The University of York Department of Computer Science

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

Implementing Test Coverage in Epsilon

Thomas Wormald

28th April 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...

Contents

1 Introduction

Welcome to the best project write-up ever.

2 Literature Review

2.1 Introduction

In this chapter an overview is given of the existing literature as well as an overview of topics that are relevant to this 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 [1].

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 [2]. Brambillia [3] believes that Model Driven Engineering is now past the 'trough of disillusionment' and into the 'slope of enlightenment' (see figure 2.1) By this they mean that new technologies often make many promises that they ultimately cannot fulfil, which is represented by the peak of inflated expectations on the graph. From the disappointment that comes with broken promises, people become disillusioned with the technology and people tend to avoid using it. Researchers continue to work on improving and maturing the technology, which is represented by the slope of enlightenment.

2.2.2 Model

A model is a representation of something that abstracts away many details that are not necessary for its use [3]. For example, the Utah Teapot [**?**] is a model of a teapot that is 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

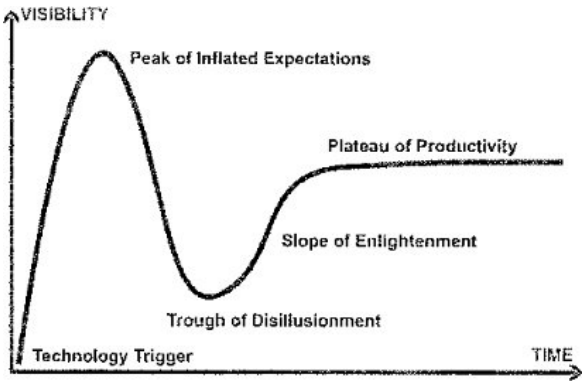


Figure 2.1: The technology hype cycle according to Brambillia [3]

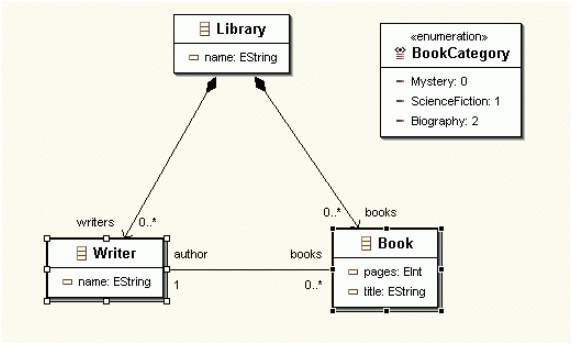


Figure 2.2: A sample model [4]

property of the teapot that will be included in the model is its 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 is not 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, methods and fields. In Figure 2.2, book is a class that has the properties pages and title. The diagram also shows that there is the association author between a writer object and a book object.

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 [5] 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 [3]. A metamodel could be represented visually or textually (as can a model), 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:

```
1 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.

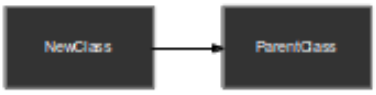


Figure 2.3: Concrete Syntax example for a modeling language

2.2.4 Epsilon

With the theory of model driven engineering covered, focus will now shift to covering the tools that can be used in model driven engineering, and that will be used in this project.

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 [6]. It is part of the Eclipse Modeling Project [**?**], and includes tools for each of its languages that integrate with Eclipse. From the Epsilon Website [6], 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

For each of these languages Epsilon provides development tools for use in Eclipse, as well as interpreters for each of the languages. As well, ANT workflows are provided which are a way of automating tasks in a sequential manner [7].

2.2.5 EUnit

EUnit is a unit testing framework that is included with Epsilon [6]. It allows systematic testing of model transformations by providing tools that can be used for checking that the output of model transformations meet certain requirements. [7] EUnit allows for different inputs and models, and could be considered as a JUnit for models.

Notably, EUnit does not include test coverage analysis, which is the focus of this project.

2.2.6 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:

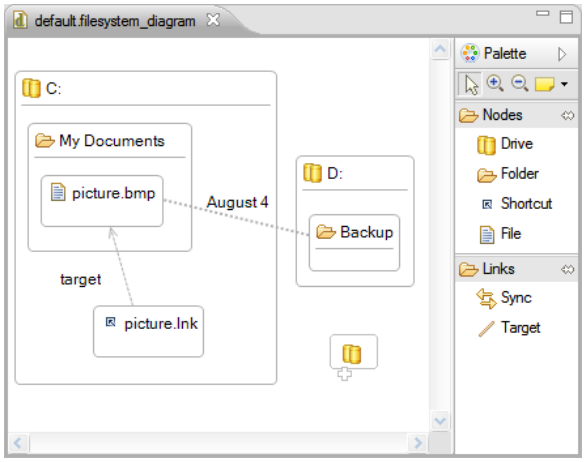


Figure 2.4: A sample gmf editor generated by EuGENia [11]

> 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.

To create these graphical editors, GMF requires 6 different files [10], including a metamodel of the models that the graphical editor will be used to create, graphics that will be used for representing models, and the tools that can be used to manipulate models.

2.2.7 EuGENia

EuGENia is one of the tools that is included with Epsilon. EuGENia takes an Ecore metamodel specification and generates a GMF editor [11]. The purpose of EuGENia, as taken from its website, is to 'shield you from the complexity of GMF and lower the entrance barrier for creating your first GMF editor.' [11] From the code in **??**, the model editor shown in figure 2.4 is generated by EuGENia.

The generated 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 associations between objects can be intuitively created.

2.2.8 Model Transformation Testing

To bring model-driven-engineering out of the 'trough of disillusionment' it has been necessary to perform testing on model transformations before they are deployed to users. Much research has been done in this area, and Benoit Baudry [12] 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 [12].

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 [12], 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. [13] 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. [13] 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. [13] 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. [14] 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. [14], 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.

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

Figure 2.5: Apple's SSL bug

2.3 Quality of Software Testing

2.3.1 Introduction

Users expect software to work without issues, and so when a bug is found the user will at best be annoyed. At worst, the bug may be in a critical piece of software that the failure of which could result in the loss of millions of dollars, or even human lives. The need for the testing of software before it is released then hardly requires justification. But how do we know when software has been tested thoroughly enough? This section will look at different approaches to determining the quality of a test suite.

2.3.2 The Need for Test Quality Metrics

To try and further justify the need for thorough software tests, I discuss two rather serious cases where software has failed where testing should have caught the bugs.

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 in 1996, 37 seconds after launch, $500 million of rocket and cargo exploded because of an integer overflow [15]. Despite having software tests, the tests were clearly insufficient as they did not find the integer overflow problem prior to the software being put into use. The software turned the rocket in the wrong direction, and to minimise the risk of fatalities the decision was quickly made to destruct the rocket. Thankfully there was no-one on board, and no-one on the ground was injured.

In 1991 though, a software error lead to the death of 28 people. The Patriot missile system was supposed to track incoming ballistic missiles and launch a rocket to destroy them before they hit their target [16]. In one instance the system lost track of an incoming missile and decided that it must have been a false reading, and so no rocket was launched. The incoming missile hit a US barrack, killing 28 American soldiers. The software for the Patriot missile lost time after being switched on for too long, meaning that it was looking in the wrong place for the incoming missile.

The final example is more recent. In February 2014 Apple issued a security update that fixed a bug in their SSL software. The bug was contained in the code in Figure 2.5. Notice that there is a duplicated `goto` statement, but because no braces were used on the above `if` statement, the duplicated statement is not conditional, and will be always executed. This is another case of insufficient testing. In this particular case, some analysis of which lines of code are executed by the test suite (i.e. line coverage analysis) would have shown that the final if statement in that block of code was never executed, because the above goto is always called.

2.3.3 Coverage

Test sets have a *coverage criterion* that measures how good a collection of sets is [17]. According to Paul Ammann [17], coverage is defined as:

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

Figure 2.6: A valid program that is all on one line.

> 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 [17] 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.

2.3.4 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% [**?**].

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 2.6. 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 [18]. Myers and Sandler [18] 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 [18] gives the code as shown in Figure 2.7. 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 2.7: The sample code provided by Myers and Sandler [18].

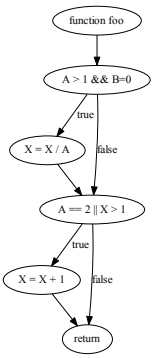


Figure 2.8: A simple control flow graph for the function foo

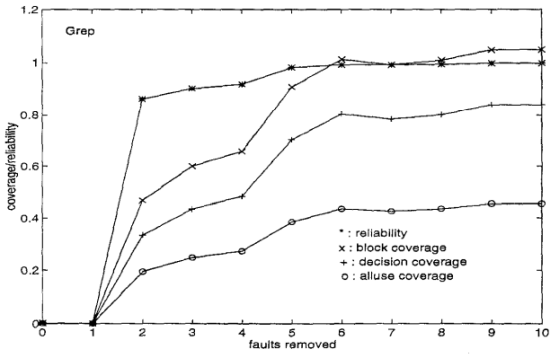


Figure 2.9: Del Frate et al. [19]'s results of testing various coverage approaches on Grep

2.3.5 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 2.8 shows the control flow graph for the code listed in Figure 2.7. 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 [18]

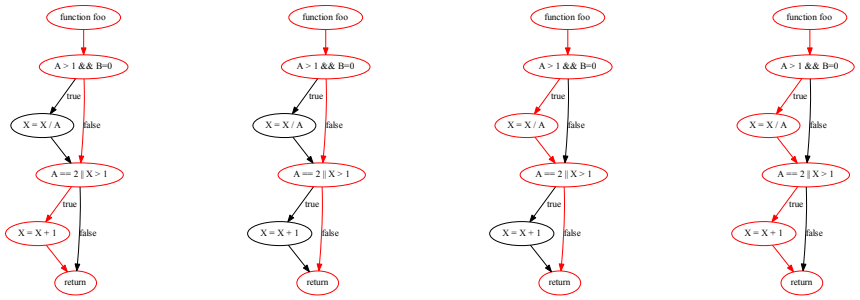
Branch Coverage

Branch coverage then is the number of branches that have been executed within the control flow graph. In Figure 2.8 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 [18].

However, Myers and Sandler [18] also argue that branch coverage can be a weak test quality metric. Going back to the code listed in Figure 2.7, 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. [19] 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 2.9 that block coverage analysis requires a higher percentage of coverage to find the same number of faults when compared to decision coverage.

2.3.6 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 [18][17]. Using the example again given by Myers and Sandler [18] in Figure 2.7, there are four possible paths through the code. Each `if` statement can evaluate to true or false. Every path through the program therefore



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

Figure 2.10: The possible paths through the program

is when the `if` statements evaluate as follows: `false, true`, `false, false`, `true, false` and `true, true`. Figure 2.10 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 [20]:

$$M = E - N + 2P$$

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 2.7 is 4. There are 7 edges, 5 nodes, and one exit point (the return statement).

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

Myers and Sandler [18] 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 her early work on the effectiveness of path analysis, Gannon [21] found that while the data used by tests will cause path coverage to be completed, it may not actually find bugs that are present. therefore recommends that tests that have good path coverage are used in conjunction with boundary input data tests.

2.3.7 Mutation Testing

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

