The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MEng in Software Engineering.

Establishing Consistency between EuGENia and EuGENia Live

Thomas Wormald

19th March 2014

Supervisor: Dr. Louis M. Rose

Number of words = 0, as counted by `wc -w`.
This includes the body of the report only.

Abstract

Something...

I would like to thank McVitie's for producing Hobnobs

Acknowledgements

Thanks for all your help

Contents

List of Figures

List of Tables

Listings

Part I

Preliminaries

1 Introduction

Welcome to the best project write-up ever.

2 Literature Review

2.1 Introduction

In this chapter I will give an overview of the existing literature that is appropriate to my project. The chapter is split into two sections. The first section gives a review of Model Driven Engineering and tools that can be used for implementation. The second section investigates software testing methods and ways of assessing the quality of software tests.

2.2 Model Driven Engineering

2.2.1 Introduction

Model Driven Engineering is a development methodology that aims to reduce the amount of time spent on projects, as well as increasing the consistency and quality of the item or system under development. One example of when model driven engineering is useful is when developing applications. A model of the system can be developed. This model can then be converted into code. Assuming that the model is correct, human typing errors are avoided and precious development time can be spent on more important aspects that hunting for trivial bugs. Maintenance is also easier, as changes can be applied to the model, and the updated code will be generated automatically from the model. Finally, the code generation can be to a multitude of languages and platforms, further reducing time spent on development [6].

In the 1980's there was a software quality crisis that lead to the search for alternative approaches to developing software. Model Driven Engineering is one solution that was of interest at the time as it provided a way to visually represent a system architecture, and from that generate code automatically. However, the return on investment that companies were expecting from model driven engineering was far too high, causing much disappointment and disillusionment, and for a while the concept was sidelined. More recently, the Object Managment Group (OMG) have promoted and developed a Unified Modeling Language (UML), and tools such as Epsilon have further promoted the use of MDE [7]. Brambillia [1] believes that Model Driven Engineering is now past the 'trough of disillusionment' and into the 'slope of enlightenment' (see figure 2.1)

2.2.2 Model

A model is a representation of something that abstracts away many details that are not necessary for its use [1]. For example, the Utah Teapot [**?**] is a model of a teapot that is

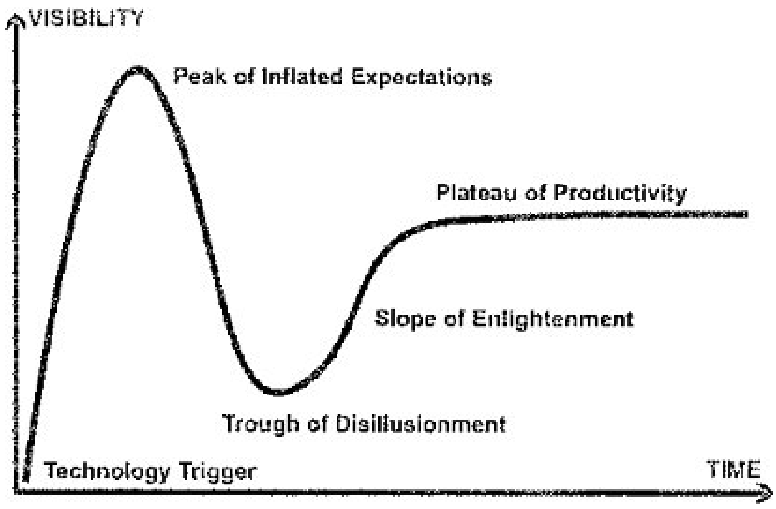


Figure 2.1: The technology hype cycle according to Brambillia [1]

