8563	46.8
10	8545	46.3

Table: Seed vs Coverage

Given below is the data for seed 0 that gave maximum branch coverage.

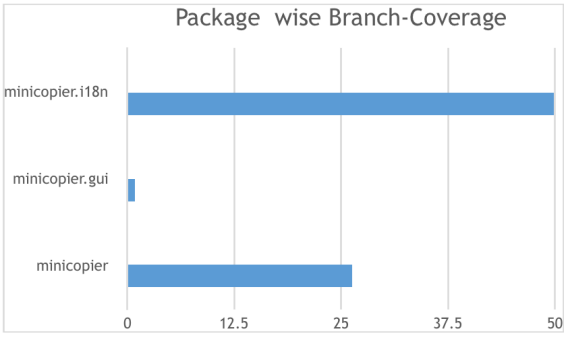


Fig:Package wise Branch-Coverage

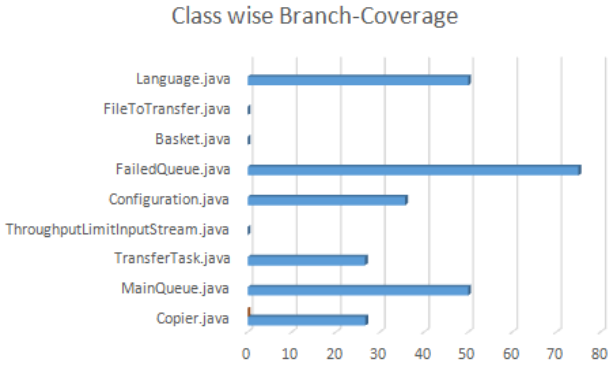


Fig: Class wise Branch-Coverage for Randoop

Coverage for TSL and CASA: Coverage is determined as method counters. Branch coverage didn't give appreciable results because there were very less branches with multiple levels.

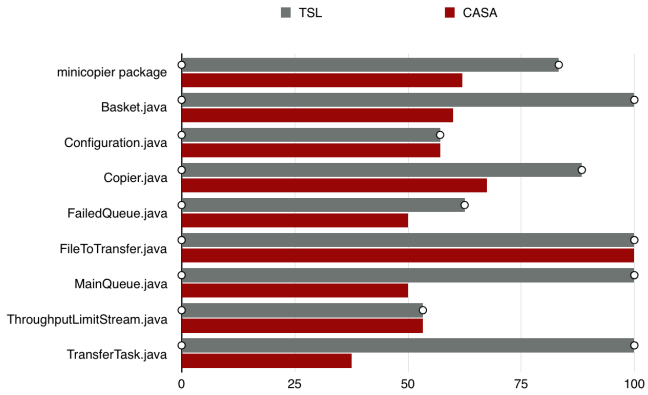


Fig: Method coverage

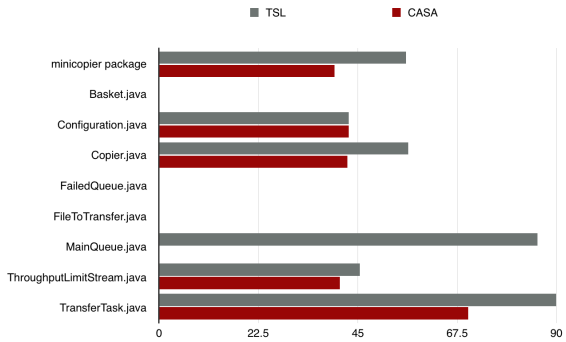


Fig: Branch Coverage

Identified Bugs:

MiniCopier is successful in providing many features for copy operation. It still has 2 limitations.

1. Percentage of copy operation is determined by size of file. If a file size of 0 bytes is copied and transfer is successful, the percentage shown is 0%. This scenario was not handled by the developer.
2. User cannot know the status of a copy operation unless there is any failure. If a file is copied from source and destination folder is same as source and Collision action is chosen as 'overwrite if older', there is no status on the copy operation.

REFERENCES

1. http://www.adriancourreges.com/projects/minicopier/
2. https://en.wikipedia.org/wiki/Black-box_testing
3. https://en.wikipedia.org/wiki/White-box_testing
4. http://sir.unl.edu//content/tsl-spec.php
5. http://web.soccerlab.polymtl.ca/log6305/protected/papers/Ostrand_CategoryPartition.pdf
6. http://cse.unl.edu/~citportal/
7. http://aetgweb.appcomsci.com/papers/1996-software.html

8. http://eclemma.org
9. http://eclemma.org/installation.html
10. http://www.cin.ufpe.br/~damorim/publications/chen-damorim-rosu-05.pdf
11. http://liacs.leidenuniv.nl/~bonsanguemm/Toos/P14_javamop-ere.pdf
12. http://arxiv.org/pdf/1404.6608.pdf
13. http://www.eecs.ucf.edu/~leavens/JML//index.shtml
14. http://www.ccs.neu.edu/home/lieber/com3362/w02/lectures/working-lectures/ECOOP2001-Overview.pdf
15. http://hugunin.net/papers/aosd-2004-cameraReady.pdf

[1] R. Ferguson and B. Korel. The chaining approach for software test data generation. ACM TOSEM, 5(1):63-86, Jan. 1996.

[2] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard.An evaluation of exhaustive testing for data structures. Technical Report MIT/LCS/TR-921, MIT Lab for Computer Science, Sept. 2003.

[3] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. IEEE TSE, 16(12):1402-1411, Dec. 1990.

[4] C. Pacheco and M. D. Ernst, Randoop: feedback-directed random testing for Java," in OOPSLA 2007 Companion, (Montreal, Canada), ACM, 2007.