```
1 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 [17]. Some example mutants might be:

```
1 y := 3x - 4 - z;
2 y := 3x - 4 + z;
3 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 [17]. The example above is greatly simplified, and in reality it is unlikely that all of the mutants would be caught by the test set.

According to Paul Ammann [17], 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. He claims that there are 11 mutation operations that should be used, regardless of the programming language in question:

ABS Absolute Value Insertion - Forces the tester to have at least one positive, one 0 and one negative value for the variable that has the `abs` function applied to it.

AOR Arithmetic Operator Replacement - Swapping between addition, subtraction, multiplication, division and modulus operators.

ROR Relational Operator Replacement - $<, \leq, ==, \geq, >, \neq$ operators are interchanged.

COR Conditional Operator Replacement - AND, OR, true and false are interchanged.

LOR Logical Operator Replacement - Bitwise operators AND (&), OR (|) and exclusive OR (^).

UOI Unary Operator Insertion - One of the unary operators $+, -, !,$ is inserted in a valid position.

UOD Unary Operator Deletion - One of the unary operators $+, -, !,$ is deleted from a position where a deletion will leave the statement valid.

SVR Scalar Variable Replacement - Each reference to a variable is replaced by another variable in scope of an appropriate type.

BSR Bomb Statement Replacement - Each statement is replaced by a bomb statement that causes the program to fail.

2.3.8 Coverage Analysis Software

There are numerous tools that can assist in calculating coverage metrics for a test suite. A selection of these are detailed in this section, and their general features will be taken into consideration during the design phase of this project.

JaCoCo

JaCoCo is a free Java code coverage library that works with Java 7 [22]. The library can be used to record statement and branch coverage of executed Java code, and will then produce a report in HTML, CSV or XML format. It provides a public API and thorough documentation with the intention that plugins make use of it for measuring coverage.

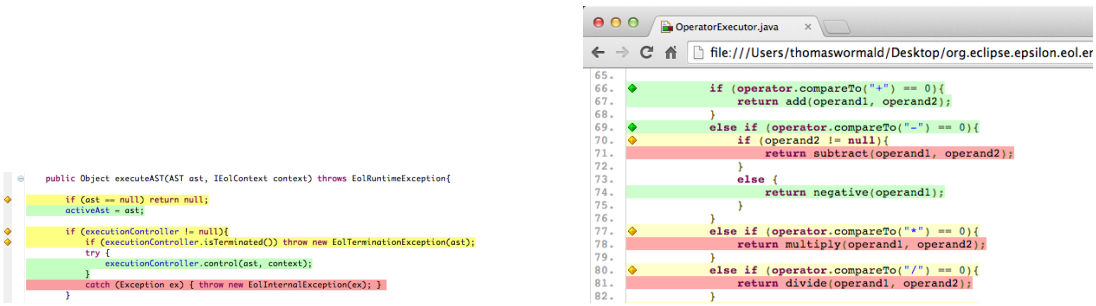


Figure 2.11: Left: EclEmma as an Eclipse plugin. Right: Output of EclEmma in HTML format

EclEmma

EclEmma is an Eclipse plugin that uses JaCoCo [23], and was developed alongside JaCoCo by the same developers. As shown in Figure 2.11 it highlights executed code in green, branches that have been partially executed in yellow, and non-executed code in red. To the left of the yellow statements is an icon, which when hovered over with the mouse tells the user how many branches there are in total, and how many of those branches were executed.

EclEmma can also export coverage reports to various formats. One option is a HTML document, as shown in Figure 2.11.

NCover

NCover is a suite of commercial code coverage tools written for use with .NET [24]. Like EclEmma, NCover offers an IDE plugin tool (for Visual Studio) that will highlight code statements to indicate the level of coverage of unit tests. NCover can output reports to HTML, and there is also a standalone desktop application that provides an 'interactive view of your coverage data so you can reduce errors, keep down your costs and deliver applications on-time' [24].

3 Analysis

3.1 Introduction

In this chapter an analysis of the problem is presented, along with a development plan.

3.2 The Problem

EuGENia has a test suite written for it. However, the test suite has never been analysed, and it is therefore not known how thoroughly tested EuGENia actually is.

In the literature review, the section titled 'Quality of Software Testing' discussed various approaches to determining how thorough a test set is. Based on that research, I will begin by attempting to perform statement coverage on the EuGENia transformation when the test set is executed.

Rather than implement statement analysis for just the EuGENia test suite, I will implement it for EOL in general, and attempt to detail a general approach for statement analysis in any language. As Epsilon has many languages, my documentation may prove useful for a future developer who wishes to implement statement analysis in another Epsilon language, or even a non-Epsilon language that does not yet have a statement analysis tool implemented.

In the book by Myers and Sandler [18], he states that statement analysis is the easiest form of coverage analysis, but is not particularly useful. Therefore I will next consider the possibility of branch analysis which is described as more difficult, but more useful that statement coverage. Once again this will be documented in the general sense in the hope that my approach can be applied to any procedural programming language.

Again, Myers and Sandler [18] is critical of branch analysis, and suggests that path coverage is a better metric of coverage. However, it is more difficult to implement than branch analysis. As with statement coverage and branch analysis, I shall attempt to document the procedure for performing the analysis in the most general sense, so that future developers can perform branch analysis on any procedural programming language.

If time permits then I will attempt to implement mutation testing. However, research suggests that this is an incredibly difficult problem that may take longer to solve than the time that I have available.

For each of the above forms of analysis I have not yet specified how exactly the coverage metric will be presented to the user. In the literature review I looked at some software packages that were plugins to Eclipse that highlighted statements and branches that were covered or not covered. If time permits then I will look into creating an Eclipse plugin. However, the main focus of my project is to actually produce various coverage metrics, and so most of my time will be focused on getting those values. If I do not have much time towards the end of the project, then I will simply output a text or HTML file of some sort that details the results of the analysis.

It is assumed that the three forms of analysis (statement, branch and path) will each be dependent on the previous. So branch analysis cannot be completed until statement analysis has been completed. While this is not strictly true, it is likely that in practice that this will be the case. As I will be new to the Epsilon source code, it makes sense to implement the easiest form of coverage first. Figure **??** is

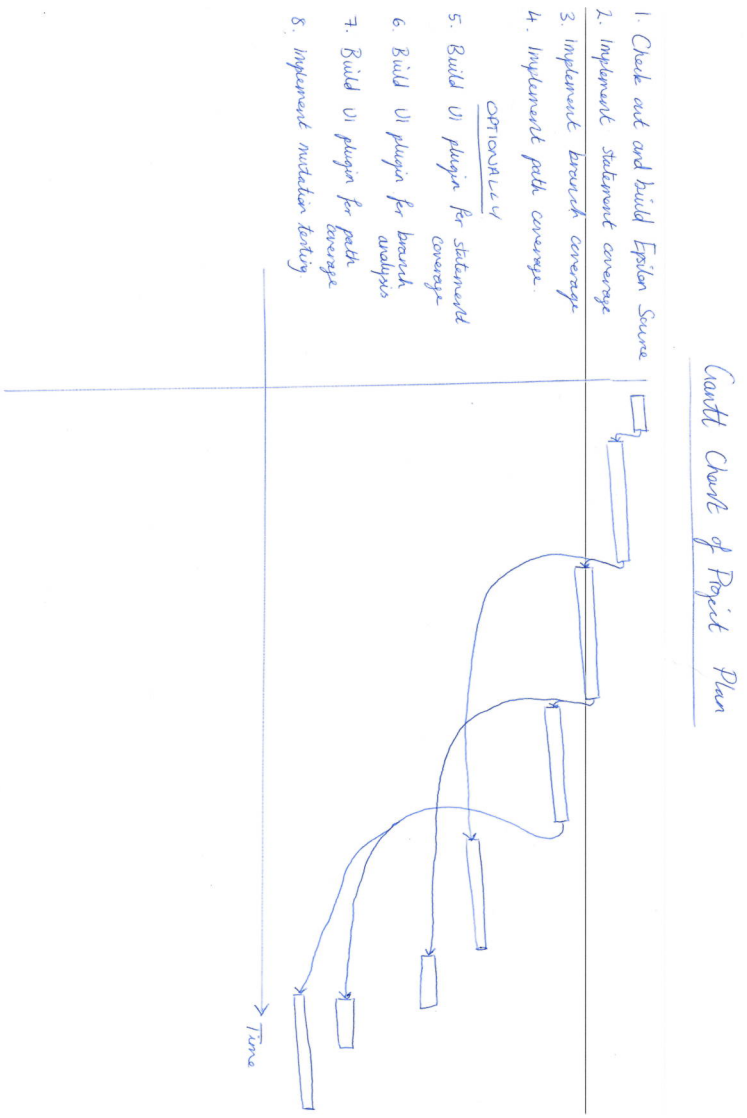


Figure 3.1: A gantt chart of the project's expected progress

a Gantt chart that details the expected progress of my project. No dates are included on the Gantt chart purposefully, as it is impossible to know at this stage how long each one will take.

4 Requirements Analysis

4.1 Introduction

In this chapter I will perform some analysis on the requirements of this project. I will begin by identifying the stakeholders in this project, and explaining their role and general aims. From this, I will then move on to detail some derived functional requirements, followed by some derived non-functional requirements.

4.2 Stakeholders

A stakeholder in the most general sense is defined by Insider [25] as:

> A person, group or organization that has interest or concern in an organization

More specifically in this project a stakeholder is someone or some group of people who have an interest in, or may benefit from, the contributions that my project makes to the field of model driven engineering or software testing.

I have identified the following stakeholders then from the above definition:

1. Enterprise Systems Group at The University of York
2. Developers who use EOL
3. Project Supervisor
4. Student

The Enterprise Systems Group is a group of academics and research students at The University of York. As outlined on their website [26], the group's primary objectives are to research and teach the fundamental objectives of enterprise systems. One of their main research areas is Model-Driven Development, which is how Epsilon came to be. Any contributions that my project make will be based around Epsilon, and should it be of a high enough standard then the outcome of my project may be incorporated into the Epsilon plugin.

The developers who use EOL is a potentially large group of people. As EuGENia is a transformation that is written in EOL, and EOL is not a language that is as widely used as say Java, the approach taken to testing will be thoroughly detailed so that another developer using EOL can use this document as a reference.

The users of EuGENia have an obvious interest in the outcome of my project. Should I find any bugs in EuGENia then I will either attempt to fix them, or at least alert the relevant people (such as a developer in the Enterprise Systems Group). This will improve the experience of EuGENia for users.

My project supervisor is a stakeholder in the project in a different way than the previously listed groups. He is currently and will continue to be involved in the project until the end. He sets deadlines, makes recommendations and suggests areas to research.

The student (myself) has a stake in the project. He is the primary researcher, developer and author of documentation. However his stake ends when the project is complete, as it is assumed that he will

have no long-term benefit from the outcome of the project. For this reason, no requirements should be derived from the view of the student.

4.3 Functional Requirements

4.3.1 A user will be able to perform statement coverage on any EOL file

Label	F-02
Description	Given an EOL file to execute, it will be possible to determine which statements within the EOL file were executed, and which were not.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	A user can find out which statements were executed, and which were not.

4.3.2 The output of statement analysis will tell the user what number of statements were executed

Label	F-03
Description	After running the statement analysis, the output to the user will include the number of statements that were executed and the total number of statements.
Source	Requirements Analysis, Literature Review
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	The number of statements that were executed is shown, as well as the total number of statements in the input EOL file.

4.3.3 The output of statement analysis will tell the user which statements were executed, and which were not

Label	F-04
Description	After running the statement analysis, the output to the user will include which particular statements were executed, and which were not.
Source	Requirements Analysis, Literature Review
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	The statements that were executed can be distinguished from those that were not executed.

4.3.4 A user will be able to perform branch coverage analysis on any EOL file

Label	F-05
Description	Given an EOL file to execute, it will be possible to determine which branches within the EOL file were executed, and which were not.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	A user can find out which branches were executed, and which were not.

4.3.5 The output of branch analysis will tell the user what number of branches were executed

Label	F-06

Description	After running the statement analysis, the output to the user will include the number of branches that were executed and the total number of branches.
Source	Requirements Analysis, Literature Review
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	The number of branches that were executed is shown, as well as the total number of branches in the input EOL file.

4.3.6 The output of branch analysis will tell the user which branches were executed, and which were not

Label	F-07
Description	After running the branch analysis, the output to the user will include which particular branches were executed, and which were not.
Source	Requirements Analysis, Literature Review
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	The branches that were executed can be distinguished from those that were not executed.

4.3.7 A user will be able to perform path coverage on any EOL file

Label	F-08
Description	Given an EOL file to execute, it will be possible to determine which paths within the EOL file were executed, and which were not.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	A user can find out which paths were executed, and which were not.

4.3.8 The output of path analysis will tell the user what number of paths were executed

Label	F-09
Description	After running the path analysis, the output to the user will include the number of statements that were executed and the total number of statements.
Source	Requirements Analysis, Literature Review
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	The number of paths that were executed is shown, as well as the total number of paths through the input EOL file.

4.3.9 The output of path analysis will tell the user which paths were executed, and which were not

Label	F-10
Description	After running the path analysis, the output to the user will include which particular statements were executed, and which were not.
Source	Requirements Analysis, Literature Review
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	The paths that were executed can be distinguished from those that were not executed.

4.4 Non-Functional Requirements

4.4.1 Statement coverage will not take an excessive amount of time to complete

Label	NF-01
Description	Statement analysis will not take an excessive amount of time to complete once the code has finished executing.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	Statement analysis completes within 5 seconds of the EuGENia transformation completing.

4.4.2 Statement coverage will not slow down the execution of an EOL file excessively

Label	NF-02
Description	Statement analysis may slow down the execution of EOL files, but not by an excessive amount.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	Execution should not take more than twice as long as it does when statement coverage is being monitored.

4.4.3 Branch coverage will not take an excessive amount of time to complete

Label	NF-03
Description	Branch analysis will not take an excessive amount of time to complete once the code has finished executing.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	Branch analysis completes within 5 seconds of the EuGENia transformation completing.

4.4.4 Branch coverage will not slow down the execution of an EOL file excessively

Label	NF-04
Description	Branch analysis may slow down the execution of EOL files, but not by an excessive amount.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	Execution should not take more than twice as long as it does when branch coverage is being monitored.

4.4.5 Path coverage will not take an excessive amount of time to complete

Label	NF-01
Description	Path analysis will not take an excessive amount of time to complete once the code has finished executing.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	Path analysis completes within 5 seconds of the EuGENia transformation completing.

4.4.6 Path coverage will not slow down the execution of an EOL file excessively

Label	NF-02
Description	Path analysis may slow down the execution of EOL files, but not by an excessive amount.
Source	Requirements Analysis
Stakeholders	Enterprise Systems at The University of York, Developers who use EOL
Satisfiable Conditions	Execution should not take more than twice as long as it does when path coverage is being monitored.

This is of course not a final list of requirements. Each is subject to change throughout the project should it be necessary. However, a reasonable attempt will be made to keep to these requirements, and any changes will be justified fully.

5 Statement Coverage

5.1 Introduction

This chapter details my effort to implement statement coverage for EOL programs. I begin by analysing the Epsilon source code that I will be working with. I then move on to detailing the design and implementation of the solution. Then I move onto testing the solution, and finish off with a conclusion on the successes and failures of the solution.

5.2 Analysis

The Epsilon source is broken into many well-organised packages. The packages `org.eclipse.epsilon.eol.*` contain all of the code that is specific to the EOL language, and so these will be the primary focus of this analysis.

The package `org.eclipse.epsilon.eol.execute` unsurprisingly contains the code to execute an EOL program. To perform statement coverage, it is necessary to determine which statements have been executed.

Some analysis of the execute package and its sub-packages has uncovered the interface `IExecutionListener`, as shown in Figure 5.2. An instance of a class that implements this interface can be added to a list of execution listeners. When a statement is about to execute, each object in the list has its `aboutToExecute` method called, and similarly after each statement has executed, each object in the list has its `finishedExecuting` method called.

An Abstract Syntax Tree (AST) object is provided to the execution listener. A syntax tree is the result of parsing the program text, and gives a data structure that is convenient for further processing [27]. Vertices of the AST contain a statement, and often also contain other relevant information. The EOL parser in Epsilon includes information in an AST vertex such as the type of vertex, and the line and line position of the statement that the vertex represents.

The first parameter of both methods in `IExecutionListener` is an abstract syntax tree object. The Abstract Syntax Tree class in Epsilon is designed in such a way that each vertex is an object of type AST, and each vertex has a list of children vertices, as well as a pointer back to the parent vertex. The

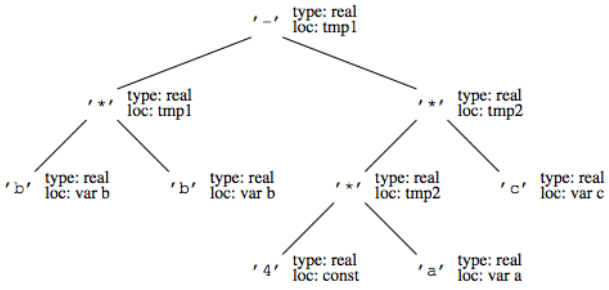


Figure 5.1: The expression b*b - 4*a*c as an AST. Taken from [27]

```
1 public interface IExecutionListener {
2
3   public void aboutToExecute(AST ast, IEolContext context);
4
5   public void finishedExecuting(AST ast, Object result, IEolContext context);
6 }
```

Figure 5.2: The public interface IExecutionListener

```
1 "Hello, World".println();
```

Figure 5.3: A simple EOL program

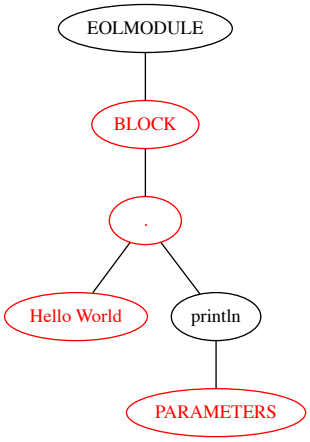


Figure 5.4: The AST of the program in Figure 5.3

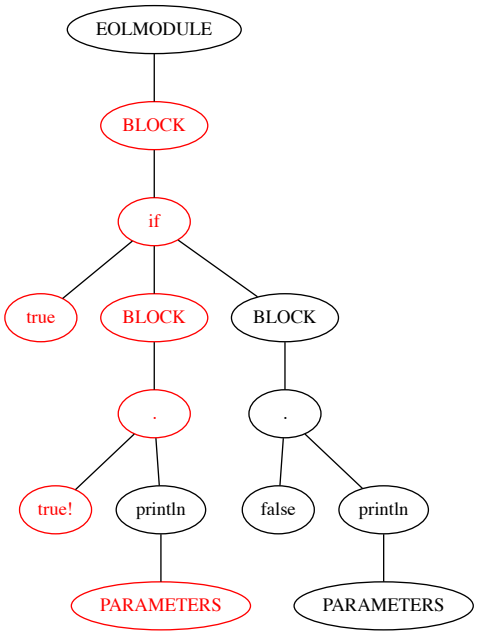
parent vertex will have a null pointer in place of a pointer to a parent vertex, and leaf of the tree will have an empty list of children.

The AST object that is passed as a parameter into both functions is a pointer to the vertex in the program's abstract syntax tree that is about to be executed or has just been executed, depending on the method being called.

One approach for determining how many statements were executed would be to keep a list of visited AST vertices, and then count the total number of vertices in the AST. There is however a problem with this approach: Consider the very simple code in Figure 5.3, and the AST that is generated for the simple program as shown in Figure 5.4. With that simple program, there is only 1 line that is going to be executed because there are no conditional statements that cause the program flow to change. However, the AST is comprised of 6 vertices. Testing shows that the execution listener's methods are only called on the red vertices. So while we know that the whole program has been executed, this naive approach will report only 4 out of 6 vertices have been executed.

Another approach that could be considered is to record which lines of the input file have been executed. This can be done because the AST class has a method called `getLine()` which as you would expect returns the line on which that statement comes from. So all of the red highlighted edges in Figure 5.4 return line 1 when `getLine()` is called. So initial analysis would suggest that 1 out of 1 lines has been covered in the simple program in Figure 5.3, which is accurate. The problem with this was discussed in the literature review, and that is that it relies on two statements not being placed on the same line. The code in Figure 5.5 and its accompanying AST in Figure 5.6 demonstrate this problem. Line coverage correctly is 100%, because 1 out of 1 lines have been executed. However, not all of that 1 line has been executed, and so this is not an accurate reflection of the coverage. With the AST, 7 out of 14 vertices have been executed, which is a lot more accurate than the line coverage.

Thankfully another option is available. The AST class has a method called `isImaginary()` which