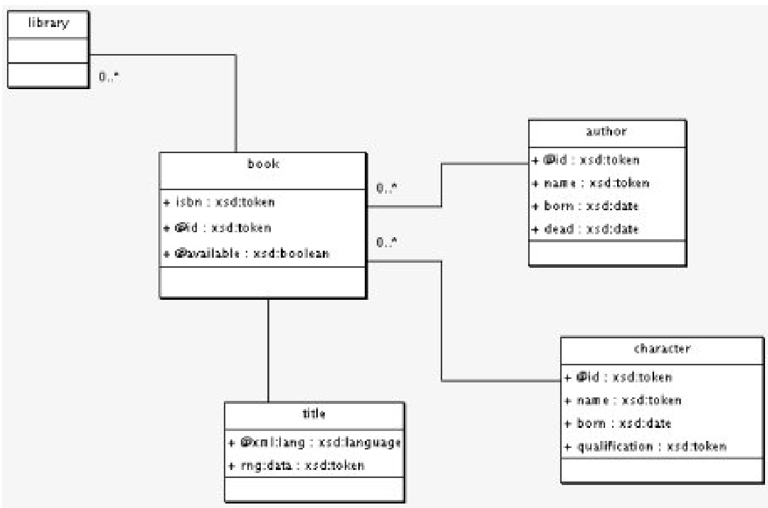


Figure 2.2: A sample model

rendered by a 3D engine. However, many aspects of the teapot are not considered in its model, as they are not necessary for a simple render. An example is that the lid is not a removable component, because for the purpose of rendering the teapot, the lid never has to be removed. Another example is that the only physical property of the teapot that will be included in the model is it's finish (texture), so that lighting and reflection can be calculated. Other details such as its weight will not be included in the model, because it isnot necessary.

UML (Unified Modeling Language) is a language that is designed specifically for representing models visually. It is ideal for object-oriented design, as it represents classes with boxes and links between classes with lines. Within the boxes there are definitions of the classes (spelling?) methods and fields. In figure 2.2, author is a class that has properties such as name, born, dead. The diagram also shows that there is a connection between author and book.

2.2.3 Metamodels

To have a modeling language, there must be a specification of that language that defines the valid syntax, constraints etc. In the case of UML, the Object Management Group provide a detailed specification [**?**] of the language, and we can check that any diagram is a UML diagram by checking it against the UML specification. A metamodel is the specification of a modeling language, in the form of a model [1]. A metamodel could be represented visually or textually, depending on the specification of the metamodeling language. As with many aspects of computer science, the metamodel is just another layer of abstraction, and we can continue to abstract to higher and higher levels. A metametamodel (known as M3) will define the specification of a metamodeling language, and the abstraction can continue as far as is required.

Going back to the example of The Utah Teapot, the metamodel in this case may define that the teapot is made up from interconnected polygons, and specify that each polygon has a location and size given in 3D space.

Abstract and Concrete Syntax

When building a metamodel, both the abstract and concrete syntax must be defined. The abstract syntax of a language is a definition of how the language components interact. For an OO language, the abstract synatax would specify that a class can inherit the properties of another class, that a class must have a constructor, and that a class must be given a name. How these requirements are met by the user is specified by the concrete syntax. The concrete syntax for allowing class inheritance would state that the colon symbol must be used after the class name:

```
class NewClass : ParentClass
```

The concrete syntax does not necessarily need to be textual. To build a modeling language you require a metamodel, which is the abstract syntax. You also require a way to visually display the model. The concrete syntax could state that a class is represented as a rectangle with the name of the class in the middle, and that to show inheritence the NewClass must have an arrow coming out of it that goes to the ParentClass that it is inheriting from.

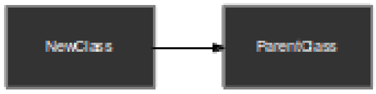


Figure 2.3: Concrete Syntax example for a modeling language

2.2.4 Graphical Modeling Framework

The Eclipse graphical modeling framework (GMF) is part of the Eclipse Graphical Modeling Project [8]. The Eclipse Wiki [9] defines GMF as:

> Using GMF, you can produce graphical editors for Eclipse. For example, a UML modeling tool, workflow editor, etc. Basically, a graphical editing surface for any domain model in EMF you would like.

2.2.5 Epsilon

To be able to perform Model Driven Engineering, we of course require some tools and languages to build and manipulate models. These tools could be built from scratch for each project, but that would be a waste of time.

Epsilon is a suite of languages and tools that provide all the necessary components to build and manipulate models. Epsilon stands for **E**xtensible **P**latform of Integrated **L**anguages for M**O**del Ma**N**agement [10]. It is part of the Eclipse Modeling Project [**?**], and includes tools for each of its languages that integrate with Eclipse. From the Epsilon Website [10], the languages that are provided by Epsilon are:

EOL Epsilon Object Language is an expression language that is used to create, query and model EMF models.

ETL Epsilon Transformation Language is a model-to-model transformation language.

EVL Epsilon Validation Language is a model constraint language.

EGL Epsilon Generation Language is a model-to-text generation language that can be used to generate code from models.

EWL Epsilon Wizard Language is similar to ETL, except that ETL performs batch operations whereas EWL works with in-place model transformations based on user selections.

ECL Epsilon Comparison Language is a model comparison language.

EML Epsilon Merging Language is used to merge models of diverse metamodels.

Epsilon Flock A rule

Together these languages provide a powerful framework for model driven engineering.

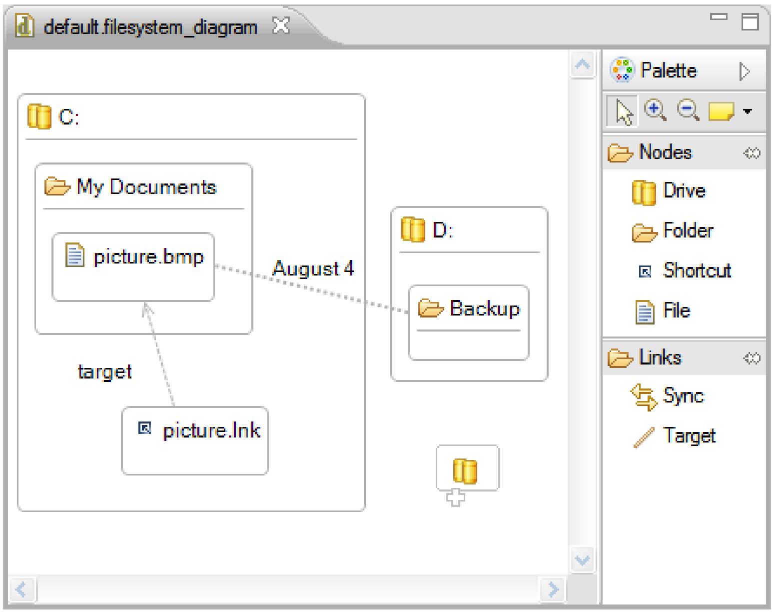


Figure 2.4: A sample gmf editor generated by EuGENia [2]

2.2.6 EuGENia

EuGENia is one of the tools that is included with Epsilon. EuGENia takes an Ecore metamodel specification and generates a GMF editor [2]. From the code in **??**, the model editor shown in figure 2.4 is generated by EuGENia.

The generator editor provides the objects shown on the right hand side of figure 2.4. These objects are then dragged to the left hand section of the editor by the user, where connections between objects can be intuitively created.

2.2.7 EuGENia Live

EuGENia Live is a web-based version of EuGENia that removes some of the complexity of getting started with EuGENia. The EuGENia Live Paper [11] describes EuGENia Live as:

> ... a tool for designing graphical DSLs

However, unlike EuGENia, EuGENia Live is a visual tool that allows you to switch back and forth between graphical editing of a DSL and the code for the DSL. Figure **??** shows EuGENia Live graphically editing a DSL, and Figure 2.6 shows the same DSL's code being edited in EuGENia Live.