```
1 if (true) { "true!".println(); } else {
      "false".println(); }
```

Figure 5.5: An if/else EOL program

Figure 5.6: The AST of the program in Figure 5.5

returns true when the current vertex is 'imaginary'. While no documentation is available, it appears that vertices that can be directly mapped back to some text from the input file are not imaginary, and those that cannot be mapped back to some text are not. Figure 5.7 shows the AST again, but this time with imaginary vertices left white, while non-imaginary vertices are filled in yellow. Quickly it becomes apparent that the definition of the method `isImaginary()` is not perfect as the vertex labelled EOLMODULE is coloured in yellow, despite not being directly mappable to a single statement in the input code.

5.3 Design

When the EOL executor calls the execution listener's pre or post execute methods, there needs to be a way to store a reference to the AST that is passed into either of the methods.

An obvious way of doing this would be to have a list of references to the AST. When one of the pre or post-execute methods is fired, a check would be performed on the list to see if a reference to that particular AST vertex was already stored, and if not, it would be added to the list.

Once the program has completed execution, in order to satisfy requirements F-03 and F-04, an analysis must then be performed to determine which vertices have been executed, and which have not. To do this, each vertex in the AST must be checked against the list of visited vertices. The AST can either be traversed by a depth or breadth-first algorithm. I have chosen to use a depth-first algorithm purely because it is easier to implement, and uses less storage space than the breadth-first (as no queue has to be stored).

At each recursion in the depth-first traversal, the list of visited vertices must be checked against the current vertex. This is therefore a $O(n^2)$ algorithm, which is not ideal as any sizeable programme can quickly slow down, which goes against requirement NF-01.

A way around this problem then is to use a hashmap to store the visited vertices. When either the pre or post-execute methods of the execution listener are fired, the hashmap would use the AST instance

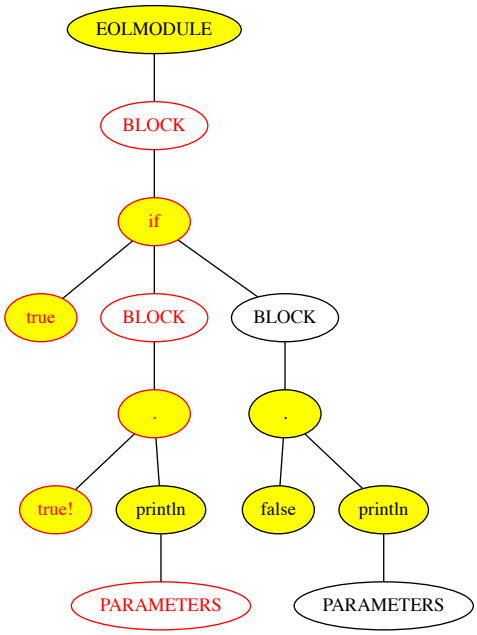


Figure 5.7: The AST with yellow vertices being 'real', and white vertices being 'imaginary'

as the key, and the value would be a boolean. To insert to the hashmap, the values `<ast, true>` would be passed in (where `ast` is the AST instance). Later during analysis, a simple call of the Java HashMap method `containsKey` would suffice, as it would return true if the AST instance has been inserted into the hashmap, or false otherwise. The use of a boolean as the value is arbitrary and in fact any value would suffice.

This approach better fits the requirement NF-02, although NF-01 could potentially still not be met. If the default hashing function is poor and causes many collisions then the performance could degrade to a similar level of the previously described approach using a list of references. A solution could be to implement a better hashing function, but it seems like a lot of effort for what may not even be a problem.

Both of the above solutions work around the Epsilon source code by storing information in new objects. However this is not actually a requirement - it is perfectly acceptable to modify the Epsilon code as long as it is justifiable. The most simple solution then is to modify the AST class so that it contains a boolean that stores whether or not that particular vertex has been executed. When initialised of course this boolean will be set to false, but when the post-execute function in the execution listener is called and it is passed an AST as a parameter, we can just change the value of the executed boolean within the AST. This has the advantage of meaning that recording that a vertex has been visited is guaranteed to be a O(1) operation (unlike the hashmap or list based solutions). Better still, when the analysis is performed after execution has completed, the lookup time to determine whether a vertex has been executed is also guaranteed to be O(1). Again this is unlike the hashmap or linked list based solutions, which were worst case O(n) lookup time.

With the core algorithms decided on, the structure of the actual code has yet to be decided upon, as well as the output from the code. There of course needs to be a class that implements the `IExecutionListener` interface. As the AST is directly being modified, then there is no need to transfer any data out of the execution listener (the AST will be available through another module). There therefore needs to be a class that takes the AST as input, and produces output of some sort. So to summarise, there will be two classes implemented. One called `StatementCoverageListener` that implements the execution listener interface, and another called `StatementCoverageAnalyser` that analyses and outputs some information.

The format of the output must satisfy requirements F-03 and F-04, that is that it will tell the user how many statements were executed, and it will also tell the user which statements were executed, and which were not. For the latter, research detailed in the literature review of other coverage tools shows that a popular way to do this is by highlighting the statements that were executed, and either leaving statements unhighlighted that have not been executed, or highlighting them in a different colour. Most of these tools worked as plugins to the Eclipse user interface. However, this is a lot of work and time is limited, so initially I will find a quicker approach. If time permits, then I will create the plugin after all other work is finished.

An easy way to output formatted text is to use HTML. HTML is easy to generate programmatically, and rendering the text is outsourced to a web browser. It would be possible to build my own text viewer, but it seems overly complicated when there are no disadvantages to generating a HTML page.

I will create a class that takes as input the executed AST, the file that has been executed, and the file to write the HTML to. The class will have to map executed vertices to actual characters that are in the executed file. The AST provides a function that make this relatively simple - there is a `getRegion` method that allows access to the start and end positions in the file of the statement that that AST vertex maps to. From this, we need to go through the input file and mark the statements that have been executed, and those that have not. There are two possible ways to go about this.

The first approach is to go through the executed file character by character, and go through every node in the AST and find if that particular character maps to a vertex. If it does, and that vertex has been executed, then highlighting is enabled (by changing the text background colour in HTML), and the character is copied to the output file. If the character cannot be mapped to a vertex, or it can be mapped but the vertex was not executed, then the character is simply copied across to the output file. The disadvantage of this approach is that it will be very time consuming to traverse the whole AST for every single character in the executed file. This will not help to meet requirement NF-01.

A better approach then will be to store for every character in the input text file whether or not that character should be highlighted. Initially every character will be set to not highlighted, but then the AST will be traversed, and any nodes that are not imaginary and that have been executed, the character will be set to be highlighted. Once the traversal has completed, the executed file's contents will be copied character for character. If the character being copied has been marked as highlighted, then the HTML highlighting tag will surround the character. The disadvantage of this approach is that it will require more memory than the previous approach, as it is now necessary to store more information for each character. However, it will be significantly quicker than the other approach, as the AST will only need to be traversed once.

In order to satisfy F-02, there will be a main method in the output class that takes an EOL file and output file as input, and executes the EOL file and outputs HTML to the output file.

5.4 Implementation

The first step was to modify the AST class. The additions are shown in Figure 5.8. Two methods (a getter and setter) have been added to modify the new boolean variable.

Following that, a class that implements the execution listener interface was created, and the post-execute method was completed to set the AST node to being visited.

Finally, the HTML outputter class was implemented. The whole class is too large to include here, so certain interesting parts are documented. The whole class can be found in the source that is included with this project, in the folder ?

The core of the class is detailed in Figure 5.10. Going through line-by-line, it initially reads in every line of the input file into a list of Strings. Then it next fills in the 'covered array', which is an array

```
private boolean statementVisited = false;

public boolean getVisited() {
   return this.statementVisited;
}

public void setVisited() {
   this.statementVisited = true;
}
```

Figure 5.8: Additions to the AST class

```
public class StatementCoverageListener implements IExecutionListener {

  @Override
  public void aboutToExecute(AST ast, IEolContext context) {
  }

  @Override
  public void finishedExecuting(AST ast, Object result, IEolContext context) {
    ast.setVisited();
  }
}
```

Figure 5.9: The StatementCoverageClass code

of strings that is exactly the same in size as the list of strings that holds the input file. So for each character in the input file, there is a position in the covered array that stores whether or not the character should be highlighted. Initially all positions in the covered array are set to a 'N' character. The function `dfAST` then performs a depth-first traversal of the AST of the executed program. It counts the number of non-imaginary vertices, as well as counts the number of non-imaginary executed vertices. As well it also changes the appropriate characters in the covered array to a 'Y' when an executed statement is found.

At this stage the AST has been traversed, and it is known which characters are to be highlighted and which are not to be. Now the HTML file is written. This is started by writing the HTML header, then the page title. Then the coverage statistics are written so to satisfy requirement F-03. Next, the code from the input file is copied to the output file, and the code that has been executed is highlighted. Finally, the HTML page footers are written and the file is written to disk.

5.5 Testing

Following the implementation of the solution, it must now be thoroughly tested for bugs, and of course if any bugs are found, then the fixes will be documented here.

Label	ST-01
Description	A simple program that outputs 'Hello, World'.
Expected Output	100% coverage, all code highlighted.
Result	Fail

Unfortunately the first test has failed. The output is shown in Figure 5.11 shows this. For the test to pass, the coverage percentage should have been 100%, but it is showing 66%. All of the code is highlighted, which is correct. Analysis of the AST for the sample program is shown in Figure 5.12. As before, yellow vertices are non-imaginary vertices that can be mapped to some text of the executed file, and vertices that are outlined in red are the ones that have been marked as executed. There is code in the analyser to ignore the `EOLMODULE` vertex, which is why it is only counting 3

```
public void analyseCoverage() {
   this.readInLines();
   this.fillCoveredArray();
   this.dfAST(ast);
   BufferedWriter writer = null;

   try {
      writer = new BufferedWriter(new FileWriter(targetFile));
      this.outputHTMLHeader(writer);
      this.outputTitle(writer);
      this.outputCoverageStats(writer);
      this.ouputEOLFile(writer);
      this.ouputHTMLFooter(writer);
      writer.flush();
      writer.close();
   }
   catch (IOException e) {
      // Not much to do here, just output stack trace
      e.printStackTrace();
   }
}
```

Figure 5.10: The core of the Statement Coverage HTML output class

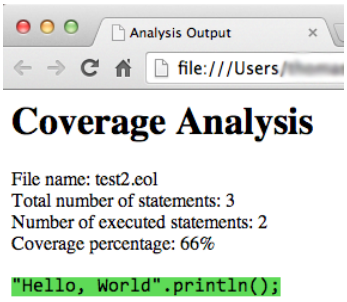


Figure 5.11: The HTML output from test ST-01, shown in Google Chrome

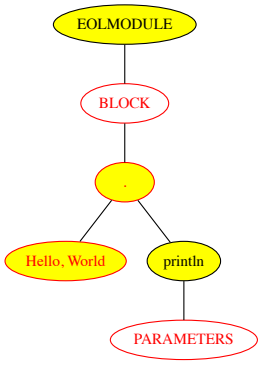


Figure 5.12: The AST for the program run in test ST-01

vertices. However, rather than the `println` vertex being marked as executed, its child `PARAMETERS` vertex has been marked as executed. This can also be seen to be the case in Figure 5.6, but was not spotted at the time.

At this stage there are a few solutions available. The first is that some code can be added to check with a `println` vertex whether or not its child `PARAMETERS` vertex has been executed. While a quick solution, further investigation shows that other operations also suffer from the same issue. Another potential solution is to modify the code of Epsilon so that the execution listener's post-execute method is fired on the `println` vertex rather than the `PARAMETERS` vertex. Unlike the simple modification that was made to the AST class earlier, this is likely to be a big job, and is probably not ideal considering that easier solutions are available. I feel then that the best solution is to modify the execution listener to detect when it has been sent a `PARAMETERS` vertex to actually mark the parent vertex as executed.

The code that was previously listed in Figure 5.9 has now been modified with the code shown in Figure 5.13.

Figure 5.13: The updated post-execution method

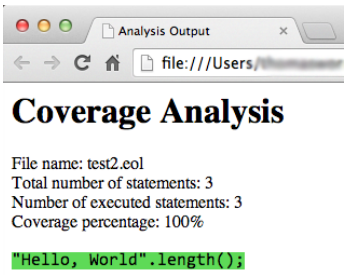


Figure 5.14: The output after re-running test ST-01

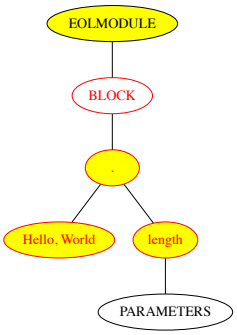


Figure 5.15: The AST for the program run in test ST-01

The results from re-running test ST-01 are shown in Figure 5.14, and the AST is shown in Figure 5.15. The test now passes.

Label	ST-02
Description	A single if-else statement, where the IF evaluates to true and so the contents of else never execute.
Expected Output	Only statements within the IF block are executed, and only those statements are highlighted.
Result	Pass

Label	ST-03
Description	A single if-else statement, where the IF evaluates to false and so the contents of it never execute, but the contents of the else block do.
Expected Output	Only statements within the ELSE block are executed, as well as the if evaluation.
Result	Pass

Label	ST-04
Description	A for-loop that contains a few statements, including one conditional statement that should execute on at least one iteration of the for-loop
Expected Output	All statements within the for-loop should be highlighted.
Result	Pass

Label	ST-05
Description	Both a context-free and context operation are defined and called.
Expected Output	All statements within each operation should be executed.
Result	Pass

5.6 Conclusions

While the tests are highlighting the code correctly, it is notable that the covered percentage value is not actually of much use to a developer. Some statements are never actually executed, and so coverage is rarely 100%. For ST-01 this was the case and was fixed, but further testing has shown that the problem is quite widespread. The output for ST-05 is shown in Figure 5.16, and the corresponding AST in 5.17. The AST is significantly larger than in previous examples, but upon close inspection there are vertices that are classed as non-imaginary, that are not executed, despite the program being

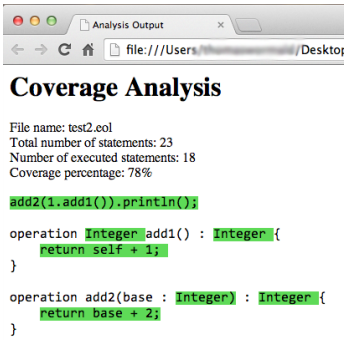


Figure 5.16: The output after from test ST-05

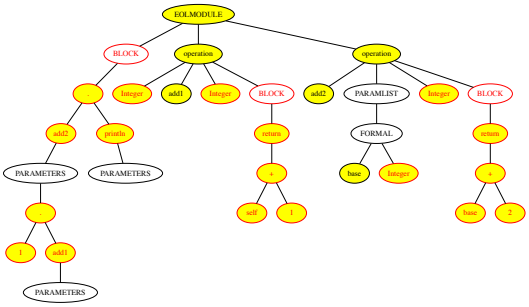


Figure 5.17: The AST for the program run in test ST-05

fully executed. Having considered this problem, I have concluded that the output is technically accurate, and thus meets requirement F-03.

Furthermore, F-04 is satisfied as it is easy to distinguish which code has been executed and what has not. With test ST-05 the user will want to see that the contents of both operations have been executed, as well as the first line that calls the operations. The fact that the operation headers are not fully covered is unlikely to be of use to the developer, and so I do not believe that this is an issue.

Requirement F-02 has been satisfied, as the main method in the HTML output class takes in an EOL file and executes it.

Anecdotal evidence suggests that NF-01 and NF-02 have also been satisfied. Further analysis of this may be required at a later date however.

6 Branch Coverage

6.1 Introduction

This chapter details my effort to implement branch coverage. The structure of the chapter is largely the same as the previous chapter. However, there is a lot more detail provided on the algorithm as it is thought to be the only detailed description on the process of converting an Abstract Syntax Tree to a Control Flow Graph.

6.2 Analysis

As detailed in the literature review, branch coverage is how many conditional statements have had all possible paths executed. So for an `if` statement, if it only ever evaluates to `true` then the branch coverage at that vertex is 50%. This becomes a problem when you have code such as in Figure **??**.If the `if` statement always evaluates to true during testing, statement coverage will show as being over 99%. However, if it ever evaluates to false then `someObject` won't be initialised, and an exception will be thrown later on if somewhere else `someObject` is referenced.

Branch coverage can counter this by looking at how many of the possible paths after all conditional statements have been executed. So in the code in Figure **??**, only 50% of possible paths from that `if` statement have been executed, and so the branch coverage is 50%.

By looking at the AST (Figure **??**) of the sample code, it would appear that by counting the number of blocks below the `if` vertex, we could determine how many branches in the code there are, and after execution we could see how many of those branches have been executed.

Unfortunately this approach is not perfect. The blocks only appear when curly braces are used. If just a single statement is placed under the `if` statement, then the block is skipped and just a vertex for the single statement appears. So this means that the code is now more complex than it was previously thought to be. Furthermore, `if` statements can have children that never need to be executed (because they just contain information about the conditional), and so my algorithm would need to include details of this. If these were the only drawbacks then I would still choose this approach. However, this kind of caveat occurs for many conditional statements, and so the code that would be produced would be rather unwieldy and difficult to maintain.

The approach therefore that I have chosen to take is actually quite difficult to justify. I suggest that the AST be converted to a Control Flow Graph (CFG), at which point the branches from each vertex will be clear. A record will be made on which edges between vertices have been executed, and the total number of edges will be counted. The edges that have not been recorded will map to the branches that were not taken. The reason that this is difficult to justify is that an extensive search has not come up with any explicit instructions on how to go about generating a control flow graph from an abstract syntax tree.Grune [27] very briefly discuss the process of 'threading' an AST to produce a CFG, but only state that there must be a subroutine that deals with each statement.

Some forward thinking means that the conversion from AST to CFG will be necessary when performing the path coverage because of the formula detailed in the literature review's path coverage section to calculate cyclomatic complexity, and so this effort will solve two problems, and will have a quicker overall development time.

```
1  Object someObject = null;
2
3  if (someBoolean) {
4  someObject = new Object();
5
6  //..............///
7  // 500 statements///
8  //..............///
9  }
10 else {
11 println("someBoolean was false");
12 }
```

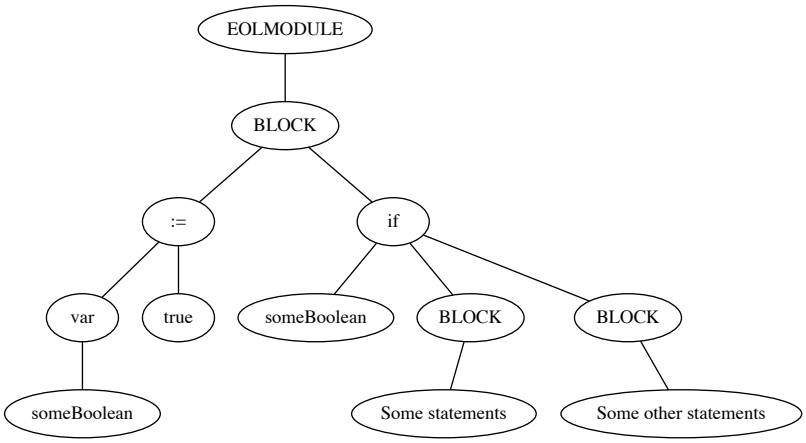


Figure 6.1: On the left is some sample pseudocode, and on the right is the AST for the sample pseudocode

Before beginning development, a list was made of the statements that need to be included. This was done by going through the Epsilon book [7] which is a complete source of EOL syntax, but also as well by going through the EuGENia source to see which statements are actually used in real EOL code. The list was then loosely ordered in priority based on the number of uses within the EuGENia source. The list as as follows:

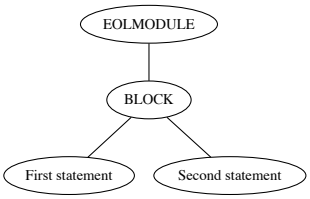
1. block
2. if
3. if .. else
4. for
5. while
6. switch
    a) case
    b) default
    c) continue
7. operation
8. return
9. break
10. breakAll
11. continue

For each of the identified statements, I will individually analyse how they can be converted from an AST to a CFG. For each statement a sample AST will be shown, as well as the desired CFG.

6.2.1 Start and End

As discussed in the literature review, a CFG start with a START vertex, and ends with an END vertex. In all the examples below of a CFG, these vertices are present. Each example represents a small subsection of an EOL program. When viewing the examples, the START vertex could be imagined to be where any part of a larger program joins up to the example CFG, and similarly the END vertex can be imagined to be the continue point of the program, where the statements following the example would join up to.

```
1 "First statement";
2 "Second statement";
```



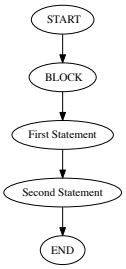


Figure 6.2: From left to right: The block's code, AST and desired CFG

6.2.2 The Block

Block is not actually a statement, but refers to a block of statements. Within a block can be any other set of statements, including other blocks. The contents of a block are often contained within { } braces, but not always (see the case statement).

The block is not conditional in any way. It can have a number of children, which are executed in order of first child (left-most) to last child (right-most). During the conversion of AST to CFG, when a block is encountered it should simply be a case of joining a block vertex from the last statement that was encountered, and joining it to the the block's first child.

The code in Figure 6.2 is a whole EOL program. The whole program is inside a block statement, as can be seen by the AST in the same figure. There are two statements in the block, and in the AST each statement is represented as a child. In the CFG, each of the children are represented sequentially, so the first child is represented after the block, and then the second child after the first child.

The block could be left out of the CFG, as arguably it does not provide any more information about control flow. For the time being this will be ignored, but at a later stage I will discuss and decide on this. For the rest of this analysis, the block statement may be used to represent any subset of vertices that does not add to the analysis. The input to the block vertex will represent input to the first vertex of the subset, and the output from the block will represent the output from any possible exit vertices from the subset.

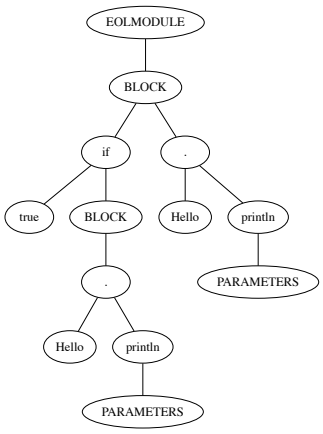
6.2.3 The if statement

The if statement is going to be treated as a separate statement to the if .. else statement, because when it comes to designing and implementing the algorithm, different code will be required for each.

The if statement evaluates its parameter, and if it evaluates to true then it executes the code in the parenthesis that follow it, or if there are no parenthesis, it executes the statement that directly follows it. While the outcome of execution is the same (when there is only one statement executed following an if statement that evaluates to true), the use of parenthesis following the if statement changes the structure of the AST in a way that must be considered during the conversion. Figure 6.3 shows the if statement that uses parenthesis, and figure 6.4 shows the statement that does not use parenthesis.

When the parenthesis are used, if the if statement evaluates to false then the statement that immediately follows the block must be the next statement to be executed. This will be the sibling of the if statement in the AST. If the if statement evaluates to true, then once it has finished executing the block it must move on to the next statement after the block, which again will be the next sibling of the if statement in the AST. This of course is assuming that a statement within either of the blocks does not divert program flow away from the next statement. A simple example would be to imagine

```
1 if (true) {
2   "Hello".println();
3 }
4
5 "Hello".println();
```



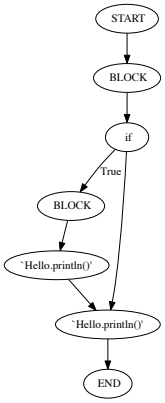
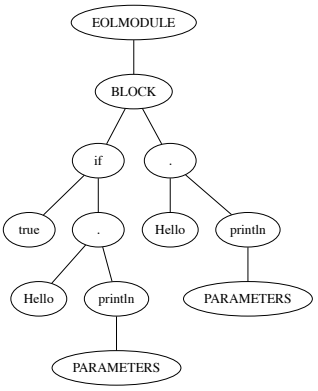


Figure 6.3: From left to right: The if statement with parenthesis' code, AST and desired CFG

```
1 if (true)
2   "Hello".println();
3
4 "Hello".println();
```



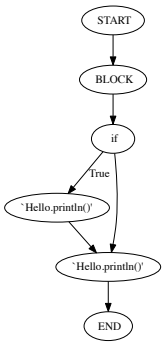


Figure 6.4: From left to right: The if statement without parenthesis' code, AST and desired CFG

that the contents of the block shown in Figure 6.3 are modified to include a `return` statement. This is not uncommon in a program, and so must be considered. In this case the the final statement will not be executed, because return will end the current program (or sub-routine).

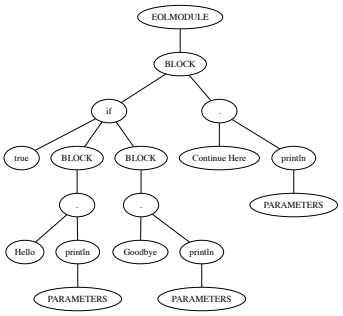
When no parenthesis are used, the same rules almost apply. Except that rather than looking for the next statement after the block, it will be the second statement following the if statement. This will still be represented in the AST as the next sibling of the if statement.

Another consideration must be made about program flow. Within the conditional part of the if statement, any number of subroutines can exist. It could be viewed as another block of code, and this is why it appears as the first child of the if statement in the AST. For the purpose of control flow, this will be ignored. Any code within the conditional will not modify the flow of the program, as it should only evaluate to either true or false.

6.2.4 The if .. else statement

The if .. else statement differs from the if statement in two ways. The first is that control flow will always go down one of two paths (see Figure 6.5), rather than potentially going down one or skipping around it. Secondly, the if statement in the AST now has three children rather than just

```
1 if (true) {
2   "Hello".println();
3 }
4 else {
5   "Goodbye".println();
6 }
7
8 "Continue Here".println();
```



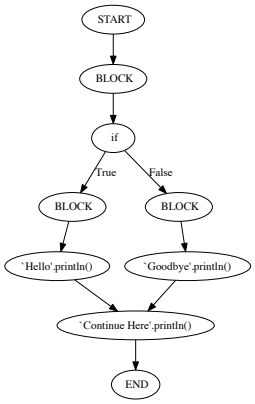
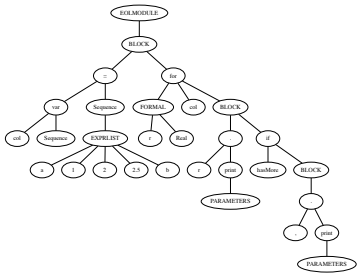


Figure 6.5: From left to right: The if .. else statement without parenthesis' code, AST and desired CFG

```
1 var col : Sequence =
      Sequence{"a", 1, 2,
      2.5, "b"};
2
3 for (r : Real in col) {
4   r.print();
5   if (hasMore){
6     ",".print();
7   }
8 }
```



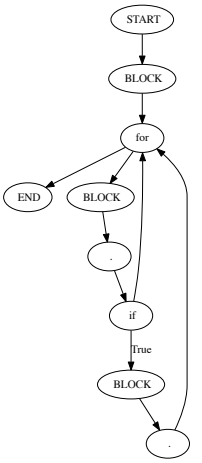


Figure 6.6: From left to right: The for loop code (taken from the Epsilon Book [7]), AST and desired CFG

two. The additional child is the block (or single statement if no parenthesis are used) for the else statement.

When implementing the conversion a check will need to be included to see how many children the if statement has. While the EOL language defines an else statement, it is never featured in the AST that is generated and therefore cannot be used to distinguish between an if statement and an if .. else statement.

Because control flow is forced down one of two paths, the final statement of both paths will now need to link to the next statement that follows the whole if .. else statement. As with the if statement, this makes the assumption that neither of the if or else blocks divert control flow. If they do, then the CFG will need to show this accordingly.

6.2.5 The for loop

The for loop in EOL works in the same way as it does in the majority of languages. It iterates over a collection of items, executing the block of code underneath it once for each element in the collection specified.

Figure 6.7: The two options for the for loop

To represent the for loop in a CFG, there will initially be a for vertex. Then from the for vertex there will be the contents of the block underneath it represented (which is the third child in the example AST in Figure 6.6).

A decision that must be made is whether the final vertex of the for loop always returns to the initial for vertex, or whether it has one edge returning to the for loop, and another edge continuing to the next statement after the for loop. For non-final iterations of the loop, the final statement within the for block will always go back to the for loop, but on the final iteration it could continue to the next part of the program. The two options are shown in Figure 6.7. The assumption that the block can be multiple statements has to be made, but you can see the two options that are available.

After some consideration (and discussion with my supervisor) I have decided to go for the option that is shown on the left in Figure 6.7. My argument is that returning to the for loop after the final iteration more accurately reflects what happens, because the program will return to the for loop to check if there are any more iterations to perform, and if not, continue on to the next statement after the for loop.

Once again statements that alter the control flow must be considered. As with the if statement, if a `return` can be called within the for loop, then this must be reflected in the CFG that is generated. There are also some additional statements that must be considered. These are: `break`, `breakAll` and `continue`. However, these will be discussed later.

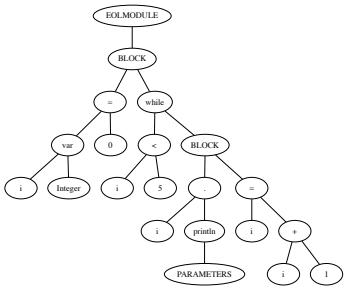
Also to be considered with the for loop is the possibility of multiple final statements within the for statement's block. In the example in Figure 6.6, there is an if statement that may or may not execute. If it does execute, then the final statement is the final statement within the if block. However, if it does not executed, the final statement is the if statement, which is why I have added an edge from the if vertex back to the for vertex. The actual implementation of this may prove to be difficult, but this will be investigated at a later time.

Up to this point, each statement has only been considered on its own. But the example in Figure 6.6 actually has an if statement within a for loop. It would be a very easy problem to solve if nested statements weren't allowed, but unfortunately it would make programs very difficult to write. Therefore the algorithm that I implement must be able to deal with any number of different types of nested statements, at any level of depth.

6.2.6 The while loop

The for loop iterates over a collection of objects, but the while loop iterates as long as its conditional statements evaluate to true. This is the case in most languages, and is the case in EOL. While there are some internal differences between the for and while loops in EOL (as detailed in the Epsilon Book [7]), in terms of control flow they are identical. The AST differs slightly because a for loop has 3 children, whereas a while loop only has 2. However, for control flow the only child of interest is the last one for each loop, as this is the block of statements that are executed.