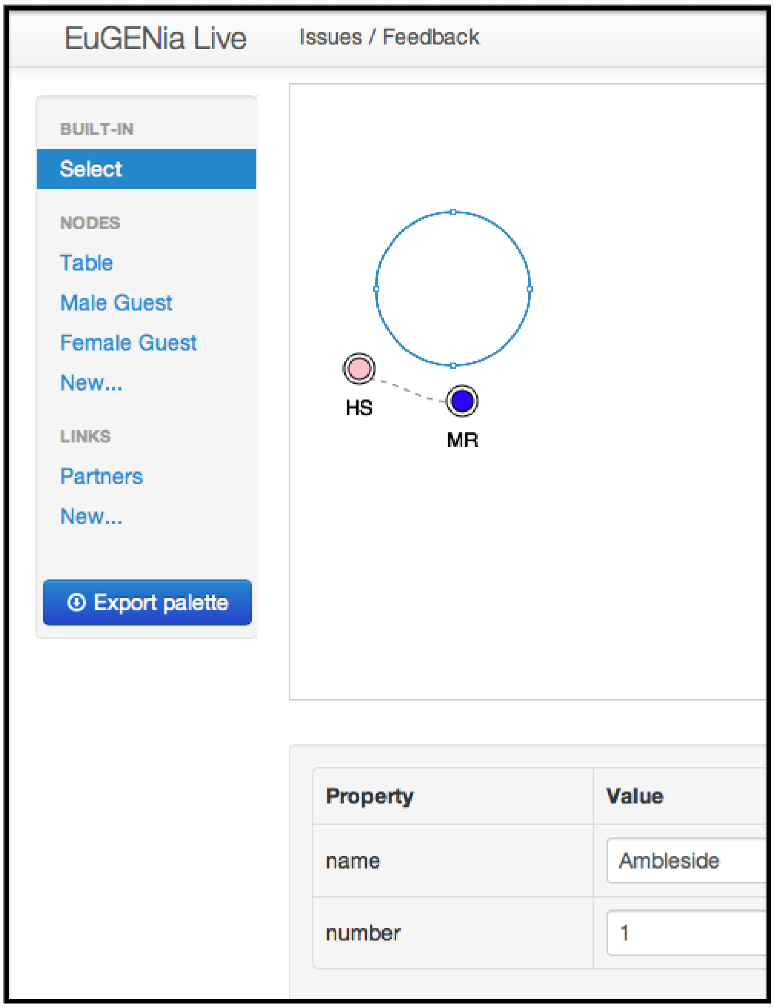


Figure 2.5: EuGENia Live's Graphical Editor [3]

2.3 Quality of Software Testing

2.3.1 Introduction

Software testing is a crucial part of the development cycle of any serious piece of software. Developers can make changes to code that make it do what they want, but could break another part of the program that uses the same code. Suites of tests can be implemented that *should* notice if a developer breaks the code, but there is the possibility that the correct tests have not been implemented to catch a certain fault.

The Ariane 5 rocket cost $7 billion to develop, and so of course any software on board would have had test suites to ensure that it did not fail. Unfortunately, 37 seconds after launch, $500 million of rocket and cargo exploded because of an integer overflow [12]. Despite having software tests, the test 'coverage' must have not been sufficient, leading to such a disaster. This is of course an extreme example, but it highlights the need for not only software testing, but good quality software testing.

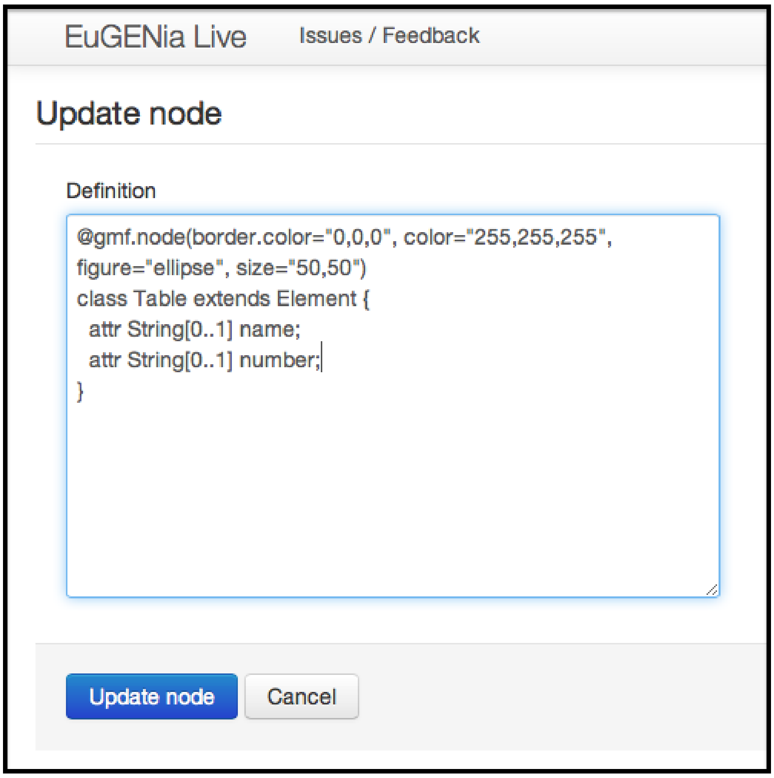


Figure 2.6: EuGENia Live's Code Editor [3]