```
1 var i : Integer = 0;
2
3 while (i < 5) {
4   i.println();
5   i = i+1;
6 }
```



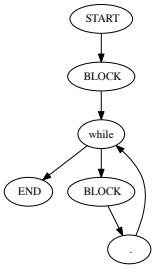


Figure 6.8: From left to right: The while loop code (taken from the Epsilon Book [7]), AST and desired CFG

Notice that the CFG example in Figure 6.8 always returns to the while vertex before continuing the program. This is the same as was decided in the for loop.

When it comes to designing the algorithm to perform the conversion, it will make sense to consider the for and while loops as the same thing, with the minor exception of the child that contains the block of statements being in a different position for each.

6.2.7 The switch statement

The switch statement is probably one of the most complex statements in EOL in terms of control flow. The EOL switch case is similar to the Pascal switch statement. The switch statement takes a value as an argument, and then the case statements within the switch statement's block are examined. If one is matched, then the block that follows that case statement is executed, and control is passed to the statement that follows the block of the switch statement.

This differs from the Java switch statement, where every single case statement is examined, and if it matches the switch statement's parameter, then it is executed. This allows for multiple case statement blocks to be executed potentially in a single case statement.

At this point, it sounds like EOL's switch statement is easier for generating a CFG, because coming out of a switch vertex is a link to each of the case vertices, and one that bypasses to the next vertex after the switch block (in case none of the case statements are executed). However, I now look at the continue statement.

```
1 var i : Integer = 0;
2
3 switch (i) {
4   case 0 : "Zero".println(); continue;
5   case 1 : "One".println();
6   case 2: "Two".println();
7 }
```

Figure 6.9: An example of a switch statement with fallthrough

If a case statement ends with continue in EOL, then the contents of every case block that follow the current case block will be executed. So in the code listing in Figure 6.9, because case 0 is executed, and case 0's block finishes with the continue statement, the actual output will be:

```
Zero
One
Two
```

So unlike Java's fallthrough that just checks every other case statement, it actually executes every following case statement, regardless of the case values. This complicates control flow quite a bit, because if a continue statement is contained in any of the case statements, then edges have to be added from the end that case statement and every subsequent case statement block to the start of the following case statement block.

As well as case, switch can also have the default statement within its block. The default statement is always after all of the case statement, and is executed when none of the case statements were executed (or when a continue statement was called in one of the case statements). In terms of program flow, this will remove the edge from the switch vertex to the statement following the switch block, because the default statement forces the execution of something within the switch block. As with all of the case statements, the final statement of the default block moves on the execute the next statement after the switch block.

The switch statement requires two examples to fully demonstrate possible flow. Figure 6.10 shows a switch statement that does not have a default statement. That means that it is possible for none of the case statements to be executed, and so there should be an edge from the switch vertex straight to the statement that follows the switch block (which in this case happens to be the end vertex because there are no more statements following the switch block). Then notice that the continue statement on the left-most case joins to the block of the next case statement, and that a vertex has been added from the end of the second case statement to the block of the third case statement.

Figure 6.11 shows a switch statement that does not contain a continue statement, and so the CFG on the right appears much tidier. Out of the switch vertex is a link to every case statement, and because there is a default statement within the switch block, there is also an edge to the default vertex. In this example there is no edge from the switch vertex to the end vertex, and this is again because the default statement is used.

6.2.8 Operations

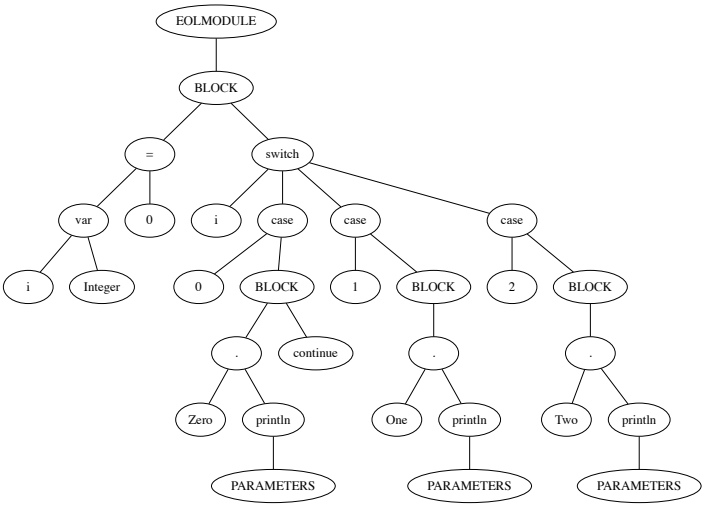
EOL allows users to define their own operations [7]. This is a standard feature of most languages, but EOL allows two different type of operations: context and context-less. Context-less operations are the closest to what is found in an object-oriented language like Java. An example adapted from The Epsilon Book DimitrosKovolos [7] is shown in Figure 6.12. The alternative is a context-type operation that extends classes with new functions. The example shown in Figure 6.13 shows that the operation `add1()` extends the Integer class, and can be called on any object of type Integer.

This is not too important in terms of control flow, however. Both types of objects can be treated in the same way, as their AST representations are very similar, and in either case the operation is called, which is all that is important for control flow. There are three options on how these operations could be included in a CFG:

1. Each time the operation is called, copy all of its CFG vertices in place of the operation call. This idea can be dismissed quickly for two reasons. The first is that any operations with more than a few vertices will make the final CFG much bigger than necessary. The second is that when considering branch and path coverage, the paths through each operation only need to be considered once. If this approach was taken then the path through each operation each time it was called would have to be considered.

2. All operations will be listed just once with start and end vertices, and when a call is made to an operation, an edge will be added from the vertex that makes the call to the start of the operation, and likewise an edge will be added from the vertex that ends the operation back to the vertex that made the operation call. The problem with this approach is that during branch and path analysis, it would be implied by the CFG that the control flow can get to an operation call, execute the operation, but then return to a different part of the program (that makes the

```
1 var i : Integer = 0;
2
3 switch (i) {
4   case 0 : "Zero".println(); continue;
5   case 1 : "One".println();
6   case 2: "Two".println();
7 }
```



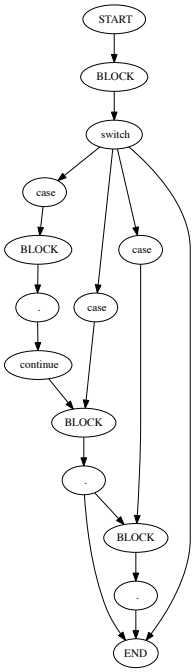
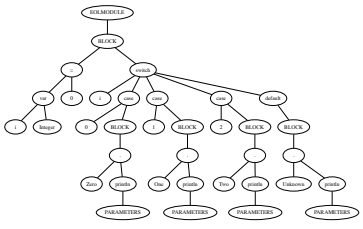


Figure 6.10: Top left: The example switch code that includes a continue statement. Bottom left: The AST of the example switch code. Right: The desired CFG for this example. Notice the continue statement on the left, and that there is an edge from the switch statement straight to the end statement.

```
1 var i : Integer = 0;
2
3 switch (i) {
4   case 0 : "Zero".println()
        ;
5   case 1 : "One".println();
6   case 2 : "Two".println();
7   default : "Unknown".
        println();
8 }
```



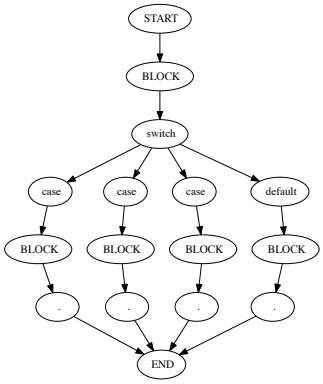


Figure 6.11: From left to right: The example switch statement that includes a default case, the AST for the example, and the desired CFG. Notice in the CFG that there is no edge from the switch vertex to the end vertex because there is a default vertex.

```
add1(1).println();

operation add1(base : Integer) : Integer {
  return base + 1;
}
```

Figure 6.12: Context-less operation

```
1.add1().println();

operation Integer add1() : Integer {
  return self + 1;
}
```

Figure 6.13: Context-type operation

same operation call)! This is of course not an option, and so special exceptions would need to be added to the branch and path analysis code, potentially making it difficult.

3. As above, all operations are listed just once, but operation call vertices do not have any edges to the operation that is being called. While maybe this isn't quite as accurate at representing program flow as the previous option, but it will simplify the process of branch and path analysis, which is the whole point of converting the program to a control flow graph in the first place.

Figure **??** gives a visual representation of the options.

6.2.9 break, breakAll, continue and return

These four operations have all been grouped together, because they all change program flow in a similar manner.

break in EOL is the same as break in most languages. When break is called within a loop (for or while), control breaks out of the loop and continues at the statement that follows the loop's block.

breakAll is like break, but rather than just breaking from the current loop, it will break from any number of nested loops, and continue execution after the outermost loop's block.

return ends execution of the current operation, or the main program if it is not called within an operation.

continue when used in the context of a loop has a different meaning to continue when used in a switch statement's block. In a loop, continue ends the current iteration within the loop and returns control to the loop conditional statement.

The Epsilon Book DimitrosKovolos [7] gives a nice example that uses break, breakAll and continue all in one small program. This is shown in Figure 6.15. Notice that in the CFG, the continue statement only has an edge back to the for loop that it is contained within, ending the current loop iteration. The break statement also breaks the control out of the inner for loop, and returns control to the outer for loop. Finally, the breakAll statement breaks out of both for loops, and execution is taken to the end of the program (because there are no statements after the outer for loop). No return statement is included in the example, but if it were at any point, then its only outbound edge would be to the end vertex, because it ends the current procedure or program.

These four statements greatly complicate the generation of a control flow graph because they can be included nearly anywhere, and change control flow significantly. When the break is found in the

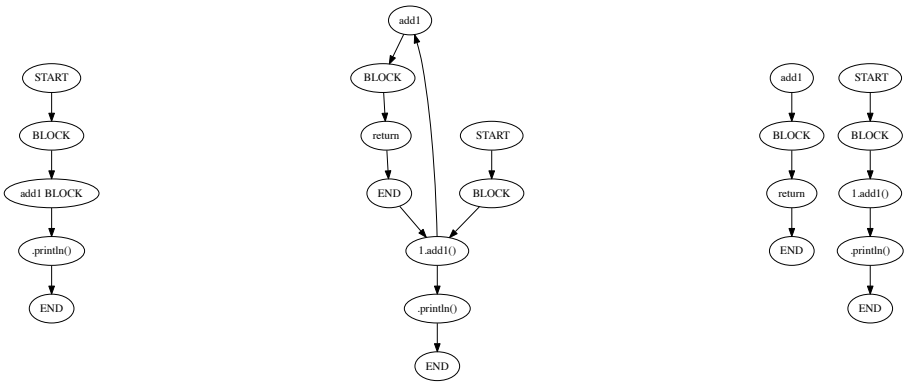
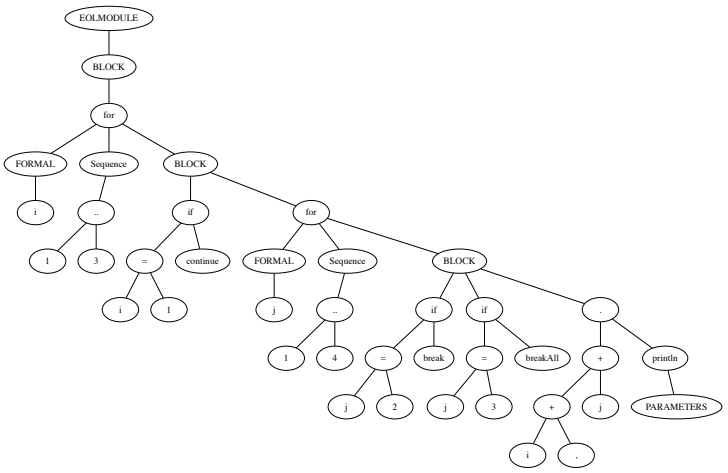


Figure 6.14: Choices of how to represent an operation in a CFG. Left: option 1, where the contents of the operation's CFG are added into the main CFG each time the operation is called, so the add1 block has been included in the main CFG. The middle CFG shows option 2, that when the operation is called, edges are added to the start and end operation vertices to the call location. Finally on the right is option 3, where operations are listed as separate CFG's.

```
1  for (i in Sequence{1..3}) {
2    if (i = 1)
3      continue;
4
5    for (j in Sequence{1..4}) {
6      if (j = 2)
7        break;
8
9      if (j = 3)
10       breakAll;
11
12     (i + "," + j).println();
13   }
14 }
```



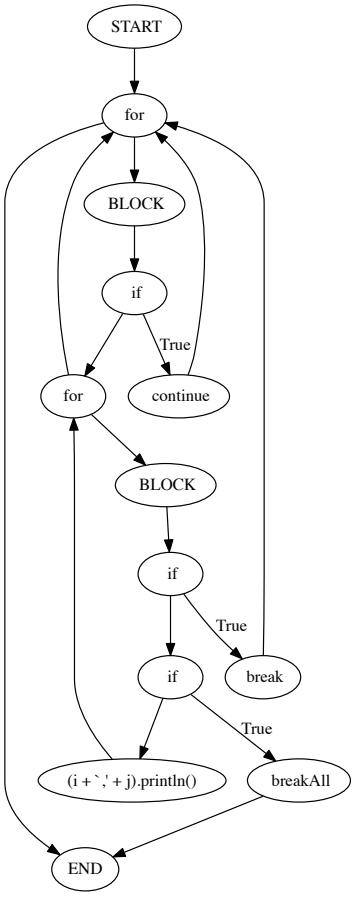


Figure 6.15: The code, AST and CFG of a program containing a continue statement, break statement and a breakAll statement.

example program, control flow will need to be diverted from the current loop to the statement that follows the containing for loop. Except that in this case, that's not quite right, because the for loop that is been broken out of is actually the last statement in another for loop, and as discussed in the section about for loops, at the end of a loop control should be returned to the loop header vertex, which in this case is the first for vertex.

It would be ridiculous to design an algorithm that caters for this specific case, and all other specific cases, because there are so many possible combinations and it would be near-impossible to maintain. Instead, a more clever and maintainable algorithm will be designed that can handle any combination of statements.

6.3 Design

In this section I begin by detailing the design for the AST to CFG conversion. I will then move on to discuss the design for actually performing branch analysis with the CFG.

6.3.1 Representing the CFG

The first step is to decide on how the control flow graph should be represented. At each vertex we need to store at least:

1. The type statement that the vertex represents
2. A label for the vertex
3. The list of children from this vertex
4. A unique ID for the vertex

These may not be all that a vertex needs to store, but at this stage it is all that is required. An AST is similar to a CFG, and for that each vertex is stored as an object with references to child AST objects. From experience, this works well for the AST, and so the design will be loosely copied for the CFG. Each vertex of the CFG will be an object of type CFG, and will contain a list of children.

6.3.2 Visualising the CFG

Drawing a CFG will be useful for debugging, and also as a form of output to the user. Drawing a graph that organises vertices neatly that don't overlap is a difficult problem. There are existing tools that take the specification of a graph, and calculate the layout accordingly. GraphViz is one such program [28] that takes a graph specification in the DOT language [29]. The DOT language is simple, which makes the generation of graphs programmatically easy.

GraphViz will be used for turning the CFG stored in memory into a visual representation. To convert the CFG to a DOT file, a depth first traversal of the CFG will take place. This traversal will need to keep a list of already visited vertices because the CFG can contain loops, and without the visited list the traversal could get stuck in an infinite loop.

6.3.3 The Basic Case

It makes sense to start with the most basic EOL program and convert that into a CFG. Epsilon usefully includes a tool called AST Explorer that gives a visual representation of the AST when the class EolParserWorkbench is executed. EolParserWorkbench has a string that points to an EOL file on disk, which I have modified to point to an EOL file that I can easily modify. For a simple Hello