2.3.2 Coverage

Test sets have a *coverage criterion* that measures how good a collection of sets is [13]. According to Paul Ammann [13], coverage is defined as:

> Given a set of test requirements TR for a coverage criterion C, a test set T satisfies C if and only if for every test requirement tr in TR, at least one test t in T exists such that t satisfies tr

In addition to coverage, coverage level is also defined by [13] as:

> Given a set of test requirements TR and a test set T, the coverage level is simply the ratio of the number of test requirements satisfied by T to the size of TR

.

There are different approaches to determining the test coverage level of a program. Below I discuss each of these.

2.3.3 Statement Coverage

Arguably the simplest approach to determining the quality of a test set is to analyse the number of lines of code that execute when the tests are run. If all lines of code are executed at least once when all tests have been run, then statement coverage is said to be 100% [?].

```
static void Main(string[] args)
{
    if (DateTime.Now.Year == 2014) Console.WriteLine("It's 2014!"); else Console.WriteLine("It's not 2014");
}
```

Figure 2.7: A valid program that is all on one line.

While simple to implement, line coverage suffers from an obvious downfall. In most programming languages it is perfectly valid to have as many operations on one line as the developer chooses. In an extreme case it would be possible for the developer to have the whole program on one line. A contrived example of this is shown in Figure 2.7. If a test was created that executed that program, the coverage should come back as 100%, regardless of whether the test tried to run the program with different date's set on the test machine or not.

An easy but non-ideal solution to this is to require that developers only place one statement on each line. Alternatively, a more complex coverage analysis tool could be used that takes this into account.

Statement coverage is the term used when talking about the number of program statements that are executed by testing [4]. Myers and Sandler [4] argue that statement coverage is 'generally useless' as a metric of test quality because of the number of problems that it can potentially miss.

The example provided by Myers and Sandler [4] gives the code as shown in Figure 2.8. They argue that a single test can provide 100% statement coverage for the code by passing in the values `A=2, B=0, X=3`, even though the code could be logically incorrect. The example that they provide is that if the first decision should be an `or` instead of an `and`, then the single test will not notice, despite providing 100% statement coverage.

```
public void foo(int A, int B, int X)
{
    if (A > 1 && B == 0)
    {
        X = X / A;
    }
    if (A == 2 || X > 1)
    {
        X = X + 1;
    }
}
```

Figure 2.8: The sample code provided by Myers and Sandler [4].

2.3.4 Branch Coverage

Control Flow Graph

Before branch coverage can be introduced, the concept of a program control flow graph must be explained. A control flow graph shows the potential paths through a piece of code. Figure 2.9 shows the control flow graph for the code listed in Figure 2.8. At the top of the control flow graph is the entry point to the graph. From there, the first conditional statement is represented as a vertex of the graph. From that vertex there are two outward arrows. One represents the case when the conditional statement evaluates to true, and the other to false [4]

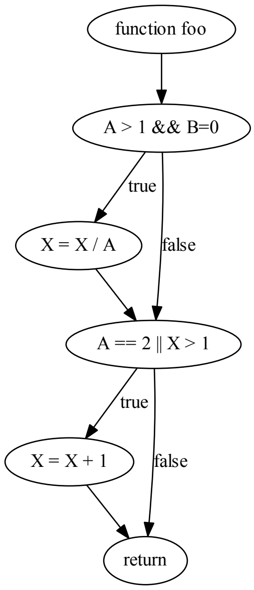


Figure 2.9: A simple control flow graph for the function foo

Branch Coverage

Branch coverage then is the number of branches that have been executed within the control flow graph. In Figure 2.9 there are two branching points - both of the conditional statements. For 100% branch coverage it is necessary for every edge to have been executed at least once by the test set. An alternative way to think of this is that at every decision point in the program, the outcome of each decision has been executed at least once. At an `if` statement, the case where the outcome is true has been executed as well as the case where the outcome is false. If branch coverage is 100%, then so should statement coverage [4].

However, Myers and Sandler [4] also argue that branch coverage can be a weak test quality metric. Going back to the code listed in Figure 2.8, branch coverage can be satisfied with the two following test cases: `A=2, B=0, X=1` and `A=3, B=1, X=1`. However, if the second conditional statement was supposed to check that $X < 1$ instead of $X > 1$, then this will not be picked up by any tests, despite the branch coverage being 100%. Del Frate et al. [5] have done a comparison of the effectiveness of decision coverage (i.e. branch coverage) and block coverage (i.e. statement coverage) after they inserted some random faults in the Unix utility Grep. Their results show in Figure 2.10 that block coverage analysis requires a higher percentage of coverage to find the same number of faults when compared to decision coverage.

2.3.5 Path Coverage

Path coverage is a stronger test quality metric. Branch coverage covers all possible decisions at a program branch, but path coverage considers every possible path through the program [4][13]. Using the example again given by Myers and Sandler [4] in Figure 2.8, there are four

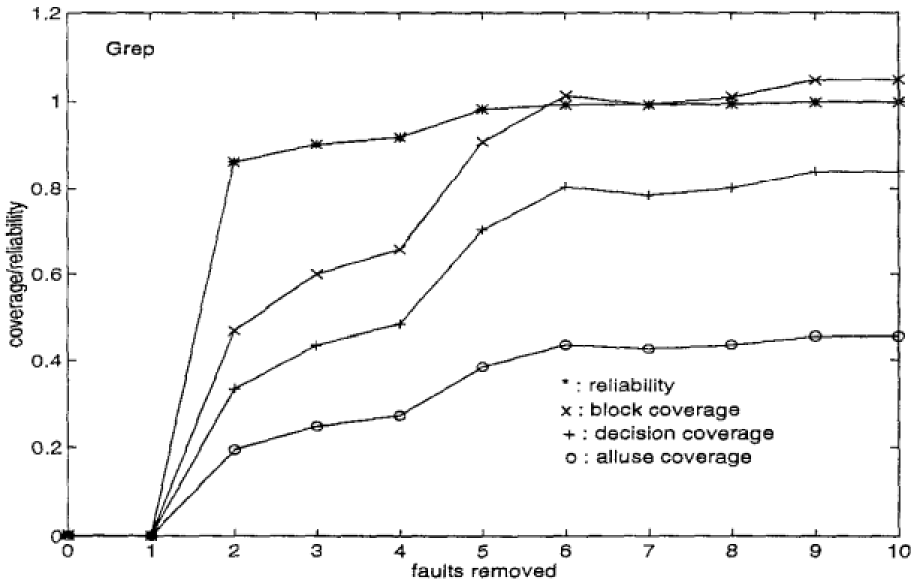
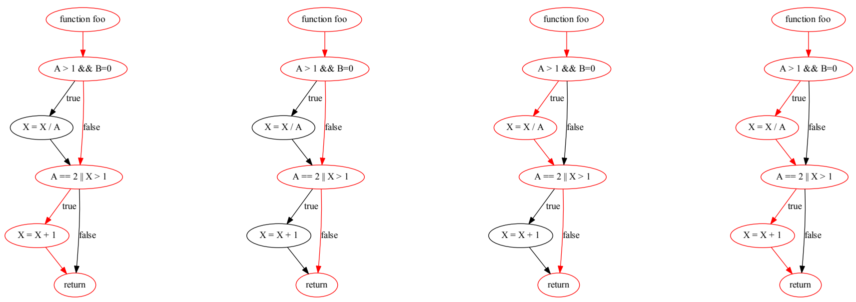


Figure 2.10: Del Frate et al. [5]'s results of testing various coverage approaches on Grep



(a) Path 1 (b) Path 2 (c) Path 3 (d) Path 4

Figure 2.11: The possible paths through the program

possible paths through the code. Each `if` statement can evaluate to true or false. Every path through the program therefore is when the `if` statements evaluate as follows: `false, true`, `false, false`, `true, false` and `true, true`. Figure 2.11 colours in red the possible paths through the program.