```
digraph sampleGraph {
  A [label="Vertex A",fillcolor=blue,
      shape=box,style="striped",
      fontcolor="white"]
  B [label="Vertex B"]
  C [label="Vertex C"]
  A -> B [label="A to B", style=dashed,
      color="red"]
  B -> C [label="B to C"]
  C -> A [label="C to A"]
}
```

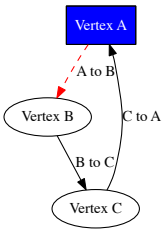


Figure 6.16: Left: The specification of a graph in the DOT language. Right: The graph generated from the code on the left.

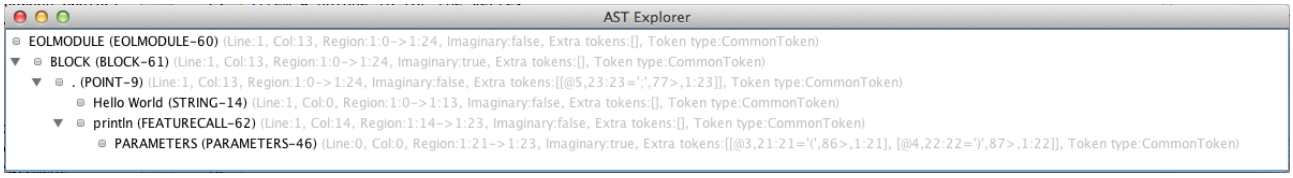


Figure 6.17: The AST explorer for a Hello World application

World application, the AST explorer shown in Figure 6.17 is shown. The AST explorer provides a lot of information, but it is not easy to quickly visualise how the tree looks, so the AST will also be visualised using graphs rendered by GraphViz. The DOT file for the AST graphs will be generated by performing a traversal of the AST and writing each recursive call as a vertex, with an edge between the last recursive call and the current one. It will differ from the CFG traversal because it won't be a directed graph, and no visited list needs to be kept, because the AST can not contain loops.

The CFG for the Hello World application will have no branching points, because there are no conditional statements. The CFG for this program should look like the CFG in Figure 6.18. This looks quite similar to what happens if you perform a depth-first traversal of the AST, shown in figure 6.19. Looking at the two graphs, there are differences between them. The first one is that the target CFG has a START and an END vertex. When performing the depth-first traversal, it would be simple to add a start vertex to the graph first, and link that to the root node of the AST. At each step, the last vertex found could be stored in a global variable (outside of the recursive depth-first traversal), and after the traversal has finished, the last vertex could be linked to an END vertex.

Another difference that must be addressed is that there are more vertices in the traversed AST than there are in the target CFG. This is because the AST contains all every detail of the program, so the code `"Hello World".println()` is actually split into 3 vertices. The first is the string, then the point, and finally the call to the operation println. In the CFG, we are only interested in the call to the println function, the other details are not relevant to control flow. Each vertex in the AST has a type associated with it, which makes it simple to filter out types of vertices that are not relevant to the CFG. There are many different types, and so rather than blacklisting certain types, I have opted to whitelist certain types of vertices. From this hello world program, I can see that I need to add block and operation call to the whitelist.

In order to correctly join up the CFG, the use of the global variable that points to the last found AST vertex will be modified slightly. At each AST vertex, when the type is contained in the whitelist, the previously found vertex will have its child list updated to include the current vertex, and then the global pointer to the last vertex will updated to point to this vertex. This is probably easier to understand with pseudocode:

```
CFG last;

depthFirstAST(AST current)
```

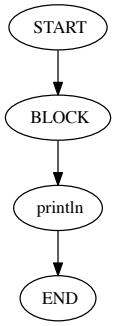


Figure 6.18: The target CFG for the Hello World program

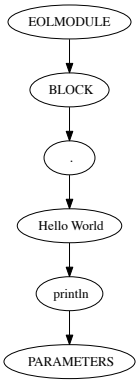


Figure 6.19: The result of a depth-first traversal of the AST

```
4   if whitelist.contains(current.type)
5     last.addChild(current.cfg)
6     last = current
7   end if
8
9   foreach child in current
10    depthFirstAST(child)
11  end foreach
12 end
```

At this point it hasn't been discussed how an AST object will link to a CFG object. Because the conversion code makes use of both, it needs to be easy to switch between accessing the two. One possible way of doing this is to use a hashtable with the AST vertex as a key, and a CFG object as the value. An alternative option is to modify the AST class to have an associated CFG class. Both approaches have their merits and drawbacks. The first method means that only the number of CFG objects need to be created as is absolutely necessary. However the second approach means that an AST can easily pass information into the constructor of the CFG, without it having to be dealt with by the class that is doing the conversion from AST to CFG. Because of this, I have opted to go for the latter approach, which is why in the pseudocode there is a call `current.cfg` that represents getting the CFG from the AST object `current`.

6.3.4 The if statement

Now that the core of the algorithm is designed, it needs to be extended to deal with statements that can send the control flow in more than one direction. I will begin by looking at the if statement, because it is one of the more simple statements.

When the depth first search reaches an if statement, it first of all needs to determine if it's an if or an if .. else statement, by looking at the number of children that it has. For the time being we assume that it has found just an if statement. The algorithm needs to add an edge from the if statement vertex to the vertex that comes after the if block. The naive approach to this would be to get the next sibling of the if statement. This doesn't work though when the if statement is the last statement within the current block. This could be fixed by looking at the parent of the sibling of the current block, but the code quickly becomes tricky to manage.

The situation is further complicated when we consider what was discussed in the analysis section about the final statement in a loop going back to the for or while loop vertex. So the problem in short then is that we don't know where the next whitelisted vertex in the AST is going to be, so we can't add an edge from the if statement to the next statement. For the time being we ignore

```
1  CFG last;
2
3  depthFirstAST(AST current)
4    if whitelist.contains(current.type)
5      last.addChild(current.cfg)
6      last = current
7    end if
8
9    foreach child in current
10     depthFirstAST(child)
11   end foreach
12
13   if current.type = EOL.IF
14     CFG endIF = new CFG("END IF")
15     current.cfg.addChild(endIF)
16     last.addChild(endIF)
17     last = endIF
18   end if
19 end
```

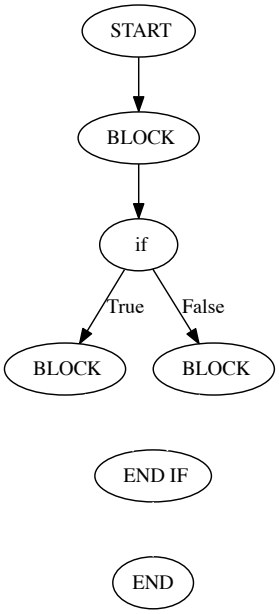


Figure 6.20: An incomplete if .. end CFG

this and look at the other path from the if statement - when it evaluates to true there will be some more vertices to add to our CFG. The depth-first traversal of the AST will add these correctly to the CFG. Because the depth-first traversal is recursive, it is quite easy to know when this process is complete. Any code that is placed after the recursive calls in a function will only be executed once the recursion has completed, so when the recursive function is called on the if statement, the CFG will have been updated after the point where the recursive call is made. In the code sample above, any code placed between the end of the foreach loop and the program end (between lines 11 and 12) would be dealing with the CFG after the if statement's children had been added to the CFG.

So why is this useful? At this point we now know that we're dealing with an if statement, and the global pointer to the last vertex to be added to the CFG points to the last vertex from the if statement's block. We still don't know where the next vertex is going to be, but we somehow need to link the if vertex and the vertex pointed to by the global last pointer to the next vertex. This can be done simply by introducing a new single vertex that mimics the next vertex and joining both vertices to it. The new vertex can be labelled 'END IF' for convenience. Both paths from the if statement ultimately end up at this 'END IF' vertex, and the global pointer to the last vertex is changed to point to the end if vertex. The code to do this loosely looks like this:

This of course now means that there is an extra vertex in the CFG. Thankfully it can be removed once the CFG has been completed, but this will be discussed in a later section.

6.3.5 The if .. else statement

The if .. else statement complicates matters slightly. If the approach described for the if statement is applied here, then it will result in an if statement that appears to execute both the true and false blocks sequentially, regardless of the value of the if statement's parameter. To prevent this from happening, when the if .. else statement is reached by the recursive statement it can immediately add an edge from the if statement to both child blocks. The CFG class has to be then modified to add a function that blocks more parents from being added. This function will be called on the false child block of the if statement.

What then happens is the depth first search of the if statement's children occurs as usual, but when it gets to the point that the final vertex of the true block tries to add the first statement of the false block as a child, it will be prevented from doing so by the newly introduced 'parent-blocking' feature. The first statement of the false block will be pointed to by the global last pointer, and the depth first traversal will continue down the if statement's children.

Once both blocks have been traversed and added to the CFG, we end up with something like what is shown in Figure 6.20. The problem is now that we no longer have a pointer to the last vertex in the true block, because it was overwritten when the false block was traversed. A naive solution would be to add another global variable that stores the position of the last vertex within the if statement's true block. Initially it sounds like a good idea, until you think about the case where we have a nested if statement. Once the inner if statement had been traversed and added to the CFG, the end of the last vertex in the true block of the outer if statement would have been overwritten when it was used by the inner block, and so this is not a good solution.

This has been a particularly difficult problem to design a solution for, and unfortunately as a result the solution that I propose is not particularly efficient. I suggest that when an if .. else vertex is reached by the depth-first procedure, that it does a depth first traversal on each of the child blocks of the if .. else statement. After each depth-first traversal has completed, the location of the final whitelisted vertex from that child block will be stored. Once this procedure has completed, we will have the locations of the final vertices for each child block of the if statement, and these can now each have an edge added from them to an 'END IF' vertex.

One final consideration must be made with the if .. else statement, which also applies to the if statement. If a break, breakAll or continue is called within either the true or false block of the if statement, then these do not want to be connected to the 'END IF' statement, and so a check will have to be performed before the edge is added.

6.3.6 Loops

As discussed in the analysis section, for and while loops will be considered as almost the same thing in the algorithm because their AST representations are very similar, and how they are to be represented in a CFG is the same.

The general case for a loop is surprisingly simple. When the recursive procedure is called with a while or for loop as its parameter, the depth-first traversal of its children can happen as normal. The contents of the loop's block are added to the loop vertex on the CFG. Once the traversal has completed, an edge is added from the global last pointer (which will point to the last vertex from the loop block) to the loop vertex. The global last pointer is then changed to point to the loop vertex, so that the next whitelisted vertex to be discovered will be added as a child to the loop vertex. This matches what was discussed in the analysis section, and the example shown in Figure 6.8.

Once again the case where a break, breakAll or continue statement exists must be considered. They are particularly relevant in this section because each of the statements is used to change control flow within a loop.

The break statement breaks out of the current loop. An edge must be added to the break vertex that connects it to the next statement to be executed after the while loop. To do this, when the recursive procedure is called with a break statement, it will need to find the next edge to be executed. Unfortunately with this we cannot store the location of the break vertex in a global variable and wait until the next whitelisted vertex is found because there could be multiple break statements before the next vertex is found. This could be solved by using a list, except that the next break statements to be found could be within another nested loop. Consider the following Java code.