Because `if` statements only have two possible outcomes - true or false - the number of paths through a program that only contains n `if` decisions is 2^n. However, the complexity is increased with statements such as `switch` that can have any number of paths. This is known as the cyclomatic complexity of the program, and the calculation to calculate this complexity from a control flow graph was given by McCabe [14]:

$M = E - N + 2P$

Where M is the cyclomatic complexity, E is the number of edges in the graph, N is the number of nodes in the graph and P is the number of exit nodes. With this formula we can verify that the number of paths through the code in Figure 2.8 is 4. There are 7 edges, 5 nodes, and one exit point (the return statement).

$M = 7 - 5 + (2 * 1)$

Myers and Sandler [4] are even more critical of path coverage than the previous test quality metrics described. Their main points of criticism are:

1. The time that it would take to even generate all possible paths through a program grows exponentially with the number of branches in the program.
2. As some decisions are dependent on the outcome of previous decisions, once all possible paths have been generated it is then necessary to do further computation to calculate the actual number of possible paths through the code.
3. Even with each possible path through the code covered, there is no guarantee that the inputs used by the tests will find every problem with the code.

In his early work on the effectiveness of path analysis, Gannon [15] found that while the data used by tests will cause path coverage to be completed, it may not actually find bugs that are present. He therefore recommends that tests that have good path coverage are used in conjunction with boundary input data tests.

2.3.6 Mutation Testing

Mutation testing is another approach to determining the quality of a test set. Consider the following line of code:

```
y := 3x + 4 − z;
```

This is the fragment of code that we want to check that our test sets sufficiently cover. The code is called the *ground string*. From the ground string, mutant strings are created. These mutant strings are based on the ground string, but have been 'mutated' in some way such that they are not the same as the ground string, but still compile [13]. Some example mutants might be:

```
y := 3x − 4 − z;
y := 3x − 4 + z;
```

```
y := 10x + 6 − i;
```

The mutants all compile, but alter the outcome of executing the function. So the quality of a test set can be determined by running the set on each of the mutants and checking how many of the mutants are rejected. The perfect test set would reject all of the mutants [13]. The example above is greatly simplified, and in reality it is unlikely that all of the mutants would be caught by the test set.

In addition to being used to determine the quality of test sets, mutation testing (and path coverage and code coverage) can be used to help develop a high quality test set [13].

2.3.7 Model Transformation Testing

EuGENia is a model transformation written in ETL. It takes an Ecore metamodel as an input and generates gmf models as an output[2]. One approach to verifying and possibly improving the consistency between EuGENia and EuGENia Live is to ensure that both applications have a good quality test suite.

Benoit Baudry [16] describe the three stages to model transformation testing:

1. Generate test data: As with any type of test, there needs to be some input to the system. In this case it will be a set of models that are to be transformed. These models will conform to the metamodel that specifies input to the model transformation, and will either be manually created by the tester, or automatically generated in the form of graphs of metamodel instances [16].
2. Define test adequacy criteria: For any modeling language beyond the very basic there will be a very large number of possible inputs. This rules out running every possible model through the transformer in to test it as it would take too long. Instead a test adequacy criteria must be defined that allows the effective seletion of test models. According to Benoit Baudry [16], there is no well-defined criteria for model transformation testing.
3. Construct an oracle: The oracle gets the output of the system and determines if it is correct (based on the test input model).

Fleurey et al. [17] propose a general framework for assessing the quality of model transformations. Their paper begins by discussing the possibility of finding 'partitions' of the transformation's input meta-model. A partition is where the model could be one of a range of values, so for example if the metamodel specifies a boolean, there is a partition as the boolean could be either true or false. With these partitions, we could check that the input test set covers each possible combination of partitions. Fleurey et al. [17] then go on to state why this isn't necessarily the most useful approach: first the complexity rises quickly with each additional partition. Secondly some of combinations of partitions will not be relevant for testing, and the tester would have to find these and remove them. Finally, some relevant combinations could be missing. Generating every single combination of partition will 'not ensure the existence of more than one composite state' they state.

What Fleurey et al. [17] propose is the idea of model and object fragments that 'define specific combinations of ranges for properties that should be covered by test models'. Their paper suggests that one of the more important aspects of model transformation testing is ensuring that input models cover the correct criteria - that they thoroughly test the transformation, while avoiding having many very similar inputs that don't test anything that has not already been tested by another input. This is of course an important aspect to consider, but for the purposes of testing EuGENia may be slightly excessive. EuGENia is not a huge transformation, and so when I decide on my test inputs I will keep this framework in mind, although I will not follow it strictly. More important I believe is the oracle aspect of testing.

The oracle can be difficult to create for any complex model transformation. Mottu et al. [18] propose six ways that an oracle could be implemented for model transformation testing:

1. Compare the output to a reference model (i.e. the expected output model for the particular input). Unfortunately this requires that the tester has to create the expected models for each test. For a large test set this could be incredibly time consuming.

2. Perform an inverse transformation on the output. This would give the original input model, if the model transformation was correct. This requires that the tester implement a reverse transformation, and also requires that the transformation is an injective function (i.e. a function that preserves distinctness). According to Mottu et al. [18], this is unfortunately unlikely.

3. Compare the output with that from a reference model transformation. This reference model transformation can produce the reference model from the test model.

4. A generic contract is a list of constraints on the output of the model transformation based on the input. Once the model transformation has completed the output model is checked against the constraints defined in the generic contract.

5. The tester could provide a list of assertions in OCL (or EVL) that can be checked on the output model. Not every detail about the output model must be provided. Doing so would be a waste of time as providing the expected output model would be quicker.

6. Model snippets could be provided by the tester. Each snippet is associated with a cardinality and logical operator so that the expected number of occurrences of each snippet can be calculated. The oracle would check that the expected number of each snippet appears in the output.

3 Analysis

To begin this project I must first investigate how thoroughly tested EuGENia is. Methods for doing this were investigated in the previous section.

Bibliography

[1] M. Brambillia, *Model-Driven Software Engineering in Practice*, 2012.

[2] E. Project, 2013. [Online]. Available: http://www.eclipse.org/epsilon/doc/eugenia/

[3] L. Rose, "Bootstrap your new graphical dsl with eugenia live."

[4] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[5] F. Del Frate, P. Garg, A. Mathur, and A. Pasquini, "On the correlation between code coverage and software reliability," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, Oct 1995, pp. 124–132.

[6] K. Lano, *Model Driven Software Engineering with UML and Java*, 2009.

[7] M. R. Blackburn, 2008. [Online]. Available: http://www.knowledgebytes.net/downloads/Whats_MDE_and_How_Can_it_Impact_me.pdf

[8] Eclipse. [Online]. Available: http://www.eclipse.org/modeling/gmp/

[9] ——. [Online]. Available: http://wiki.eclipse.org/Graphical_Modeling_Framework_FAQ

[10] Y. U. E. Team, 2013. [Online]. Available: https://www.eclipse.org/epsilon/

[11] R. P. Louis Rose, Dimitrios Kolovos, "Eugenia live: A flexible graphical modelling tool."

[12] D. N. Arnold. [Online]. Available: http://www.ima.umn.edu/~arnold/disasters/ariane.html

[13] J. O. Paul Ammann, *Introduction to Software Testing*.

[14] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[15] C. Gannon, "Error detection using path testing and static analysis," *Computer*, vol. 12, no. 8, pp. 26–31, Aug 1979.

[16] F. F. R. F. Y. L. T. J.-M. M. Benoit Baudry, Sudipto Ghosh, "Barriers to systematic model transformation testing."

[17] F. Fleurey, B. Baudry, P.-A. Muller, and Y. Traon, "Qualifying input test data for model transformations," *Software & Systems Modeling*, vol. 8, no. 2, pp. 185–203, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10270-007-0074-8

[18] J.-M. Mottu, B. Baudry, and Y. Le Traon, "Model transformation testing: oracle issue," in *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, 2008, pp. 105–112.

A
EuGENia Sample Code

```
mespace(uri="filesystem", prefix="filesystem")
package filesystem;

@gmf.diagram
class Filesystem {
    val Drive[*] drives;
    val Sync[*] syncs;
}

class Drive extends Folder {

}

class Folder extends File {
    @gmf.compartment
    val File[*] contents;
}

class Shortcut extends File {
    @gmf.link(target.decoration="arrow", style="dash")
    ref File target;
}

@gmf.link(source="source", target="target", style="dot", width="2")
class Sync {
    ref File source;
    ref File target;
}

@gmf.node(label = "name")
class File {
    attr String name;
}
```