```
1 private void loopSample() {
2   int i = 0;
3   while (i < 5) {
```

```
4     int j = 0;
5     if (i == 4)
6       break;
7     while (j < 5) {
8       j++;
9       if (j == 3)
10        break;
11    }
12  }
13 }
```

The code is rather contrived, but still valid. When the first break statement is found, it could be added to a list of discovered break statements with the intention that the the recursive function at the first statement after that while loop could check the list for any break statements, and if there were any it could remove them and add an edge from the listed break statement to the current vertex. This is fine, except that then the while loop on line 7 would be the first to check this list after the break statement on line 6 had been added. So a check could be added to see if the global last pointer points to a loop vertex. If it does, then the list is checked. However this does not work, because the increment operator statement on line 8 now satisfies that condition, and so an edge from the break on line 6 is added to the assignment on line 8!

Unfortunately the actual solution is not a tidy one. Thinking about the position of a break statement in the AST, first of all the loop that contains the break statement must be found, which can be done by looking at the parent, then the parent's parent, the parent's parent's parent etc, until a loop is discovered. Once that happens, a check can be done to see if the loop has a sibling, and if not then the parent can be checked. If the parent is a loop then that is the statement to link the break statement to. If not, then if there is a sibling of that parent then that is the statement to link the break statement to. If not, then the checking of parents and their siblings continues until a suitable candidate is found, or until there is nothing more to check, in which case the break statement is linked to the program's end statement. This is a difficult procedure to explain, but it should become clear in the implementation section when there is some code to explain.

For the breakAll statement the process is similar, except that the outermost loop must be discovered. This is done by going through all the parents of the breakAll statement, recording the position of the last discovered for or while loop, until the root of the AST is found. Then the statement that follows the outermost loop's block is found by looking at the sibling of the loop. Again if this does not exist, then an edge is added from the breakAll statement to the end vertex of the CFG.

The continue statement is relatively simple. From the continue statement's position in the AST its parents are searched until the loop that it is contained within is found. Then an edge is added from the continue vertex in the CFG to the loop vertex.

For each of the break, breakAll and continue statements, there can be no more children than the ones that were added once they were discovered. The depth-first traversal of the AST will continue after their discovery, but the next discovered whitelisted vertex should not be added as a child to the continue, break or breakAll vertices. The CFG class already has a 'parent-blocking' feature, and now it must also have a 'child-blocking' feature. This will enabled on the CFG when a break, breakAll or continue statement has been discovered and its correct child been found by the approach just described, and will prevent other children from correctly being added to those vertices.

6.3.7 Switch statements

The switch statement is unique in that it can potentially any number of children from its vertex in a CFG. Remember that EOL by default is not a fallthrough switch statement, unlike Java. The first step in the conversion from AST to CFG for the switch statement is to find every case statement and add an edge from the switch vertex the discovered case vertices. As well as case statements, if a default statement is present then is should also be added as a child to the switch vertex.

Once this is done, in each case statement's block the final vertex needs to link to an 'END SWITCH' block (for the same reasons as detailed in the if statement section). Unfortunately the way to do this is not particularly efficient. When each case statement is reached, a separate depth-first traversal will need to find the final whitelisted vertex within that case statement's block, and add an edge from it to the END SWITCH vertex. This could wait until the case block has been fully traversed by the normal depth-first procedure, but the end switch block would need to be kept in scope. This is more of a programming matter, and will be discussed in more detail in the implementation section.

The next consideration to make about the switch statement is that the default statement may or may not be present. If it is, then it will be included, but if it is not, then an edge must be added from the switch vertex to the END SWITCH vertex to represent the situation when none of the case statements are executed.

Finally, the continue statement within a case block has to be accounted for. If a continue statement is found, then it will have the 'child-blocker' feature enabled to prevent it from linking to the END SWITCH statement. Then a the next case statement will be found by looking at the AST to find the case vertex that contains the continue statement. The next sibling of this case statement will be found, and the block vertex under the case statement will be added as a child to the continue vertex. From there, the last whitelisted statement within that case block will need to be found, and have the next case statement's block vertex added as a child. This process will repeat until there are no more case statements (or a default statement is found).

6.3.8 Operations

In the analysis section it was decided that operations will be listed as separate graphs to the main program. This means that essentially they are sub-programs, and can be treated as such when it comes to converting them from an AST to a CFG.

The AST to CFG conversion main class can be built in such a way that it takes in the root vertex of an AST, and produces a CFG from that. So when an operation definition is discovered, it's vertex can be passed into a new instance of the AST to CFG class, and the CFG that is produced can be added alongside the main program CFG when the graphical output is produced.

One small change must be made to the AST that is passed into the new instance of the AST to CFG class, and that is that the root vertex must have the link back to its actual parent removed. This is because various statements (such as breakAll) work in a way that they search as far as the root node, which is identified as the node which has no parent associated with it. If the operation vertex was passed in without removing the link back to its parent, then such searches could link to vertices that are outside of the scope of the operation definition.

6.3.9 Branch Coverage

Once the CFG has been generated, it can be used to perform branch coverage analysis. As with statement coverage a class that implements `IExecutionListener` will be implemented. A record needs to be kept about which branches have been taken. This information can be stored either in the execution listener class, or the CFG class could be modified to store it.

If the information about which branches have been executed is stored in the execution listener, then a new class will need to be defined that stores which vertex the execution started at and which one it went to. It is going to be a lot tidier to modify the CFG class to also include which branches have been executed. When the execution listener is fired on a statement, the associated CFG and parent CFG will be found, and the parent CFG will store that the child was executed.

There is a slight complication that must be considered, which is that unlike an AST, a CFG vertex can have multiple parents, but only one of them will need to store that the child has been executed.

To solve this problem, a pointer to the last executed AST vertex will be stored in a global variable. The parent CFG will be accessed through the last executed AST vertex.

Once the EOL program has been executed, the number of branches in total and the number of executed branches will be counted, to satisfy requirement F-06. Satisfying F-07 is a bit more difficult, because giving a visual representation of which branches have executed is tricky. In EclEmma the branches are highlighted in Eclipse, and when you hover over them it displays how many of the branches were executed. This would be ideal, but as with statement coverage, implementing an Eclipse plugin is very time-consuming. Instead I will initially colour the edges of the output graph red if they have not been executed, or green if they have been.

6.4 Implementation

Now that an analysis and design of the conversion have been done, the implementation can be detailed. The last section of this report covered the high level algorithm details, but this section will drill down into more detail in areas where it helps for the understanding of the algorithm. All coding was done in Java because Epsilon is written in Java, and to make use of the AST class is easier if the rest of the code is in Java.

All of the code is included in the appendices. Where appropriate, small sections of code will be copied into this section, but for the most part the appendices should be referred to.

6.4.1 Code Structure

There are two main classes. The first is the class that does the conversion, and the second is the class CFG that represents a vertex in a CFG.

The conversion class can be broken down into three parts: Pre-recursion, recursion and post-recursion. To begin with, the recursive function is called with the root of the AST to be converted. Within the recursive function, a check is done to see if the node type is whitelisted. If so, it adds it as a child to the last whitelisted vertex (if there is one, if not, the start vertex is used). Then the `handleAST` procedure is called with three parameters. The first is the current AST, the next is the CFG associated with the current AST (if it is whitelisted, otherwise the previously found whitelisted CFG is passed in). Finally, a list of CFG objects is passed in. Sometimes the `handleAST` needs to store pointers to relevant CFG objects, and so this list is used to do that.

Once the pre-recursion methods have finished, the recursive method is called with every child of the current AST vertex. The recursive method return a list of CFG objects that may be of use in the post-recursive method of the parent, although it may also return an empty list.

Finally, the post recursive method is called. This ties up any loose ends, such as joining the end of each case statement to an END SWITCH vertex or adding the END IF statement and joining the branches of the IF statement to it.

Once the CFG has been created, a final pass is made to search for any END IF or END SWITCH statements. If any are found, then they are removed and their parent CFGs are connected to their child CFGs. If any CFGs are found that don't have a child, they are linked to the program end vertex.

6.4.2 Extensibility

The code as it is written at the moment is not particularly extensible. Within the pre and post recursion methods a switch statement is used so that different code can be executed on different

types of statement. If time permits then I will look at refactoring the code in a way that allows easy modification and maintenance. This could be useful if the conversion is changed to work on another language, such as one of Epsilon's other languages.

6.4.3 Execution Listener

The approach to recording which branches have been executed was discussed in the analysis section. There were some additional complications however that had to be catered for, and so the code is not quite as straightforward as was suggested in the analysis. One example complication is the switch statement. When a switch statement is reached, the `aboutToExecute` function in the execution listener is fired for each of the case statements. This means that with the algorithm outlined in the analysis, each of the case statements is marked as executed, despite that not being true. To counter this, the code was modified to ignore case statements, and when a block was executed it was checked if it is the child of a case statement, and if so, that case statement is marked as executed.

Once execution of the EOL program has taken place, a depth first search is performed on the CFG (with a list of checked vertices to make sure that a vertex isn't visited twice, because a CFG can have loops). At each vertex, if the number of children is greater than one, then the number of children is counted, and the number of executed children is also counted. These figures are then printed to the standard output.

When the DOT file is being generated to display the CFG, if a show branch boolean is set to true, then branches are coloured in green if they have been executed, or red otherwise.

When a test is executed, most of the code will only be passed over once at most. Branch coverage is only useful if it works across multiple tests and combines the results into one graph, and so a class was created that contains stores which branches have been executed. These results can then be merged to create a CFG that contains all of the executed branches, which is used to produce the final output.

6.5 Testing

As with statement coverage, the solution must be tested. Each of the basic statements will be tested individually, and then some combinations of statements will be tested. Obviously there is an infinite number of input combinations, and only a small subsection of these can be tested. I will attempt to choose interesting inputs that require difficult to produce CFGs, rather than something less interesting, such as 10 sequential if statements. Automatic verification of the produced CFGs would be very difficult to perform, and so all CFGs will be checked by hand. This of course means that there is a risk of human error, but at this moment I can see no way to alleviate this risk.

6.5.1 Individual Statements

The criteria for a test to pass are as follows:

- There is a START vertex at the top of the CFG which has no parents, and at least one child.
- There is an END vertex that has at least one parent, and no children.
- The produced CFG matches the one that was shown in the analysis section for that statement.
- Other than the end vertex, all vertices have at least one child

Testing begins with the block vertex. This will be done by having a single line of code in a program, with no conditional statements.

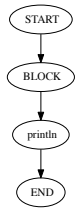


Figure 6.21

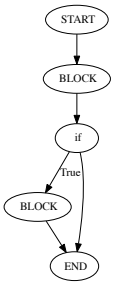


Figure 6.22

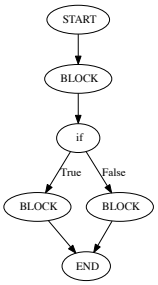


Figure 6.23

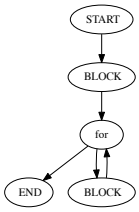


Figure 6.24

Label	BT-01
Statement under Test	BLOCK
Expected Output	See figure 6.2
Result	Pass (see Figure 6.21)

Code used for testing:

```
1 "This is a test".println();
```

Of note about this test is that the operation call `println` is printed, but not the string that is to be printed. This is because the type string is not on the whitelist, but the operation call type is. If both were on the whitelist, then two vertices would be used to represent that line of code.

Label	BT-02
Statement under Test	if
Expected Output	See figure 6.3
Result	Pass (see Figure 6.22)

Code used for testing:

```
1 if (true) {
2
3 }
```

The if statement does not require any code in its block for this test, because leaving the space between the parenthesis empty produces a block vertex, which is enough for this test. The if statement above will always execute because its parameter is `true`, but again for testing this is not relevant because the generated CFG does not take this into account.

Label	BT-03
Statement under Test	if .. else
Expected Output	See figure 6.5
Result	Pass (see Figure 6.23)

Code used for testing:

```
1 if (true) {
2
3 }
4 else {
5
6 }
```

Label	BT-04
Statement under Test	for
Expected Output	See figure 6.6
Result	Pass (see Figure 6.24)

Code used for testing:

```
1 for (i in Sequence{1..3}) {
2
3 }
```

Label	BT-05
Statement under Test	switch (with fallthrough, without a default statement)
Expected Output	See figure 6.10
Result	Pass (see Figure 6.26)

Code used for testing:

```
1 var i : Integer = 0;
2 switch (i) {
3   case 0 : "Zero".println(); continue;
4   case 1 : "One".println();
5   case 2 : "Two".println();
6 }
```

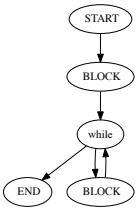


Figure 6.25

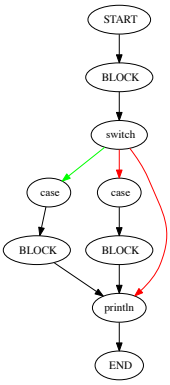


Figure 6.26

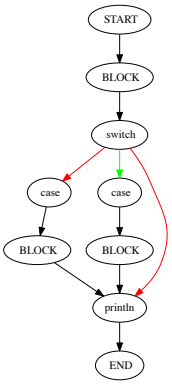


Figure 6.27

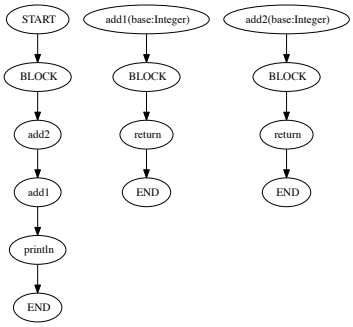


Figure 6.28

Label	BT-06
Statement under Test	switch (without fallthrough, with a default statement)
Expected Output	See figure 6.11
Result	Pass (see Figure 6.27)

Code used for testing:

```
var i : Integer = 0;

switch (i) {
  case 0 : "Zero".println();
  case 1 : "One".println();
  case 2 : "Two".println();
  default : "Unknown".println();
}
```

Both context-free and context-type operations were tested in the following test. Using the example from The Epsilon Book [7] one operation of each type was defined, and then both operations called.

Label	BT-07
Statement under Test	Operation call
Expected Output	See figure 6.28
Result	Pass (see right-hand CFG of Figure **??**). See below.

Code used for testing:

```
add2(add1(1)).println();

operation add1(base : Integer) :
    Integer {
  return base + 1;
}

operation add2(base : Integer) :
    Integer {
  return base + 2;
}
```

Notice that the operations are actually shown in the wrong order in the generated CFG. `add1` would be executed first, because it is passed as a parameter to `add2`. Looking at the AST for this code it is easy to see why. `add1` is a child of `add2`, because it is a parameter. When a depth-first traversal is performed, `add2` is found before `add1`, and so it is added to the CFG first. While this is technically incorrect, for purposes of calculating branch coverage it does not matter. However, for the more general purpose use, the operation call function would need to traverse its children to find any other operation call vertices, and add them to the CFG in reverse order. This could be done easily by recursively calling a function on the children of the AST vertex, and after the recursion checking to see if the type is an operation call. If so, then add it to the CFG and update the global last pointer to that vertex.

For the break, breakAll and continue statements, they were placed in an if statement so that the control flow was also created for when they are not executed. In separate testing that is not documented due to space constraints, they were also tested on their own and proved to work as expected.

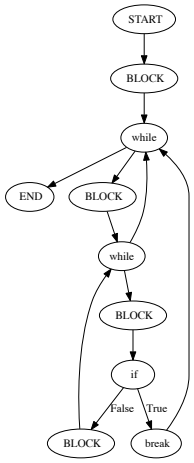


Figure 6.29

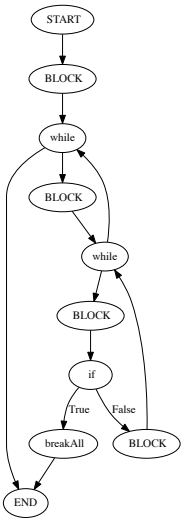


Figure 6.30

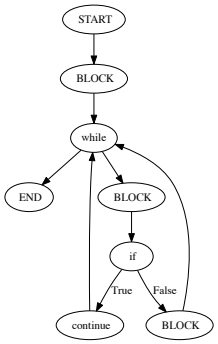


Figure 6.31

Label	BT-08
Statement under Test	Break
Expected Output	See figure **??**
Result	Pass (see right-hand CFG of Figure 6.29). See below.

Code used for testing:

```
while (true) {
  while (true) {
    if (true) break;
    else { }
  }
}
```

For break, a nested while loop was used to prove that the break statement only breaks out of one while loop, unlike the breakAll statement.

Label	BT-09
Statement under Test	BreakAll
Expected Output	See figure **??**
Result	Pass (see Figure 6.30). See below.

Code used for testing:

```
while (true) {
  while (true) {
    if (true) breakAll;
    else { }
  }
}
```

For breakAll, a two while loops were used to prove that the breakAll statement can break out of more than just the one loop that the break statement does.

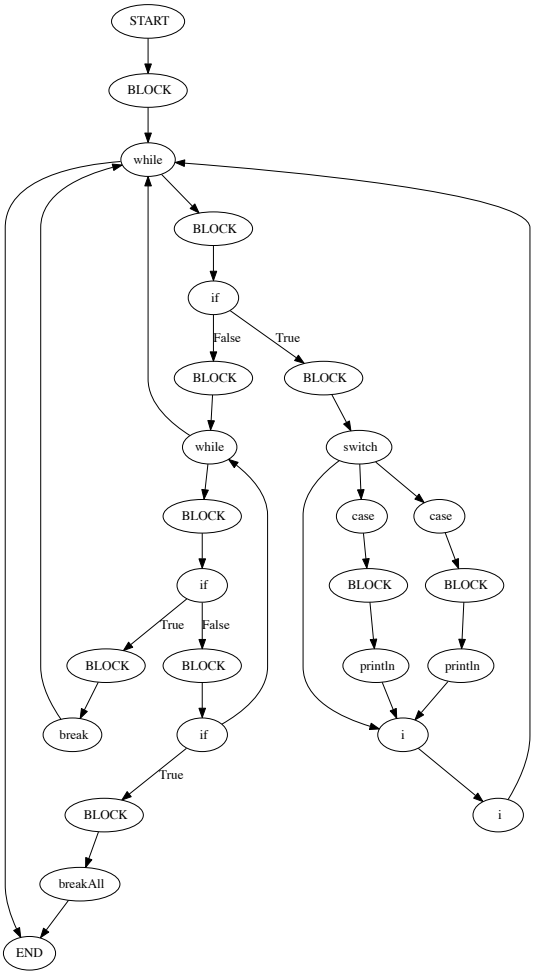
Label	BT-10
Statement under Test	Continue
Expected Output	See figure **??**
Result	Pass (see Figure 6.31). See below.

Code used for testing:

```
while (true) {
  if (true) continue;
  else { }
}
```

6.5.2 Multiple Statements Testing

During the previous section I showed that individual statements are dealt with as they should be. Some of the tests also had multiple statements in, such as the break statement. Now that this has been proven, bigger test case will be executed. During development of course many were tested, but once again due to the space constraints in this document I can only record one of them. The even bigger test will be when EuGENia is used as a case study, but that is more about testing how well the coverage metrics work, rather than how well the AST to CFG conversion works.



Code used for testing:

```
1  while (true) {
2    if (true) {
3      var i : Integer = 0;
4      switch (i) {
5        case 0: "zero".println();
6        case 1: "one".println();
7      }
8      i = i + 1;
9    }
10   else {
11     while (true) {
12       if (true) {
13         break;
14       }
15       else {
16         if (true) {
17           breakAll;
18         }
19       }
20     }
21   }
22 }
```

Figure 6.32: The test complex code and the generated CFG

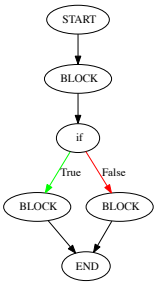
This test has been designed to use most of the statements that are covered in the analysis section. The code is highly contrived, and if code like this were seen in a real application then you would probably want to question the programmer's technique. However, that doesn't mean that my code shouldn't be able to handle it, and so in Figure 6.32 is the generated CFG on the left, and the code used on the right.

A manual walkthrough of the CFG confirms that it does as is expected. The only point of interest is at the bottom right, where there are two vertices labelled 'i'. This maps to the `i = i + 1;` statement on line 8 of the code. Analysis of the AST using the AST Explorer shows that the vertex i is of type Feature Call, which is the same as is used for operation calls. I could remove operation calls from the CFG whitelist, but I think that it is useful to have them in the CFG, and so I will leave it as is.

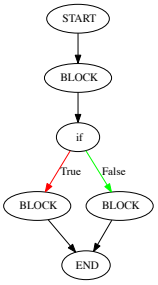
6.5.3 Branch Coverage Testing

Now that I am satisfied that the CFG is correctly, it can be used to perform branch analysis.

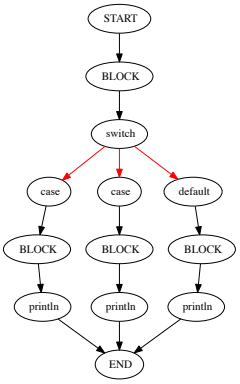
Label	BT-11
Description	An if .. else statement that just executes the if's true block
Expected Output	if .. else CFG with the edge from the if vertex to the true block vertex coloured in green, and the other side in red.
Result	Pass



Label	BT-12
Description	An if .. else statement that just executes the if's false block
Expected Output	if .. else CFG with the edge from the if vertex to the true block vertex coloured in red, and the other side in green.
Result	Pass



Label	BT-13
Description	A switch statement that only executed the default block
Expected Output	Switch CFG with the edge between the switch vertex and the default vertex coloured in green.
Result	Fail



Test BT-13 has failed because the default branch was not coloured in green, meaning that it has not been marked as executed. After some investigation, the bug was found and fixed. When a switch statement is executed, the first child of each case statement is executed. This initially caused every case statement to be marked as executed, and so an exception was added to prevent this. Unfortunately this code then broke the code that marked the default branch as being executed. A fix was implemented, and the switch statement thoroughly tested, as shown in Figure 6.33

Label	BT-14
Description	A while loop that contains a break statement that will be executed.
Expected Output	The edge into the while block should be green, but because the break statement is called the other edge from the while vertex will be red.
Result	Pass

Label	BT-15
Description	A while loop that contains a break statement that will be executed.
Expected Output	The edge into the while block should be green, but because the break statement is called the other edge from the while vertex will be red.
Result	Pass

Many more tests were carried out, but unfortunately due to space requirements they cannot be detailed here.

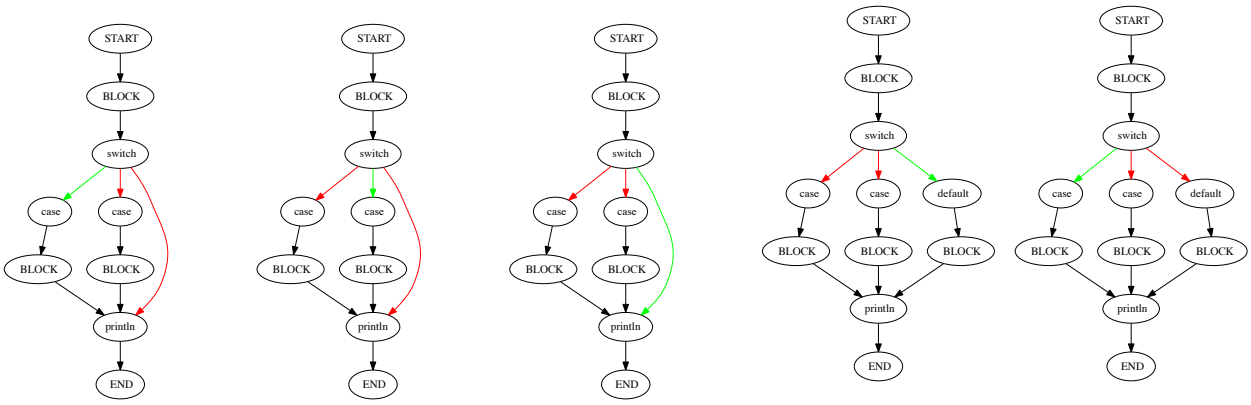


Figure 6.33: The thoroughly tested switch statement

6.6 Conclusions

The task of converting an abstract syntax tree to a control flow graph was difficult and took a lot of thought and planning. That the algorithm has not been detailed before (as far as could be found) is surprising as it seems like an algorithm that would be useful in a few situations.

Requirement F-05 has been satisfied, as the branch analysis can be run on any EOL file. F-06 has been met because the number of branches that were executed out of the total number of branches, and F-07 has also been met because the user can also distinguish between branches that have been executed, and those that have not.

NF-03 and NF-04 appear to have been satisfied, although during the case study in the next section that will be formally determined.

The code for the conversion method is included in the Appendix. As was mentioned earlier, the structure of the code does not lend itself to future extensions, and ideally needs refactoring. If an interface was implemented that included methods for pre and post-recursion, then new statements could be added to the conversion process easily. The large switch statements that exists at the moment are too big to maintain easily, but were written initially with testing the designed algorithm out.

7 Evaluation

7.1 Introduction

Testing so far has comprised of using small contrived programs to test that statements work individually, and that a few statements work together as they should. However, to test that the designed coverage tools work in the real world, a real piece of software that is written in EOL will be studied. Conveniently Epsilon includes EuGENia which is written in EOL, and also has a test suite, making it the ideal candidate for a case study.

7.2 Implementation

All tests up to this point have been executed by creating an instance of EolParserWorkbench, and then calling functions that parse and execute a test file. However, we now need to execute EuGENia's test suite which is a bit more difficult. To do this, the class EugeniaActionDelegate was modified to include code that adds the execution listeners for statement and branch analysis before execution occurs, but after the EuGENia source has been parsed so that the AST of the program is available.

The EuGENia source code has two import statement at the top. These import statements were removed, and the files that they referred to were appended to the main source code file. This is because both statement and branch coverage have both been designed with only one source file in mind, which is a shortcoming that will be discussed in the further work section.

7.3 Problems

Statement coverage worked as expected, but branch coverage had two major problems that were spotted after some inspection of the output.

7.3.1 Conditional Branches

The first problem was that when a conditional statement contained an operation call as part of the condition statement, this executing was causing the branch from the conditional statement to

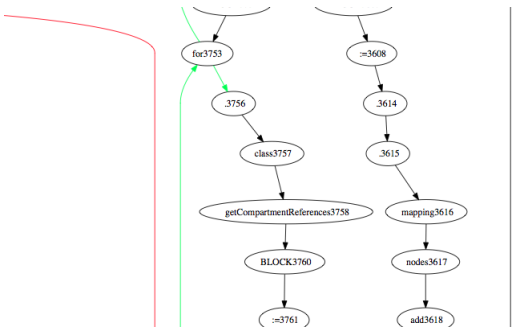


Figure 7.1: The first problem with branch analysis of EuGENia

be marked as executed, regardless of whether the block of the conditional statement was actually being executed. An example of this is shown in Figure 7.1. The for loop header contains the code `class.getCompartmentReferences()` which is split into 3 vertices immediately following the for loop vertex. These are evaluated every time the for loop is reached, and if they return true then the for loop block is executed, and if not the loop is skipped. However, the branch execution listener code was noting that when any of the three vertices were executed that they satisfied the criteria:

1. The executed vertex is a child of the last executed vertex

2. The CFG of the executed vertex is a child of the CFG of the last executed vertex

3. The executed vertex is a valid type (i.e. whitelisted)

With these conditions satisfied, the branch was being marked as having been executed. Two possible solutions were available. The first is to modify the AST to CFG conversion so that the vertices in the conditional statement are not executed. An attempt was made to implement this solution, but because of how the conversion algorithm deals with the case where there is no code following a loop block, it was not possible to do this 'cleanly' (i.e. without lots of nasty conditional statements that would likely introduce bugs). Instead, the branch execution listener was modified so that the extra condition was included:

4. If the parent vertex is of type HELPERMETHOD then this vertex must be of type BLOCK

The additional condition ensures that only the execution of a block under a conditional statement will cause the branch to be marked as executed.

7.3.2 Operations

Simple operation calls were covered during testing, and were thought to work correctly. However, while studying the output of branch analysis on the EuGENia source code, it was apparent that something was not working correctly as some branches were showing that no children had been executed, despite the fact that there definitely should have been at least one child executed (such as at an if .. else statement). An investigation of the cause showed that the problem was limited to when there was an operation call just prior to the problem conditional statement. In the branch execution listener there is a pointer to the last executed vertex that is used when calculating which two branches have been executed. When an operation is called, the pointer will be updated within that operation. When control returns to the code that called the operation, the pointer will still be pointing to a vertex within the previously called operation. So when a branch is executed, the branch execution listener will decide that there is no connection between the last executed vertex pointer and the branch that has been executed, and not mark the branch as been executed as it should do.

To solve this problem, a stack was introduced. The idea was that when an operation call is executed the last executed vertex pointer is pushed to the stack, and when the operation call has completed, the stack is popped and stored in the last executed vertex pointer. The implementation was not quite as easy as it could have been. A function had to be implemented that calculated whether an operation call had been executed by looking at the relative positions of the last executed vertex and the current executed vertex in the AST. The contents of the stack also have to be checked at each vertex execution because it seems that sometimes the execution listener doesn't always fire the end of an operation call vertex, and it can be necessary to pop more than one pointer from the stack on some occasions.

7.4 Results

7.4.1 Statement Coverage

The results of the statement coverage are found in Appendix **??**. The coverage of the existing test suite for EuGENia is 49%. The output from the analysis clearly shows which section of the code have not been executed, and what needs to be tested in new test suites. For example, the for loop that 'Processes EAttributes' (according to the comment above it) is never executed, and so none of the tests must contain models with EAttributes.

A lot of the non-executed statements are those in functions that were originally in the includes, Formatting.eol and ECoreUtil.eol.

7.4.2 Branch Coverage

Out of 235 branches in EuGENia, 143 were executed (60.9%). The graph that is produced is very large, too big to include as a whole graph. Instead the output code was modified to print operations as separate graphs so that they can be included more easily in Appendix.

8 Conclusions

Calculating code coverage for unit tests is not a new idea. This project has introduced two new things. The first is a framework for statement and branch coverage for EOL programs. The second is a general algorithm for converting an Abstract Syntax Tree to a Control Flow Graph. The algorithm is tailored for EOL syntax, but the descriptions are abstract enough that the process could be applied to any language with a similar syntax.

A very high level algorithm was provided by Grune [27] in the book 'Modern Compiler Design', but this was the only literature that could be found that even mentioned the conversion process. The algorithm given in the book is very high level, and does not provide any level of detail, except for an example about an if statement. In this project I have detailed each statement and explained the considerations that must be made.

8.1 Further Work

8.1.1 Conversion Algorithm

An attempt has been made to detail the conversion algorithm so that it is applicable to any language with a syntax that uses statements common to most programming languages (if, else, switch etc.). Despite this, it has designed around EOL, and therefore may require some modification before it can be used in other languages. One notable case of this is the switch statement that implements fallthrough in a different way to Java and most other modern object-oriented languages. Therefore before the algorithm could be used for Java, how the switch statement should be represented as a CFG would need to be considered.

Some statements that are common across most OO languages are not included in EOL. One prime example is the do .. while statement. The CFG of a do .. while statement would be unlike any other loop because flow would immediately go to the block of the loop, and then at the end the evaluation of the conditional statement would take place.

8.1.2 Languages

Epsilon has many languages for performing operations on models. EOL is the core language, upon which the other languages are built atop [7]. It should not be difficult then to extend the conversion algortihm to work with the other languages and provide coverage for all Epsilon languages.

To adapt the algorithm to work with non-EOL based languages, all that is required is the ability to perform a traversal of the AST for the program, and the ability to identify different types of statements in the tree.

8.1.3 Output

The implementation of the conversion algorithm currently only outputs DOT files that can be rendered in GraphViz. Likewise for statement coverage, the only available output is a HTML file. From a user-perspective, an Eclipse plugin similar to EclEmma that provides a quick visual overview

of coverage would be useful. Currently my code provides no public API for getting coverage information, but it could be easily adapted to have one like JaCoCo, and tools could be created that generate prettier outputs.

Bibliography

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

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

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

[4] Eclipse. Creating an emf 1.1 model using a graphical editor. [Online]. Available: http://www.eclipse.org/modeling/emf/docs/1.x/tutorials/glibmod/glibmod_emf1.1.html

[5] O. M. Group, *Unified Modeling Language*. [Online]. Available: http://www.omg.org/spec/UML/

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

[7] DimitrosKovolos, *Epsilon Book*, 2013.

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

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

[10] M. Gouyette. Chapter 2. getting started with gmf. [Online]. Available: http://www.kermeta.org/docs/fr.irisa.triskell.kermeta.samples.fsm.documentation/build/html.chunked/KerMeta-Create-FSM-Graphical-Editor-With-GMF/index.html

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

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

[13] 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

[14] 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.

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

[16] T. H. H. Wolpe. Gao report: Patriot missile system. [Online]. Available: http://www.fas.org/spp/starwars/gao/im92026.htm

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

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

[19] 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.

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

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

[22] M. R. Hoffmann. Jacoco - java code coverage. [Online]. Available: http://www.eclemma.org/jacoco/trunk/index.html

[23] ——. Elcemma - java code coverage for eclipse. [Online]. Available: http://www.eclemma.org/

[24] Ncover | .net code coverage. [Online]. Available: https://www.ncover.com/

[25] B. Insider, "What is stakeholder?" Online, 2014. [Online]. Available: http://www.businessdictionary.com/definition/stakeholder.html

[26] T. U. of York, "Enterprise systems group," Online. [Online]. Available: https://www.cs.york.ac.uk/research/research-groups/es/

[27] D. Grune, *Modern compiler design*, ser. Worldwide series in computer science. John Wiley, 2000. [Online]. Available: http://books.google.co.uk/books?id=maVQAAAAMAAJ

[28] Graphviz - graph visualization software. [Online]. Available: http://www.graphviz.org/

[29] The dot language. [Online]. Available: http://www.graphviz.org/doc/info/lang.html